Tiny Tapeout IHP 26a Datasheet

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

March 24, 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/ttihp26a-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]

Neuromorphic Tile

by **Justin Long**

0005 HDL Project

`github.com/crockpotveggies/neuron-ttihp`

"Event-driven neuromorphic neuron tile with selectable modes and on-chip learning"

This document describes the live RTL implementation of tt_um_crockpotveggies_neuron.sv.

The active design is a programmable, event-driven neuron core with:

- a TinyTapeout-facing wrapper
- a 4-bit saturating datapath
- a 16-entry ternary-clamped weight bank
- a 16-word microcode store
- a single-op microsequencer
- a reusable 2-entry event FIFO

The old primitive-composition architecture is no longer the active design.

Active Module Structure

Wrapper

- tt_um_crockpotveggies_neuron.sv

The wrapper owns the TinyTapeout boundary:

- pin mapping
- synchronized input capture
- host request de-duplication
- command decode into `tt_cmd_t`
- forwarding the held output beat

Reusable common blocks

- tt_io_frontend.sv
- tt_event_decode.sv
- tt_event_fifo.sv
- rv_if_t.vh
- event_types.vh

These are shared infrastructure for the wrapper boundary, event queue, and packed structs/constants.

Core hierarchy

- neuron.sv
- neuron_csr.sv
- neuron_ucode_store.sv
- neuron_weight_bank.sv
- neuron_state.sv
- neuron_exec.sv

`neuron.sv` is the core top. It owns the event scheduler and sequences one micro-op per cycle for each in-flight event.

Execution Model

The design is event-driven, not free-running.

- `CMD_EVENT` pushes events into the 2-entry FIFO.
- If the FIFO is non-empty and no output beat is being held, the core starts servicing one event.
- The sequencer executes one micro-instruction per cycle.
- Intermediate state is kept in the core's working registers while the event is in flight.
- The architectural RF, metadata, output latch, and weight bank commit once, at the end of the event.

That gives the core:

- a smaller combinational cone than the older unrolled executor
- stable per-event semantics, because commit is still atomic

Important constraints:

- there is no branch instruction
- control flow is still only `vector_base[tag]` plus `ucode_len_r`
- the datapath step in `neuron_exec` is combinational, but the overall core is a clocked state machine

Architectural State

Register file

The core stores eight signed 4-bit registers:

- `R0 = V`
- `R1 = I`
- `R2 = TH`
- `R3 = R`
- `R4 = T0`
- `R5 = T1`
- `R6 = W`

- `R7 = AUX`

All arithmetic saturates to `-8..+7`.

Non-RF state

The core also stores:

- `last_sid[3:0]`
- `last_tag[1:0]`
- `last_time[5:0]`
- `spike_flag`
- `have_out`
- `out_data_r[7:0]`

Weight memory

neuron_weight_bank.sv stores 16 signed 4-bit entries, but the committed legal values are always ternary:

- `-1`
- `0`
- `+1`

Both direct host writes and `STDP_LITE` writeback are clamped into that set.

Microcode memory

neuron_ucode_store.sv stores 16 words of 16 bits each.

- programming is byte-oriented through `CMD_UCODE`
- `ucode_ptr_r[0]` selects low vs high byte
- `ucode_ptr_r[4:1]` selects one of the 16 words

Programming Surfaces

The core is programmed through:

- `CMD_CSR`
- `CMD_WEIGHT`
- `CMD_UCODE`

The event path (`CMD_EVENT`) only feeds the FIFO.

CSR-owned state

neuron_csr.sv owns:

- `CSR_CTRL` pulse decode
- `ucode_ptr_r`
- `ucode_len_r`
- `vector_base0_r..vector_base3_r`
- `init_rf_flat`

Default reset image:

- `V = 0`
- `I = 0`
- `TH = +7`
- `R = 0`
- `T0 = 0`
- `T1 = 0`
- `W = 0`
- `AUX = 0`

Reset And Enable Behavior

Hard reset

- `rst_n = 0`

This clears:

- CSR state
- microcode store
- weight bank
- FIFO contents
- runtime neuron state

Disable

- `ena = 0`

This clears runtime state aggressively:

- FIFO cleared
- weights cleared to zero
- `neuron_state` reloads `init_rf_flat`
- metadata cleared
- held output cleared

Soft runtime reset

Triggered by `CSR_CTRL.bit0`.

This reloads the live runtime state from `init_rf_flat` and clears:

- `last_sid`, `last_tag`, `last_time`
- `spike_flag`

It does not erase microcode or the persistent weight bank.

Output and FIFO clear pulses

`CSR_CTRL` also provides:

- `bit1`: clear held output beat
- `bit2`: clear the event FIFO

Numeric And Transport Encodings

Arithmetic precision

- RF values: signed 4-bit
- weight storage: signed 4-bit, ternary committed values
- event metadata: `sid[3:0]`, `tag[1:0]`, `event_time[5:0]`

Host-side ternary weight encoding

- `00` -> `0`
- `01` -> `+1`
- `11` -> `−1`
- `10` -> treated as `0`

Output beat format

`uo_out[7:0]` is the held output beat:

- `uo_out[7] = 1`
- `uo_out[6:5] = emitted tag`
- `uo_out[4:1] = last_sid`
- `uo_out[0] = spike_flag`

The beat is held until acknowledged on `uio_in[1]`.

Notes On Time

`event_time` is captured by `RECV` and stored in `last_time`, but the current core does not automatically compute decay from timestamp deltas.

All decay remains explicit and shift-based:

- `LEAK`
- `TDEC`
- the non-spike decay branch of `REFRACT`

Related Documentation

- command_protocol.md
- isa.md
- layer_examples.md

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
`0`	IN_DATA[0] (addr[0])	OUT_DATA[0]	IN_REQ (in) / IN_ACK (out)

#	Input	Output	Bidirectional
1	IN_DATA[1] (addr[1])	OUT_DATA[1]	OUT_ACK (in) / OUT_REQ (out)
2	IN_DATA[2] (addr[2])	OUT_DATA[2]	CFG_OP[0] (in)
3	IN_DATA[3] (addr[3])	OUT_DATA[3]	CFG_OP[1] (in)
4	IN_DATA[4] (addr[4])	OUT_DATA[4] (type[0])	CFG_ARG[0] (in)
5	IN_DATA[5] (addr[5])	OUT_DATA[5] (type[1])	CFG_ARG[1] (in)
6	IN_DATA[6] (polarity)	OUT_DATA[6] (type[2])	CFG_ARG[2] (in)
7	IN_DATA[7] (is_tick)	OUT_DATA[7] (valid)	CFG_ARG[3] (in)

Borg - Tiny GPU

by **Andreas Wendleder**

0013 HDL Project

github.com/gonsolo/Borg

"A RISC-V SoC with a GPU featuring vertex shading, rasterization, and RGB fragment shading"

How it works

Borg is a tiny GPU built around a RISC-V SoC (TinyQV). It renders a 10-frame animation of a rotating RGB triangle, fully self-contained in firmware — no host-side rendering needed.

Architecture

- **CPU**: TinyQV (RV32EC), runs the application and GPU driver firmware
- **GPU core (Borg)**: FP16 FMA unit with 16 registers, 8-instruction IMEM, MMIO-accessible
- **Shaders**: Compiled from GLSL → SPIR-V → SPIR-B (custom binary format), embedded in firmware

Rendering Pipeline

1. **Vertex shader** — rotates triangle vertices using sin/cos (computed via LUT)
2. **Triangle setup** — computes edge vectors and 1/area (Newton-Raphson reciprocal on the FPU)
3. **Rasterization** — evaluates edge functions per pixel to determine coverage
4. **Fragment shader** — interpolates per-vertex RGB colors using barycentric coordinates (runs 3× per pixel, once per channel)
5. **Framebuffer** — 16×16 RGB (FP16), written to PSRAM for host readback

Software Stack

The firmware is structured like a Vulkan application:

```
const borg_vertex_t vertices[3] = {
    { .pos = { ... }, .color = { FP16_ONE,  FP16_ZERO,
FP16_ZERO } },  // red
    { .pos = { ... }, .color = { FP16_ZERO, FP16_ONE,
FP16_ZERO } },  // green
    { .pos = { ... }, .color = { FP16_ZERO, FP16_ZERO,
FP16_ONE  } },  // blue
};

int main() {
```

```
    borg_init(vert_borg, ..., frag_borg, ...);
    for (int frame = 0; frame < 10; frame++) {
        borg_set_angle(&draw, angle);
        borg_cmd_draw(&draw, vertices, frame);
        borg_present(frame);
        angle = fp16_add(angle, FP16_36DEG);
    }
}
```

How to test

In `./fpga`:

```
make triangle
```

This programs the firmware, boots the FPGA, waits for all 10 frames to render, then reads back 10 PPM images (`triangle_00.ppm` through `triangle_09.ppm`).

External hardware

- QSPI PMOD from the Tiny Tapeout store
- For FPGA development: pico-ice board and Raspberry Pi Debug Probe

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Interrupt 0	UART TX	Flash CS
1	Interrupt 1	UART RTS	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

SoilZ v1 Lock-In Impedance Analyzer

by **TechHU**

0015 Analog Project

github.com/TechHU-GS/analog-trial

"1-bit IQ lock-in sigma-delta impedance analyzer for soil moisture, IHP SG13G2 130nm"

How it works

SoilZ v1 is a 1-bit IQ lock-in sigma-delta impedance analyzer for soil moisture sensing, implemented in IHP SG13G2 130nm BiCMOS.

Signal chain: PTAT current source → cascode mirror → H-bridge excitation → external soil probe → chopper demodulator → CT-ΣΔ integrator (OTA + C_fb + R_in) → Strong-arm comparator → SR latch → TG DAC feedback → digital divider chain (VCO ÷2/÷4/÷8/÷16 with I/Q) → bitstream output.

Key blocks:

- 5-stage ring VCO (9 MHz) with PTAT bias for temperature compensation
- Programmable current source (3-bit binary weighted, 3.7-24 µA)
- Chopper-based lock-in demodulation at excitation frequency
- First-order CT-ΣΔ modulator with 1 pF MIM feedback capacitor
- Digital frequency divider with quadrature output selection

How to test

1. Connect a resistive load (1-27 kOhm) between ua[0] (probe_p) and ua[1] (probe_n).
2. The internal VCO generates the excitation clock; the digital divider selects the measurement frequency.
3. The bitstream density output on uo[] is proportional to the probe impedance.
4. Use an ESP32 or similar to count the bitstream density over a measurement window.

External hardware

- Soil moisture probe or resistive test load connected to ua[0] and ua[1]
- MCU (e.g., ESP32) to read bitstream density from digital outputs

Project Pinout

Digital Pins

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

Analog Pins

ua#	analog#	Description
0	15	Soil probe positive (H-bridge excitation + chopper sensing)
1	14	Soil probe negative (H-bridge excitation + chopper sensing)

AdEx Neuron NCS

by **Chenxi Wen**

0019 Analog Project

github.com/sunny441/ttihp26a-adex-neuron-ncs

"Adaptive Exponential Integrate-and-Fire Neuron made using DPI circuits in subthreshold regime"

How it works

This is a DPi circuit based Mixed Signal Adaptive Exponential Neuron.

How to test

This neuron is controlled by analog biases controlled by an SPI register.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	NN_ack_VABI[0]	NN_req_VABO[0]	MUX_sel0_VSI
1	NN_ack_VABI[1]	NN_req_VABO[1]	MUX_sel1_VSI
2	NN_ack_VABI[2]	NN_req_VABO[2]	NN_ack_VABI[8]
3	NN_ack_VABI[3]	NN_req_VABO[3]	NN_ack_VABI[9]
4	NN_ack_VABI[4]	NN_req_VABO[4]	NN_ack_VABI[10]
5	NN_ack_VABI[5]	NN_req_VABO[5]	NN_req_VABO[8]
6	NN_ack_VABI[6]	NN_req_VABO[6]	NN_req_VABO[9]
7	NN_ack_VABI[7]	NN_req_VABO[7]	NN_req_VABO[10]

Analog Pins

ua#	analog#	Description
0	13	BUF_prob_VX1O
1	12	NN_ref_VNI
2	11	NN_tau_VNI
3	10	V2I_cm_VI
4	9	V2I_bias_VPI
5	8	V2I_in_VI

Wedgetail TCDE REV01

by **M. L. Young**

0032 2 MHz HDL Project

`github.com/SiliconPlatformsLab/wedgetail-tester-tt-ihp26a`

"Test, Calibration and Design Exploration (TCDE) for the Wedgetail project"

How it works

Wedgetail is a project that is part of my PhD thesis. This particular design is for Test, Calibration and Design Exploration (TCDE), to assess the effectiveness of the project on real silicon.

The design consists of a configurable array of ring oscillators, a Digital Phase Locked Loop (DPLL), and an SPI register file generated with SystemRDL.

The intent is to:

- Verify the correctness of all of these components on real silicon; particularly the SPI and ring oscillator array
- Design and verify the fun logo-stamping workflow

How to test

Ring Oscillator Mux

The first 4 pins, `ROSC SEL[3:0]`, are a 4-bit mux that can be used to select a particular ring oscillator test.

Currently, the selectable options are:

Binary value	Name	Description
0	ROSC_NONE	No output
1	ROSC_32_1	First 32 stage osc
2	ROSC_32_2	Second 32 stage osc
3	ROSC_64	64 stage osc
4	ROSC_16	16 stage osc
5	ROSC_32_OR	ROSC_32_1 and ROSC_32_2 OR'd together
6	ROSC_31	31 stage osc
7	ROSC_128	128 stage osc

8	ROSC_32_AND	ROSC_32_1 and ROSC_32_2 AND'ed together
9	ROSC_32_DRIVE_8	32 stage osc with 8x drive current inverter
10	ROSC_32_DRIVE_16	32 stage osc with 16x drive current inverter

Note: Before you get mad at me for saying it won't oscillate because it's even, in all of these designs, there is an extra +1 inverter from the feedback tap. So a 32-stage oscillator has 32 inverters in the loop, plus 1 feedback inverter, making a 33-stage design. This was confirmed with full parasitics GDS-level SPICE simulation to oscillate. On the other hand, the 31-stage oscillator has +1 = 32 inverters total, so may not oscillate.

DPLL

A digital-phased lock loop is included, written by jsloan256. Clock the main module at 2 MHz, then pass a 300 KHz signal into the `DPLL CLK 300 KHz` input port. The output port `DPLL CLK` will have the signal passing through the DPLL, and the port `DPLL CLK FMULT` will have the signal passing through an 8x frequency multiplier.

SPI

A simple SPI interface and register file (generated with SystemRDL) is included.

Specifications:

- Rated frequency: 2 MHz
- Absolute maximum frequency: 33 MHz
- CS: Active low
- Clock pin: Same as system clock

For the register file documentation, see the end of this document.

SPI Programmable Ring Oscillator

A ring oscillator is included that can be programmed on the fly by SPI. Write to the `ROSC_EN_SEL` register to configure the "coding" of the ring oscillator. In this coding, each bit in `ROSC_EN_SEL` represents two inverters in the ring oscillator. For example, if `ROSC_EN_SEL[0] == 1`, then `inverter[0]` AND `inverter[1]` will be powered on.

The ring oscillator output is routed to `ROSC SPI OUT`.

LFSR

The pin `LFSR` is the 1-bit output of a 16-bit LFSR. It cannot be turned off, lol, sorry. But you can reset it with the TinyTapeout system reset.

Warnings

- Do not test the DPLL and SPI at the same time, as they run off the same clock.

External hardware

- None required

wedgetail_spi address map

- Absolute Address: 0x0
- Base Offset: 0x0
- Size: 0x4

Wedgetail SPI interface for Wedgetail TCDE REV01

Offset	Identifier	Name
0x0	SYS_CTRL	Reset
0x1	ECHO1	ECHO1
0x2	ECHO2	ECHO2
0x3	ROSC_EN_SEL	Ring Oscillator Enable Select

SYS_CTRL register

- Absolute Address: 0x0
- Base Offset: 0x0
- Size: 0x1

Bits	Identifier	Access	Reset	Name
7:0	RESET	w	0x0	—

RESET field

When any value is written to this register, a power on reset will be performed on the entire device

ECHO1 register

- Absolute Address: 0x1
- Base Offset: 0x1
- Size: 0x1

Bits	Identifier	Access	Reset	Name
7:0	DATA	rw	0x0	—

DATA field

Read/write echo register, for SPI debugging

ECHO2 register

- Absolute Address: 0x2
- Base Offset: 0x2
- Size: 0x1

Bits	Identifier	Access	Reset	Name
7:0	DATA	rw	0x0	—

DATA field

Read/write echo register, for SPI debugging

ROSC_EN_SEL register

- Absolute Address: 0x3
- Base Offset: 0x3
- Size: 0x1

Bits	Identifier	Access	Reset	Name
7:0	DATA	rw	—	—

DATA field

Select the bits enabled by the configurable ring oscillator.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	ROSC SEL 0	ROSC MUX OUT	—
1	ROSC SEL 1	LFSR	—
2	ROSC SEL 2	ROSC 32 NO MUX	—
3	ROSC SEL 3	DPLL CLK	—
4	DPLL CLK 300 KHz	DPLL CLK FMULT	—
5	MOSI	MISO	—
6	CS	ROSC SPI OUT	—
7	—	—	—

Two Song Buzzer Player

by **Simon**

0033 | 50 MHz | HDL Project

github.com/CT4111/test_WOKWI_Circute

"Plays a part Never Gonna Give You Up and Imperial March on a buzzer, switchable via input"

How it works

saves notes and the sequenz to execute them in the hardware and with the clock input adjustet to the note frequency and and the duration counted with it the chip should be able to play a part of the imperial march and never gonna give you up

How to test

set the clock to 50MHz and a switch at input 0 and 1 (to switch songs and pause) connecting a buzzer to out zero should result in a looping song beeing played

External hardware

piezo buzzer

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Song select (0=NGGU, 1=Imperial March)	Buzzer PWM output	—
1	Pause	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Oscillating Bones

by **Uri Shaked**

0034 HDL Project

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

"A stylish ring oscillator built from SkullFET transistors"

How it works

A simple yet stylish ring oscillator that uses a chain of 21 SkullFET inverters to generate a square wave output. Based on simulation, the oscillator should have a frequency of around 150.5 MHz.

Pin	Expected frequency
osc_out	150.5 MHz
osc_div_2	75.3 MHz
osc_div_4	37.6 MHz
osc_div_8	18.8 MHz

How to test

Connect an oscilloscope to one of the output pins (eg. `osc_div_8` / `uo_out[3]`) and enjoy the show.

Simulation results

The following graph shows the output of the oscillator and the divided outputs. It was generated by running `make -C docs/layout_sim.png`:

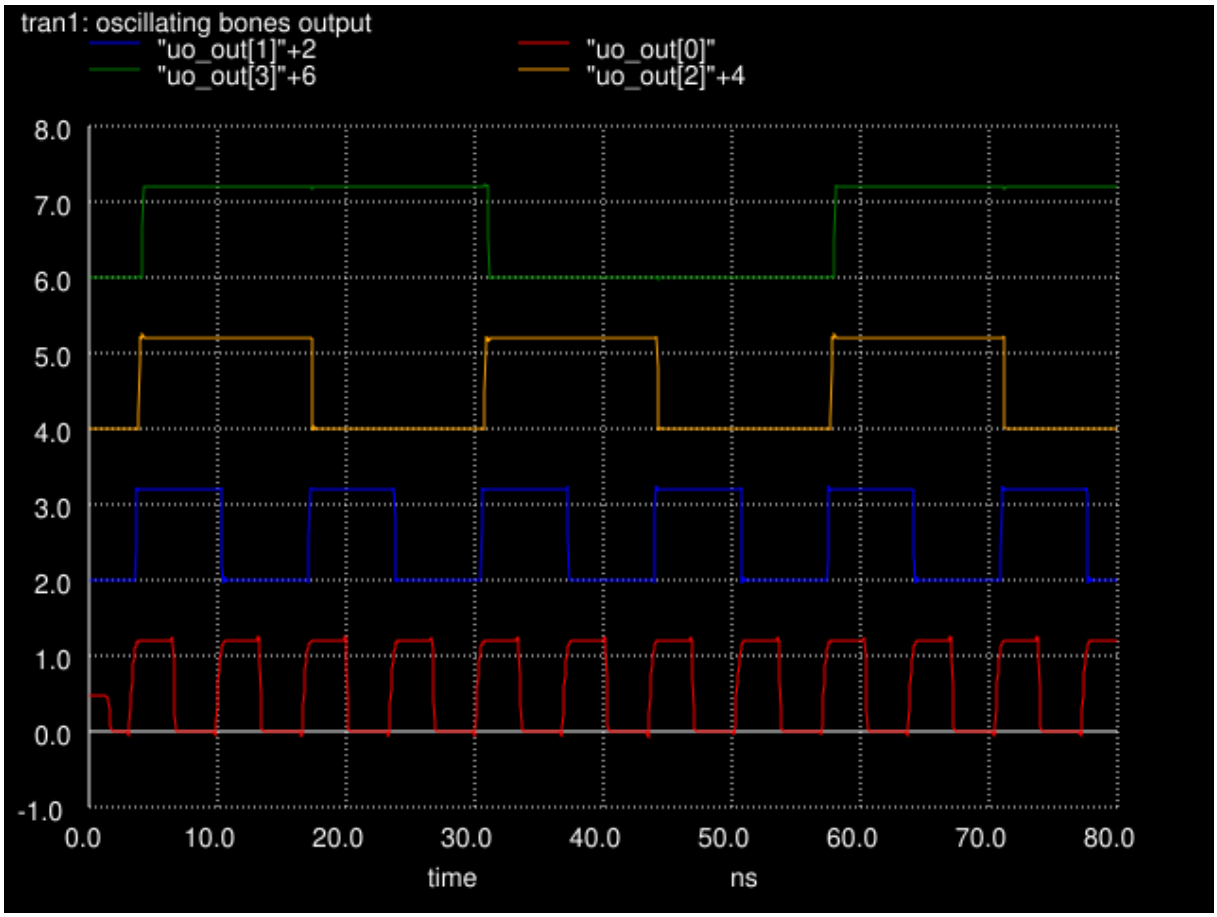


Figure 34.1: Simulation results

The outputs are shifted by 2 volts to make them easier to see in the graph. “uo_out[0]” is the main output of the oscillator and “uo_out[1]”/“uo_out[2]”/“uo_out[3]” are the divided outputs.

Please note that the simulation results do not account for all parasitics, only the primary ones. Consequently, the actual frequency of the oscillator is likely to be lower than the simulated value.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	osc_out	—
1	—	osc_div_2	—
2	—	osc_div_4	—
3	—	osc_div_8	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

RTX 8090

by **Ziyang Zhao**

0035 Wokwi Project

github.com/agentzz1/GDS_tinytapeout

wokwi.com/projects/455291738750219265

“descripstion tbd”

How it works

Hello i am making a chip lol ## How to test Explain how to use your project yessir

test hello

External hardware

hello

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

Project Pinout

Digital Pins

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

gatekeeping the gates

by **orangerot**

0037 10 kHz Wokwi Project

github.com/Geronymos/tinytapeout

wokwi.com/projects/455291728585320449

“a got your gates”

How it works

How do I know? This is just some stuff I’m doing during a workshop.

How to test

Now these points of data make a beautiful line.

External hardware

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input a	out 0	—
1	input b	out 1	—
2	input c	out 2	—
3	input d	out 3	—
4	input e	out 4	—
5	input f	out 5	—
6	input g	out 6	—
7	input h	out 7	—

FPGA

by **Can Lehmann**

0039 | 25 MHz | HDL Project | Medium Danger

github.com/can-lehmann/tt-fpga

"A tiny FPGA"

This project can damage the ASIC under certain conditions.

Since the FPGA implements a random bitstream at startup, it may lead to combinatorial loops. See the documentation for how to prevent this.

How it works

This project implements a tiny FPGA. The design is essentially a crossbar which connects 8 Inputs, 8 LUTs and 4 output ports. Because of the small size, I decided not to do any routing fabric besides the crossbar.

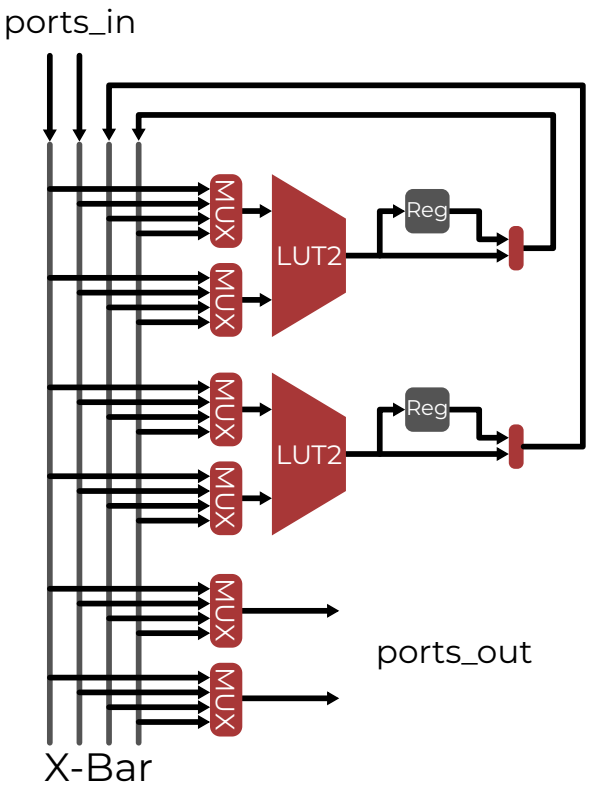


Figure 39.1: Overview of the FPGAs architecture. In reality we have 8 inputs, 8 LUTs and 4 outputs instead of just 2.

How to test

Important: Tie `Virtual Reset` high before powering on the chip.

1. Tie the `Virtual Reset` pin high to ensure that you don't accidentally get a combinatorial loop on startup.
2. Power on the chip.
3. Load in a bitstream using the `Program Data` and `Program Enable` pins. The configuration is stored in a shift register, so you can load in the configuration one bit at a time.
4. Disable `Virtual Reset` and observe your design in action!

Some of examples of what might be possible:

- Simple combinatorial logic.
- Toggling an output.
- A 4-bit counter that counts from 0 to 15 and then wraps around.

See the testbench for examples.

External hardware

No external hardware is required to run this project.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Program Data	out[0]	in[0]
1	Program Enable	out[1]	in[1]
2	Virtual Reset	out[2]	in[2]
3	—	out[3]	in[3]
4	—	—	in[4]
5	—	—	in[5]
6	—	—	in[6]
7	—	—	in[7]

WIP

by **Vincent Li**

0041 10 kHz Wokwi Project

github.com/DebuggingDisaster/wowki

wokwi.com/projects/455303914893650945

"slow down clock frequency and let the 7-seg display spin"

How it works

10.000 hz is slowed to XXXhz and the 7-seg Display shows a rotation of the outer columns

How to test

Have a look at the 7-seg display

External hardware

7-seg display

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	sw1	7-seg A	—
1	sw2	7-seg B	—
2	sw3	7-seg C	—
3	sw4	7-seg D	—
4	sw5	7-seg E	—
5	sw6	7-seg F	—
6	sw7	7-seg G	—
7	sw8	7-seg H	—

just copy 4 not gates

by **Peilong Li**

0043 10 kHz Wokwi Project

github.com/RussLi-IDK/Create-the-GDS

wokwi.com/projects/455303526551376897

"nothing"

How it works

A NOT gate is connected to input a A AND gate is connected to input b and input c A OR gate is connected to input d and input e ## How to test

External hardware

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input a	output not	—
1	input b	output and	—
2	input c	output or	—
3	input d	—	—
4	input e	—	—
5	—	—	—
6	—	—	—
7	—	—	—

WIP Title

by **Paul Harter**

0045 10 kHz Wokwi Project

github.com/plhrtr/TinyTapeOutTest

wokwi.com/projects/455291641368904705

“_”

How it works

WIP

How to test

WIP

External hardware

TODO

Project Pinout

Digital Pins

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

uCore

by **Julian**

0047 100 kHz HDL Project

`github.com/yJulian/tiny-tapeout`

"*A simple ALU with embedded registers*"

How it works

Counts up a 4-bit counter and outputs to a 7-seg display

How to test

Press button on pin 1

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	DATA IN[0]	DATA OUT[0]	IMM [0]
1	DATA IN[1]	DATA OUT[1]	IMM [1]
2	DATA IN[2]	DATA OUT[2]	IMM [2]
3	DATA IN[3]	DATA OUT[3]	OP [0]
4	DATA IN[4]	DATA OUT[4]	OP [1]
5	DATA IN[5]	DATA OUT[5]	OP [2]
6	DATA IN[6]	DATA OUT[6]	—
7	DATA IN[7]	DATA OUT[7]	—

Multiply-Add

by **yNiklas**

0049 HDL Project

`github.com/yNiklas/tt-chip-design`

"*Muliply-Add*"

How it works

Input three 8-bit numbers A, B, C sequentially at consecutive positive clock edges. Upon the fourth positive clock edge, the result out=(A+B)*C is provided.

How to test

Insert 3 numbers, check result.

External hardware

None

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Input (A, B, or C)	(A+B)*C (after 4. clock posedge)	—
1	Input (A, B, or C)	(A+B)*C (after 4. clock posedge)	—
2	Input (A, B, or C)	(A+B)*C (after 4. clock posedge)	—
3	Input (A, B, or C)	(A+B)*C (after 4. clock posedge)	—
4	Input (A, B, or C)	(A+B)*C (after 4. clock posedge)	—
5	Input (A, B, or C)	(A+B)*C (after 4. clock posedge)	—
6	Input (A, B, or C)	(A+B)*C (after 4. clock posedge)	—
7	Input (A, B, or C)	(A+B)*C (after 4. clock posedge)	—

Bernstein-Yang Modular Inverse (secp256k1)

by **Corey Hahn**

0050 | 50 MHz | HDL Project

`github.com/coreyhahn/tto_submission_mar_2026`

"Hardware modular inverse accelerator for secp256k1 using the Bernstein-Yang algorithm"

How it works

This design computes the modular multiplicative inverse over the secp256k1 prime field using the Bernstein-Yang divstep algorithm. Given a 256-bit input `a`, it computes `a^(-1) mod p` where `p` is the secp256k1 prime `0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F`.

Core algorithm

The core iterates a single combinational divstep unit up to 742 times sequentially. Opportunistic double-step and triple-step optimizations fold extra iterations when the intermediate value `g` has trailing zero bits, reducing the typical cycle count to 335 clock cycles. Early termination occurs if `g` reaches zero before all iterations complete. After iteration, a sign-correction step reduces the result into `[0, p)`.

Byte-serial interface

Data is loaded and read via an 8-bit byte-serial interface using a 272-bit shift register (34 bytes). Control signals use rising-edge detection, so holding `wr` or `rd` high for multiple cycles only triggers one shift. The FSM has three states:

- **S_LOAD**: Accepts 34 input bytes (MSB-first), shifting each into the register on a `wr` rising edge.
- **S_BUSY**: Runs the divstep iterations. The `valid` flag is low. A second input can be pipelined during this state (see below).
- **S_READ**: Result is available. The MSB byte appears on `uo_out` immediately; pulse `rd` to shift out the remaining 33 bytes.

Parity error detection

Input and output data are protected by a single even-parity bit. The 34-byte input contains 256 data bits and 1 parity bit (bit 15 of the shift register). On load, the design XORs all data bits with the received parity bit. If the result is non-zero, a `parity_error` flag is set on `uio_out[4]` and embedded in the output. The output also includes a parity bit computed from the result so the reader can verify data integrity.

Pipelined input loading

During S_BUSY, the `ready` signal remains high, allowing a second 34-byte input to be loaded while the first computation runs. When the first result is ready, a single `wr` pulse in S_READ kicks off the second computation without re-loading.

Output modes

The `mode` pins (`uio_in[5:4]`) select what appears in the output shift register:

Mode	Description
`00`	Normal inverse result (256-bit result + parity bit + parity_error flag)
`01`	Performance counters: total cycles, double-step count, triple-step count, iteration count, and a status byte (`{parity_error, trng_ready, 6'b0}`)
`10`	TRNG random bytes (one byte per `rd` pulse, gated by `trng_ready`)
`11`	Reserved (falls through to normal result)

TRNG

A true random number generator provides random bytes via mode `10`. In hardware, three ring oscillators (5, 7, and 9 inverters) are XOR'd for raw entropy, then passed through a Von Neumann debiaser to produce unbiased bits. The ring oscillators are gated by the `ena` signal — when `ena` is low, the oscillators stop and no entropy is generated. The `trng_ready` flag (`uio_out[6]`) indicates when a full byte is available. Pulse `rd` in TRNG mode to consume the byte and begin filling the next.

In simulation, an LFSR replaces the ring oscillators for deterministic testability.

How to test

1. Assert `rst_n` low for at least 2 clock cycles, then release. Ensure `ena` is high.
2. Confirm `ready` (`uio_out[0]`) is high, indicating the S_LOAD state.
3. Write 34 bytes MSB-first: place each byte on `ui_in[7:0]` and pulse `wr` (`uio_in[2]`) high then low. The 34 bytes encode 256 data bits, 1 parity bit, and 15 padding bits.
4. After the 34th byte, computation starts automatically. Wait for `valid` (`uio_out[1]`) to go high (335 clock cycles typical).
5. Read the first result byte directly from `uo_out[7:0]`.
6. Pulse `rd` (`uio_in[3]`) 33 times to shift out the remaining bytes, reading `uo_out` after each pulse.
7. To compute another inverse, start again from step 3. For pipelined operation, load the next input during step 4 (while `ready` is still high).

Performance counter readout

Set `uio_in[5:4] = 01` before the computation completes. The result shift register will contain performance data instead of the inverse result. Read 34 bytes as normal.

TRNG readout

After reaching S_READ, set `uio_in[5:4] = 10`. Wait for `trng_ready` (`uio_out[6]`) to go high, then read the byte from `uo_out[7:0]`. Pulse `rd` to consume and request the next byte.

External hardware

A microcontroller or FPGA is needed to drive the byte-serial interface (provide clocked `wr`/`rd` pulses and supply/read data bytes). The `ena` signal must be held high for the TRNG ring oscillators to operate. No other external hardware is required.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	data_in[0]	data_out[0]	ready (output)
1	data_in[1]	data_out[1]	valid (output)
2	data_in[2]	data_out[2]	wr (input)
3	data_in[3]	data_out[3]	rd (input)
4	data_in[4]	data_out[4]	parity_error (output)
5	data_in[5]	data_out[5]	—
6	data_in[6]	data_out[6]	trng_ready (output)
7	data_in[7]	data_out[7]	—

Switch Puzzle

by **Rimaito's Lab**

0051 Wokwi Project

`github.com/RimaitosLab/TinyTapeoutWorkshop`

`wokwi.com/projects/455293379343017985`

"Control a 7 segment display tile where each switch toggles one segment and its neighbors"

How it works

This is a simple puzzle game where your goal is it to get the seven segment display to display all the numbers from 0-9. Flipping an input always toggles a corresponding segment and all neighboring segments.

How to test

Connect microswitches to i0 to i6 and connect the seven segment display to o0-o6 (the dot of the display isn't used if it has one)

External hardware

- seven segment led display
- seven microswitches

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	TM	TM	—
1	TR	TR	—
2	BR	BR	—
3	BM	BM	—
4	BL	BL	—
5	TL	TL	—
6	MM	MM	—
7	—	—	—

Tudor BCD Test

by **Tuda05**

0097 Wokwi Project

github.com/Tuda05/TinyTapeout

wokwi.com/projects/455291631430488065

"A simple BCD with wowki for TinyTapeout"

How it works

You input the 4 digit binary number using pins 0 - 3 with input pin 0 being LSB and input pin 3 MSB. This way you can show any number between 0 and 9 (max binary is 1010 = 9)

How to test

Enter any number you want and it will be displayed on the 7 segment display

External hardware

One single digit 7 segment display.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input1	—	—
1	input2	—	—
2	input3	—	—
3	input4	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Tiny Tapeout 1

by **Markus S** & **Marcel B**

00**99** **50 MHz** **Wokwi Project**

`github.com/makrs11/TinyTapeout-1`

`wokwi.com/projects/455300137923517441`

"TBD"

How it works

Clock devider as simple counter.

How to test

Give it a clock and see if the outputs are binary numbers counting up.

External hardware

Output LEDs.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	NN data in 0	NN data out 0	NN data in 8
1	NN data in 1	NN data out 1	NN data in 9
2	NN data in 2	NN data out 2	NN data out 8
3	NN data in 3	NN data out 3	NN data out 9
4	NN data in 4	NN data out 4	Config and Control Bit 0
5	NN data in 5	NN data out 5	Config and Control Bit 1
6	NN data in 6	NN data out 6	Config and Control Bit 2
7	NN data in 7	NN data out 7	Config and Control Bit 3

little frequency divider

by **Paul Döller**

0101 | 10 kHz | Wokwi Project

github.com/pauld0503/paul_tiny_tapeout

wokwi.com/projects/455300818767153153

"this is a little freqency divider using D-Flipflops"

How it works

switch input0 to high to activate the frequency divider

How to test

You should see a smaller freqency at the output

External hardware

blablabla

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input0	output0	—
1	—	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Test - clock divider

by **Sam**

0103 1 Hz Wokwi Project

github.com/sam-m7/tinyTapeoutChip1

wokwi.com/projects/455291748212573185

"Quick clock divider test"

How it works

Clock to out7. Clock/4 to out 0.

How to test

In0 is low, wait for 4s. Change In0 to high, press rst for 1 second.

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	Enable	Clock/4	—
1	—	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	Clock	—

Counter

by **Rebecca**

0105 1 Hz Wokwi Project

github.com/Estel64/Beccas_tinytapeout

wokwi.com/projects/455291611094391809

"hopefully counts"

How it works

Hopefully it counts in the future

How to test

No clue as it does not work yet, if you are able to, you are a magician.

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	—	output0 dr flipflop	—
1	—	output1 dr flipflop	—
2	—	output2 dr flipflop	—
3	—	output3 dr flipflop	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

SotaSoC

by **SotaTek**

`0106` `64 MHz` `HDL Project`

`github.com/sotatek-dev/ttihp-SotaSoC`

"RISC-V 32-bit embedded SoC with RV32IC_Zicsr_Zifencei core featuring SPI, I2C, UART, PWM, and GPIO peripherals"

How it works

SotaSoC is a compact RISC-V System-on-Chip (SoC) targeting **Tiny Tapeout** tape-out and also capable of running on **Artix 7 FPGA** with Vivado. Suitable for custom boards, teaching, and as a base for your own SoC. Software support includes **FreeRTOS**, **MicroPython** (in development), and **bare-metal**.

Supported ISA Extensions

- **I** — RV32I: 32-bit RISC-V base integer instruction set with 32 general-purpose registers.
- **C** — RISC-V Compressed instructions.
- **Zicsr** — Control and Status Register extension.
- **Zifencei** — Instruction-fetch fence extension.

Peripherals

- **QSPI Flash** and **QSPI PSRAM** — 128 Mbit Flash for code and data, 64 Mbit PSRAM for runtime memory
- **UART** — programmable baud rate via 10-bit clock divider; default 115200 at 64 MHz
- **48-bit timer** (mtime)
- **13× GPIO** — 1 bidirectional (in/out), 6 input (with interrupt), 6 output
- **PWM** — 16-bit period and duty (in clock cycles), configurable frequency and duty cycle per channel
- **SPI** — master; full mode support (CPOL/CPHA), clock up to 16 MHz, configurable; 4-byte buffer
- **I2C** — master; clock configurable via 8-bit prescaler — 100 kHz, 400 kHz, 1 MHz, and others; START, STOP, repeated START, byte read/write with ACK/NACK

Board Support Package (BSP)

A BSP is available for **FreeRTOS** and **bare-metal** development:

- FreeRTOS BSP: https://github.com/sotatek-dev/SotaSoC-BSP/tree/main/examples-freertos

- Bare-Metal BSP: https://github.com/sotatek-dev/SotaSoC-BSP/tree/main/examples-baremetal

Demo

SotaSoC is capable of driving real-world applications such as a **320×240 ST7789 LCD** display at 10 FPS via SPI at 16 MHz clock.

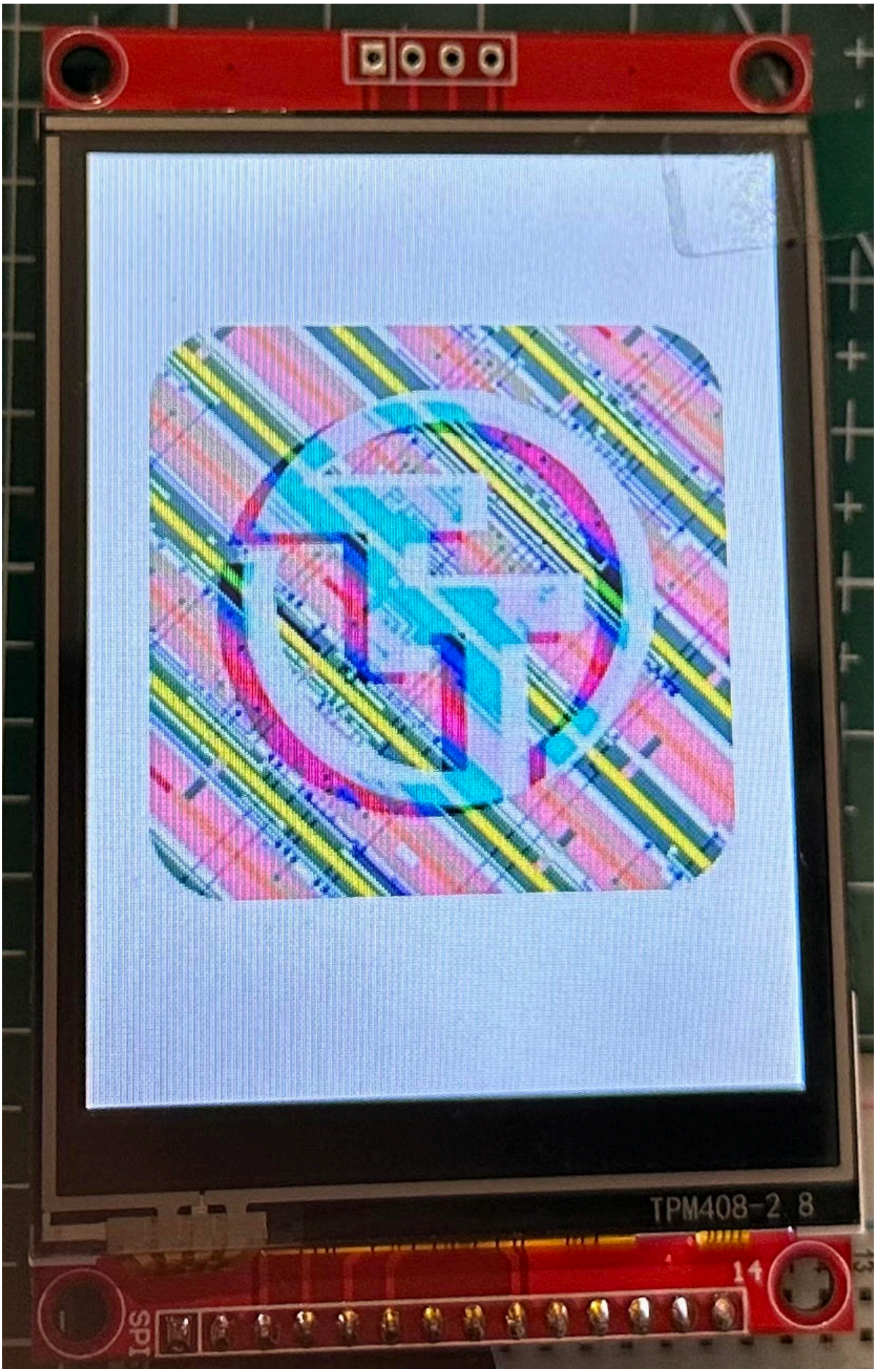


Figure 106.1: LCD demo (320×240 ST7789)

The photo above was taken from a test on an Artix 7 FPGA; the tapeout chip is not yet available.

More examples and demos are available in the SotaSoC-BSP (https://github.com/sotatek-dev/SotaSoC-BSP) repository.

For more detailed technical information, see https://github.com/sotatek-dev/SotaSoC.

How to test

Prerequisites for testing:

- A **QSPI Pmod** is required.
- **System clock** is set to **32 MHz**.
- Pins **ui_in[5]** and **ui_in[6]** are **pulled down**. These two pins configure the read delay for QSPI data. When the system clock is above 32 MHz, try **ui_in[6:5]** in order—**00**, **01**, **10**, **11**—to see which value gives reliable operation. **Note:** This value is sampled only once, immediately after reset.

Blink

This test verifies the basic functionality of the design by blinking an LED.

1. **Write firmware to Flash**

   Download the blink firmware: https://github.com/sotatek-dev/SotaSoC-BSP/blob/main/examples-baremetal/blink-tt/build/blink-tt.bin, then write it to Flash at address **0x0000_0000**.

2. **Connect two LEDs to the board**

   Connect two LEDs (each with a suitable series resistor): one to **uo_out[1]** and one to **uo_out[2]**.

3. **Reset and run**

   Reset the board. The LED connected to **uo_out[2]** will blink.

   If there is an error related to Flash and PSRAM, the LED connected to **uo_out[1]** will light up.

ST7789 LCD test

This test verifies the ability to drive the ST7789 LCD via SPI. Follow the instructions below:

1. **Write firmware to Flash**

   Download the firmware from https://github.com/sotatek-dev/SotaSoC-BSP/blob/main/examples-baremetal/spi-st7789-tt/build/spi-st7789-tt.bin, then write it to Flash at address **0x0000_0000**.

2. **Wiring**

Connect the LCD to the development board as follows:

LCD Pin	Development Board Pin
VCC	VCC
GND	GND
CS	uo_out[3]
SCK	uo_out[4]
SDI (MOSI)	uo_out[5]
DC	uo_out[6]
RST	uo_out[7]
LED	VCC

3. **Expected Result**

   After reset, you will see some content displayed on the LCD as shown in the figure below:

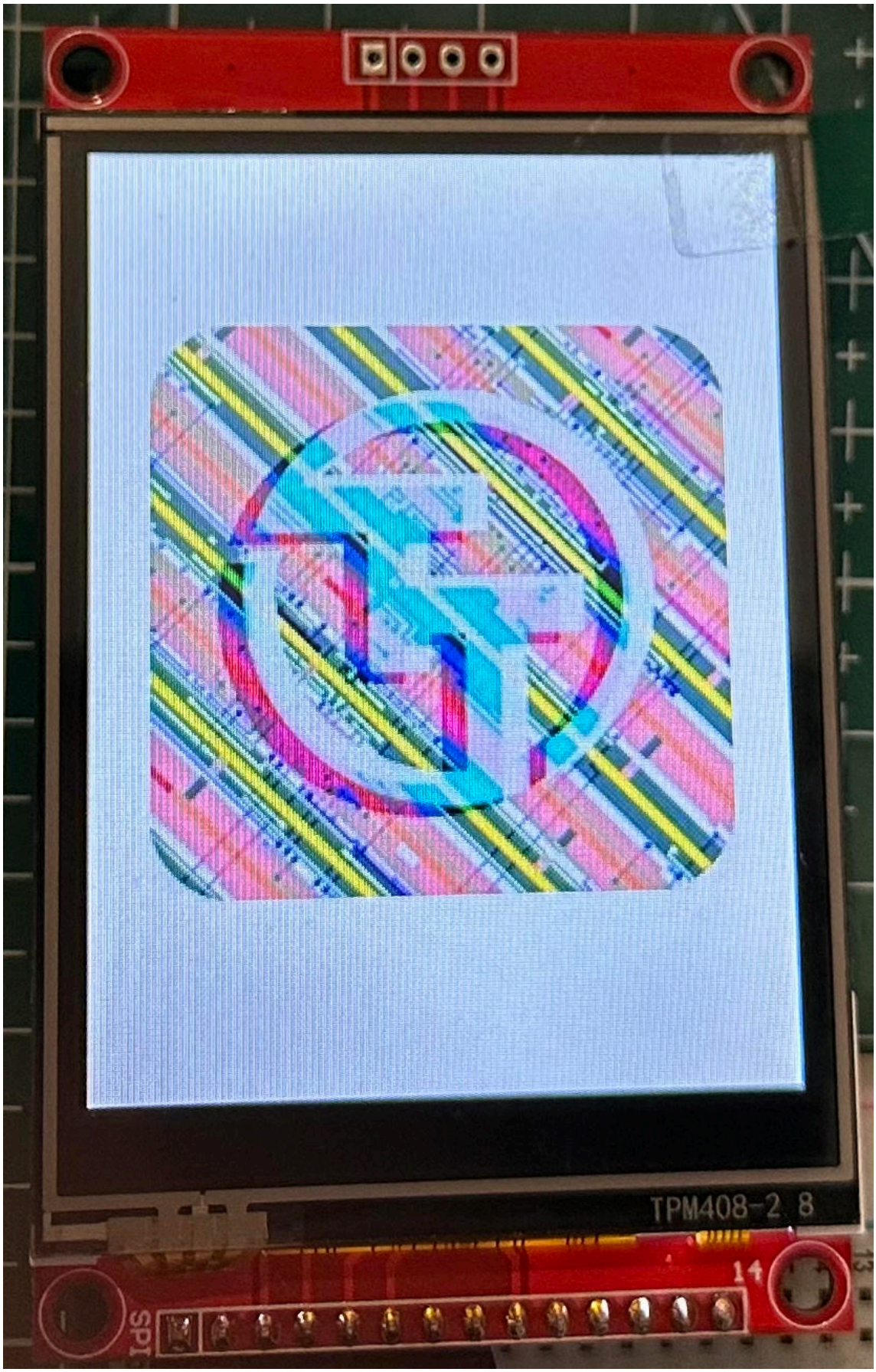


Figure 106.2: LCD demo (320×240 ST7789)

Other examples

The https://github.com/sotatek-dev/SotaSoC-BSP repository provides other sample firmware (e.g. UART, PWM, I2C). You can download any of them and write the binary to flash at address **0x0000_0000** to run different demos or test other peripherals.

Important note: To test **I2C** or **GPIO[0]**, you need to **cut the PSRAM B trace on the QSPI Pmod**, because I2C and GPIO[0] are using pin **uio[7]**.

External hardware

To test **blink**: you need a **QSPI Pmod** and **two LEDs** connected to **uo_out[2]** and **uo_out[1]** as described in How to test above.

To test **ST7789 LCD**: you need a **320×240 ST7789 LCD** (SPI). Connect it to the development board as described in the ST7789 LCD test section above.

To test **other peripherals** (UART, PWM, SPI, I2C, etc.), refer to the specific examples in the https://github.com/sotatek-dev/SotaSoC-BSP repository.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	SPI_MISO	UART0_TX	FLASH_CS_N
1	GPIO_IN[0]	ERROR_FLAG	BUS_IO[0]
2	GPIO_IN[1]	GPIO_OUT[0]/I2C_SCL	BUS_IO[1]
3	GPIO_IN[2]	GPIO_OUT[1]/SPI_CS_N	BUS_SPI_SCLK
4	GPIO_IN[3]	GPIO_OUT[2]/SPI_SCLK	BUS_IO[2]
5	GPIO_IN[4]	GPIO_OUT[3]/SPI_MOSI	BUS_IO[3]
6	GPIO_IN[5]	GPIO_OUT[4]/PWM[0]	RAM_CS_N
7	UART0_RX	GPIO_OUT[5]/PWM[1]	GPIO_IO[0]/I2C_SDA

WIP

by **WIP**

0107 Wokwi Project

github.com/kmosta19/TinyTapeoutProject

wokwi.com/projects/455331155298538497

“WIP”

How it works

This design implements the TEA (Tiny Encryption Algorithm) block cipher in hardware. It encrypts a 64-bit data block using a 128-bit key over 32 rounds, using only 32-bit add (mod 2^32), XOR and shifts.

How to test

Follow the official Tiny Tapeout workshop “activate and test a design” flow. Use the project’s defined I/O protocol (as documented in your repository) to load key/plaintext, run encryption, and read back the ciphertext for comparison.

Known TEA reference vector:

- key = 0x00000000_00000000_00000000_00000000
- plaintext = 0x00000000_00000000
- ciphertext= 0x41EA3A0A_94BAA940

External hardware

None required.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	DIN0	DOUT0	—
1	DIN1	DOUT1	—
2	DIN2	DOUT2	—
3	DIN3	DOUT3	—
4	DIN4	DOUT4	—
5	DIN5	DOUT5	—
6	DIN6	DOUT6	—
7	DIN7	DOUT7	—

VGA Tiny Logo Roto Zoomer

by **Renaldas Zioma**

0108 25.175 MHz HDL Project

github.com/rejunity/ttihp26a-vga-tiny-logo-rotozoomer

"Large 480x480 pixels Tiny Tapeout logo with bling and dithered colors crammed into 1 tile!"

How it works

Compressed VGA Logo

How to test

Connect to VGA monitor and run it!

External hardware

TinyVGA PMOD, VGA monitor

Project Pinout

Digital Pins

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

VGA demo

by **Matt Venn**

0109 25.175 MHz HDL Project

`github.com/mattvenn/ihp26a-vga-test`

"A quick test of the new advanced workshop guide"

How it works

Based on VGA playground, this project generates a stripe pattern.

How to test

Connect a monitor and play with the input pins to change the pattern.

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	—

VGA Squares

by **Matthias Fischer**

0111 25.175 MHz HDL Project

github.com/matth-fischer/TT_VGA

"ETH TT WS VGA Squares"

How it works

Generates VGA squares animation (based on VGA playground rings template)

How to test

Either using the testbench or connect final design. Ui[0] can control the speed and Ui[1] controls the direction of the square animation.

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

Genesis

by **brunny95**

0112 Analog Project

`github.com/brunny95/Genesis`

"8-phase buck converter + ultra low power BG"

How it works

This design is a mixed-signal 8-phase buck converter controller. The digital block (`top_dig`) manages startup sequencing, soft-start, phase rotation, and digital configuration, while the analog blocks generate the control-oriented signals used by the digital logic and implement the regulation loop.

At top level, the system is composed of:

- a bias/reference generation path,
- level shifters between digital low-voltage control and analog control domains,
- an analog control block (`ctrl`),
- the digital sequencing block (`top_dig` / `top_dig2`).

Although `cot` and `hs_req` are inputs of the `top_dig` module, they are not external system inputs at top level. They are generated internally by the analog control path:

- `cot` is produced by the analog control block and indicates when a new switching event is needed,
- `hs_req` is generated by the analog control path and is used by the digital block to terminate soft-start and enter normal operation.

Digital control flow

The digital controller is organized around a 4-state FSM:

- **IDLE**: waiting for controller enable.
- **DEADTIME**: after enable, the controller forces the comparator high for a fixed 1.6 ms startup delay.
- **SOFTSTART**: switching activity is introduced gradually through a DDS-based ramp.
- **RUN**: normal operation.

If soft-start bypass is enabled, the controller skips the DEADTIME and SOFT-START phases and goes directly to RUN.

Soft-start operation

During soft-start, the digital block does not forward switching activity directly from the analog request path. Instead, pulse generation is modulated by an internal DDS ramp:

- a programmable initial increment sets the startup pulse rate,
- the DDS increment increases over time,
- the pulse density progressively ramps up until `hs_req` indicates the end of soft-start.

This provides controlled startup of the converter and avoids abrupt energy delivery.

Phase management

The controller supports from 1 to 8 active phases, configured through `cfg_n_ph`.

A phase counter rotates across the enabled phases and generates a one-hot phase selection for the high-side outputs. This distributes switching events across the active phases in round-robin fashion.

Output generation

The high-side outputs are exposed on `uio_out[7:0]`, with corresponding enables on `uio_oe[7:0]`.

Depending on configuration:

- in forced CCM mode, the controller emits one-hot phase pulses,
- otherwise, enabled phases remain statically asserted according to the selected operating mode.

Unused phases can optionally be driven low through `cfg_unused_force_low`.

Configuration interface

The digital block contains a 38-bit serial configuration shift register. Configuration is loaded through:

- `SR_DATA`,
- `SR_CLK`,
- `SR_COMMIT`.

The register controls:

- soft-start bypass,
- external reference forcing,
- soft-start startup frequency,
- TON selection,
- number of active phases,
- reference trim and reference source selection,

- external capacitor enable,
- high-gm mode,
- forced CCM,
- debug mux routing.

Analog / digital interaction

The analog block `ctrl` implements the regulation path and generates the internal control signals observed by the digital sequencer. In particular:

- the analog comparator and delay path generate `cot`,
- the analog startup/control path generates `hs_req`,
- the digital block returns control signals such as `force_comp_high`, `ss_done`, `en_ctrl2_1p2v`, and `sel_ton_1p2v` back to the analog domain.

This makes the overall architecture a closed mixed-signal control loop rather than a purely digital pulse generator.

Debug outputs

Several internal digital signals can be routed to the dedicated outputs through a debug mux, including:

- `cot`,
- `hs_req`,
- FSM state bits,
- `force_comp_high`,
- `ss_active`,
- `ss_done`,
- `hs_pulse`,
- DDS `tick`,
- control enable signals.

This is useful for bring-up, mixed-signal validation, and lab/debug correlation.

How to test

The design should be tested as a mixed-signal controller, verifying both the digital sequencing and the interaction with the internally generated analog control signals.

Basic startup test

1. Reset the design and keep `en_ctrl_1p2v` low.
2. Release reset and enable the controller.
3. Verify the FSM transitions:
    - `IDLE -> DEADTIME -> SOFTSTART -> RUN`
4. Check that:
    - `force_comp_high` is asserted only during DEADTIME,
    - `ss_active` is asserted only during SOFTSTART,

- `ss_done` is asserted only during RUN.

Soft-start bypass test

1. Enable soft-start bypass through `ss_bypass_ext` or `cfg_ss_bypass`.
2. Enable the controller.
3. Verify that the state machine enters RUN directly.

Analog interface test

1. Verify that `cot` is generated by the analog `ctrl` block and observed correctly by the digital controller.
2. Verify that `hs_req` is generated internally by the analog control path.
3. Check that deasserting `hs_req` causes the transition from SOFTSTART to RUN.

DDS soft-start test

1. Run with soft-start enabled.
2. Observe `tick`, `hs_pulse`, `ss_active`, and `hs_req`.
3. Verify that:
   - DDS is initialized at SOFTSTART entry,
   - the pulse activity starts from the programmed initial rate,
   - the pulse density increases over time,
   - soft-start terminates only when `hs_req` ends soft-start.

Phase rotation test

1. Program `cfg_n_ph` from 0 to 7.
2. Verify that the number of enabled phases is respectively 1 to 8.
3. In RUN mode, verify that switching requests are distributed round-robin across active phases only.

CCM mode test

1. Test with `cfg_fccm = 0` and `cfg_fccm = 1`.
2. Verify that `FORCE_CCM` is asserted only in RUN when enabled by configuration.
3. Check output behavior in both normal and forced-CCM operation.

Configuration register test

1. Shift known values into the serial configuration register.
2. Commit the configuration.
3. Verify correct update of:
   - TON selection,
   - number of phases,
   - reference settings,
   - external capacitor enable,
   - high-gm mode,
   - force-CCM mode,

- debug mux selections.

Debug test

1. Route internal signals to the debug outputs.
2. Verify visibility of relevant signals such as `cot`, `hs_req`, `ss_active`, `ss_done`, `tick`, and `hs_pulse`.

Disable / reset test

1. Disable the controller from each FSM state.
2. Verify return to IDLE and reset of internal counters and startup state.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	EN_BIAS_1P2V	FORCE_CCM	HS_OUT[0]
1	EN_CTRL_1P2V	DISB_N	HS_OUT[1]
2	VC_PULL_DOWN	SS_ACTIVE	HS_OUT[2]
3	SS_BYPASS_EXT	SR_OUT	HS_OUT[3]
4	THWN	DBG_4	HS_OUT[4]
5	SR_COMMIT	DBG_5	HS_OUT[5]
6	SR_DATA	DBG_6	HS_OUT[6]
7	SR_CLK	DBG_7	HS_OUT[7]

Analog Pins

ua#	analog#	Description
0	0	VDDA
1	1	EXT_REF
2	2	FB_VOUT
3	3	FB_SW
4	4	VC

VGA Maze Runner

by **Philippe Sauter**

`0113` `25.175 MHz` `HDL Project`

`github.com/phsauter/vga-playground-maze`

"Generate a Maze and let players compete against solvers"

How it works

The default design generates a perfect maze on-chip using a procedural binary-tree maze generator. The current top-level configuration uses a `10x10` maze. The maze is shown live while it is being built, so walls can visibly change during the reveal. After generation finishes, the player controls the green dot while a single wall-following solver can race as well.

Controls

- D-pad: move the player
- A: start or stop the solver
- B: reset player and solver positions
- SELECT: generate a new maze
- START: speed up generation while the maze is still being built

How to test

Connect VGA and the PMOD gamepad, then reset the design and play the maze game. Press SELECT to generate a fresh maze and START during generation to accelerate the live build animation.

External hardware

A PMOD gamepad and VGA are required.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	R1	—
1	—	G1	—
2	—	B1	—
3	—	VSync	—
4	pmod_latch	R0	—
5	pmod_clk	G0	—

#	Input	Output	Bidirectional
6	pmod_data	B0	—
7	—	HSync	—

Operational Amplifier Test Circuits

by **Pascal Gesell**

0114 Analog Project

github.com/gfcwfzkm/ttihp_opamp_bfh_gesep1_mht1_msm9

"The test circuit contains an operational amplifier for external feedback, a voltage follower, and an R2R DAC."

The goal of this analogue circuit project was to design a low-power operational amplifier (op-amp) using only open-source EDA tools targeting the IHP SG13G2 OpenPDK and to measure its performance using three test circuits: an inverting opamp circuit with to be connected external feedback, a voltage follower circuit, and an R2R Digital-to-Analogue Converter (DAC).

How it works

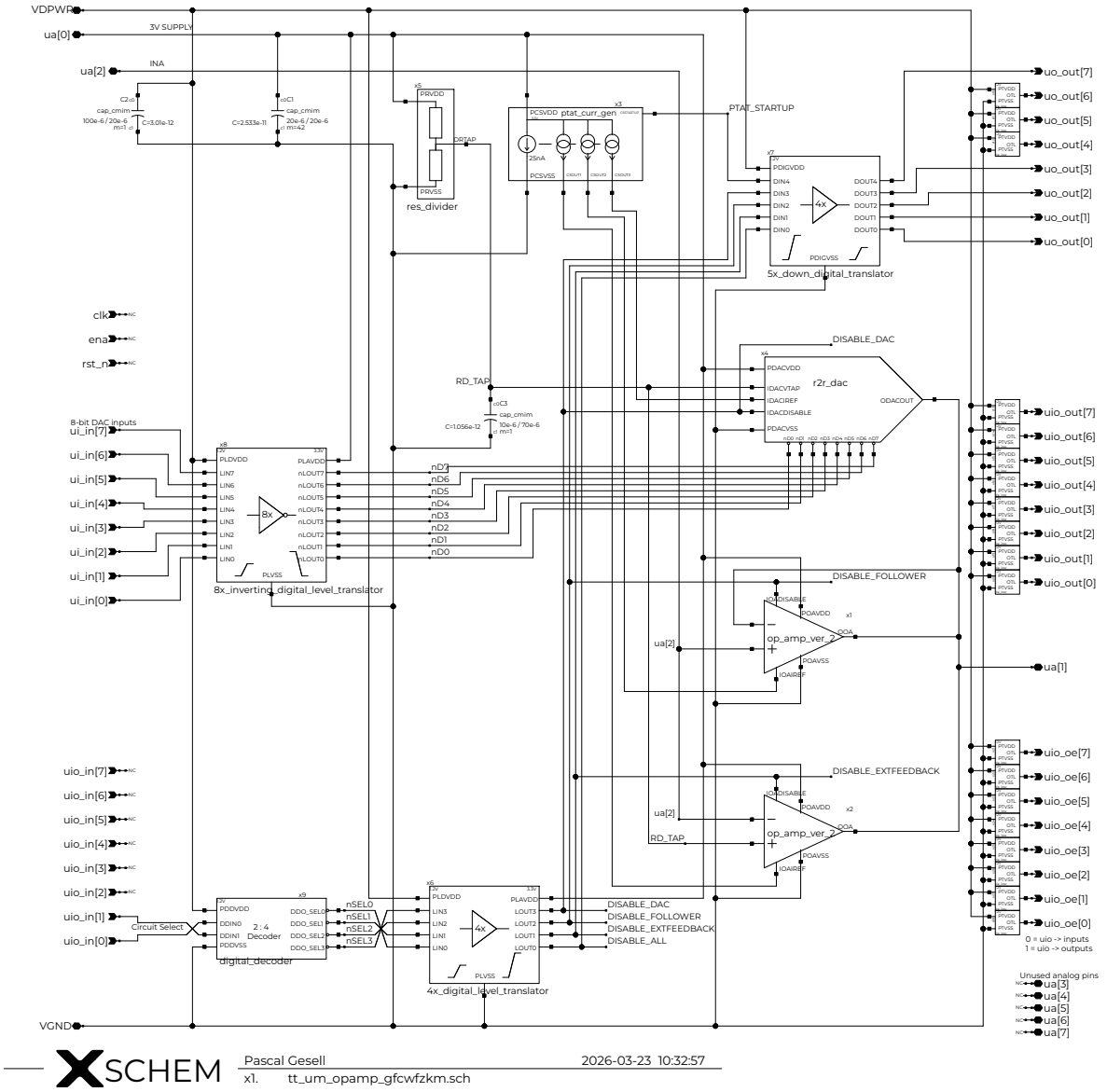


Figure 114.1: Schematic of the Tiny Tapeout tile

The above schematic shows the Tiny Tapeout analogue tile, containing the three operational amplifier (op-amp) circuits and the auxiliary circuits. The three operational amplifier circuits are configured as a simple inverting amplifier with external feedback, a voltage follower and an R2R DAC. The auxiliary circuits include a Proportional to Absolute Temperature (PTAT) current source for biasing the analogue circuits, a resistor divider, digital level shifters and a 2-to-4 decoder.

Next to the standard digital I/Os, provided by the Tiny Tapeout template, the tile includes three analogue pins `ua[0]` to `ua[2]`. The pin `ua[0]` is used as a 3.3 V power supply input for the analogue circuit, due to the lack of an 3.3 V supply rail on this shuttle (TTIHP26a). The pin `ua[1]` is used as the output of the three op-amp circuits, which are connected together. The pin `ua[2]` is

used as an input for the inverting amplifier and the voltage follower circuit. It is not used for the R2R DAC circuit. Due to the limited number of analogue pins, an internal resistor divider is used to create a reference voltage of 1.65 V, which is used as the 'virtual ground' for the inverting amplifier and the R2R DAC.

Circuit selection

All three op-amp circuits share the same op-amp design and their outputs are connected to the same output pin `ua[1]`. Using the digital input pins `uio_in[1]` and `uio_in[0]`, the user can select which of the three operational amplifier circuits to test. The decoded, active-low output of the 2-to-4 decoder is available on the output pins `uo_out[0]` to `uo_out[3]` and can be used to view which circuits are disabled. The following table shows the decoding of the input signals to select the circuit to test:

`uio_in[1]`	`uio_in[0]`	Description
0	0	All op-amps disabled, output `ua[1]` high-impedance
0	1	External feedback op-amp active
1	0	Voltage follower active
1	1	R2R DAC active

To safe space and due to some LVS problems with the standard cell library, the 2-to-4 decoder is implemented directly using 1.2 V transistors.

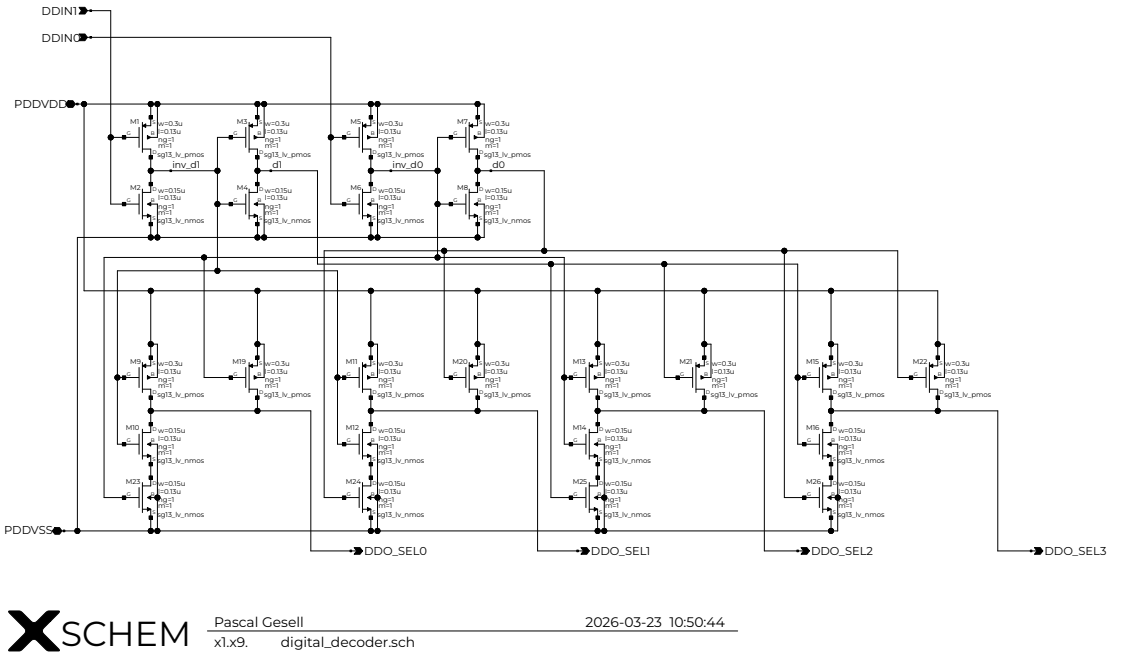


Figure 114.2: 2-to-4 decoder

The decoded 1.2 V signals are then level-shifted to 3.3 V using a simple, non-inverting level shifter design.

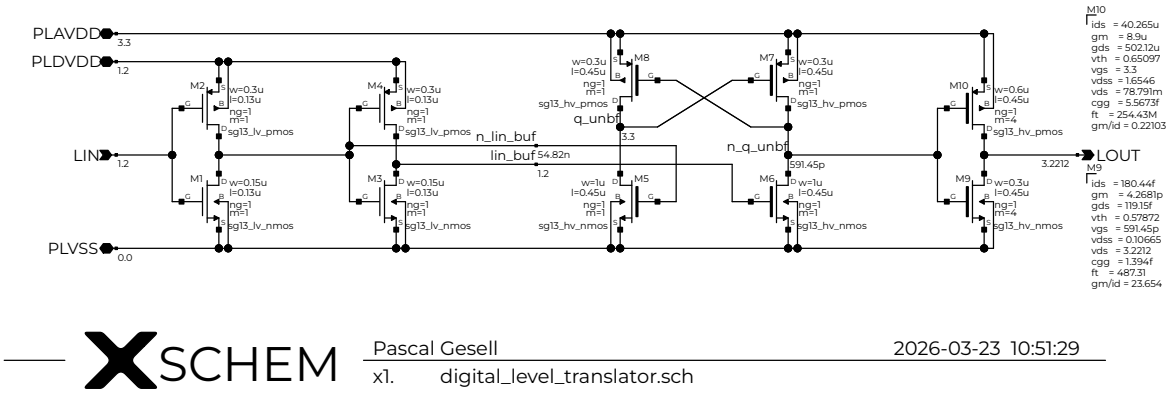


Figure 114.3: Non-inverting level shifter

PTAT current source

The PTAT current source is used to provide a stable current source, containing three current mirror outputs with a target current of approx. 25 nA. The PTAT current source includes a start-up circuit to ensure that the current source starts up correctly and does not get stuck in a zero-current state.

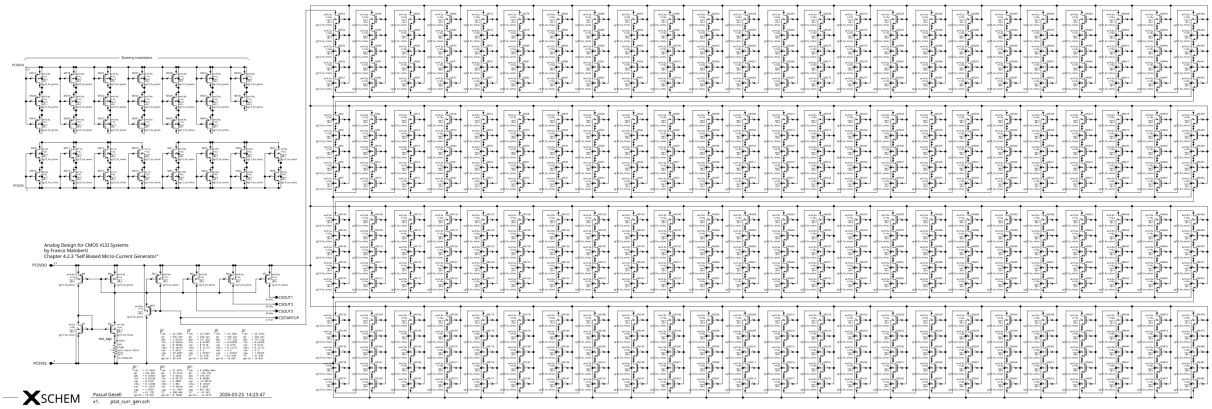


Figure 114.4: PTAT current source

While the core circuit of the PTAT current source is quite compact, the start-up circuit takes up a lot of space due to the voltage drop required to operate the start-up circuit and due to the small current targeted by the PTAT current source. Thus, the start-up circuit alone requires 480 transistors configured at the minimal width and maximal length allowed by the PDK (w=0.3µm, l=10µm) to achieve the required voltage drop and to minimize the current consumption of the start-up circuit.

The circuit not only contains the three mirrored current outputs, but also a start-up signal output, which is used to indicate a successful start-up of the current source (active-high). The start-up signal is available on the output

pin `uo_out[7]` and can be used to check if the current source has started up correctly.

Op-amp design

The design of the operational amplifier is based on a simple two-stage, low-power design. The first stage is a differential pair with a current mirror load, while the second stage is a rail-to-rail output stage. The circuit is designed to operate at a supply voltage of 3.3 V with a bias current of 25 nA, which is provided by the PTAT current source. In addition to the typical op-amp pins (current-bias input, non-inverting input, inverting input and output), the op-amp also includes a disable pin (active-high), which is used to disable the internal op-amp circuit and to put the output in a high-impedance state.

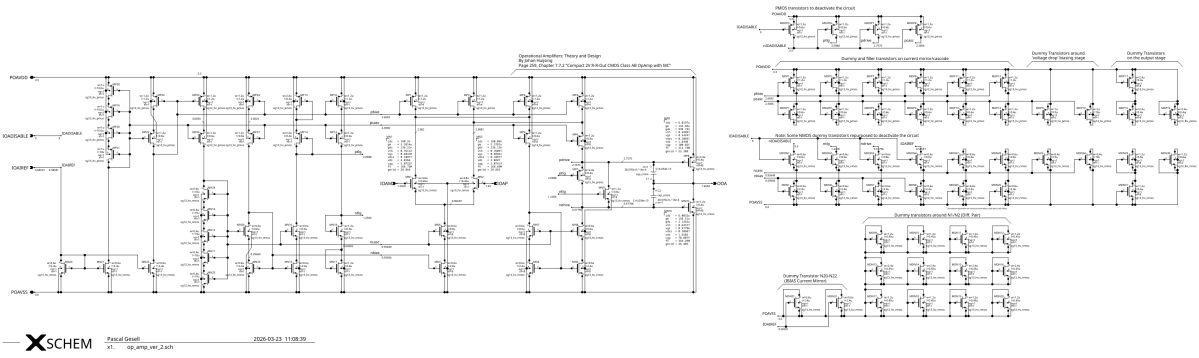


Figure 114.5: Operational amplifier design

The process corner simulations of the designed op-amp in the voltage follower configuration show unity gain up to 339 kHz in the worst case with a test load of 10 MΩ and 50 pF. Note that these preliminary simulation results were done with the schematic excluding any extracted parasitics.

R2R DAC design

The 8-bit R2R DAC is designed using a simple resistor ladder structure, with the 8-bit input directly connected to the resistor ladder. The R2R DAC is designed to operate at a supply voltage of 3.3 V and to provide an output voltage range close to the supply range. An inverting operational amplifier circuit is used as a buffer stage. Thus, the input bits of the R2R DAC circuit itself are active-low (while the binary value is set via the active-high input ui_in[7:0]). The amplification of the op-amp is set to 0.933 to ensure that the output voltage is within the optimal range of the op-amps output stage. To correctly disable the circuit when the R2R DAC is not selected, an additional transmission gate disconnects the feedback path of the output, as the R2R resistor ladder would otherwise still load the analogue output pin and affect the performance of the other op-amp test circuits when they are selected.

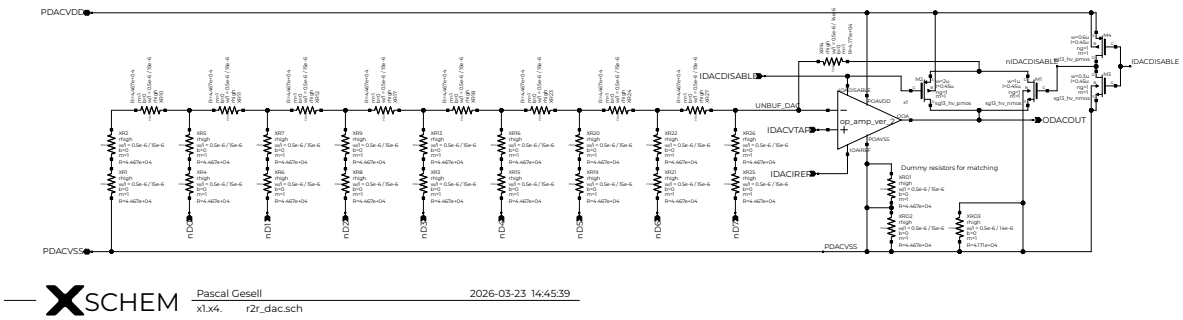


Figure 114.6: R2R DAC design

Since the R2R DAC does not contain any additional buffers/drivers at the input stage and requires 3.3 V logic levels, the parallel input `ui_in[7:0]` of the Tiny Tapeout tile are inverted and shifted up by the logic level shifter, before being fed into the R2R DAC circuit. The output buffer stage of the logic level shifter is designed to provide a strong drive strength to ensure the correct operation of the R2R DAC circuit.

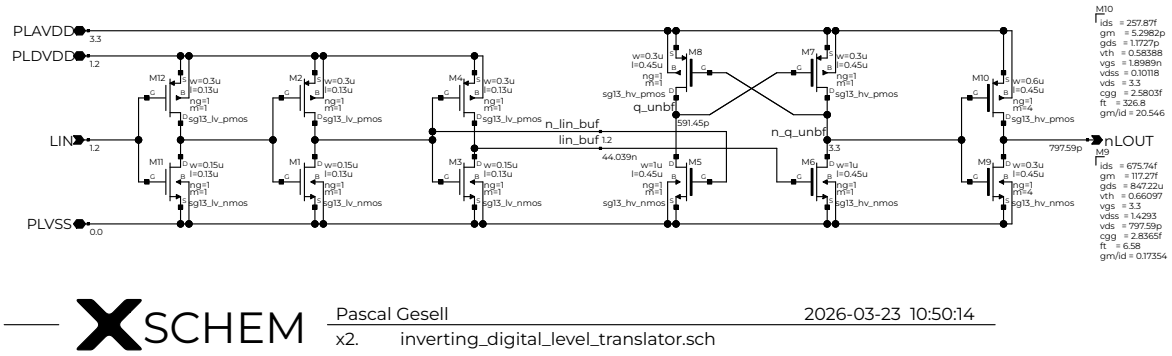


Figure 114.7: Inverting logic level shifter for R2R DAC inputs

Note: Some images are shown as PNGs instead of SVGs due to the large size of some SVG files and the 1 MB size limit of the Tiny Tapeout project datasheet. The original SVG files can be found in the xschem/svg directory.

Simulation results

All designed circuits come with a ngspice/xschem test bench in the xschem directory using the IIC-OSIC-TOOLS docker image:

- **tb_op_amp.sch** - Operational amplifier test bench
- **tb_r2r_dac.sch** - R2R DAC test bench
- **tb_ptat_curr_gen.sch** - PTAT current source test bench
- **tb_ptat_opamp_startup.sch** - PTAT op-amp startup test bench
- **tb_digital_decoder.sch** - 2-to-4 decoder test bench
- **tb_digital_level_translator.sch** - Level shifter test bench
- **tb_res_divider.sch** - Resistor divider test bench

- **tb_tie_low.sch** - Tie low test bench
- **tb_tt_um_opamp_gfcwfzkm.sch** - Full Tiny Tapeout op-amp tile test bench

How to test

See the info.yaml file for the pinout and connect the required test equipment to the right pins. For the input and bidirectional input port of the TinyTapeout Demo Board, you can connect a basic DIP switch Pmod (e.g., the 1BitSquared PMOD DIP Switch) to set the DAC input bits and to select the circuit to test. For the digital output pins, connect a simple LED Pmod (e.g., the Digilent Pmod 8LD) to visualize the currently disabled circuits and the start-up status of the PTAT current source.

For the analogue output pins, you must provide a 3.3 V power supply to the internal analogue circuit using the pin `ua[0]`. The output of the op-amp circuits can be measured on the pin `ua[1]` using an oscilloscope or a multimeter. To test the non-inverting amplifier and the voltage follower circuit, you can provide an input signal to the pin `ua[2]` using a function generator. Note that the analogue input pin is not in use if the R2R DAC circuit is selected.

External hardware

To test and use this project, you will need the following hardware:

- 2 × 1BitSquared PMOD DIP Switch : A 8-bit DIP switch Pmod
- 1 × Digilent Pmod 8LD : A 8 LED Pmod
- A 3.3 V power supply to power the analogue test circuits
- A function generator to provide input signals for the inverting amplifier and the voltage follower circuit
- An oscilloscope or a multimeter to measure the output of the op-amp circuits

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	DAC_0	CIRC_ACTIVE	CIRC_SEL_0
1	DAC_1	CIRC_EFB_DISABLED	CIRC_SEL_1
2	DAC_2	CIRC_FOL_DISABLED	—
3	DAC_3	CIRC_DAC_DISABLED	—
4	DAC_4	—	—
5	DAC_5	—	—
6	DAC_6	—	—

#	Input	Output	Bidirectional
7	DAC_7	PTAT_STARTUP	—

Analog Pins

ua#	analog#	Description
0	5	DD3
1	6	OUTA
2	7	INA

Silly demo

by **James Buchanan**

0115 10 MHz HDL Project

`github.com/BoredSemiRetiredEngineer/ttihp_submission`

"Zero to ASIC demo project"

How it works

silly2 creates divide by 2 clk signals, silly1 allows user to pick which signals get outputed.

How to test

Apply various signals onto the inputs and see what comes out the outputs. Outputs should be the appropriately divide by 2 clock.

External hardware

Use FPGA to drive and read signal outputs.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	ui_in[0]	uo_out[0]	—
1	ui_in[1]	uo_out[1]	—
2	ui_in[2]	uo_out[2]	—
3	ui_in[3]	uo_out[3]	—
4	ui_in[4]	uo_out[4]	—
5	ui_in[5]	uo_out[5]	—
6	ui_in[6]	uo_out[6]	—
7	ui_in[7]	uo_out[7]	—

ttihp-HDSISO8

by **Yann Guidon**

0160 | 50 MHz | HDL Project

`github.com/ygdes/ttihp-HDSISO8`

"High density Shift register - DLHQ"

What it is

This tile delays a bit's value by 502 cycles at speeds above 100MHz (according to the synthesiser, to be tested). It is a baseline for storage packing density, as well as a test architecture for asynchronous shift registers, not made out of large DFF cells. This version packs 672 standard latches and a controller, filling 87% of the tile's surface.

How it works

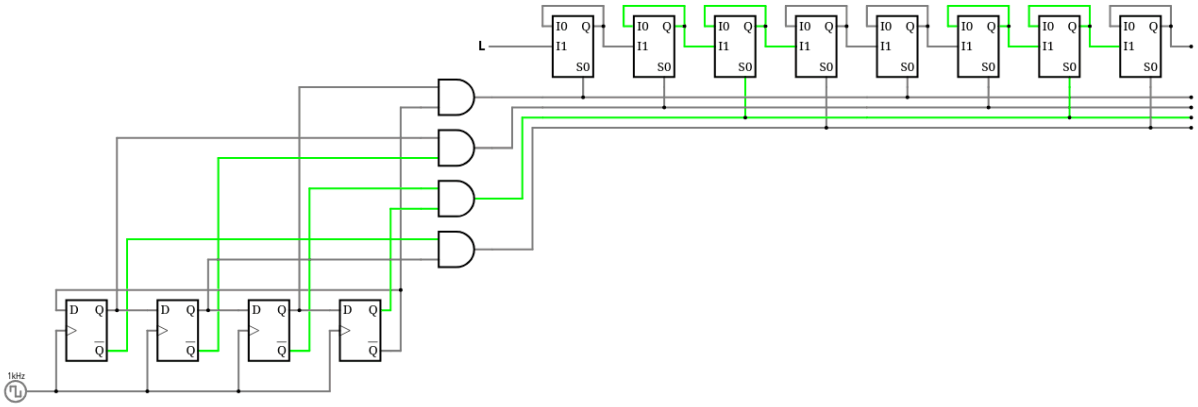
As the name implies, it's a high density shift register for deep digital delays. According to the PDK for CMOS IHP at https://github.com/IHP-GmbH/IHP-Open-PDK/blob/main/ihp-sg13g2/libs.ref/sg13g2_stdcell/doc/sg13g2_stdcell_typ_1p20V_25C.pdf

- Area of sg13g2_dfrbpq_1 : 48.98880
- Area of sg13g2_dlhq_1 : 30.84480
- Area of sg13g2_mux2_1 : 18.14400
- Area of sg13g2_a21oi_1 : (×2) = 18.14400 (same as sg13g2_o21ai_1)

MUX2 is almost 3× smaller than the DFF gate and could be used as a latch by feeding its output back to an input (just like with the old antifuse Actel FPGAs such as A1xxx). This trick is rejected by the tools but in the same area, I could also implement a SR latch with enable, using combined and compact OR/AND gates. This is done in a different tile (see https://github.com/ygdes/ttihp-HDSISO8RS) but first I need a reliable reference point.

This project "tt_um_ygdes_hdsiso8_dlhq" uses the conventional transparent latch DLHQ, whose size is in-between. The shift register uses 4 latches to store 3 bits at a given time and 4 non-overlapping "clock" pulses perform the shifting. Slowly. Just like below, but with 8 parallel chains.

"Tranches" are provided with 16, 64 or 256 latches and must be used in "odd-even" pairs so you get 24, 96 or 384 cycles of delay. You can chain them as long as surface allows (to a degree). The controller adds another 20 cycles. The 16x tranches need 4 extra inverters if used alone.



The apparent complexity comes from the 8-phase clock, which is brought to the “asynchronous” domain. Each of the 8 lanes is 8× slower (which relaxes timing constraints) but the overall throughput is preserved by an intricate demultiplexer and multiplexer. So it “should” work at “full speed”, we’ll see.

Compared to a shift register with normal DFF cells, it could store up to twice the same amount of bits per unit of surface, without the need of full-custom cells, as the controller’s (sequencer, mux and demux) size becomes insignificant when the chain gets longer. Depths of several kilobits are possible without too much hassles (if the synth agrees), without a mad clock network, reducing simultaneous switching noise... Not only are the pulses slower, their traces are also shorter: each pulse affects only 1/8th of the cells at any time.

Ideally, the 8 chains should be manually placed (or with a script), not thrown at random. For implementation, I use a “tuned” Verilog workflow and instantiate cells directly from https://github.com/IHP-GmbH/IHP-Open-PDK/blob/main/ihp-sg13g2/libs.ref/sg13g2_stdcell/verilog/sg13g2_stdcell.v . For simulation, parts of this file are copy-pasted to gate-specific files to remove some warnings (find them in /test, thank you Jeremy!).

You will get a “Synthesis warnings : Warning: There are XXX unclocked register/latch pins.” This is normal.

How to test

Good to know:

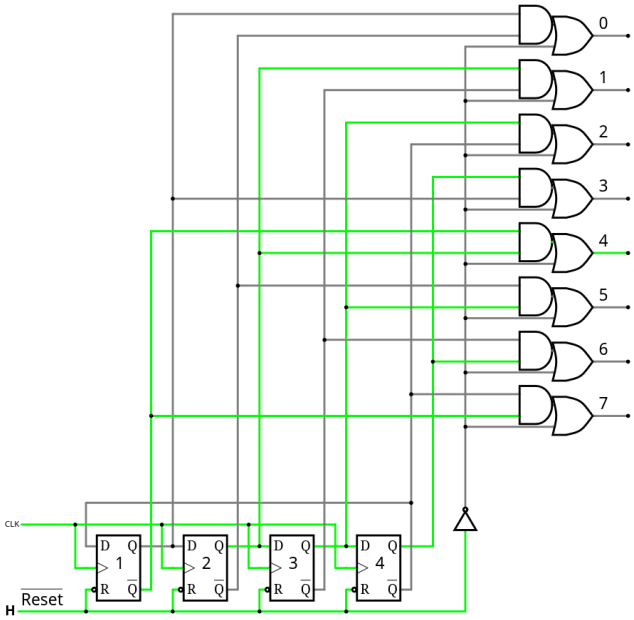
- Clock and Reset can be asserted by external pins and internal signals.
- The pin CLK_OUT copies the currently selected clock (negated), for external triggering and troubleshooting. If it oscillates, you’re good.
- External reset pin EXT_RST (asserted at 0 like the internal one) overrides the internal reset, don’t let it float. A weak pull-up to 1 is advised.
- External clock (pin EXT_CLK) can be selected when pin CLK_SEL=1 (don’t let them float).
- Always assert EXT_RST (to 0) while changing the state of CLK_SEL.

Startup sequence:

- EXT_RST asserted (0)
- Choose CLK_SEL's value
- Run that clock
- Release EXT_RST (to 1, and RESET is internally clock-resynchronised so give it a couple of cycles to come into effect)
- Input a '1' or a '0' on D_IN, and observe the value appearing on D_OUT after 502 clock cycles.

Extra insight and observability:

- When SHOW_LFSR=0, the IO port shows the 8 internal staggered pulses, turning from 0 to 1 and back to 0 in a linear sequence. It's just like a 4017 but 8 bits, since it's a Johnson counter too.
- 4 output pins provide the internal state of that 4-bit Johnson counter, or ring counter, thus you should observe a pretty pattern where only one pin changes at each clock cycle.
- You can measure the routing latency of the pins/pads/internal wires because CLK_OUT is inverted so just tie it to EXT_CLK with pin CLK_SEL=1. Probe with an oscillocsope and voilà, you have a free-running oscillator and you can directly measure the low and high times, each corresponding to one trip on the in or out wire.



Note in the diagram above that RESET forces all the outputs to 1, thus flushing the whole delay line in less than a microsecond.

Bonus: LFSR

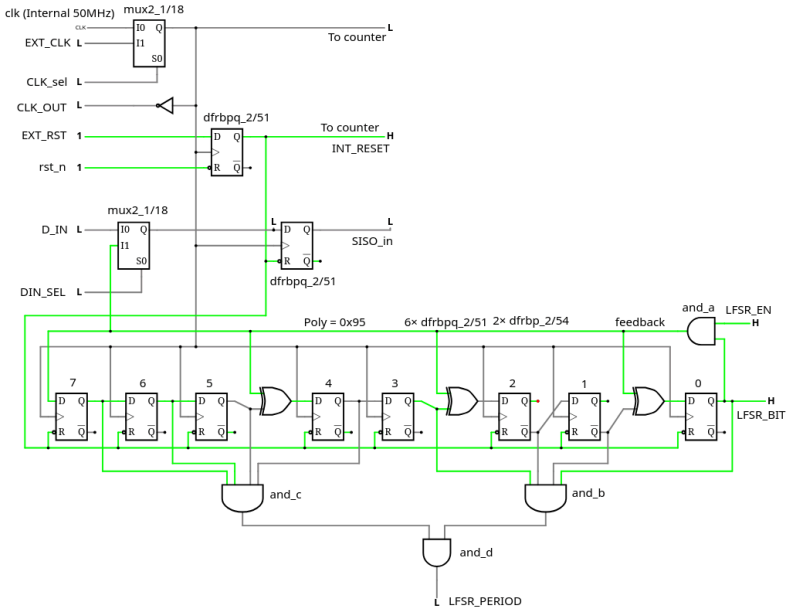
An 8-bit LFSR is integrated to ease testing. Thus an oscilloscope and a variable frequency oscillator are enough to characterise the achieveable speed. To use it,

- Assert the external reset EXT_RST (0)
- Select the desired clock (CLK_SEL)
- Unlock the internal LFSR by asserting pin LFRS_EN to 1
- Assert pin DIN_SEL (1) to internally route the LFSR bitstream to the SISO input
- Run the clock (internal or external, depending on CLK_SEL)
- Release EXT_RST (1) (now it should be started)
- Connect an oscilloscope to probe the signals D_OUT and LFSR_BIT while triggering on LFSR_PERIOD (which pulses every 255 clock cycles)
- See if both traces match (add 8 cycles of delay on D_OUT).
- Send me pictures of your scope traces!

Note 1: 8 bits gives a period of 255, almost half of the SISO's depth of 502, a small shift is expected (the SISO output precedes the LFSR by 8 cycles) and the SISO should store twice the whole LFSR period, but the output should align anyway.

Note 2: The LFSR_PERIOD pulse should appear 193 clock cycles after the release of the RESET pin.

Note 3: The RESET signal clears the contents of the SISO. Give it a few cycles for the 0 to propagate through all the latches while it flushes after releasing the RESET.



External hardware

A basic custom test board will be put together, to hook the variable frequency generator and the oscilloscope probes.

Optionally, if you only want to make a “light chaser”, hook 8 LED to the IO port, select the external clock and add a 555. Or you can have a more funky pattern by displaying the LFSR’s state by setting SHOW_LFSR to 1.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	CLK_SEL	D_OUT	PULSE #0
1	EXT_CLK	CLK_OUT	PULSE #1
2	EXT_RST	Johnson #0	PULSE #2
3	D_IN	Johnson #1	PULSE #3
4	—	Johnson #2	PULSE #4
5	SHOW_LFSR	Johnson #3	PULSE #5
6	LFSR_EN	LFSR_PERIOD	PULSE #6
7	DIN_SEL	LFSR_BIT	PULSE #7

Lab and Lectures SoC

by **Aloke Kumar Das**

0162 50 Hz HDL Project

github.com/alokerdas/tt_um_ihp26a_LnL_SoC

"A tiny SoC comprising of a cpu, memory, pwm, timer and SPI protocol"

How it works

This project implements a tiny system on chip. It has a 16 bit microprocessor, a boot rom, a PWM, a timer and a spi protocol.

The boot rom has 32 words. After reset it runs a program to get input from outside and display to outside. The program has all the instructions that this processor supports. This tapeout is done to test the microprocessor on silicon. The SPI, PWM and timers are memory mapped. The processor writes the data to SPI, PWM and timers so that those IPs can be tested also.

The SPI protocol can be used for serial communication. The data can be loaded to and from cpu. This IP is mapped at 0020. If the cpu attempts to write to the address 0020 the data will be transmitted through the SPI protocol. It can accept data from outside of the SoC as specified in the spi protocol. The signals load and unload can be used to enable this IP.

The PWM resolution is 8. The duty cycle can be varied from 12.5 to 87.5 percent. It is memory mapped at the address 0040. It has a 3-bit register which can be written by the processor to set the duty cycle value. The timer is 8-bit without any pre-scalar. The timer is auto reload and can not be stopped. The output signals can be chosen from devide by 2/4/..../128. It is memory mapped at 0080. It has a 3-bit register which can be written by the processor to set the devisor value.

The microprocessor is a basic one. The data bus is 16-bits, address bus is 12-bits. Adderss and data busses are connected to internal boot rom, RAM and SPI. They cannot access outside memory. There is a parallel input port of 8-bits which is also input of the SoC. Similarly, there is a parallel output port of 8-bits that is also output of SoC. The Instructions that are supported are as follows: LDA - Load the content of a memory location to accumulatorAC ADD - Add the content of a memory location to AC AND - And the content of a memory location with AC STA - Store the content of AC to a memory location BUN - Branch unconditionally BSA - Branch to a memory location storing the return address ISZ - Increment the content of a memory location and check if zero, skip the next instructio Indirect addressing mode of all the above instruction are also supported.

CLA - Clear the content of the AC CLE - Clear the overflow flag E CMA - Complement the content of the AC CME - Complement the overflow flag E CIR - Shift right the content of AC and E, circular CIL - Shift left the content of AC and E, circular INC - Increment the content of AC SPA - Skip next instruction if the content of AC is positive SNA - Skip next instruction if the content of AC is negative SZA - Skip next instruction if the content of AC is zero SZE - Skip next instruction if E is zero INP - Accept 8-bit input from input port if inp flag is high OUT - Send 8-bit output to output port and set the outp flag SKI - Skip next instruction if input flag is high SKO - Skip next instruction if output flag is high HLT - Halt the cpu

How to test

After power on the cpu starts running automatically. No extra effort is required. The boot rom has a program inbuit. It check for input. If input flag is high the 8-bit value is written to accomulator from ui_in pins. Immidiately the same value is output to uo_out pins so that it can be displayed on 7-segment. After that all the other instrustions are executed. Those tests the direct as well as indirect addressing modes. The program write addresses 0020, 0040 and 0080. This is the space for SPI, timer and PWM. The data comes out serially of uio_out[5] pin (mosi of spi), uio_out[4] and uio_out[3].

External hardware

Keypad, 7-segment or LCD or LED. Some kind of storage or data source. To be dicided later.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	keyboard 0	display 0	cpu keyboard in flag
1	keyboard 1	display 1	miso of spi
2	keyboard 2	display 2	ssn in of spi
3	keyboard 3	display 3	clock of spi (future use)
4	keyboard 4	display 4	ssn out of spi
5	keyboard 5	display 5	mosi of spi
6	keyboard 6	display 6	sclk of spi
7	keyboard 7	display 7	cpu display flag

m6502 Microcontroller

by **Chris Moos**

0163 50 MHz HDL Project

github.com/chrismoos/tt-led-controller

"*Complete 6502 CPU with bus multiplexer, GPIO, Timer, and UART*"

A complete MOS Technology 6502-compatible CPU with integrated peripherals, designed specifically for TinyTapeout. Features a bus multiplexer architecture to efficiently expose the full 64KB address space and peripheral functions through the limited 24-pin interface.

Features

- **Complete 6502 CPU**: Cycle-accurate implementation with all documented opcodes
- **Bus Multiplexer**: 4-phase multiplexing reduces pin requirements from 24 pins to 8 data pins + control
- **External Memory Support**: Full 64KB address space via multiplexed bus (RP2040 controller recommended)
- **Integrated Peripherals**:
  - **GPIO**: 6 pins (2 input-only, 4 output-only on TinyTapeout)
  - **UART**: 8N1 serial with 4-byte TX/RX FIFOs, configurable baud rate
  - **Timer**: 16-bit timer with prescaler, auto-reload, and interrupt support
  - **Clock Control**: Runtime CPU clock division for power management
- **Pin Multiplexing**: UART can be routed to any GPIO pin via mode registers
- **TinyTapeout Optimized**: Fits in 2×2 tile allocation (2,900 lines RTL)

How it Works

System Architecture

The 6502 MCU consists of three main components:

1. **6502 CPU Core**: Executes instructions and drives the internal bus
2. **Bus Multiplexer**: Time-multiplexes 16-bit address + 8-bit data into 8 shared pins
3. **Memory-Mapped Peripherals**: Decode addresses 0xA000-0xA047 for internal devices

All memory accesses outside the peripheral range (0xA000-0xA047) are routed to the external bus via the multiplexer, allowing up to 64KB of external RAM/ROM.

Bus Multiplexing

The multiplexer reduces pin count by sequencing through 4 phases within each CPU cycle:

MUX_SEL	Phase	Direction	Data
01	ADDR_HI	MCU → Ext	Address[15:8]
00	ADDR_LO	MCU → Ext	Address[7:0]
10	DATA_IN	Ext → MCU	Read data
11	DATA_OUT	MCU → Ext	Write data

The external controller (RP2040 on TinyTapeout demo board) monitors PHI2 and sequences MUX_SEL accordingly. **There is no performance penalty** - all phases complete within one CPU cycle.

Memory Map

Address Range	Size	Description
0x0000-0x9FFF	40KB	External memory
0xA000-0xA00B	12B	GPIO registers
0xA010-0xA017	8B	Reserved
0xA020-0xA027	8B	Timer
0xA030-0xA033	4B	Clock control
0xA040-0xA047	8B	UART
0xA048-0xFFFF	22KB	External memory

Typical configuration: RAM at 0x0000-0x7FFF, ROM at 0x8000-0xFFFF with reset vector at 0xFFFC.

Pin Configuration

Input Pins (ui_in[7:0])

Pin	Function	Description
0	MUX_SEL[0]	Bus phase select bit 0
1	MUX_SEL[1]	Bus phase select bit 1
2	RDY	CPU ready signal (active high)
3	NMI_N	Non-maskable interrupt (active low)
4	IRQ_N	Interrupt request (active low)
5	SO_N	Set overflow flag (active low)
6	GPIOA0	GPIO pin 0 input (unidirectional)

7	GPIOA1	GPIO pin 1 input (unidirectional)

Output Pins (uo_out[7:0])

Pin	Function	Description
0	PHI1	CPU phase 1 clock
1	PHI2	CPU phase 2 clock
2	R/W	Read/write signal (1=read, 0=write)
3	SYNC	Opcode fetch indicator
4	GPIOA2	GPIO pin 2 output (unidirectional)
5	GPIOA3	GPIO pin 3 output (unidirectional)
6	GPIOA4	GPIO pin 4 output (unidirectional)
7	GPIOA5	GPIO pin 5 output (unidirectional)

Bidirectional Pins (uio[7:0])

Pin	Function	Description
0-7	MUX_DATA[7:0]	Multiplexed address/data bus

How to Test

Minimal Setup

1. **Power**: 3.3V I/O, 1.2V core
2. **Clock**: 20MHz nominal (configurable)
3. **Reset**: Assert reset_n low, then high
4. **Memory Controller**: RP2040 or similar providing:
    - MUX_SEL[1:0] sequencing
    - External RAM/ROM contents
    - Proper bus timing

Peripheral Registers Quick Reference

GPIO (0xA000-0xA00B)

- **0xA000**: OE - Output Enable (0=input, 1=output)
- **0xA001**: OUT - Output Data
- **0xA002**: IN - Input Data (read-only)
- **0xA004-0xA00B**: MODE_PIN0-7 - Pin function select
    - 0x00 = GPIO, 0x01 = UART_TX, 0x02 = UART_RX
    - On TinyTapeout only PIN0-5 are connected to physical pins

Timer (0xA020-0xA027)

- **0xA020**: CTRL - Control (ENABLE | AUTO_RELOAD | IRQ_ENABLE | LOAD)

- **0xA021**: STATUS - Bit 0 = OVERFLOW (write 1 to clear)
- **0xA022-0xA023**: COUNT_LO/HI - Counter value (read-only)
- **0xA024-0xA025**: RELOAD_LO/HI - Reload value
- **0xA026**: PRESCALER - Clock prescaler (0-255)

Clock Control (0xA030-0xA033)

- **0xA030**: CPU_DIV - CPU clock divisor (0-255)
  - cpu_freq = cpu_clk / (CPU_DIV + 1)
- **0xA032**: STATUS - Bit 0 = CPU_LOCKED (always 1)

UART (0xA040-0xA047)

- **0xA040**: CTRL - TX_EN | RX_EN | TX_IRQ_EN | RX_IRQ_EN
- **0xA041**: STATUS - TX_READY | RX_READY | TX_EMPTY | RX_FULL | TX_ACTIVE | RX_ERROR
- **0xA042**: DATA - FIFO access (read/write)
- **0xA043-0xA044**: BAUD_LO/HI - Baud divisor
  - baud = sysclk / (16 × (divisor + 1))

Testing

Testbenches use cocotb:

```
cd test
make
```

Tests cover:

- CPU instruction execution
- Peripheral register access
- Bus multiplexer protocol
- UART TX/RX
- Timer operation
- GPIO modes

Architecture

Technology: IHP SG13G2 130nm **Die Size**: 2×2 TinyTapeout tiles **Clock**: 20 MHz nominal

External Resources

- **Datasheet**: m6502_datasheet.pdf - Complete technical reference
- **Upstream Project**: m6502 - Full MCU implementation with RP2040 memory controller example
- **6502 Reference**: 6502.org - Instruction set and programming guides
- **W65C02S Datasheet**: Western Design Center
- **MOS 6502 Programming Manual**: Original MOS Technology documentation

Development Tools

- **cc65**: C compiler and assembler suite for 6502
- **ACME**: Cross-assembler
- **py65**: Python-based 6502 simulator for testing code before hardware

What is TinyTapeout?

Tiny Tapeout is an educational project that makes it easier and cheaper to get your digital designs manufactured on a real chip. Learn more at tinytapeout.com.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	MUX_SEL[0] - Bus phase select bit 0	PHI1 - CPU phase 1 clock	MUX_DATA[0] - Multiplexed address/ data bus
1	MUX_SEL[1] - Bus phase select bit 1	PHI2 - CPU phase 2 clock	MUX_DATA[1] - Multiplexed address/ data bus
2	RDY - CPU ready signal	R/W - Read/write signal (1=read, 0=write)	MUX_DATA[2] - Multiplexed address/ data bus
3	NMI_N - Non-maskable interrupt	SYNC - Opcode fetch indicator	MUX_DATA[3] - Multiplexed address/ data bus
4	IRQ_N - Interrupt request	GPIOA2 - GPIO pin 2 output (unidirectional)	MUX_DATA[4] - Multiplexed address/ data bus
5	SO_N - Set overflow flag	GPIOA3 - GPIO pin 3 output (unidirectional)	MUX_DATA[5] - Multiplexed address/ data bus
6	GPIOA0 - GPIO pin 0 input (unidirectional)	GPIOA4 - GPIO pin 4 output (unidirectional)	MUX_DATA[6] - Multiplexed address/ data bus
7	GPIOA1 - GPIO pin 1 input (unidirectional)	GPIOA5 - GPIO pin 5 output (unidirectional)	MUX_DATA[7] - Multiplexed address/ data bus

tiny_tester

by **jalcim**

0164 HDL Project

`github.com/jalcim/tiny_tester`

"multiple_tester_for_ihp"

How it works

3 input

- 4bit input_A
- 4bit input_B
- 1bit Cin

1 ouput

- 5bit out

How to test

iverilog brent-kung.v test.v && ./a.out && gtkwave signal_brent_kung.vcd

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	in	out	in
1	in	out	in
2	in	out	in
3	in	out	in
4	in	out	in
5	in	out	in
6	mux	out	opt
7	mux	out	opt

Undecided

by **Martin Schoeberl**

0166 HDL Project

`github.com/schoeberl/ttihp-undecided`

"I have not yet decided what this will be"

How it works

This is just the template demo.

How to test

One segement will blink.

External hardware

None

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	a	d	—
1	b	e	—
2	c	g	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

test project

by **tarkor111**

`0168` `1 Hz` `Wokwi Project`

`github.com/tarkor111/tiny-tapeout-design`

`wokwi.com/projects/456572126761001985`

"led blink test"

How it works

It just blinks an LED on the 7-segment display, when the switch 1 is set to on.

How to test

There is nothing to test right now.

External hardware

None until now.

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	—

Smart LED digital

by **Christian Hoene**

0170 24 MHz HDL Project

github.com/hoene/tt_um_hoene_smart_led_digital

“i am trying to learn a bit”

How it works

It is a digital LED circuit.

How to test

First, run the software tests. More hardware plans will be added later.

External hardware

Add a reset source, a clock source, and a RGB LEDs with current of 4mA for each color.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	IN0	IN0SELECTED	STATE0
1	IN1	LOW_PASS_OUT	STATE1
2	—	DECODER_DATA	PROTOCOL_ERROR
3	—	DECODER_CLK	PROTOCOL_DATA
4	—	DECODER_ERROR	LED_RED
5	—	FRAME	LED_GREEN
6	—	TEST_MODE	LED_BLUE
7	—	PWM_SET	ENCODER_OUT

Workshop

by **Mikkel Heise Kofoed**

`0172` `Wokwi Project`

`github.com/MikkelKofoed/TinyTapeoutWorkshop`

`wokwi.com/projects/456572032892464129`

"Testing"

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	ui0	uo0	—
1	ui1	uo1	—
2	ui2	uo2	—
3	ui3	uo3	—
4	ui4	uo4	—
5	ui5	uo5	—
6	ui6	uo6	—
7	ui7	uo7	—

Tiny Tapeout Test

by **Kristoffer Matsumoto Mulbjerg**

0174 | 1 Hz | Wokwi Project

github.com/kurimm/GitHub-Wokwi-Template

wokwi.com/projects/456571983499280385

"Segment Blink"

How it works

1 Hz segment blink when 1st dip switch is ON.

How to test

Turn the 1st dip switch ON.

External hardware

Dip switch, 7 segment display

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	a	a	—
1	b	b	—
2	c	c	—
3	d	d	—
4	e	e	—
5	f	f	—
6	g	g	—
7	h	h	—

My first design

by **Vasilis Ntinas**

0176 10 kHz Wokwi Project

github.com/vntinas/My_Test_design

wokwi.com/projects/456576548374933505

"Some freq div and random logic"

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

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

Project Pinout

Digital Pins

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

fullAdder

by **Boibo**

0178 Wokwi Project

github.com/Nazgul-0/tinytapeoutGDS

wokwi.com/projects/456571746867102721

"fulladder"

How it works

Full Adder is a combinational circuit that adds three inputs and produces two outputs. The first two inputs are A and B and the third input is an input carry as C-IN. The output carry is designated as C-OUT and the normal output is designated as S which is SUM.

How to test

Set the inputs and check the outputs match

insert full-adder truth table

External hardware

regular LEDs

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	A	S	—
1	B	Cout	—
2	Cin	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

KianV SV32 TT Linux SoC

by **Hirosh Dabui** & **Uri Shaked**

0179 HDL Project

github.com/splinedrive/kianv-sv32-tt-linux-soc

"RISC-V Linux SoC on TT"

How it works

This design integrates a **KianV RV32IMA RISC-V processor** with an SV32 MMU and hardware caches (icache + dcache). The system is capable of booting **Linux, µLinux (uClinux), and xv6**, providing a compact Linux-capable SoC platform. The processor supports virtual memory through the SV32 MMU with two-level page table translation and separate instruction and data TLBs (8 entries each). The cache hierarchy uses a 512-set direct-mapped icache and dcache backed by IHP SG13G2 SRAM macros. The SoC includes **32 MiB of external QSPI PSRAM (4 × 8 MiB banks)**, **16 MiB of external SPI NOR flash**, and memory-mapped peripherals such as **UART, SPI, and GPIO**.

System Memory Map

Address	Size	Purpose
0x02000000	64 KiB	CLINT (timer/interrupts)
0x10000000	0x06	UART Peripheral
0x1000000C	0x0C	UART Divider / CPU info
0x10000700	0x0C	GPIO Peripheral
0x10500000	0x08	SPI Peripheral
0x10600000	0x04	QSPI PSRAM Control
0x11100000	0x04	Reset / HALT control
0x20000000	16 MiB	SPI NOR Flash
0x80000000	32 MiB	QSPI PSRAM

The system boots from SPI NOR flash. After reset, the CPU starts executing code from 0x20000000.

UART Peripheral registers

Address	Name	Description
0x10000000	UART_DATA	Write to transmit, read to receive
0x10000005	UART_LSR	UART line status register

0x1000000C	UART_DIV0	Clock divider for UART baud rate
0x10000010	UART_DIV1	Clock divider (alternate)
0x10000014	CPU_FREQ	CPU frequency register (Q8.8 MHz)
0x10000018	MEMSIZE	Memory size register

SPI Peripheral registers

Address	Name	Description
0x10500000	SPI_CTRL0	SPI control
0x10500004	SPI_DATA0	SPI data

GPIO Peripheral registers

Address	Name	Bits	Description
0x10000700	GPIO_DIR	-	Direction (output enable), currently unused in HW
0x10000704	GPIO_OUT	[9]	Output value driven on `uo_out[1]`
0x10000708	GPIO_IN	[7:0]	Read all 8 input pins (`ui_in[7:0]`)

Notes: GPIO_DIR is implemented in the register file but the output enable has no hardware effect. Only a single output bit exists (`uo_out[1]`), controlled via bit 9 of GPIO_OUT. All 8 `ui_in` pins are readable via GPIO_IN; pins 2 and 7 are shared with SPI MISO and UART RX respectively.

QSPI PSRAM Control register

Address	Name	Description
0x10600000	QSPI_CTRL	Bit 0: half_clock (1=div4 slow, 0=div2 fast clock)

CPU control register

Address	Name	Description
0x11100000	CPU_RESET	Write 0x7777 to reset the CPU, 0x5555 to halt the CPU

How to test

We will provide a pre-built system image + instructions how to build your own image.

External hardware

- Machdyne QSPI Pmod (4 x 8 MiB PSRAM banks)
- SPI NOR flash Pmod

- SD Card Pmod
- Ethernet Pmod (optional)

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	GPIO IN 0	UART TX	KVSMEM SS_N
1	GPIO IN 1	GPIO0 OUT	KVSMEM SIO0
2	SPI MISO	SPI CS1	KVSMEM SIO1
3	GPIO IN 3	SPI MOSI	KVSMEM SCLK
4	GPIO IN 4	SPI CS0	KVSMEM SIO2
5	GPIO IN 5	SPI SCLK	KVSMEM SIO3
6	GPIO IN 6	SPI CS2	KVSMEM CSN0
7	UART RX	SPI CS3	KVSMEM CSN1

7-Segment-Wokwi-Design

by **Marvin**

0193 Wokwi Project

github.com/MarvinBrth/7-Segment-Wokwi-Design

wokwi.com/projects/455301361268988929

"Flashing the letters (M,a,r,v,i,n)"

How it works

The 3 input pins can be used to decide which of the 6 letters should be displayed. The logic circuit then converts the input into specific letters.

"000 = M" "001 = a" "010 = r" "011 = v" "100 = i" "101 = n"

How to test

Simply enter a 0 or 1 on the first 3 pins and connect the output to a 7-segment display.

External hardware

Only switches for the input and a 7-segment display as output

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	1.Bit for Letter selection	a-pin on the 7-Segment	—
1	2.Bit for Letter selection	b-pin on the 7-Segment	—
2	3.Bit for Letter selection	c-pin on the 7-Segment	—
3	—	d-pin on the 7-Segment	—
4	—	e-pin on the 7-Segment	—
5	—	f-pin on the 7-Segment	—
6	—	g-pin on the 7-Segment	—
7	—	—	—

(Hexa)Decimal Counter

by **Maximilian Fuchs**

0195 10 MHz HDL Project

`github.com/elFuchso/tt_hdl_hex-cnt_elfuchso`

"Counts up using the slowed down clock and displays it in a configurable base on the 7 segment display"

How it works

D Flipflops are used to slow down the clock. The DIP switches are used to select the base of the counter that counts up in the 7 segment display

How to test

Look at the segment display and see if it counts up correctly in the selected base.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Up (1) / Down (0)	seg_a	unused
1	Hex (1) / Dec (0)	seg_b	unused
2	—	seg_c	unused
3	—	seg_d	unused
4	—	seg_e	unused
5	—	seg_f	unused
6	—	seg_g	unused
7	—	dot (Hex Mode Ind)	unused

move VGA square

by **Tobias Greiser**

0197 | 25 MHz | HDL Project

`github.com/tobiasgreiser/tiny_tapeout_workshop`

"Move the square as you like"

How it works

Iagine sitting in front of a computer screen, moving the mouse. That's exactly what I tried to replicate, except that you can change the cursor size when you have a hard time finding it. The cursor is supposed to be white on a gray background.

Inputs 0 to 5 are used:

- 0: Move the cursor to the left
- 1: Move the cursor to the right
- 2: Move the cursor to the top
- 3: Move the cursor to the bottom
- 4: Increase cursor size
- 5: Decrease cursor size

How to test

Connect a VGA monitor. Set the inputs according to your preference.

External hardware

Tiny VGA Pmod and switches.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Move left	R1	—
1	Move right	G1	—
2	Move up	B1	—
3	Move down	VSync	—
4	Decrease square size	R0	—
5	Increase square size	G0	—
6	—	B0	—
7	—	HSync	—

Temporary Title

by **Clemens Lerchl**

0199 Wokwi Project

github.com/IsAreWhoKey/TTWorkshopThing

wokwi.com/projects/455291603750154241

"Nothing currently"

How it works

It currently does not

How to test

One does not

External hardware

NA

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input a	out xor	—
1	input b	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

RGB PWM

by **Johannes Nekes**

0201 Wokwi Project

`github.com/JohannesNekes/Tiny-Tapeout-RGB-PWM`

`wokwi.com/projects/455297270449581057`

"A small PWM module"

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	Bit 7 (MSB)	Out	—
1	Bit 6	—	—
2	Bit 5	—	—
3	Bit 4	—	—
4	Bit 3	—	—
5	Bit 2	—	—
6	Bit 1	—	—
7	Bit 0 (LSB)	—	—

Primitive clock divider

by **Alexey**

0203 100 Hz Wokwi Project

github.com/alexey-serdyuk/tiny_tapeout_workshop

wokwi.com/projects/455291640579325953

"Very primitive clock divider"

How it works

This is design of a very simple clock divider.

How to test

Just apply 100 Hz clock and release reset.

External hardware

It will toggle segmend G and segment DP of the digit indicator.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	—	—
1	—	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	output out6	—
7	—	output out7	—

Tiny Ape Out

by **Byte Nibbleson**

0205 | 10 MHz | Wokwi Project

github.com/Byte-01100110/TT-Ape-out

wokwi.com/projects/455291654653337601

"a small gorilla escaped the zoo and is on a rampage"

How it works

It's a blinky. with a clock downsamper. Input 6 & 7 are connected to an xor.

How to test

idk

External hardware

7 segment display, 8 switches for input

Project Pinout

Digital Pins

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

TinyMOA: RISC-V CPU with Compute-in-Memory Accelerator

by **Ezra Wolf**

206 | 50 MHz | HDL Project

github.com/EzraWolf/TinyMOA-IHP26a

> *"RISC-V CPU based on TinyQV with an integrated SRAM-based compute-in-memory (CIM) accelerator for performing efficient analog matrix multiplications."*

How it works

WIP

How to test

WIP

External hardware

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

Setup

- Download VSCode or alternative code editor
- Create the IHP26a implementation through the template here https://github.com/TinyTapeout/ttihp-verilog-template
- Create the main TinyDCIM repo
- Setup code skeletons & link main repo to the IHP26a repo through submodule
- Get dummy counter sanity test working with GDS and submit through TT site
- Download KLayout
- Download IHP PDK https://github.com/IHP-GmbH/IHP-Open-PDK
- Copy right files (?) to KLayout

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	enable	dbg_pc[0]	—
1	up_down	dbg_pc[1]	—
2	clk	dbg_pc[2]	—
3	—	dbg_pc[3]	—
4	—	dbg_pc[4]	—
5	—	dbg_state[0]	—
6	—	dbg_state[1]	—
7	—	dbg_state[2]	—

adder

by **semommes**

0207 | 100 Hz | Wokwi Project

`github.com/stsar/tiny_tapeout`

`wokwi.com/projects/455291618175430657`

"Full single bit adder"

How it works

it doesnt work

How to test

u cant test it

External hardware

no hardware

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	a	—
1	—	b	—
2	—	—	—
3	—	—	—
4	—	—	—
5	a	—	—
6	b	c	—
7	c	d	—

test

by **Peter von Kürten**

`0209` `10 Hz` `Wokwi Project`

`github.com/DasKunstUngetuem/tt`

`wokwi.com/projects/455291689699908609`

“test AND”

How it works

AND and clock

clc= out7

How to test

Input A & B Output 00 0 01 0 10 0
11 1

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	In0	And	—
1	In1	—	—
2	In2	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	clc	—

quad-sieve

by **Christopher Swenson**

0210 50 MHz HDL Project

github.com/swenson/ttihp-quad-sieve

"TinyTapeout Quadratice Sieve chip based on 'A Pipeline Architecture for Factoring Large Integers with the Quadratic Sieve Algorithm' by Pomerance, Smith, and Tuler"

Block Processor for Quadratic Sieve Factoring

This project implements a **Block Processor (BP)** from the classic 1988 paper "A Pipeline Architecture for Factoring Large Integers with the Quadratic Sieve Algorithm" by Pomerance, Smith, and Tuler.

Overview

The quadratic sieve is one of the fastest known algorithms for factoring large composite numbers. This implementation creates a hardware accelerator for the sieving stage of the algorithm, which is the computational bottleneck.

How it works

The quadratic sieve algorithm finds numbers that factor completely over a small "factor base" of primes. The BP performs the following operation repeatedly:

1. For each prime power $q \leq B$ and address $A \equiv A_q \pmod{q}$:
   - Read S[A] from memory
   - Compute $S[A] \leftarrow S[A] + \lambda(q)$ where $\lambda(q) = \log(q)$
   - Write S[A] back to memory
   - Update $A \leftarrow A + q$
2. When S[A] exceeds a threshold T, the value at position A likely factors completely over the factor base.

This "sieving" operation is embarrassingly parallel and well-suited to hardware acceleration.

Design Features

- **External SPI RAM Interface**: Uses a 64KB SPI SRAM (such as 23LC512) for sieve memory storage
- **Pipeline Architecture**: Implements the Block Processor design from the 1988 paper
- **Multiple Operating Modes**:
  - **IDLE** (00): Low power state

- **INIT** (01): Initialize all 64K bytes of sieve memory to zero
- **SIEVE** (10): Main sieving operation
- **REPORT** (11): Re-sieving mode for identifying successful factorizations

Pin Configuration
Dedicated Inputs (ui_in[7:0]):

- `ui_in[1:0]`: Mode select (00=IDLE, 01=INIT, 10=SIEVE, 11=REPORT)
- `ui_in[2]`: Data valid input signal
- `ui_in[7:3]`: Command/data input (5 bits)

Dedicated Outputs (uo_out[7:0]):

- `uo_out[0]`: Busy signal (1 when processing)
- `uo_out[1]`: Valid output (1 when operation complete)
- `uo_out[2]`: Report valid (1 when threshold exceeded)
- `uo_out[7:3]`: Status/data output (5 bits)

Bidirectional I/O (uio[7:0]):

- `uio[0]`: SPI CS (Chip Select) - Output
- `uio[1]`: SPI SCK (Clock) - Output
- `uio[2]`: SPI MOSI (Master Out) - Output
- `uio[3]`: SPI MISO (Master In) - Input
- `uio[7:4]`: Reserved for future use

How to test

1. **Connect external SPI RAM** (see External hardware section)
2. **Initialize the sieve memory:**
   - Set `ui_in[1:0] = 01` (INIT mode)
   - Wait for `uo_out[0] = 0` (not busy)
   - This takes approximately 1.3 seconds at 10 MHz
3. **Perform sieving:**
   - Set `ui_in[1:0] = 10` (SIEVE mode)
   - Set `ui_in[2] = 1` (data valid)
   - Set `ui_in[7:3]` = starting address
   - Monitor `uo_out[2]` for reports (threshold exceeded)
4. **Check results:**
   - When `uo_out[2] = 1`, read `uo_out[7:3]` for address
   - This address contains a value that likely factors completely
5. **Run automated tests:**

```
cd test
make
```

External hardware

Required:

- **23LC512 64KB SPI SRAM** or compatible (e.g., 23LC1024)
  - Microchip part number: 23LC512-I/P or 23LC512-I/SN
  - Connect as follows:
    - Pin 1 (CS) → uio[0]
    - Pin 2 (SO/MISO) → uio[3]
    - Pin 3 (NC) → Not connected
    - Pin 4 (VSS) → Ground
    - Pin 5 (SI/MOSI) → uio[2]
    - Pin 6 (SCK) → uio[1]
    - Pin 7 (HOLD) → VCC (3.3V)
    - Pin 8 (VCC) → 3.3V
  - Add 10kΩ pull-up resistor on CS line (recommended)

Optional:

- **Logic analyzer** to observe SPI transactions
- **LED indicators** on output pins to visualize operation

Power Requirements:

- 3.3V supply for both ASIC and SRAM
- Typical current: 10mA (SRAM), plus ASIC current

Historical Context

This implementation is based on groundbreaking research from 1988, when the authors predicted that custom hardware could factor 100-digit numbers in less than a month. The paper estimated a custom $50,000 device could factor 100-digit numbers in two weeks, representing an order of magnitude speedup over 1988-era supercomputers.

References

Pomerance, C., Smith, J.W., and Tuler, R. “A Pipeline Architecture for Factoring Large Integers with the Quadratic Sieve Algorithm.” *SIAM Journal on Computing*, Vol. 17, No. 2, April 1988, pp. 387-403.

Built with Tiny Tapeout

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	in0	out0	cs

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

Just logic

by **Jannis Weiner**

0211 Wokwi Project

github.com/Jannis-Uni/TinyTapeout

wokwi.com/projects/455291713774178305

"shows stuff on 7 segment display with code 1-2-4-6"

How it works

Uses Nand and And To check for correct input and activates 7 segment display ## How to test

use 1-2-4-6 as code the 7 segment display will show a 7

External hardware

None

Project Pinout

Digital Pins

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

FOMO

by **algofoogle (Anton Maurovic)**

0224 HDL Project

`github.com/algofoogle/ttihp26a-fomo`

"*Fear Of Missing Out on TTIHP26a!*"

How it works

31-stage ring oscillator with controllable current sources.

How to test

Set `ui_in = 0xFF` and `ring_enb=1` to make things oscillate on `uo_out[5:0]`.

External hardware

Maybe just an oscilloscope to measure timing?

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	sel_n[0]	div8	ring_enb
1	sel_n[1]	div16	VGND
2	sel_n[2]	div32	VGND
3	sel_n[3]	div64	VGND
4	sel[0]	div128	VGND
5	sel[1]	div256	VGND
6	sel[2]	—	VGND
7	sel[3]	—	VDPWR

Basic Oszilloscope and Signal Generator

by **Pascal Gesell**

`225` `25 MHz` `HDL Project`

`github.com/gfcwfzkm/ttihp-scope`

"*Basic oscilloscope & signal generator on an ASIC*"

Authors: Pascal Gesell, Dr. Torsten Maehne, Dr. Theo Kluter

How it works

This is a basic oscilloscope design using the experimental VHDL template. It samples the input signal from channel 1 of an ADC Pmod (Digilent PmodAD1) and buffers the samples on an external FRAM. The captured signal is output on screen via a BlackMesa HDMI Pmod. Test signals are generated using Direct Digital Synthesis and are output on channel 1 of the DAC Pmod (Digilent PmodDA2). Four buttons and two switches allow to control the oscilloscope and choose the test signal to generate.

When the trigger button is pressed, a single-shot measurement is taken when the trigger criteria is met. The trigger criteria can be the vertical and horizontal position as well as the trigger level (positive edge or negative edge). The data is buffered onto the external FRAM, with the goal to contain 32k samples before the trigger event and 32k samples after the trigger event. After the data is collected, the data is displayed on the HDMI screen.

Since an external FRAM memory is used with no buffers on the chip, the displayed oscilloscope screen is actually rotated by 90° to the right. Thus only one sample needs to be read from the FRAM per output video line. A Python script is provided for convenience to read the video frames captured by an USB HDMI video grabber, rotate them by 90° to the left and display them on the screen.

The signal generator supports a few basic waveforms: sine, square, triangle and sawtooth. The frequency and amplitude can be adjusted using the buttons and switches. The signal generator is also used to test the trigger functionality and the display of the oscilloscope.

The scope settings are continuously output via UART at 9600 baud (8N1) on `uo_out(3)`.

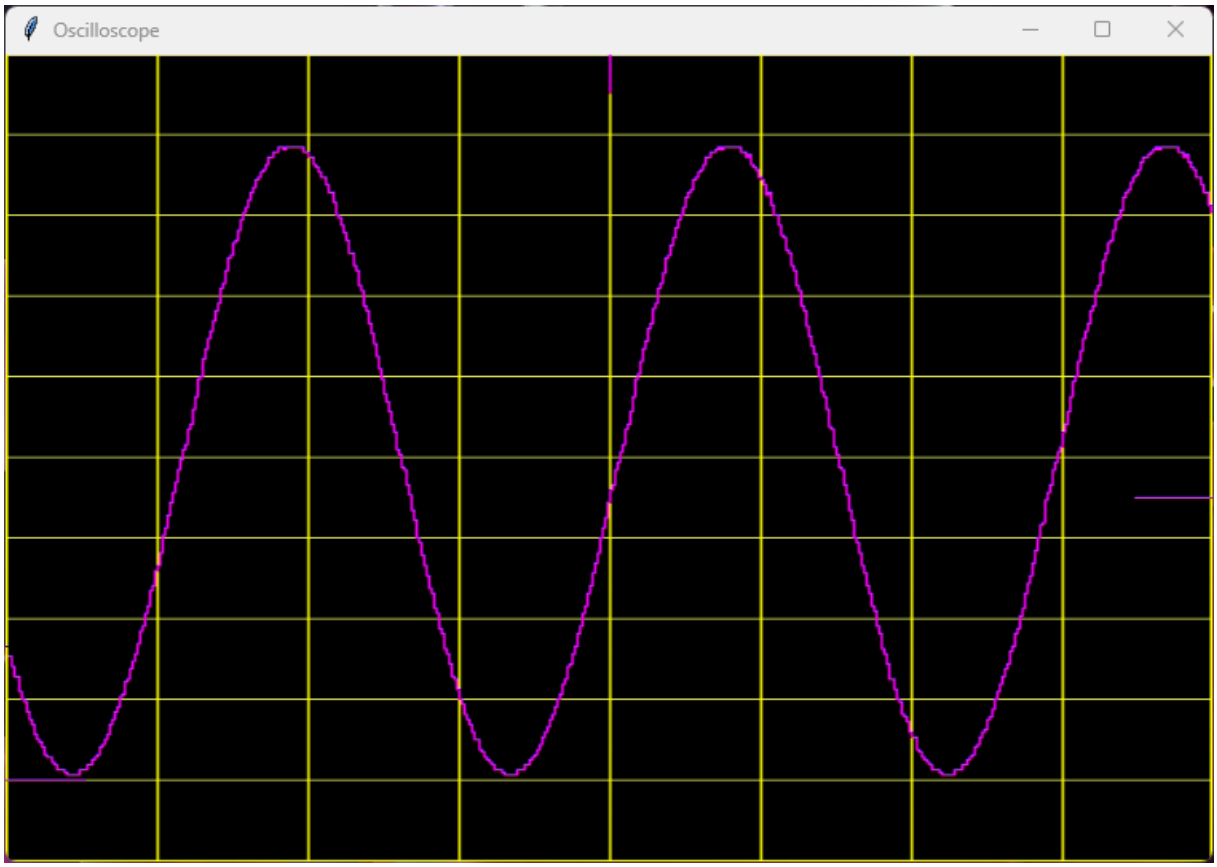


Figure 225.1: Image of Scope

How to test

Connect the various Pmods to the TinyTapeout 4+ demo board or FPGA board according to the pinout description in the info.yaml file. Connect the output of the DAC to the input of the ADC and connect the HDMI Pmod to a screen or HDMI capture card. Run the trigger to capture a single-shot measurement and display the data on the screen.

External hardware

To test and use this project, you will need the following hardware:

- 1 × BlackMesaLabs 3-bit HDMI Pmod : A 3-bit HDMI Pmod
- 1 × Digilent PmodAD1 : A 12-bit ADC Pmod
- 1 × Digilent PmodDA2 : A 12-bit DAC Pmod
- 1 × FM25W256G : 32k x 8 FRAM Pmod
- 1 × Digilent PmodBTN : A 4 Buttons Pmod
- 1 × Digilent PmodSWT : A 4 Switches Pmod, of which only 2 are used
- Optionally, an HDMI capture card to display the HDMI output on a computer screen

Attention: The above Pmods cannot be directly connected to the TinyTapeout 4+ demo board! The Pmods' pins need to be individually connected to

the right Pmod pins of the TinyTapeout 4+ demo board, as documented in the IO section.

FPGA Implementation

The design has been implemented and tested on a Sipeed Tang Nano 9k FPGA board using my own base PCB to have enough available Pmods to test the design.

Acknowledgements

This work was realized under the supervision of Dr. Torsten Maehne and Dr. Theo Kluter as part of my project work in the 5th term of my Bachelor studies of electrical engineering and information technology at Berner Fachhochschule (BFH), Biel/Bienne, Switzerland.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	FRAM MISO	ADC CS	HDMI Pmod Green
1	Button 1	DAC MOSI	HDMI Pmod Clock
2	Button 3	ADC SCLK	HDMI Pmod HSYNC
3	Switch 1	FRAM SCLK	UART_TX Settings Info (9600bps, 8N1)
4	ADC MISO	DAC CS	HDMI Pmod Red
5	Button 2	DAC SCLK	HDMI Pmod Blue
6	Button 4	FRAM CS	HDMI Pmod DE
7	Switch 2	FRAM MOSI	HDMI Pmod VSYNC

ADPLL

by **Christian Birk Sørensen**

0226 HDL Project

github.com/chrbirks/tt-2026

"All-Digital PLL"

How it works

An All-Digital Phase-Locked Loop (ADPLL).

```
clk_ref ──►[Bang-Bang PD]──► early/late ──►[Loop Filter]──►
freq_ctrl[6:0]
                ▲                          (up/down counter)
|
                |
▼
          [Divider /8] ◄── dco_clk ◄──────────────────────[Ring Osc
DCO] ──► clk_pll_o
```

The bang-bang phase detector compares the divided DCO output against the reference clock. If the DCO is too fast, the loop filter decrements `freq_ctrl` (adding delay stages, slowing the DCO). If too slow, it increments. At lock, `freq_ctrl` dithers by +/-1 around the target value.

Target specs

- Reference clock: 5-10 MHz
- DCO range: 300-600 MHz (7-stage ring oscillator)
- Output clock: 40-75 MHz (with /8 divider)

See README.md for more details.

How to test

The PLL will start to lock onto the input reference clock after reset is deasserted.

The generated PLL can be measured on output pin uo[0].

uo[0]: "clk_pll_o" uo[1]: "locked_o" uo[2]: "clk_ref_o"

The clock frequency is adjusted by a 7-bit control that can be read on the bi-dir pins for debugging:

uio[0]: "freq_ctrl_o[0]" uio[1]: "freq_ctrl_o[1]" uio[2]: "freq_ctrl_o[2]" uio[3]: "freq_ctrl_o[3]" uio[4]: "freq_ctrl_o[4]" uio[5]: "freq_ctrl_o[5]" uio[6]: "freq_ctrl_o[6]"

External hardware

Oscilloscope nedded for measuring output frequency.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	clk_pll_o	freq_ctrl_o[0]
1	—	locked_o	freq_ctrl_o[1]
2	—	clk_ref_o	freq_ctrl_o[2]
3	—	—	freq_ctrl_o[3]
4	—	—	freq_ctrl_o[4]
5	—	—	freq_ctrl_o[5]
6	—	—	freq_ctrl_o[6]
7	—	—	—

Tiny MMU

by **Yuri Honegger**

0227 | 50 MHz | HDL Project

`github.com/recursivetree/tt-tmmu`

"A tiny MMU for a TTL logic chip cpu"

How it works

This is a TLB (translation lookaside buffer) for a TTL IC breadboard CPU using latch memory.

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	vaddr0	paddr0	write_en
1	vaddr1	paddr1	read_valid
2	vaddr2	paddr2	tlb_hit
3	vaddr3	paddr3	—
4	vaddr4	paddr4	—
5	vaddr5	paddr5	—
6	vaddr6	paddr6	—
7	vaddr7	paddr7	—

Photo Frame

by **Mike Bell**

`0228` `25 MHz` `HDL Project`

`github.com/MichaelBell/ttihp26a-photo-frame`

"Display images from flash"

How it works

Reads pixel data from QSPI flash using a DTR read. Displays on VGA.

Timing and latency are very configurable, hopefully allowing full resolution images at up to 720p and 1024x768 resolutions.

The image format is RGB332, which can be either truncated or dithered to the RGB222 format required by the Tiny VGA PMOD.

There are two config shift registers that control the design. The first controls the VGA timing parameters, plus a trigger count of how many cycles before the active display region the QSPI read should be started. The second sets the address of the QSPI read, whether to use full res mode, and whether to dither.

The active config register is selected by in7. This should allow quick changing of the address without affecting VGA configuration.

How to test

Flash an RGB332 image to the QSPI flash (e.g. using the Tiny Tapeout flasher), set the config registers, and enable.

The image address can be set to any multiple of 128kB, allowing multiple images to be stored and switched between by changing the config register. This should allow short animations to be displayed with a simple script on the RP2.

You can create RGB332 images using the `make_img_bin.py` script in the repo.

The `photo.py` script in the `upy` directory gives an example of how to configure the design.

By default, images should be half the resolution of the configured timing mode. For full resolution images you must double the clock rate, double all the horizontal timing parameters, and set the full res bit in the QSPI config register.

External hardware

QSPI PMOD, Tiny VGA PMOD

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Config clk	R[1]	CS
1	Config data	G[1]	SD0 / SCK
2	Display enable	B[1]	SD1 / SD0
3	Select QSPI pinout	vsync	SCK / SD1
4	QSPI latency 0	R[0]	SD2
5	QSPI latency 1	G[0]	SD3
6	QSPI latency 2	B[0]	Unused CS
7	Config register selection	hsync	Unused CS

tt_gian_alu

by **Gianmarco Fortunelli**

`0229` `50 MHz` `HDL Project`

`github.com/GianmarcoFortunelli/TT_coprocessor`

"Tiny Tapeout 32-bit ALU with integrated multiplier and nibble-based serial I/O interface."

Authors

- **Gianmarco Fortunelli**
- **Francesca Di Giuseppe**
- **Mattia Cesaraccio**

How it works

This project implements a 32-bit Arithmetic Logic Unit (ALU) for Tiny Tapeout.

The ALU supports arithmetic, logic, shift, comparison, and multiplication operations.
Operands are 32-bit wide, while the operation code is provided as a 6-bit function field (`FUNC_BITS`).

Supported operations are:

- Arithmetic: `ADD`, `ADDU`, `SUB`, `SUBU`
- Logic: `AND`, `OR`, `XOR`
- Shifts: `SLL`, `SRL`, `SRA`
- Comparisons: `SEQ`, `SNE`, `SLT`, `SGT`, `SLE`, `SGE`, `SLTU`, `SGTU`, `SLEU`, `SGEU`
- Multiplication: `MUL`

Multiplication

For the `MUL` operation, only the lower 8 bits of each operand are used internally by the multiplier, producing a 16-bit product.
This product is then zero-extended to 32 bits on the ALU output.

Tiny Tapeout interface

The project uses a simple custom serial protocol with 4-bit parallel transfers ("nibble interface").

Inputs

- `ui_in[0]` = `ext_progr`
  Starts and keeps active a transaction
- `ui_in[7:4]` = input nibble (`in_data`)
- `ui_in[3:1]` = unused

Outputs

- `uo_out[7:4]` = output nibble (`out_data`)
- `uo_out[2]` = `busy`
- `uo_out[1]` = `frame_error`
- `uo_out[0]` = `result_valid`
- `uo_out[3]` = unused

Bidirectional IO

- `uio_in[7:0]` unused
- `uio_out[7:0]` unused
- `uio_oe[7:0]` all disabled

Transaction behavior

When `ext_progr` is asserted high, the wrapper starts a transaction.

Each transaction has two phases:

1. **Read previous result**
   - The circuit first outputs the result computed in the previous transaction
   - The result is sent on `uo_out[7:4]`
   - It is transmitted as 8 nibbles, most significant nibble first
2. **Load new operation**
   - The user sends:
     - operand A = 8 nibbles
     - operand B = 8 nibbles
     - function code = 2 nibbles
   - After internal processing, the new result is stored
   - **That result will be returned at the beginning of the next transaction**

How to test

To use the project, drive the interface synchronously with the main clock.

Start a transaction

- Set `ui_in[0] = 1` (`ext_progr = 1`)

Step 1: read previous result

For 8 clock cycles:

- sample `uo_out[7:4]`
- reconstruct the 32-bit result nibble by nibble
- transmission order is **MS nibble first**

Step 2: send operand A

For 8 clock cycles:

- place one nibble of operand A on `ui_in[7:4]`
- send the most significant nibble first

Step 3: send operand B

For 8 clock cycles:

- place one nibble of operand B on `ui_in[7:4]`
- send the most significant nibble first

Step 4: send function code

For 2 clock cycles:

- send the 6-bit function code packed into 2 nibbles:
  - first nibble = `{2'b00, FUNC[5:4]}`
  - second nibble = `FUNC[3:0]`

Step 5: wait internal processing

- keep `ext_progr = 1` for 2 more clock cycles

Step 6: close the transaction

- set `ui_in[0] = 0`

Result timing

The result of the operation just submitted is **not returned immediately**.
It will be available as the “previous result” at the beginning of the next transaction.

Status bits

- `uo_out[2] = busy`: transaction in progress
- `uo_out[1] = frame_error`: protocol error
- `uo_out[0] = result_valid`: at least one valid result has been computed

Function codes

Operation	FUNC_BITS
SLL	`000100`
SRL	`000110`
SRA	`000111`
ADD	`100000`
ADDU	`100001`
SUB	`100010`
SUBU	`100011`

AND	100100
OR	100101
XOR	100110
SEQ	100111
SNE	101000
SLT	101001
SGT	101010
SLE	101011
SGE	101100
SLTU	111010
SGTU	111011
SLEU	111100
SGEU	111101
MUL	111110

External hardware

No external hardware is required. The protocol can be driven by a microcontroller.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	ext_progr (start/hold transaction)	result_valid	unused
1	unused	frame_error	unused
2	unused	busy	unused
3	unused	unused	unused
4	in_data[0] (LSB of input nibble)	out_data[0] (LSB of output nibble)	unused
5	in_data[1]	out_data[1]	unused
6	in_data[2]	out_data[2]	unused
7	in_data[3] (MSB of input nibble)	out_data[3] (MSB of output nibble)	unused

Canright SBOX

by **Gijs Burghoorn**

0230 | 50 Hz | HDL Project

github.com/coastalwhite/tinytapeout-ihp-canright

"*Canright SBOX for the AES block cipher.*"

How it works

The design is driven by commands on `uio_in[3:0]`.

`uio_in[3:0]`	Command	Description
`0000`	`IDLE`	No-operation command. Generally assumed to be default command.
`0001`	`DATA_IN`	Load the data from `ui_in` into the internal `data_i` register.
`0010`	`KEY_IN`	Load the data from `ui_in` into the internal `key` register.
`0011`	`MASK_IN`	Load the data from `ui_in` into the internal `mask_i` register.
`0100`	`PRD_7_0_IN`	Load the data from `ui_in` into `prd_i[7:0]` of the internal `prd_i` register.
`0101`	`PRD_15_8_IN`	Load the data from `ui_in` into `prd_i[15:8]` of the internal `prd_i` register.
`0110`	`PRD_17_16_IN`	Load the data from `ui_in[1:0]` into `prd_i[17:16]` of the internal `prd_i` register.
`1000`	`UNMASKED_DATA_OUT`	Drive the data output of the unmasked SBOX on `uo_out`.
`1001`	`MASKED_DATA_OUT`	Drive the data output of the masked SBOX on `uo_out`.
`1010`	`MASKED_MASK_OUT`	Drive the mask output of the masked SBOX on `uo_out`.

Every `posedge` of the `clk` a command will be read and executed.

There are two SBOXes that are boxes that are triggered using `uio_in[7:6]`.

If `uio_in[7]` is set to `1`, the unmasked SBOX will evaluate `UNMASKED_SBOX(data_i ^ key)`. If `uio_in[6]` is set to `1`, the masked SBOX will evaluate `MASKED_SBOX(data_i ^ key, mask_i, prd_i)`.

How to test

The design is tested using `cocotb` by passing in a bunch of test vectors and asserting the correct output value. This can be ran using `make` or by running `python3 runner.py` from the `test` directory.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	DATA_I[0]	DATA_O[0]	STATE[0]
1	DATA_I[1]	DATA_O[1]	STATE[1]
2	DATA_I[2]	DATA_O[2]	—
3	DATA_I[3]	DATA_O[3]	—
4	DATA_I[4]	DATA_O[4]	—
5	DATA_I[5]	DATA_O[5]	—
6	DATA_I[6]	DATA_O[6]	—
7	DATA_I[7]	DATA_O[7]	—

8-bit RISC-V Lite CPU

by **Wiwathana Son**

`0231` `10 MHz` `HDL Project`

`github.com/XxWiwaxX/chip1`

"8-bit CPU with hardware multiplier, 32B RAM, and custom branching instructions."

How it works

This project is a custom **8-bit RISC-V-inspired CPU** optimized for high-density synthesis on the IHP26a process (TinyTapeout 1x2 tile). It utilizes a multi-cycle Finite State Machine (FSM) to handle fetching, decoding, and execution.

Architecture Features:

- **Registers:** 8 general-purpose 8-bit registers (r_0 to r_7). r_0 is hardwired to zero at the logic level to save area and ensure ISA compatibility.
- **RAM:** 24 bytes of internal storage, implemented as a flip-flop array for high-speed access.
- **Hardware Multiplier:** A dedicated 8-bit combinational multiplier allows for single-cycle multiplication during the `EXECUTE` phase.
- **Safety:** Includes bounds-checking to prevent out-of-range memory writes to the 24-byte RAM.

Instruction Set (16-bit):

Instructions are stored in **Little-Endian** format. The bit-fields are organized as follows: `[Opcode: 4 bits] [Unused: 2 bits] [rd: 3 bits] [rs1: 3 bits] [imm/rs2: 4 bits]`

Opcode	Mnemonic	Description
0000	ADD	rd = rs1 + rs2
0001	ADDI	rd = rs1 + imm (4-bit)
0010	MUL	rd = rs1 * rs2 (8-bit result)
0011	LW	rd = RAM[rs1]
0100	SW	RAM[rs1] = rd
0101	BEQ	If rd == rs1, PC = PC + imm
0110	BNE	If rd != rs1, PC = PC + imm

How to test

The CPU features a built-in bootloader mode. Follow these steps to load and run a program:

1. **Reset:** Pull `rst_n` LOW. This resets the Program Counter (PC) to 0 and enters `LOAD` mode.
2. **Program Loading:**
   - Keep `uio[7]` (Mode Select) **LOW**.
   - Set the target RAM address (0-23) on `uio[4:0]`.
   - Set the instruction/data byte on `ui[7:0]`.
   - Toggle the clock (`clk`) to write the byte to RAM.
3. **Execution:** Set `uio[7]` to **HIGH**. The CPU will exit the `LOAD` state and begin fetching instructions from RAM address 0.
4. **Monitoring:** * `uo[7:0]` continuously outputs the value of the destination register (`rd`) of the current instruction.
   - `uio[4:0]` outputs the current Program Counter (PC) for debugging.

External hardware

- **Logic Analyzer/LEDs:** Connect to `uo[7:0]` to verify calculation results.
- **Clock Source:** 10MHz recommended for IHP26a, though it can run at much lower speeds for manual step-through debugging.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	data_in[0]	reg_out[0]	addr[0]
1	data_in[1]	reg_out[1]	addr[1]
2	data_in[2]	reg_out[2]	addr[2]
3	data_in[3]	reg_out[3]	addr[3]
4	data_in[4]	reg_out[4]	addr[4]
5	data_in[5]	reg_out[5]	addr[5]
6	data_in[6]	reg_out[6]	addr[6]
7	data_in[7]	reg_out[7]	mode_sel

8-bit SEM Floating-Point Multiplier

by **Jordan Delos Reyes**

`0232` `50 MHz` `HDL Project`

`github.com/DelosReyesJordan/ttihp26a-FP8-SEM-Multiplier`

"Custom 8-bit Sign-Exponent-Mantissa floating-point multiplier with NaN and zero handling."

8-bit SEM Floating-Point Multiplier

What it does

This project implements an 8-bit floating-point multiplier using a custom Sign–Exponent–Mantissa (SEM) format.

Each 8-bit number is structured as:

S EEEE MMM

- 1 sign bit
- 4 exponent bits
- 3 mantissa bits

The design multiplies two SEM numbers and produces an 8-bit SEM result. The architecture is divided into three major blocks:

- Sign computation
- Mantissa multiplication and normalization
- Exponent addition and adjustment

The design also handles two special cases:

1. NaN
   If either input equals `S.1111.111`, the output is forced to `0_1111_111`.
2. Zero
   If either input equals `S.0000.000`, the output is forced to `0_0000_000`.

Priority:
NaN > Zero > Normal multiplication

How to use

Inputs

- `ui_in[7:0]` → Operand A (SEM format)
- `uio_in[7:0]` → Operand B (SEM format)
- `clk` → Clock

- `rst_n` → Active-low reset

Output

- `uo_out[7:0]` → Result (SEM format)

Operation

1. Apply operands A and B.
2. Pulse reset low to initialize the system.
3. On the next clock edge, the multiplier produces the result.

The design is fully synchronous and updates on the rising edge of `clk`.

Example

Input A: `0_0101_010`
Input B: `0_0011_001`

Output: `S_EEEE_MMM` (result of SEM multiplication)

If either input is:

`x_1111_111` → Output = `0_1111_111` (NaN)

`x_0000_000` → Output = `0_0000_000` (Zero)

External hardware

No external hardware is required.

The module is fully self-contained and purely digital.

Design details

The multiplier is implemented using three synchronous submodules:

- `sign`
  Computes the result sign as the XOR of the input signs.
- `mantissa`
  Multiplies mantissas, normalizes the result, and generates an exponent increment.
- `exponent`
  Adds exponents and applies normalization adjustment.

The final result is registered for synchronous output.

Limitations

- No support for infinities.
- Only a single canonical NaN is supported.
- Exponent overflow behavior depends on internal saturation logic.

Verification

The design was verified through simulation of:

- Normal multiplication cases
- Zero input cases
- NaN input cases
- Sign combinations

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	A_M0	R_M0	B_M0
1	A_M1	R_M1	B_M1
2	A_M2	R_M2	B_M2
3	A_E0	R_E0	B_E0
4	A_E1	R_E1	B_E1
5	A_E2	R_E2	B_E2
6	A_E3	R_E3	B_E3
7	A_S	R_S	B_S

NYAN CAT

by **tippfehlr**

`0233` `10 MHz` `HDL Project`

`github.com/tippfehlr/tinytapeout-nyancat`

"Plays the NYAN CAT theme, hopefully. Written by GitHub Copilot, probably GPT-5.3-Codex"

How it works

All 16 output pins (8 dedicated outputs `uo` and 8 bidirectional `uio` configured as outputs) blink together to transmit the message **"HELLO WORLD"** in Morse code, then repeat continuously.

Timing (standard Morse code at 20 WPM, 16 Hz clock)

Symbol	Duration
Dot (·)	1 unit ≈ 62.5 ms (1 clock cycle)
Dash (−)	3 units ≈ 187.5 ms
Element gap (between dots/dashes within a character)	1 unit
Character gap	3 units
Word gap	7 units

Morse code sequence

Letter	Code
H	....
E	.
L	.-..
L	.-..
O	---
(space)	7-unit gap
W	.--
O	---
R	.-.
L	.-..
D	-..

After the final letter D, a 4-word-space gap (4 × 7 = 28 units) is inserted before the sequence repeats.

How to test

Connect LEDs (or a logic analyser) to any of the 16 output pins. Power the device with a 16 Hz clock. After releasing reset (`rst_n` high), all outputs will blink the Morse code for "HELLO WORLD" at 20 WPM and loop indefinitely.

No inputs are required; `ui_in` and `uio_in` are ignored.

External hardware

LEDs connected to `uo[0..7]` or `uio[0..7]` (with appropriate current-limiting resistors) will visibly display the Morse code message.

Project Pinout

Digital Pins

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

One One

by **Gustav Kolind**

0234 HDL Project

`github.com/gusish89/tinytapeout_wokwi0`

"*First Asic design, we will see where this goes*"

How it works

The design receives a 1-bit serial input stream on `ui[0]`, collects 16 bits into a sample, and applies either a low-pass or high-pass filter selected by `ui[1]`. The filtered sample is then shifted out serially on `uo[0]`.

How to test

Apply reset, then drive input bits on `ui[0]` one bit per clock cycle. Set `ui[1]=1` for low-pass or `ui[1]=0` for high-pass before streaming starts. After the first 16-bit word is received, output bits begin appearing on `uo[0]` with pipeline delay.

External hardware

No external hardware is required.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Waveform input	Filtered output	—
1	Lowpass/Highpass select	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Three Body Solution

by **Jan**

0235 | 25.175 MHz | HDL Project

`github.com/jknotrowling/TinyTapeoutVGA`

"Basic three body simulation using manhattan distances and bucketed acceleration values"

How it works

yAdda yadda

How to test

dont

External hardware

blabla

Project Pinout

Digital Pins

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

Count To Ten

by **Isaak**

0236 Wokwi Project

github.com/Isk1337/Count-To-Ten

wokwi.com/projects/457062377137305601

"*Count To Ten*"

How it works

asdf Explain how your project works

How to test

fasdf Explain how to use your project

External hardware

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

Project Pinout

Digital Pins

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

RO-based security primitives

by **Simon Pankner**

`0237` `50 MHz` `HDL Project`

`github.com/thravax/tt-test`

"Two physical security primitives based on Ring-Oscillators: RO-PUF and TRNG."

How it works

This tile implements two physical security primitives using a bank of 8 **ring oscillators (ROs)**. No cryptographic algorithm is required, all security properties emerge from manufacturing process variation, which makes each chip unique.

Primitives

Mode	`uio_in[4:3]`	Description
RO-PUF	`00`	4 oscillator pairs compared for a 4-bit device fingerprint in `result[3:0]`
TRNG	`10`	8 measurement rounds; XOR of all 8 counter LSBs → 1 random byte

Ring oscillator architecture

Each of the 8 ROs is modelled as a clock-divided flip-flop chain with a distinct half-period of `(i + 2)` clock cycles (i = 0..7). This provides deterministic, distinct frequencies for simulation and testing. On actual silicon, the physical ring oscillators should exhibit frequency variation caused by gate delay variation introduced during manufacturing.

RO-PUF

All 8 oscillators run for a configurable measurement window (default: 64 clock cycles). The number of rising edges is counted per oscillator in an 8-bit counter. The 4 pairs are compared:

```
PUF bit[i] = 1  if  cnt[2i] > cnt[2i+1]   (i = 0..3)
result[3:0] = PUF bits;  result[7:4] = 0
```

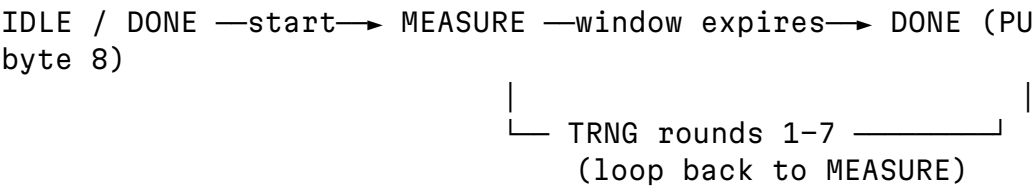
The frequency difference between paired oscillators is caused by gate delay variation introduced during manufacturing.

TRNG

The TRNG accumulates 1 bit per measurement round. After each window, the LSBs of all 8 counters are XORed together. Jitter (thermal noise in the

oscillators) causes the LSB to vary unpredictably between measurements. After 8 rounds, 1 byte of random data is available.

Internal state machine



Interface

State is configured and read one byte at a time through the 8-bit data bus. An internal 5-bit address counter advances automatically on each `we` or `re` pulse.

Signal	Direction	Description
`ui_in[7:0]`	Input	Data byte to write
`uo_out[7:0]`	Output	Data byte to read
`uio_in[0]`	Input	**we** — write `ui_in` to config[addr], addr++
`uio_in[1]`	Input	**re** — output result to `uo_out`, addr++
`uio_in[2]`	Input	**start** — begin measurement (resets addr to 0)
`uio_in[4:3]`	Input	**mode** — `00`=RO-PUF, `10`=TRNG
`uio_in[5]`	Input	**addr_clr** — reset byte address to 0
`uio_out[6]`	Output	**busy** — measurement in progress
`uio_out[7]`	Output	**done** — result ready to read

`uio_oe = 0xC0` — only bits 7 and 6 are driven by the tile.

Config address map (write)

addr	Register	Default
0	`window[7:0]` — measurement duration in clock cycles	64

Result address map (read)

addr	Content
0	`result[7:0]` — 4 PUF bits (bits [3:0]) or TRNG byte

How to test

Minimal sequence — RO-PUF

```
1. rst_n=0 for ≥3 cycles, then rst_n=1.

2. (Optional) Write config:
   ui_in=64, uio_in=0x01 (we=1)  → window = 64 cycles
```

```
   uio_in=0x00

3. Start:
   uio_in = 0x04  (start=1, mode=00=RO-PUF)
   rising edge
   uio_in = 0x00

4. Wait for done:
   Poll uio_out[7].  Asserts after (window + 2) clock cycles.

5. Read result:
   uio_in = 0x20  (addr_clr=1)
   rising edge
   uio_in = 0x02  (re=1)
   rising edge
   Read uo_out  →  4 PUF bits in uo_out[3:0]
   uio_in = 0x00
```

TRNG byte

```
1. Start TRNG:
   uio_in = 0x14  (start=1, mode=10=TRNG)

2. Wait done:
   TRNG runs 8 measurement rounds → done after ~8 × window cycles.

3. Read byte:
   addr_clr, then re → uo_out holds random byte.
```

External hardware

.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	data_in[0]	data_out[0]	we (IN) — write data_in to config register, addr++
1	data_in[1]	data_out[1]	re (IN) — read result to data_out, addr++
2	data_in[2]	data_out[2]	start (IN) — begin measurement
3	data_in[3]	data_out[3]	mode[0] (IN) — mode LSB: 00=RO-PUF, 10=TRNG
4	data_in[4]	data_out[4]	mode[1] (IN) — mode MSB
5	data_in[5]	data_out[5]	addr_clr (IN) — reset byte address to 0
6	data_in[6]	data_out[6]	busy (OUT) — measurement in progress
7	data_in[7]	data_out[7]	done (OUT) — result ready

Glitcher

by **Philip Åkesson**

0238 50 MHz HDL Project

`github.com/pakesson/tt_glitcher`

"Fault injection pulse generator with trigger input and configurable parameters, controllable over UART."

Introduction

This project is a pulse generator with configurable parameters, intended for use in voltage or electromagnetic fault injection attacks.

Fault injection is a hardware attack technique in which a brief disruption to a microcontroller's power supply or electromagnetic environment creates faults (glitches) that can cause it to skip instructions, corrupt computations, or bypass security checks. This can be used to potentially reveal cryptographic keys, bypass secure boot, or unlock otherwise inaccessible functionality.

In a typical voltage fault injection setup, the target microcontroller's power rail is momentarily pulled low (or sometimes spiked high) for a tiny fraction of a second. If timed correctly, this can cause a single instruction to execute incorrectly or not at all.

Similarly, electromagnetic fault injection (EMFI) uses a coil placed near the target IC to induce transient currents in the die.

In practice, the correct glitch parameters are rarely known in advance. The useful fault parameters often have to be found experimentally by sweeping across different delay values, pulse widths, pulse counts, and pulse spacing until the target shows an interesting response.

The pulse generator can be configured and controlled over a standard UART serial connection at 115200 baud, making it easy to drive from a microcontroller, a single-board computer like a Raspberry Pi, or a USB-to-serial adapter.

Pulses can be triggered either by UART commands or by an external trigger input.

The pulse generator runs at 50 MHz, giving a timing resolution of 20 ns. This is sufficient for precise, repeatable glitch placement on a wide range of targets.

Hardware Interface

Inputs

Pin	Signal	Description
`ui[0]`	Trigger In	External hardware trigger. When the trigger is armed, a high signal on this pin starts the pulse sequence.
`ui[1]`	UART RX	Serial input from the host (115200 8N1). Connect to your host's TX pin.

Outputs

Pin	Signal	Description
`uo[0]`	UART TX	Serial output to the host (115200 8N1). Connect to your host's RX pin.
`uo[1]`	Pulse Out	Glitch pulse output, active high.
`uo[2]`	Target Reset	Target power-cycle output, active high. Hold high for the configured reset duration.
`uo[3]`	Pulse EN	Single-cycle strobe, goes high for one clock cycle (20 ns) at the start of each triggered glitch sequence. Useful for oscilloscope triggering.
`uo[4]`	Busy	High whenever the glitcher is executing a sequence (reset, delay, pulse, or spacing). Low only when idle.
`uo[5]`	Armed	High when the hardware trigger input is armed and waiting for a high signal on `ui[0]`.
`uo[6]`	Pulse Out or Target Reset	Active high during both glitch pulse output and target reset.
`uio[0]`	Pulse Out (Inverted)	Inverted version of `uo[1]`. Use with active-low circuits or MOSFET drivers that require an inverted input.
`uio[1]`	Target Reset (Inverted)	Inverted version of `uo[2]`.

External Hardware

The `Pulse Out` (`uo[1]`) output can be used with an N-channel MOSFET or analog multiplexer/switch for voltage fault injection, or connected to a ChipSHOUTER for EMFI.

Use `Pulse Out (Inverted)` (`uio[0]`) when your driver circuit expects an active-low enable signal.

In cases where the pulse output might drive (all or parts of) a target during reset, use the combined `Pulse Out or Target Reset` (`uo[6]`) output, which is high during both pulse generation and target reset.

To use the `Target Reset` (or `Target Reset (Inverted)`), connect it to a suitable reset pin or a power switch for the entire target.

UART Protocol

All communication is at 115200 baud, 8N1 (8 data bits, no parity, 1 stop bit). Commands are single bytes, optionally followed by parameter bytes for configuration values. All multi-byte parameters are big-endian (high byte first).

All parameter values are either 8-bit or 16-bit unsigned integers.

All timing values are specified in clock cycles at 50 MHz, where 1 cycle = 20 ns.

The minimum effective duration for all timing parameters is 1 clock cycle (20 ns). Both 0 and 1 produce a 1-cycle duration; to set a 2-cycle duration, use the value 2, and so on.

The maximum value for 8-bit timing parameters is 5.10 µs, while the 16-bit timing parameters go up to 1.31 ms.

Configuration

Configuration commands only update the stored parameter values.

Command	Byte	Parameters	Default	Description
`d`	`0x64`	2 bytes (16-bit)	`0x0000`	Set delay before first pulse
`w`	`0x77`	1 byte (8-bit)	`0x01`	Set pulse width
`n`	`0x6E`	1 byte (8-bit)	`0x01`	Set number of pulses
`s`	`0x73`	2 bytes (16-bit)	`0x0000`	Set spacing between pulses
`r`	`0x72`	2 bytes (16-bit)	`0x0000`	Set target reset duration

Actions

Command	Byte	Parameters	Default	Description
`t`	`0x74`	none	—	Trigger pulse sequence
`a`	`0x61`	none	—	Toggle arm/disarm
`p`	`0x70`	none	—	Reset (power cycle) target using the configured reset duration.

Reset Modes

The reset mode determines what happens after the target reset command has completed. By default, a pulse sequence is started directly after resetting the target, but it is also possible to set the armed state and wait for a trigger input, or do nothing at all.

Command	Byte	Parameters	Default	Description
`u`	`0x75`	none	—	Reset mode: Pulse (default)
`i`	`0x69`	none	—	Reset mode: Arm
`y`	`0x79`	none	—	Reset mode: None

Other

Command	Byte	Parameters	Default	Description
`h`	`0x68`	none	—	Hello! Returns `Erika`
other	—	—	—	Unknown commands are echoed back over UART

How It Works

Internally, the glitcher is implemented as a small state machine with five main phases: idle, target reset, delay, pulse active, and pulse spacing. The `Busy` (`uo[4]`) output is high whenever the design is not idle, the `Armed` (`uo[5]`) output is high when the external trigger path is waiting for `Trigger In` (`ui[0]`), and `Pulse EN` (`uo[3]`) generates a one-clock strobe at the moment a pulse sequence starts.

When in the target reset state, the `Target Reset` (`uo[2]`) output is high.

When in the pulse active state, the `Pulse Out` (`uo[1]`) output is high.

Note that the trigger input is synchronized internally, so there is an initial delay of two clock cycles before starting the pulse sequence. The pulse sequence then has a minimum of one additional delay cycle before the pulse output goes high, giving a minimum time from trigger input to pulse output of 60-80 ns (three to four clock cycles), depending on the timing of the trigger, when configured with zero additional delay.

The trigger is activated when `Trigger In` (`ui[0]`) is high, not just on a rising edge. This means that if the trigger input is set to a constant high signal, the system will trigger as soon as the synchronized trigger is seen after arming.

Manual Trigger Over UART

In the simplest use case, the host first configures the pulse parameters over UART and then sends the `t` command to start a sequence immediately. The glitcher waits for the configured delay, generates the requested number of

pulses with the configured width and spacing, and then returns to the idle state.

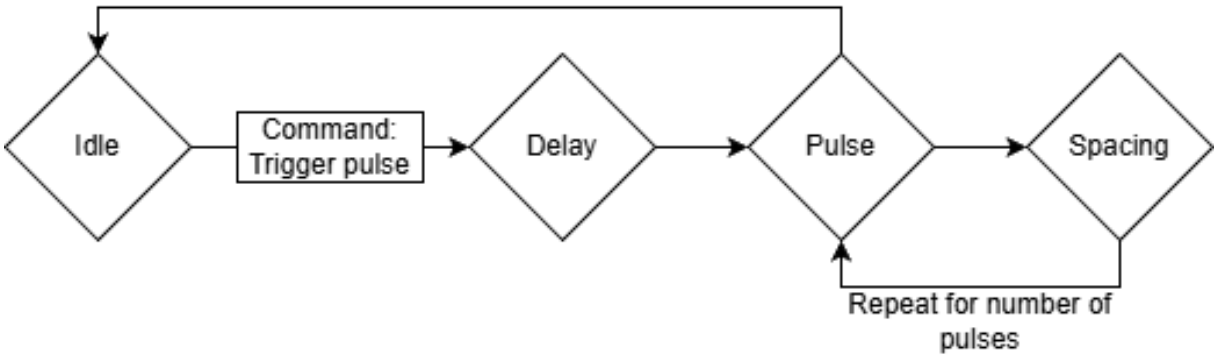


Figure 238.1: Diagram showing manual triggering over UART

Arm Over UART, External Trigger Input

For external trigger inputs, the host sends `a` to arm the trigger logic. The glitcher then waits in the idle state with `Armed` (`uo[5]`) high until `Trigger In` (`ui[0]`) goes high, at which point it starts the same delay-and-pulse sequence as a manual UART trigger. Arming is automatically cleared when the sequence begins, so each arm command corresponds to one trigger event.

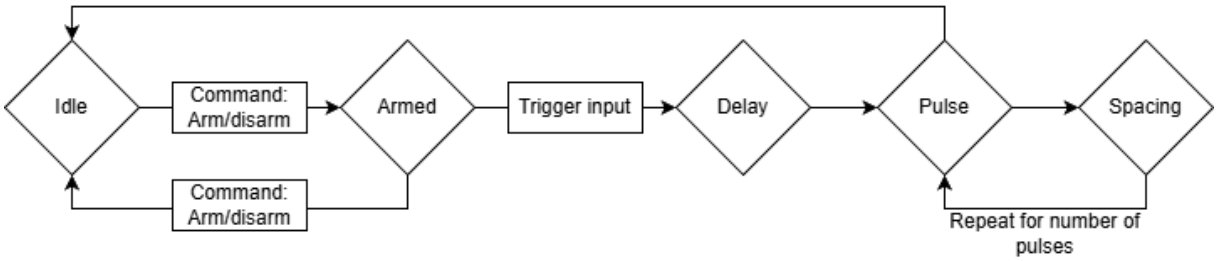


Figure 238.2: Diagram showing arming over UART and then using an external trigger

Reset Mode: None

When reset mode is set to `None`, the `p` command only asserts `Target Reset` (`uo[2]`) for the configured reset duration. After that time has passed, the glitcher returns directly to idle without generating any pulse sequence and without arming the external trigger input.

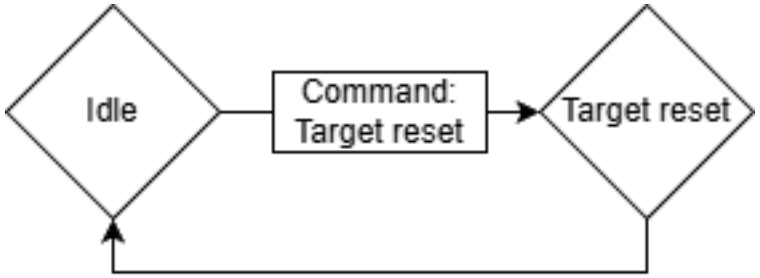


Figure 238.3: Diagram showing the “None” reset mode

Reset Mode: Pulse

When reset mode is set to `Pulse`, the `p` command first resets the target and then automatically starts the configured pulse sequence. After reset is released, the normal pulse delay is still applied before the first pulse, which makes it possible to place the glitch at a controlled offset relative to the end of the reset interval.

This is the default reset mode.

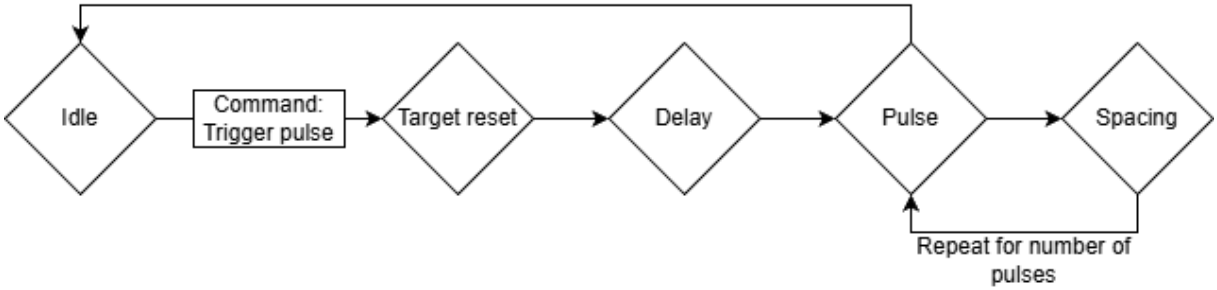


Figure 238.4: Diagram showing the “Pulse” reset mode

Reset Mode: Arm

When reset mode is set to `Arm`, the `p` command resets the target and then returns to idle with the trigger logic armed. This is useful when the target should be reset first, but the actual glitch should not occur until a later external event on `Trigger In` (`ui[0]`).

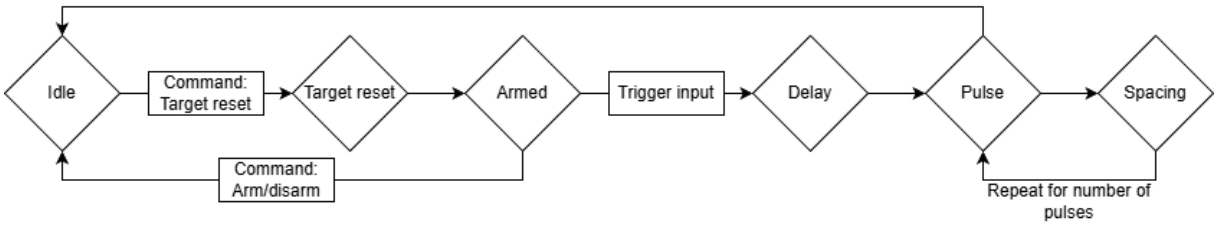


Figure 238.5: Diagram showing the “Arm” reset mode

How to Test

All examples and oscilloscope captures in this section were taken with the project running on a Lattice UP5K FPGA on the Tiny Tapeout FPGA Breakout.

Basic Use Cases

The project can be tested by connecting a microcontroller or USB-to-serial adapter to the UART RX and TX pins (`ui[1]` and `uo[0]`, respectively).

First, test that the UART works by sending `x` (hex byte `78`), which should be echoed back because it is an unknown command, or `h`, which should return the string `Erika`.

To test a basic pulse, set the pulse width to 100 clock cycles (2 µs) with `w\x64` (hex bytes `77 64`) and trigger the pulse with `t` (hex byte `74`). This should result in a pulse on the Pulse Out pin (`uo[1]`).

Full sequence:

```
w 0x64
t
```

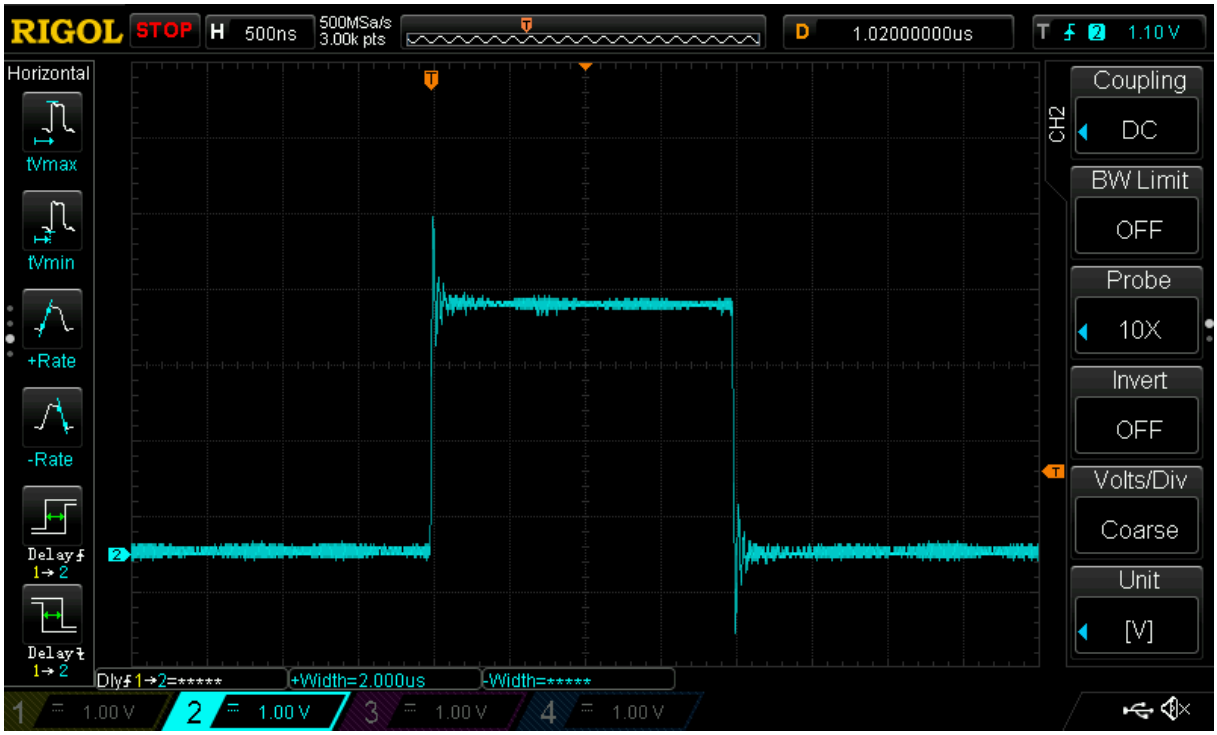


Figure 238.6: Oscilloscope capture of a single pulse, with width set to 100 clock cycles

To test multiple pulses, set the width to 50 clock cycles (1 µs) with `w\x32` (hex bytes `77 32`), spacing to 32 clock cycles (640 ns) with `s\x00\x20` (hex bytes `73 00 20`) and number of pulses to 3 with `n\x03` (hex bytes `6E 03`), then trigger the pulse with `t` (hex byte `74`).

Full sequence:

```
w 0x32
s 0x00 0x20
n 0x03
t
```

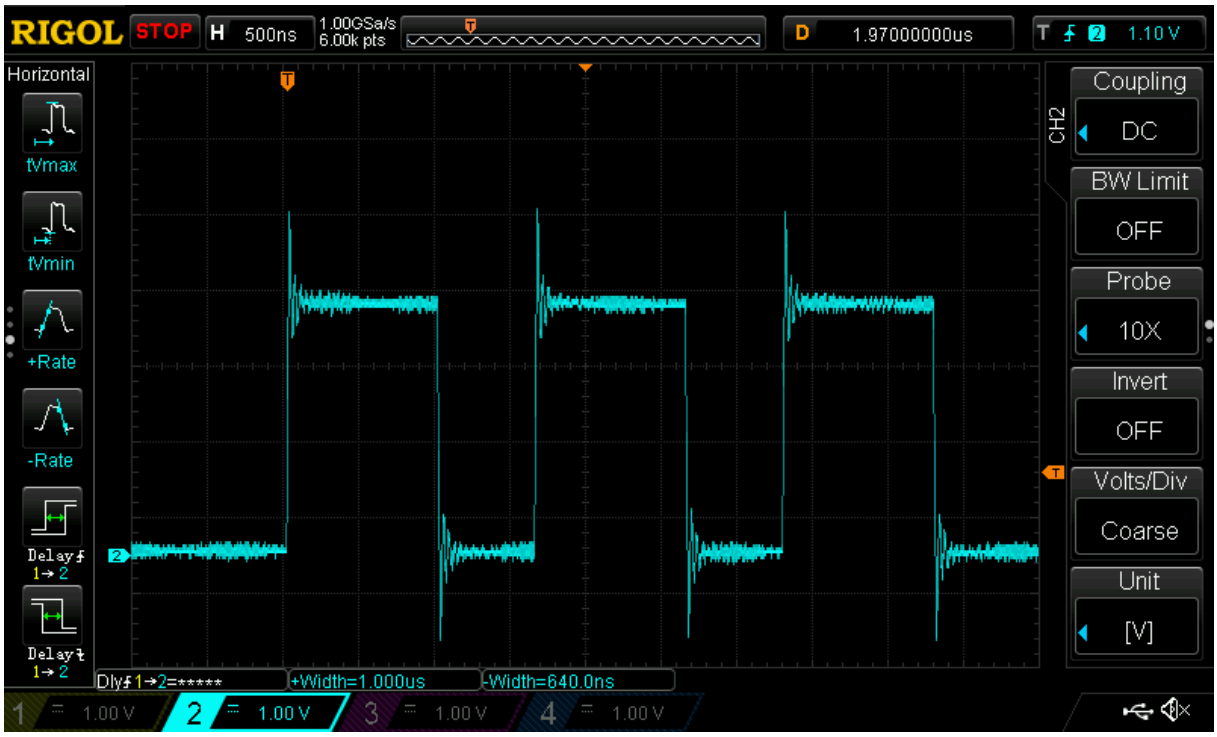


Figure 238.7: Oscilloscope capture of multiple pulses

To test trigger arming, send `a` (hex byte `61`). This should make the armed pin (`uo[5]`) go high. Sending `a` again should disarm the trigger. While armed, setting the trigger input pin (`ui[0]`) to high will trigger a pulse sequence with the configured parameters.

Full sequence:

```
w 0x32
s 0x00 0x20
n 0x03
a
```

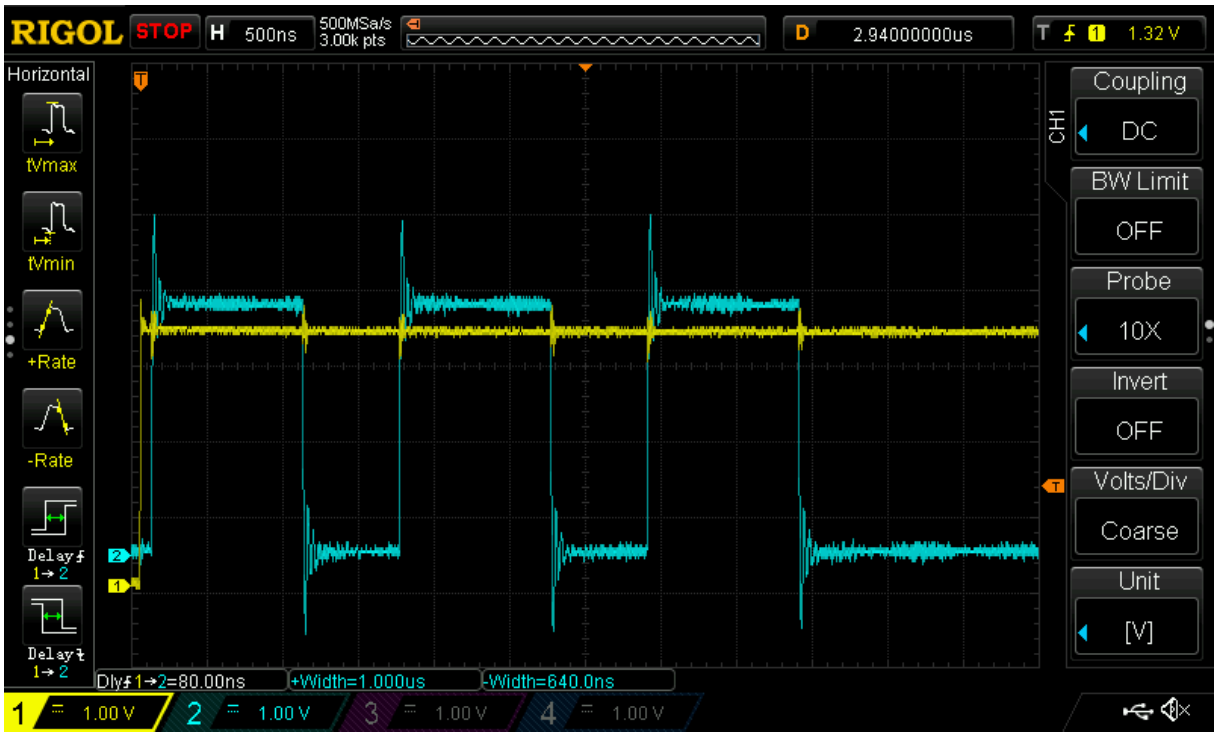


Figure 238.8: Oscilloscope capture of both trigger signal and pulse output with three pulses, pulse width 50 clock cycles and spacing 32 clock cycles.

This can be combined with a delay. Set the delay to 100 clock cycles (2 µs) with `d\x00\x64` (hex bytes `64 00 64`), pulse width to 50 clock cycles (1 µs) with `w\x32` (hex bytes `77 32`), spacing to 32 clock cycles (640 ns) with `s\x00\x20` (hex bytes `73 00 20`) and number of pulses to 3 with `n\x03` (hex bytes `6E 03`). Arm with `a` (hex byte `61`) and then set the trigger input pin (`ui[0]`) to high.

Full sequence:

```
d 0x00 0x64
w 0x32
s 0x00 0x20
n 0x03
a
```

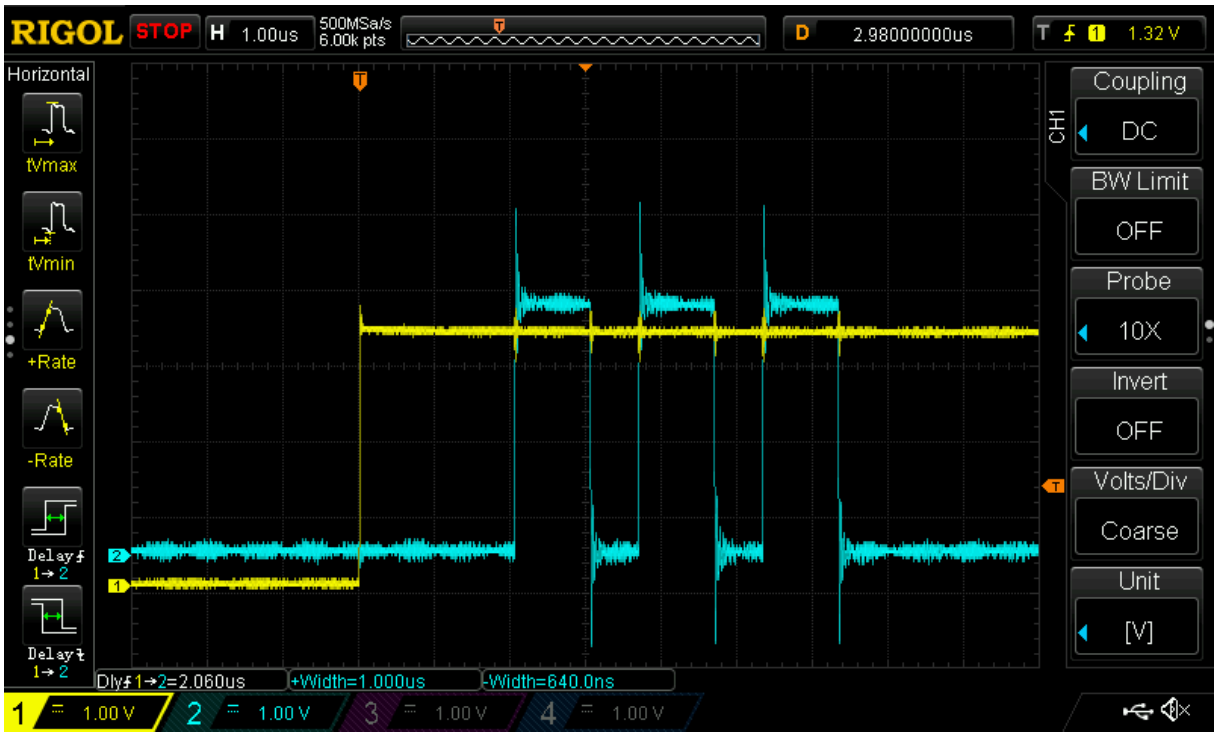


Figure 238.9: Oscilloscope capture of both trigger signal and pulse output with delay 100 cycles, three pulses, pulse width 50 clock cycles and spacing 32 clock cycles.

See the cocotb tests for more examples.

Using with MicroPython on the TT Demo Board

The Tiny Tapeout demo board includes an RP2350 running MicroPython, which can be used to test most of the functionality.

First, set `mode = ASIC_RP_CONTROL` in `config.ini` (or manually in the REPL) to allow the RP2350 to drive the project inputs.

To use UART from the MicroPython REPL, initialize it like this:

```
>>> from machine import UART
>>> uart = UART(0, baudrate=115200, tx=tt.pins.ui_in1.raw_pin,
rx=tt.pins.uo_out0.raw_pin)
>>> _ = uart.read() # Clear read buffer
```

The `h` command can be used to verify that the project is running:

```
>>> uart.write(b'h')
1
>>> uart.read()
b'Erika'
```

Unknown commands are echoed back directly:

```
>>> uart.write(b'x')
1
```

```
>>> uart.read()
b'x'
```

The `Armed` signal can be found on `uo[5]`, and the trigger input is on `ui[0]`. Here is a quick sanity check for these:

```
>>> tt.uo_out[5]      # Check if armed (0 = not armed, 1 = armed)
<Logic ('0')>
>>> uart.write(b'a') # Arm trigger
1
>>> tt.uo_out[5]      # Trigger is now armed
<Logic ('1')>
>>> tt.ui_in[0] = 1   # Set the trigger input
>>> tt.uo_out[5]      # No longer armed
<Logic ('0')>
```

Acknowledgments and Similar Projects

This project had several sources of inspiration, including:

- The "NXP LPC1343 Bootloader Bypass" series of blog posts by Dmitry Nedospasov is where I first saw this type of glitcher implemented in an FPGA.
- The Wallet.fail presentation at 35C3 (watch the presentation on YouTube), by Thomas Roth, Dmitry Nedospasov, and Josh Datko, used a very similar FPGA glitcher.
- ... and so did the Chip.Fail presentation at Black Hat USA 2019, by Thomas Roth and Josh Datko. The code for this can be found on GitHub.
- I attended one of Dmitry's in-person "Hardware hacking with FPGAs" training courses in 2019 as well, which was also a great source of inspiration.

If you are looking for fault injection tooling that works out-of-the-box, also check out the ChipWhisperer by Colin O'Flynn (NewAE Technology) or the Faultier by Thomas Roth (Hextree.io).

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Trigger Input	UART TX	Pulse Out (Inverted)
1	UART RX	Pulse Out	Target Reset (Inverted)
2	—	Target Reset	—
3	—	Pulse EN	—
4	—	Busy	—
5	—	Armed	—

#	Input	Output	Bidirectional
6	—	Pulse Out or Target Reset	—
7	—	—	—

My first Wokwi design

by **PolykarposV**

0239 Wokwi Project

github.com/PolykarposV/tt_workshop

wokwi.com/projects/453674671092670465

"7 segment displays the first letter of my name when the input is set to a secret code"

How it works

Used the template to customize the display of the 7 segment. Activating the correct inputs, makes the 7 segment display the letter 'P'.

How to test

The code is a secret, try to find it out yourself.

External hardware

No external hardware

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	sw1:1a	tttout:EXTOUT0	—
1	sw2:2a	tttout:EXTOUT1	—
2	sw3:3a	tttout:EXTOUT2	—
3	sw4:4a	tttout:EXTOUT3	—
4	sw5:5a	tttout:EXTOUT4	—
5	sw6:6a	tttout:EXTOUT5	—
6	sw7:7a	tttout:EXTOUT6	—
7	sw8:8a	tttout:EXTOUT7	—

8-bit Prime Number Detector

by **Niklas Oberhuber**

`0240` `HDL Project`

`github.com/obrhubr/ttihp-submission`

"Set 8 DIP switches to any number 0-255. The 7-segment display shows its hex digit; the decimal point lights up if the number is prime."

How it works

This is a purely combinational 8-bit prime number detector. It tests every number from 0 to 255 in hardware using a 256-bit lookup table (one bit per number, pre-computed at synthesis time). The result is available with zero latency — no clock required.

The 7-segment display shows the lower hex nibble of the input number (0–F), and the **decimal point lights up when the number is prime**. There are 54 primes in the range 0–255, from 2 up to 251.

The entire design uses 100 logic gates: a 256-bit ROM for the prime lookup and a 16-entry hex decoder for the display.

How to test

1. Set `ui[7:0]` to any number using the DIP switches.
2. Read the lower nibble from the 7-segment display (0–F).
3. **Decimal point lit = prime. Decimal point off = not prime.**

Try scanning through numbers by flipping switches — you can find all 54 primes between 0 and 255by watching the decimal point.

Some interesting numbers to try:

Number	Prime?	Notes
2	yes	Smallest prime; only even prime
7	yes	—
9	no	3 × 3 — a common mistake
127	yes	Mersenne prime ($2^7 - 1$)
128	no	2^7
251	yes	Largest prime ≤ 255
255	no	3 × 5 × 17

External hardware

None required. The 7-segment display is built into the TinyTapeout demo board.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Number bit 0 (LSB)	Segment a	—
1	Number bit 1	Segment b	—
2	Number bit 2	Segment c	—
3	Number bit 3	Segment d	—
4	Number bit 4	Segment e	—
5	Number bit 5	Segment f	—
6	Number bit 6	Segment g	—
7	Number bit 7 (MSB)	Decimal point (1 = prime)	—

VGABlock

by **Enrico Zelioli**

0241 | 25.175 MHz | HDL Project

`github.com/ezelioli/tt-efcl-workshop`

"Draws moving blocks across screen"

How it works

The design outputs moving squared blocks on the VGA output. A green and blue checkerboard pattern is moving towards the top right corner, while larger red blocks are moving to the left.

How to test

Connect to a 640x480 screen and observe the pattern moving as described above.

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	—

True(er) Random Number Generator (TRNG)

by **Angelo Nujic**

0242 | 50 MHz | HDL Project

`github.com/anujic/anujic_ttihp26a`

"Generates a random byte based on bits sampled from ring oscillators."

How it works

This project implements a fully digital True Random Number Generator (TRNG) designed to fit within a single Tiny Tapeout tile. Since analog components are not available in a standard cell digital flow, this design harvests entropy from the phase noise (jitter) of free-running, unconstrained ring oscillators.

The raw, asynchronous bitstream is sampled by the synchronous system clock, whitened to remove systemic `0` or `1` biases, and shifted into a complete 8-bit word.

Architecture

IO: Top Level Interface

The module communicates using a standard valid/ready handshake protocol to ensure the receiving system only reads fully assembled, valid random bytes.

```
    input  logic [7:0] ui_in,    // ui_in[0] is used for ready_i
signal, rest is UNUSED
    output logic [7:0] uo_out,   // 8-bit Random Byte payload
    input  logic [7:0] uio_in,   // UNUSED
    output logic [7:0] uio_out,  // uio_out[0] is used for valid_o
signal, rest is UNUSED
    output logic [7:0] uio_oe,   // uio_oe[0] is 1'b1, rest is
UNUSED
    input  logic       ena,      // always 1 when the design is
powered, so you can ignore it
    input  logic       clk,      // clock
    input  logic       rst_n     // reset_n - low to reset
```

Ring Oscillator (Entropy Source)

To guarantee entropy, the source consists of multiple mutually prime length ring oscillators (e.g., lengths of 3, 5, and 7).

Each ring oscillator is built with `2*DEPTH + 1` standard IHP 130nm (`sg13g2_inv_1`) inverter cells. These standard cells are manually instantiated with `(* keep = "true" *)` attributes to prevent the synthesis tool (Yosys) from optimizing away the combinatorial loops.

The asynchronous outputs from these rings are XOR'd together into a single raw bitstream which is captured by D-Flip-Flops clocked by `clk_i`.

Von Neumann Whitener

Raw ring oscillators often exhibit a slight bias (e.g., naturally preferring `1`s over `0`s due to microscopic process variations). The Von Neumann extractor eliminates this bias by reading the raw bits in non-overlapping pairs.

Extraction Logic: The whitener generates a bit according to the following truth table. Discarded bits yield no output (`valid_o` remains LOW).

Bit 1 (Cycle N)	Bit 2 (Cycle N+1)	`bit_o`	`valid_o`	Action
0	0	0	0	**Discard** (No entropy)
0	1	1	1	**Keep 1**
1	0	0	1	**Keep 0**
1	1	0	0	**Discard** (No entropy)

Watchdog Timer (Failsafe)

Because the Von Neumann extractor drops `00` and `11` pairs, there is a statistical possibility (especially if the oscillators lock up) that the process suffers from "entropy starvation" and fails to converge on a full 8-bit byte in a reasonable timeframe.

To prevent the system from hanging indefinitely:

- A Watchdog Timer increments on every `clk_i` cycle.
- It resets to `0` every time the whitener successfully outputs a `valid` bit.
- **Timeout:** If the counter reaches **1024 clock cycles** without seeing a valid bit, it triggers a timeout.
- **Reset Mechanism:** Upon timeout, the watchdog pulls an internal soft-reset line. This flushes the whitener's state machine and clears the current shift register progress, restarting the byte generation process from scratch to recover from the stall.

How to test

Testing the physical silicon requires observing the handshake protocol.

1. Assert `rst_ni` LOW, then HIGH to reset the module.
2. Assert `ena_i` HIGH to enable the oscillators and sampling logic.

3. Assert `ready_i` HIGH from your receiving device to indicate you are ready to receive data.
4. Wait for `valid_o` to go HIGH.
5. On the clock cycle where `valid_o` is HIGH, read the 8-bit value on `byte_o`. This is your true random byte.
6. To verify randomness, capture several megabytes of output data and process it using a statistical test suite like **NIST SP 800-22** or **Dieharder**.

Design Verification (DV) Plan

Pre-silicon verification is handled via a Python-based Cocotb testbench. Because digital simulators cannot natively process analog phase noise, the RTL leverages a \ifdef SIM`block to model the oscillators using fractional#` delays.

The DV suite verifies the digital plumbing using two main tests:

1. **Continuous Generation Check:** The testbench drives the `ready_i` signal HIGH and loops 100 times, waiting for the `valid_o` edge to capture sequential random bytes and confirming the handshake can operate continuously.
2. **Statistical Sanity Checks (Variance):** Validates the behavioral entropy by checking for output uniqueness. Out of the 100 sampled bytes, the test ensures that more than 5 unique values are generated, proving the simulated oscillators are drifting and the FSM is not stuck emitting a constant value.

Gate-Level Simulation (GLS) Limitations

Note: Simulating this design at the gate level (GLS) presents a fundamental EDA challenge. Synthesis tools generate a physical netlist for the combinatorial inverter loops without an initial value. Digital simulators evaluate this unknown initial state as `X`.

Because a pure combinatorial loop has no reset pin, the `X` state remains permanently locked, propagating through the synchronizer flip-flops and crashing the Von Neumann logic. Consequently, our Cocotb testbench actively monitors the output bus for unresolvable `X` states. If an `X` state is detected (indicating a GLS environment where the oscillator failed to initialize), the testbench logs the limitation and gracefully skips the remainder of the simulation to prevent CI pipeline failures.

External hardware

- **Tiny Tapeout Demo Board:** To provide the system clock, power, and physical pin breakouts.

- **RP2040 Microcontroller (built into the demo board):** Required to interface with the `valid`/`ready` handshake protocol and quickly stream the generated random bytes over USB to a host PC for statistical validation.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	ready_i	bit0_o	valid_o
1	—	bit1_o	—
2	—	bit2_o	—
3	—	bit3_o	—
4	—	bit4_o	—
5	—	bit5_o	—
6	—	bit6_o	—
7	—	bit7_o	—

Second TT experiment

by **Mddde**

0243 Wokwi Project

github.com/fschh/tt-experiment

wokwi.com/projects/455292199922428929

"Simple project"

How it works

Simple 7 Segment display

How to test

Click reset button

External hardware

7 segment display

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	sw0	dis0	—
1	sw1	dis1	—
2	sw2	dis2	—
3	sw3	dis3	—
4	sw4	dis4	—
5	sw5	dis5	—
6	sw6	dis6	—
7	sw7	dis7	—

Chisel Async Test

by **Tjark Petersen**

320 | 5 MHz | HDL Project

`github.com/tjarker/ttihp-feb-2026`

"*Testing several asynchronous primitives*"

How it works

C Elements

The tile contains three attempts of implementing Müller C-elements using standard cells. The C-element is a primitive for asynchronous circuits that synchronizes two input transitions. It outputs a '1' only when *both* inputs have transitioned to '1', and outputs a '0' only when *both* inputs have transitioned back to '0' again. If the inputs are different, the output retains its previous state. The three implementations are:

1. `c_gate_ao`: uses aoi primitive with feedback
2. `c_gate_mux`: uses a 2:1 mux with feedback
3. `c_gate_sr_nor_latch`: uses an SR latch with NOR gates

All C-elements share the same input signals `a` and `b`, and their outputs are `c_gate_ao`, `c_gate_mux`, and `c_gate_sr_nor_latch` respectively.

C Element Ring Oscillators

To test the speed of the C-elements, rings with 128 stages are implemented for each type. The `c_ring_en` signals enables the oscillation of the rings.

Arbiter Test

Asynchronous arbitration is difficult, since no relation is assumed between the request signals and metastability can occur. A SR-latch can serve as an arbiter, but it is not guaranteed to be metastability-free. [1] suggests using 4-input NOR gates to create metastability filters after a NAND-based SR-latch. To test this setup, two requests are registeded into flip-flops during a rising edge of `launch_arbiter` (clocked by clk). The outputs of the metastability filters are checked for being both high. If they both are high, a seconds SR-latch is set to output `arbiter_test_bad` as '1', indicating a failure of the arbiter. The second SR-latch is reset when a new arbitration is launched.

[1] Y. Zhang et al., "Design and Analysis of Testable Mutual Exclusion Elements," in 2015 21st IEEE International Symposium on Asynchronous Circuits and Systems, May 2015, pp. 124–131. doi: 10.1109/ASYNC.2015.28.

Mouse Trap FIFO

A mousetrap pipeline implements asynchronous 2-phase handshakes using latches and xor gates. A transition of `req` signals a new request and a transition of `ack` signals an acknowledgment. Combining multiple stages creates a FIFO buffer. The FIFO is 4 stages deep, and exposes the input and output handshakes. The data is only a single bit.

How to test

C Elements

The C-elements can be tested by applying all combinations of input transitions and checking the output. For example we can apply the following sequence of inputs:

- `a=0, b=0` (initial state)
- `a=1, b=0` (output should remain 0)
- `a=1, b=1` (output should transition to 1)
- `a=0, b=1` (output should remain 1)
- `a=0, b=0` (output should transition back to 0)

C Element Ring Oscillators

The ring oscillators can be tested by enabling the ring and checking for oscillation on the outputs `uio_out[2:0]`.

Arbiter Test

The arbiter can be brought into a known state using:

- `launch_arbiter=0` (initial state)
- `a=1, b=0` (set either a or b)
- `launch_arbiter=1` (apply testing stimulus)
- `launch_arbiter=0` (finish launching)

Checking for metastability can be done by applying requests on both inputs and checking if `arbiter_test_bad` is '1'.

- `a=1, b=1` (apply requests on both inputs)
- `launch_arbiter=1` (apply testing stimulus)
- `launch_arbiter=0` (finish launching)
- Check if `arbiter_test_bad` is '1' (indicates a failure of the arbiter)

Mouse Trap FIFO

The FIFO can be tested by applying a sequence of requests and checking the corresponding acknowledgments. For end-to-end testing we can insert data and expect it on the output:

- `fifo_in_req=0` (initial state)
- `fifo_in_data=?` (set data)

- `fifo_in_req=1` (toggle request)
- `await fifo_out_req=1` (wait for output request)
- check if `fifo_out_data=?` (check if output data matches input)

The input and output interfaces can also be driven independently to test the throughput of the FIFO. One driver should apply eagerly insertions while the other driver should apply eager removals. For example, for the input side:

- `toggle(fifo_in_data)` (set new data)
- `toggle(fifo_in_req)` (toggle request)
- `awaitToggle(fifo_in_req)` (wait for acknowledgment)
- repeat (apply more insertions)

On the output side we expect that the data output toggles between every request:

- `awaitToggle(fifo_out_req)` (wait for output request)
- check if `fifo_out_data` toggled
- `toggle(fifo_in_ack)` (toggle acknowledgment)
- repeat (apply more removals)

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	a	c_gate_ao	c_ring_ao
1	b	c_gate_mux	c_ring_mux
2	launch_arbiter	c_gate_sr_nor_latch	c_ring_latch
3	fifo_in_req	arbiter_test_bad	uio3
4	fifo_in_data	fifo_out_data	uio4
5	fifo_out_ack	fifo_out_req	uio5
6	c_ring_en	fifo_in_ack	uio6
7	ui7	uo7	uio7

O2ELHd 7segment display

by **OzelHD**

322 | 1 Hz | Wokwi Project

`github.com/OzelHD/wokwi_test`

`wokwi.com/projects/458140717611045889`

"Uses the 7 segment display to write my name"

How it works

Using a clock and a binary counter, this design counts from 0 to 6 and then resets back to 0. For every value of the counter, a different letter is displayed on the 7-segment display. The mapping from counter value to letter is as follows:

- 0: O
- 1: 2
- 2: E
- 3: L
- 4: H
- 5: d
- 6: (blank)

How to test

You can manually test the inputs using the switch:

SW	Function
1	Reset
2	a
3	b
4	c
5	N/A
6	N/A
7	N/A
8	N/A

Set the clock to 1 Hz and observe the display. You should see the letters O, 2, E, L, H, d, and blank as you increment the counter from 0 to 6.

External hardware

LED display (7-segment) button, clock, and reset button.

Project Pinout

Digital Pins

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

Tschai's Tic-Tac-Toe

by **tschai-yim**

`324` `HDL Project`

`github.com/tschai-yim/ttihp-mill`

"The classic Tic-Tac-Toe game implemented in ASIC for maximum performance!"

How it works

This project is a hardware-accelerated, two-player Tic-Tac-Toe implementation. The design is optimized for high throughput and performance to meet the extreme demands of the legendary three by three board game (some might even say it's NP-complete).

Visual States:

- **IDLE:** A single LED sequence rotates around the 3x3 board.
- **Player 1:** Solid ON.
- **Player 2:** Slow Pulsing (50% duty cycle).
- **Win:** The three LEDs forming the winning line flash rapidly.
- **Draw:** If the board is full with no winner, the Player 2 LEDs flash rapidly (Sparkle pattern).
- **Error:** If an occupied cell is selected, that LED strobes rapidly for one second.

How to test

1. **Power On:** Upon reset, the board enters `IDLE` mode. You will see a single LED "rotating" around the 3x3 grid.
2. **Start Game:** Player 1 presses any of the 9 buttons. Their mark is placed immediately, and the game transitions to Player 2's turn.
3. **Gameplay:**
   - Players take turns pressing buttons to occupy empty cells.
   - If you press an already-occupied cell, that cell will strobe for one second to indicate an error, and the turn remains with the current player.
4. **Winning:** Once a line of three is formed, the game enters `GAME_OVER`. The winning line will flash rapidly.
5. **Reset:** Press any button during the `GAME_OVER` state to clear the board and return to the `IDLE` rotation animation.

External hardware

The project requires 9 momentary pushbuttons and 9 LEDs arranged in a 3x3 grid.

Pinout Mapping:

Component	ASIC Pin(s)	Description
Buttons 0-7	`ui_in[7:0]`	Inputs for the first 8 cells (Top-Left to Bottom-Middle).
Button 8	`uio_in[0]`	Input for the 9th cell (Bottom-Right).
LEDs 0-7	`uo_out[7:0]`	Outputs for the first 8 LEDs.
LED 8	`uio_out[7]`	Output for the 9th LED.

Wiring Requirements:

- **Pull-down Resistors:** Each input pin (`ui_in` and `uio_in[0]`) should have a 10kΩ pull-down resistor to Ground. The buttons should connect the pin to VCC (3.3V) when pressed.
- **Current Limiting:** Each LED output must have a current-limiting resistor (recommended 330Ω to 470Ω) to stay within the safe 4mA drive strength of the ASIC I/O pads.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Button Top Left	LED Top Left	Button Bottom Right
1	Button Top Center	LED Top Center	—
2	Button Top Right	LED Top Right	—
3	Button Middle Left	LED Middle Left	—
4	Button Middle Center	LED Middle Center	—
5	Button Middle Right	LED Middle Right	—
6	Button Bottom Left	LED Bottom Left	—
7	Button Bottom Center	LED Bottom Center	LED Bottom Right

microlane demo project

by **htfab**

`326` `5 Hz` `HDL Project`

`github.com/htfab/ttihp-microlane-demo`

"Scrolls a message on the 7-segment display. Hardened using microlane."

How it works

Scrolls the text "hardened using python" over the 7-segment display. This is a demo project for the microlane flow.

How to test

Select the project and provide a slow clock (say, 5 Hz).

External hardware

None

Project Pinout

Digital Pins

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

tiny_dino

by **RongGi**

`328` `25.175 MHz` `HDL Project`

`github.com/RongGi/tiny_dino`

> *"a vga project, trying to simulate the dino game, you cannot lose"*

How it works

using vga color scheme to build a oversimplified Dino game

How to test

connect a button to pin 0 and vga to output and let the dino jump over the obstracle

External hardware

Display, same as template (vga connection) Button with power source

Project Pinout

Digital Pins

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

Infinity Core

by **FilteredNoise**

`0330` `50 MHz` `HDL Project`

`github.com/FilteredNoise/tt-infinity-core`

"*Audio visualizer and algorithmic pattern generator*"

How it works

The Infinity Core is a hardware-native audio visualizer. It processes a digital audio trigger pulse through a custom ASIC architecture comprising:

1. **Heartbeat Engine:** A master 24-bit counter that derives all timing signals for PWM and SPI.
2. **PWM Engine:** A digital comparator that drives the infinity mirror LED brightness.
3. **SPI Shift Register:** A hardware-native serializer that pushes graphical pixel data directly to an OLED display without software, RAM, or a CPU.

How to test

1. Apply 3.3V/5V power to the breakout board and connect the SPI OLED display to the designated SPI pins (SCK, MOSI, DC).
2. Feed a digital "beat" pulse into `ui[0]`.
3. Observe the SPI OLED display rendering geometric patterns and the infinity mirror LEDs reacting to the trigger pulse.

External hardware

- **SPI OLED Display:** 128x64 pixels (SSD1306-compatible)
- **Rotary Encoders:** Two encoders connected to `ui[1..4]` to adjust decay and pattern variables.
- **Potentiometer:** Analog sensitivity control (fed through an external comparator circuit).
- **LEDs:** High-brightness LEDs for the infinity tunnel, driven by `uo[0]`.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
`0`	Audio Trigger In (Digital Pulse)	LED PWM Out	—

#	Input	Output	Bidirectional
1	Encoder 1 Pin A (Brightness/ Density)	SPI Clock (SCK)	—
2	Encoder 1 Pin B (Brightness/ Density)	SPI Data (MOSI)	—
3	Encoder 2 Pin A (Decay Speed)	SPI Data/Command (DC)	—
4	Encoder 2 Pin B (Decay Speed)	SPI Chip Select (CS)	—
5	—	Debug Heartbeat Blink	—
6	—	—	—
7	—	—	—

4-bit processor

by **Alessio**

332 HDL Project

github.com/alessio8132/IHP26a-4-bit-processor

"*Basic 4 bit processor*"

How it works

This is a 4 bit CPU with very basic opcodes for running programs. The input on ui_in is of the form 4 bits opcode - 4 bits operand. The output uo_out is of the form 4 bits program counter - 4 bits value. All instructions are performed on the 4-bit accumulator, which can then be sent to the output (OUT) or stored into memory (STORE). The CPU also has 16 words of RAM on the chip. It is intended to use an Arduino or some other microcontroller as the program storage, to be able to run different programs on the chip. A list of all opcodes:

- Opcode 0001: LDA (Load operand into Accumulator)
- Opcode 0010: ADD (Add value at operand memory address to Accumulator)
- Opcode 0011: SUB (Subtract value at operand memory address from Accumulator)
- Opcode 0100: JZ (Jump to program counter value of operator if accumulator is Zero)
- Opcode 1100: JMP (Jump to program counter value of operator)
- Opcode 0110: SHL (Shift Left - Multiply value in accumulator by 2)
- Opcode 0111: XOR (Exclusive OR, compared bitwise against value in the operand memory address)
- Opcode 1011: AND (And, compared bitwise against value in the operand memory address)
- Opcode 1101: OR (Or, compared bitwise against value in the operand memory address)
- Opcode 1000: OUT (Write accumulator to output register)
- Opcode 0101: STORE (Write accumulaor to memory at operand memory address)
- Opcode 1010: LOAD (Load value at operand memory address into accumulator)

How to test

Since I'm not entirely sure how the testing will actually work, and what has to be done to ensure sucess, I can currently only give rough instructions. Once I'm able to test the design myself though, I will of course update this section. The Idea is to write a little Program using the opcodes provided and the RAM on the chip to write a little program. One could even code up a small assembler to write programs for the CPU. This program can then be fed to the CPU by a microcontroller, which also has access to the process counter, in order to send the correct instructions through ui_in. To ensure correct timing, it may be necessary to have the clock signal generated by the Arduino. I will put my arduino program(s) on my GitHub as well.

External hardware

- Arduino or other microcontroller

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	operand[0]	out_reg[0]	—
1	operand[1]	out_reg[1]	—
2	operand[2]	out_reg[2]	—
3	operand[3]	out_reg[3]	—
4	opcode[0]	pc[0]	—
5	opcode[1]	pc[1]	—
6	opcode[2]	pc[2]	—
7	opcode[3]	pc[3]	—

UART interfaced 8ch PWM controller

by **Andrei Tudoroi**

0334 12 MHz HDL Project

github.com/ADDTDR/tt-pwm-controller

"UART interfaced 8ch PWM controller"

How it works

8 chanel 8bit (0-254) pwm controller uart control interface

How to test

send 8 values 0 to 254 write to memory send 0xff, 255

External hardware

LED pmod or 7 segment pmod

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	PWM channel 1	—
1	—	PWM channel 2	—
2	—	PWM channel 3	—
3	—	PWM channel 4	UART input
4	—	PWM channel 5	—
5	—	PWM channel 6	—
6	—	PWM channel 7	—
7	—	PWM channel 8	—

UART-ALU Processor

by **Lukas Kämpf**

`0336` `50 MHz` `HDL Project`

`github.com/lukkaempf/tt_um_uart_alu`

"8-bit ALU controlled via UART – send opcode + two operands, get the result back"

How it works

This design is a small UART-driven ALU processor. It receives three bytes over UART (opcode, operand A, operand B), computes the result with an 8-bit ALU, and sends the result byte back over UART.

- **Protocol:** 8N1 (idle=1, start=0, 8 data bits LSB first, stop=1). Default baud rate is 115200 at 50 MHz clock.
- **Frame:** Send exactly 3 bytes: Byte 1 = opcode (0x00–0x07), Byte 2 = A, Byte 3 = B. The chip responds with 1 byte (the 8-bit result).

Opcodes

Opcode	Name	Operation (8-bit)
0x00	NOP	0
0x01	ADD	A + B
0x02	SUB	A - B
0x03	AND	A & B
0x04	OR	A | B
0x05	XOR	A ^ B
0x06	SHL	A << B[2:0]
0x07	SHR	A >> B[2:0]

How to test

1. Connect UART: **RX** = chip receive (your host TX), **TX** = chip transmit (your host RX). Set RP2040 clock to 50 MHz and UART adapter to 115200 baud 8N1.
2. Send three bytes: opcode, A, B (e.g. `0x01 0x14 0x1E` for ADD 20 + 30).
3. Read one byte from the chip; that is the result (e.g. 50 for the example above).

Example (hex): send `01 14 1E` → expect `32` (50).

Simulation: The full test (`make test-full`) uses a synchronous UART sender in the testbench (`uart_tb_sender.v`) with overridden parameters (100 kHz clock, 4800 baud) so RX is driven with exact 16 cycles/bit on the same clock as the DUT.

External hardware

- UART adapter (e.g. USB–serial at 115200 8N1, 3.3 V) connected to uio[0] (RX) and uio[1] (TX).

Project Pinout

Digital Pins

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

A fully functional ALU (Arithmetic logic unit)

by **Tobias Rickenbacher**

338 | 100 kHz | HDL Project

github.com/RK4444/tt_ALU_t_rick_ihp_ver

> *"A fully functional 4 bit ALU which implements all the usual operations (add, subtract, and, or, xor, ones complement, twos complement, logic and arithmetic shifts in both directions, rotate and rotate through carry in both directions)."*

How it works

The operations are applied to the input ports A and B (port A is the lower half byte of the digital inputs and port B is the upper half byte) and output to the lower half byte of the digital outputs. On the upper half byte of the outputs, the mini status register is found, with the usual flags (carry, zero, negative and overflow). The bidirectional pins are all declared as inputs, with the first bit for the subtract signal and the following four bits for the operation selection. If an unary operation (e.g. the ones complement) is performed, port A is usually used. Only exception is the twos complement, where port B is used. The output mux selects the operations in the following order, starting from 0: Add and subtract (depends on the subtract mode), and, or, xor, ones complement, shift left (same for arithmetic and logic shift), shift right arithmetic, shift right logic, rotate left, rotate right, rotate left through carry and finally rotate right through carry.

How to test

The digital inputs are divided in a lower and an upper half byte. Set the lower half byte to some number and do so with the upper half byte. Select an operation with the operation selection mux (bits 1 to 4 of the bidirectional pins). Choose the operation from the list in the previous section. If nothing is selected, the add/subtract mode is selected by default. The result of the operation is displayed at the lower half byte of the digital outputs. The upper half byte holds the flags. Be aware that most of the operations don't utilize them. They are only used in the add/subtract mode and the rotate through carry mode. When in add/subtract mode, the pin 0 of the bidirectional pins indicates whether to perform addition (low) or subtraction (high). To get the twos complement, use subtraction mode and subtract form 0 (port A is set to 0).

External hardware

Not used.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Port A bit 0	result of the operation bit 0	subtract mode on
1	Port A bit 1	result of the operation bit 1	output mux bit 0
2	Port A bit 2	result of the operation bit 2	output mux bit 1
3	Port A bit 3	result of the operation bit 3	output mux bit 2
4	Port B bit 0	Carry flag	output mux bit 3
5	Port B bit 1	Zero flag	—
6	Port B bit 2	Negative flag	—
7	Port B bit 3	Overflow flag	—

Code Lock

by **Jesper Kristensen**

`0353` `50 MHz` `HDL Project`

`github.com/jmkr-ece-git/tt_hdl_ihp26a`

"*A simple code lock implementation*"

How it works

A simple exercise with a code lock application using statemachines. Codes are hardcoded, in code_lock_fsm. The application allows three attempts, after which it locks until reset

How to test

First code is setup on inputs, and confirmed via enter button, the second code is setup and confirmed. Upon correct code, the lock output goes low.

External hardware

Preferebly buttons and leds, but digilent analog discovery can also be used

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	code button 0	—	—
1	code button 1	—	—
2	code button 2	—	—
3	code button 3	open lock output	—
4	enter button	debug output0	—
5	—	debug output1	—
6	—	debug output2	—
7	—	—	—

Bday Candle Chip

by **Bianca Anita Ceolin**

0355 Wokwi Project

github.com/biancaceolin/Bday-Chip

wokwi.com/projects/459303685175910401

"Lights up birthday candle for Linus"

How it works

I check if the input number equals to 2904: I use 8 ui and 8 uio to represent 4 BCD. if it's a match, the LED birthday candle lights up! ## How to test it D3: 0010 D2: 1001 D2: 0000 D1: 0100. You can also try to input an invalid number and observe it doesnt work.

External hardware

LED

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	D2d	MATCH_2904	D0d
1	D2c	—	D0c
2	D2b	—	D0b
3	D2a	—	D0a
4	D3d	—	D1d
5	D3c	—	D1c
6	D3b	—	D1b
7	D3a	—	D1a

1-4 Counter

by **Melih**

0357 Wokwi Project

github.com/mlhktp/ttihp-wokwi

wokwi.com/projects/455293203542942721

"1-4 Counter"

How it works

I have created a simple clock divider. The output, and the negated output of the clock divider is connected to two different flip flop. With two different flip flops, we can generate four different states. The output of the flip flops is connected to a combinational logic circuit, which generates the number from 1 to 4.

How to test

Press the simulation button.

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	—	output 0	—
1	—	output 1	—
2	—	output 2	—
3	—	output 3	—
4	—	output 4	—
5	—	output 5	—
6	—	output 6	—
7	—	output 7	—

2 Bit Adder

by **Zeynep Demirdag**

`#359` `Wokwi Project`

`github.com/zeynepdemirdag/tiny-tapeout-submission`

`wokwi.com/projects/454935456504261633`

"2 Bit Addition Operation"

How it works

This is a simple 2 bit adder circuit. AND, OR and XOR gates are used in this circuit.

How to test

This circuit implements a two-bit adder, where each input has one bit.

`A + B + Cin = S with Cout`

Cin	A	B	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

External hardware

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input 0	output 0	—
1	input 1	output 1	—
2	input 2	output 2	—

#	Input	Output	Bidirectional
3	input 3	output 3	—
4	input 4	output 4	—
5	input 5	output 5	—
6	input 6	output 6	—
7	input 7	output 7	—

74LS138

by **Tianrui**

0361 10 kHz Wokwi Project

github.com/terrywtr/Tianrui-Wang

wokwi.com/projects/455291660779120641

"try to build a 3-8 Decoder"

How it works

NOT,XOR gates are connected with a,b,c,d e,f,g Explain how your project works

How to test

set the clock into 10khz Explain how to use your project

External hardware

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input a	—	—
1	input b	—	—
2	input c	output not	—
3	input d	output not	—
4	—	output nand	—
5	input e	output nand	—
6	—	output nand	—
7	input f	—	—

Mein Hund Gniesbert

by **Gniesbert**

`0363` `Wokwi Project`

`github.com/sisarikaya/mytiny`

`wokwi.com/projects/455291682546516993`

"Adder that only work for number 3"

How it works

Simple Adder. Only works for number 3. Displays all possible sums of the number 3

How to test

Use Input 1, 2 and 3 for all possible summations of 3

External hardware

7 Segment display

Project Pinout

Digital Pins

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

Cyber EMBEDDEDINN

by **Vysakh P Pillai**

364 | 25.175 MHz | HDL Project

`github.com/vppillai/tt-vga-submission`

"VGA Bouncing Text with Generative Font."

A single-tile VGA design for Tiny Tapeout featuring bouncing "EMBEDDEDINN" text with a parallax starfield background.

Key Features: Procedural font generation, 640x480 @ 60Hz VGA output, animated starfield.

Figure 364.1: VGA Preview

What it does

This project generates a 640x480 @ 60Hz VGA output displaying animated bouncing text with a dynamic starfield background. The design uses procedurally generated fonts (no ROM required) and creates a retro-style visual effect perfect for demonstrations.

How it works

VGA Timing Generator

The design implements standard VGA 640x480 @ 60Hz timing using a state machine that tracks horizontal and vertical pixel positions. It generates HSYNC and VSYNC signals according to the VGA specification.

Generative Font Engine

Characters are generated using combinational logic that defines primitive shapes (bars, corners) and combines them to form letters. Each character occupies a 32x40 pixel slot, and the text “EMBEDDEDINN” is rendered by selecting appropriate shape primitives based on the current character index.

Animation System

A frame counter synchronized to VSYNC drives the animation. The text position (`tx`, `ty`) updates each frame based on a selectable speed. The text bounces off screen boundaries, creating smooth, stable motion.

Interactive Controls

The design features interactive controls via the `ui_in` pins:

- **Speed Control (`ui_in[1:0]`)**:
  - `00`: Normal speed
  - `01`: Fast speed (2x)
  - `10`: Slow speed (0.5x)
  - `11`: Pause animation
- **Color Palettes (`ui_in[3:2]`)**:
  - `00`: Classic (Deep Blue/Purple)
  - `01`: Cyberpunk (Neon Pink/Cyan)
  - `10`: Forest (Green/Emerald)
  - `11`: Monochrome (Grayscale)
- **Scanline Toggle (`ui_in[4]`)**: Toggle the retro scanline effect (OFF when pin is high).

Parallax Starfield

The background features a two-layer parallax starfield created using XOR patterns that change with the frame counter, giving a sense of depth and motion.

Color Mixing

The design outputs 2-bit color (4 levels) for red, green, and blue channels. Text and background colors are adjustable via the interactive controls.

How to test

Connect the TinyVGA PMOD to the output pins and a VGA monitor. The design will automatically start displaying bouncing text.

- **Expected output**: Bouncing "EMBEDDEDINN" text with parallax starfield
- **Controls**: Toggle `ui_in` pins to change speed, colors, and scanlines.
- **Timing**: 640x480 @ 60Hz (25.175 MHz pixel clock)
- **Inputs**: `ui_in[1:0]` speed, `ui_in[3:2]` palette, `ui_in[4]` scanline toggle

The cocotb tests verify:

- VGA timing compliance (HSYNC/VSYNC periods)
- Proper frame generation
- Color output during active video regions

External hardware

- **TinyVGA PMOD** (mole99/tiny-vga)
- VGA monitor supporting 640x480 @ 60Hz
- VGA cable

The TinyVGA PMOD provides the necessary resistor DAC for 2-bit RGB output and VGA connector.

Project Pinout

Digital Pins

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

Tiny Tapeout Full Adder

by **nusli7**

0365 Wokwi Project

github.com/nusli7/TT-Project

wokwi.com/projects/455291649462874113

"1 bit full adder"

How it works

1 bit Full Adder using NAND, XOR and OR gates.

How to test

Input 1 bit binary numbers to the input and watch the outputs.

External hardware

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input a	output sum	—
1	input b	output carry	—
2	input carry	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

VGA Rings

by **Uri Shaked**

366 HDL Project

`github.com/urish/tt-rings`

"*Hypnotic concentric rings effect*"

How it works

This project generates a mesmerizing VGA display of hypnotic concentric rings that expand or contract from the center of the screen. It's designed as a demoscene-style visual effect for the Tiny Tapeout platform.

Technical Details

Distance Calculation: The distance from center is approximated using the formula: max(|x|, |y|) + min(|x|, |y|)/2, which provides a reasonable octagonal approximation of Euclidean distance without requiring multiplication or square roots.

Animation: A frame counter increments each video frame. The animated radius is computed by adding (or subtracting, depending on direction) the frame counter to the distance value, creating the illusion of expanding or contracting rings.

Color Generation: The 8-bit animated radius value is mapped to RGB222 color outputs by selecting different bit slices for each color channel, creating smooth color cycling through the rings.

How to test

1. Connect a VGA PMOD (such as the Tiny VGA PMOD) to the output pins
2. Connect a VGA monitor
3. Apply power and release reset

Controls

- **ui_in[0]**: Speed control
  - 0 = Normal speed (1 frame step per vsync)
  - 1 = Fast speed (2 frame steps per vsync)
- **ui_in[1]**: Direction control
  - 0 = Rings expand outward from center
  - 1 = Rings contract inward toward center

External hardware

- Tiny VGA PMOD (or compatible RGB222 VGA PMOD)

- VGA monitor or display with VGA input

Project Pinout

Digital Pins

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

Yturkeri_Mytinytapeout

by **Yigitcan Türkeri**

`0367` `5 kHz` `Wokwi Project`

`github.com/Duwandervall/yturkeri_mytinytapeout`

`wokwi.com/projects/455293410637770753`

"Test of Logic gates"

How it works

The main output is outputb it lights down right corner if the input combination is correct

How to test

Try different combinations if it is correct outputb lights together with the line on oppositing site.

External hardware

Your hand (and I find numpad useful.)

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	inputa	outputa	—
1	inputb	—	—
2	inputc	outputb	—
3	inputd	—	—
4	inputd	outputc	—
5	inpute	—	—
6	inputf	—	—
7	inputg	outputd	—

Johnson counter

by **Ehsan Khodadad**

`0368` `10 kHz` `Wokwi Project`

`github.com/EhsanKhodadad/ttihp-wokwi`

`wokwi.com/projects/456019228852926465`

"An eight bit Johnson counter"

How it works

It is a simple Johnson counter, which need only clock to count. A Johnson counter is defined as a digital sequential logic circuit where the complement output of the last flip-flop is fed back to the input of the first flip-flop. (https://www.electrical4u.com/johnson-counter/)

How to test

Just Turn the circuit on

External hardware

Nothing

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	OUT0	—
1	—	OUT1	—
2	—	OUT2	—
3	—	OUT3	—
4	—	OUT4	—
5	—	OUT5	—
6	—	OUT6	—
7	—	OUT7	—

2-Bit Adder

by **Daniel Hönlinger**

369 Wokwi Project

`github.com/DeeJayTF/deejay-tinytapeout`

`wokwi.com/projects/455291642978471937`

"Adds 2 2-bit numbers and outputs a 2-bit result with a 1-bit carry out signal"

How it works

A half-adder and a full-adder are used to build a 2-bit adder with a carry out signal.

Two 2-bit numbers will be added and produced result will be given as 2 Bits and a 1 Bit carry out signal.

How to test

Set the inputs according to the following truth table and check if the outputs match the truth table:

input a and b	output result	output cout
00 00	00	0
00 01	01	0
00 10	10	0
00 11	11	0
01 00	01	0
01 01	10	0
01 10	11	0
01 11	00	1
10 00	10	0
10 01	11	0
10 10	00	1
10 11	01	1
11 00	11	0
11 01	00	1
11 10	01	1
11 11	10	1

External hardware

Switches or buttons for input signals

LEDs for output signals

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	First Number - Bit 0	Result - Bit 0	—
1	Second Number - Bit 0	Result - Bit 1	—
2	First Number - Bit 1	Carry Out	—
3	Second Number - Bit 1	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

8Bit Posit MAC Unit

by **Ripunjay Singh**, **Muzaffar Asim Ansari**, **Aniket Singha**, **Sachin**, **Ashutosh Anand**, **Pradyut Kumar Sanki**, **Gaurav Kaushal** & **Biswabandhu Jana**

0370 16 MHz HDL Project

github.com/RipunjayS109/posit_mac

"*Its a MAC unit that uses Posit Number system fot operation.*"

How it works

Our Project is MAC unit that uses Posit number system for multiply and accumulate operation.

How to test

It requires two 8bit inputs that are successively undergo MAC operation

External hardware

No external hardware needed.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	ui_in[0]	uo_out[0]	uio_in[0]
1	ui_in[1]	uo_out[1]	uio_in[1]
2	ui_in[2]	uo_out[2]	uio_in[2]
3	ui_in[3]	uo_out[3]	uio_in[3]
4	ui_in[4]	uo_out[4]	uio_in[4]
5	ui_in[5]	uo_out[5]	uio_in[5]
6	ui_in[6]	uo_out[6]	uio_in[6]
7	ui_in[7]	uo_out[7]	uio_in[7]

4-Bit Adder

by **Mads A. Pedersen**

`0371` `Wokwi Project`

`github.com/madsamtoft/TinyTapeout2`

`wokwi.com/projects/456118923713667073`

"4 sequentially placed adders placed to construct a 5-bit LED output"

How it works

Using an 8-bit switch input, that represents two individual 4-bit numbers SW0-SW3 and SW4-SW7 respectively, two integers can be added together

How to test

To test this project, incrementally flip the 8 switches and verify that the numbers are added together correctly

External hardware

None used

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	IN0	OUT0	—
1	IN1	OUT1	—
2	IN2	OUT2	—
3	IN3	OUT3	—
4	IN4	OUT4	—
5	IN5	—	—
6	IN6	—	—
7	IN7	—	—

Herald

by **Pranav M**

`386` `50 MHz` `HDL Project`

`github.com/pranav0x0112/ttihp26a-Herald`

"A simple DSP Core for TinyTapeout"

Overview

Herald is a hardware digital signal processing (DSP) accelerator that combines two computational engines:

1. **CORDIC Unit** - COordinate Rotation DIgital Computer for trigonometric operations
2. **MAC Unit** - Multiply-Accumulate unit for arithmetic operations

The design provides 8 hardware-accelerated math operations accessible through a simple serial byte-oriented interface. All computations use **Q12.12 fixed-point arithmetic** (24-bit values: 12 integer bits + 12 fractional bits).

Architecture

Herald consists of three main components working together:

1. Control FSM

A finite state machine that manages the command/data protocol and orchestrates operations between the CORDIC and MAC engines. The FSM implements six states:

- **IDLE** - Waiting for command byte
- **CMD_WRITE** - Command received, ready for operands
- **DATA_WRITE_A** - Receiving first operand (3 bytes, LSB-first)
- **DATA_WRITE_B** - Receiving second operand (3 bytes, if needed)
- **EXECUTE** - Running computation (3-phase: start, wait_busy, get_result)
- **RESULT_READY** - Result available for byte-by-byte readout

2. CORDIC Engine (`mkCORDICHighLevel`)

Implements the CORDIC algorithm using iterative micro-rotations with only shifts and adds (no multipliers). The engine operates in two modes:

- **Rotation Mode**: Rotates a vector by a given angle (used for sin/cos)
- **Vectoring Mode**: Rotates a vector to align with x-axis (used for atan2, sqrt, normalize)

Each CORDIC operation completes in approximately 16-32 clock cycles depending on the precision required.

Supported Operations:

- `sin_cos(angle)` - Simultaneous sine and cosine computation
- `atan2(y, x)` - Arctangent returning angle from coordinates
- `sqrt_magnitude(x, y)` - Vector magnitude $\sqrt{(x^2 + y^2)}$
- `normalize(x, y)` - Returns normalized unit vector plus original magnitude

3. MAC Engine (`mkMAC`)

A fixed-point multiply-accumulate unit with internal accumulator register. Uses Bluespec-generated multipliers optimized for hardware synthesis.

Supported Operations:

- `multiply(a, b)` - Simple multiplication (accumulator unchanged)
- `mac(a, b)` - Multiply-accumulate: `acc += a × b`
- `msu(a, b)` - Multiply-subtract: `acc −= a × b`
- `clear_accumulator()` - Reset accumulator to zero

Block Diagram

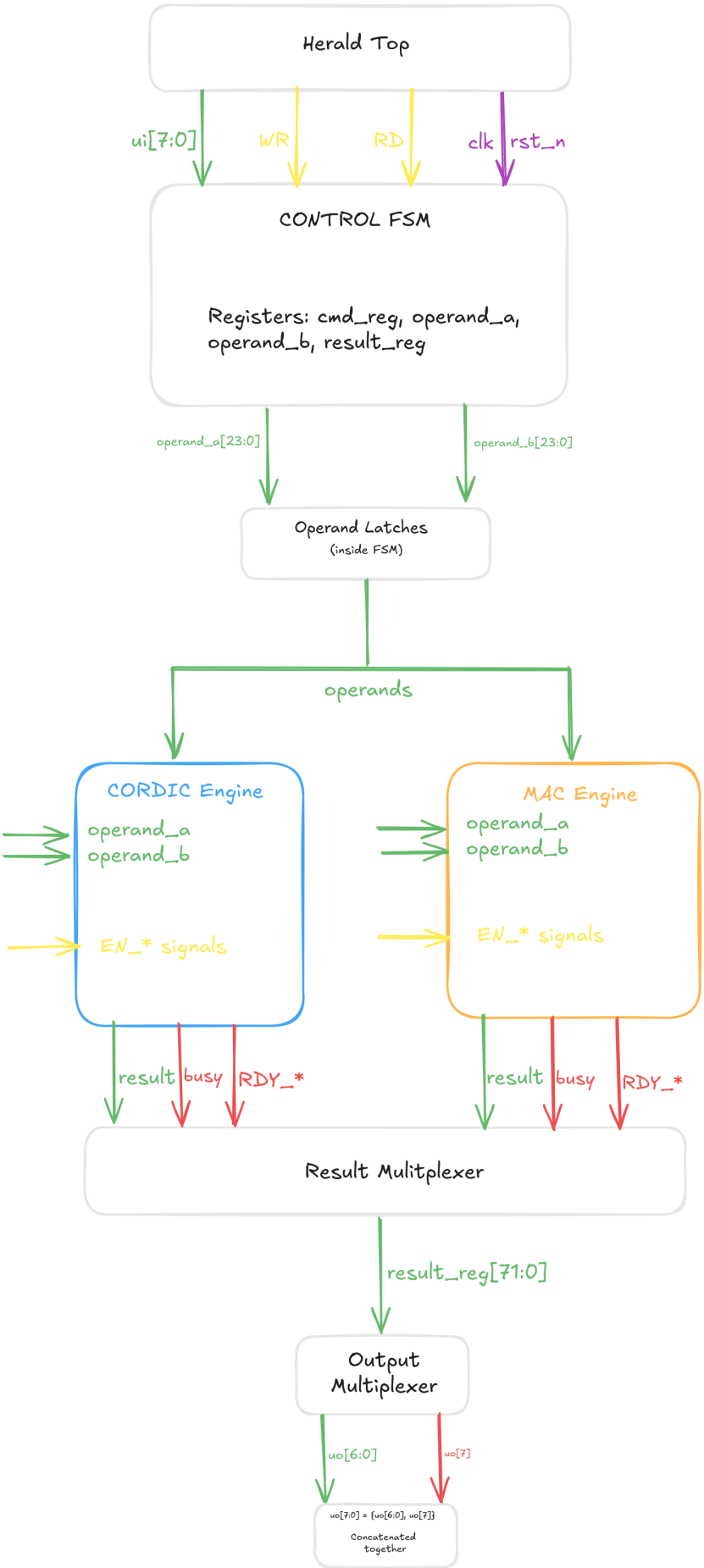


Figure 386.1: Herald Block Diagram

Figure 1: Herald architecture showing Control FSM, CORDIC Engine, MAC Engine, and data paths*

Interface Specification

Pin Assignment

Pin Group	Direction	Width	Description
`ui[7:0]`	Input	8 bits	Data/command input bus
`uo[7:0]`	Output	8 bits	Data output bus**uo[7] = BUSY flag**
`uio[0]`	Input	1 bit	**WR** - Write strobe (rising edge trigger)
`uio[1]`	Input	1 bit	**RD** - Read strobe (rising edge trigger)
`uio[7:2]`	Unused	6 bits	Reserved for future use
`clk`	Input	1 bit	System clock (up to 50 MHz)
`rst_n`	Input	1 bit	Active-low asynchronous reset
`ena`	Input	1 bit	Chip enable (TinyTapeout standard)

Signal Descriptions

Input Data Bus (`ui[7:0]`)
8-bit bidirectional bus for writing command bytes and operand data. Data is latched on the rising edge of the WR strobe.

Output Data Bus (`uo[7:0]`)
8-bit output bus providing result data and status:

- **Bits [6:0]**: Result data (when BUSY = 0)
- **Bit [7]**: BUSY flag
  - `1` = Herald is processing a command (busy)
  - `0` = Herald is idle or result ready for reading

Write Strobe (`uio[0]`)
Rising edge triggers data capture from `ui[7:0]`. Used to write command bytes and operand bytes.

Read Strobe (`uio[1]`)
Rising edge triggers output of next result byte on `uo[7:0]`. Used to read result data byte-by-byte.

Reset Behavior
On reset (`rst_n = 0`):

- FSM returns to IDLE state
- All registers cleared (command, operands, results)

- BUSY flag set to 0
- Output bus set to 0x00
- Both CORDIC and MAC engines reset

Communication Protocol

Herald uses a **serial byte-oriented protocol** with write/read strobes for transferring commands, operands, and results.

Write Sequence (Host → Herald)

Step 1: Write Command Byte

```
1. Set ui[7:0] = command opcode (e.g., 0x10 for SINCOS)
2. Pulse uio[0] (WR) high then low (rising edge latches data)
3. Wait 1 clock cycle
4. Check uo[7] = 1 (BUSY flag set, FSM enters CMD_WRITE state)
```

Step 2: Write Operand Bytes

```
For each operand byte (always LSB-first, little-endian):
1. Set ui[7:0] = data byte
2. Pulse uio[0] (WR)
3. Wait 1 clock cycle
4. Repeat for all operand bytes

Operand A: 3 bytes (bits [7:0], [15:8], [23:16])
Operand B: 3 bytes (only if command requires second operand)
```

Step 3: Wait for Completion

```
1. Poll uo[7] until it becomes 0
2. Herald is now in RESULT_READY state
3. Result is available for reading
```

Read Sequence (Herald → Host)

Step 4: Read Result Bytes

```
For each result byte (LSB-first, little-endian):
1. Pulse uio[1] (RD)
2. Wait 1 clock cycle
3. Read uo[7:0] to get result byte
4. Repeat for all result bytes

After reading the last byte, Herald automatically returns to IDLE.
```

Command Operand Requirements

Command	Operand A	Operand B	Result Bytes
SINCOS (0x10)	angle	-	6
ATAN2 (0x11)	y	x	3
SQRT (0x12)	x	y	3

NORMALIZE (0x13)	x	y	9
MULTIPLY (0x20)	a	b	3
MAC (0x21)	a	b	3
CLEAR (0x22)	-	-	0
MSU (0x23)	a	b	3

FSM State Machine

The control FSM implements the following state transitions:

```
IDLE
  ↓ (WR strobe with command byte)
CMD_WRITE
  ↓ (WR strobe with operand bytes)
DATA_WRITE_A (receive operand A, 3 bytes)
  ↓ (if command needs operand B)
DATA_WRITE_B (receive operand B, 3 bytes)
  ↓
EXECUTE (3 phases: start → wait_busy → get_result)
  ↓
RESULT_READY (output bytes on RD strobe)
  ↓ (after last byte read)
IDLE
```

State Descriptions

IDLE

- Waiting for command byte
- BUSY = 0
- Transitions on WR strobe

CMD_WRITE

- Command byte received
- BUSY = 1
- Determines if operands needed
- CLEAR command skips directly to EXECUTE

DATA_WRITE_A

- Collecting first operand (3 bytes)
- Bytes stored LSB-first
- After 3rd byte, checks if operand B needed

DATA_WRITE_B

- Collecting second operand (3 bytes)
- Only for commands requiring two operands

- After 3rd byte, transitions to EXECUTE

EXECUTE

- Three-phase execution:
  - **Phase 0**: Assert start enable signal to CORDIC or MAC
  - **Phase 1**: Wait for engine busy flag to clear
  - **Phase 2**: Assert get_result enable and latch result
- Transitions to RESULT_READY when result captured

RESULT_READY

- Result available for reading
- BUSY = 0
- Each RD strobe outputs next byte
- Returns to IDLE after last byte read

State Diagram

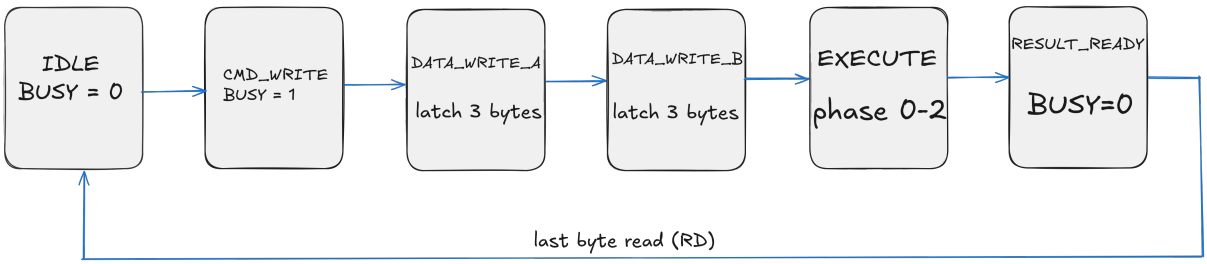


Figure 386.2: Herald FSM State Diagram

Figure 2: FSM state transitions for command processing*

Data Format: Q12.12 Fixed-Point

All operands and results use **24-bit Q12.12 fixed-point** representation:

```
Bit Layout:
[23]     [22:12]      [11:0]
Sign     Integer      Fractional
1 bit    11 bits      12 bits
```

Format Details

- **Sign bit [23]**: 0 = positive, 1 = negative (two's complement)
- **Integer bits [22:12]**: Signed integer part (range: −2048 to +2047)
- **Fractional bits [11:0]**: Unsigned fractional part (units of 1/4096)

Numeric Range and Resolution

- **Range**: −2048.0 to +2047.999755859375
- **Resolution**: 1/4096 ≈ 0.000244140625
- **Smallest positive value**: 0x000001 = 1/4096
- **Largest positive value**: 0x7FFFFF ≈ 2047.9998

- **Most negative value**: 0x800000 = −2048.0

Encoding Examples

Decimal Value	Hex (Q12.12)	Binary Breakdown
0.0	0x000000	0 00000000000 000000000000
1.0	0x001000	0 00000000001 000000000000
0.5	0x000800	0 00000000000 100000000000
−1.0	0xFFF000	1 11111111111 000000000000
3.14159	0x003243	0 00000000011 001001000011
π/2 (1.5708)	0x001921	0 00000000001 100100100001
−0.25	0xFFFC00	1 11111111111 110000000000

Conversion Functions

Floating-point to Q12.12:

```
int32_t float_to_q12_12(float value) {
    return (int32_t)(value * 4096.0f);
}
```

Q12.12 to floating-point:

```
float q12_12_to_float(int32_t q_value) {
    return (float)q_value / 4096.0f;
}
```

Byte Order (Endianness)

All multi-byte values are transmitted **LSB-first (little-endian)**:

- **Byte 0**: Bits [7:0] (least significant)
- **Byte 1**: Bits [15:8]
- **Byte 2**: Bits [23:16] (most significant)

Example: Value 0x003243 (π ≈ 3.14159) is transmitted as:

```
Byte 0: 0x43
Byte 1: 0x32
Byte 2: 0x00
```

Command Reference

CORDIC Commands

0x10: CORDIC_SINCOS

Compute sine and cosine of an angle simultaneously

Operands:

- `angle` (3 bytes, Q12.12, radians)

Result: 6 bytes

- Bytes 0-2: `sin(angle)` (Q12.12)
- Bytes 3-5: `cos(angle)` (Q12.12)

Range: Input angle should be in range [-π, +π] for best accuracy. Values outside this range will wrap.

Example:

```
Input:  angle = π/4 ≈ 0.7854 = 0x000C91
Output: sin = 0.707 ≈ 0x000B50
        cos = 0.707 ≈ 0x000B50
```

0x11: CORDIC_ATAN2
Compute arctangent of y/x (angle from coordinates)

Operands:

- y (3 bytes, Q12.12)
- x (3 bytes, Q12.12)

Result: 3 bytes

- Angle in radians (Q12.12), range [-π, +π]

Special Cases:

- If x = 0 and y > 0: returns +π/2
- If x = 0 and y < 0: returns -π/2
- If x = 0 and y = 0: returns 0

Example:

```
Input:  y = 1.0 = 0x001000
        x = 1.0 = 0x001000
Output: angle = π/4 ≈ 0.7854 = 0x000C91
```

0x12: CORDIC_SQRT
Compute magnitude of vector (x, y) using √(x² + y²)

Operands:

- x (3 bytes, Q12.12)
- y (3 bytes, Q12.12)

Result: 3 bytes

- Magnitude (Q12.12)

Note: Both x and y are squared internally, so sign doesn't affect result. This is effectively computing the Euclidean distance from origin.

Example:

```
Input:  x = 3.0 = 0x003000
        y = 4.0 = 0x004000
Output: magnitude = 5.0 = 0x005000
```

0x13: CORDIC_NORMALIZE

Normalize vector and return both normalized components plus magnitude

Operands:

- `x` (3 bytes, Q12.12)
- `y` (3 bytes, Q12.12)

Result: 9 bytes

- Bytes 0-2: `x_normalized` (Q12.12, unit vector x-component)
- Bytes 3-5: `y_normalized` (Q12.12, unit vector y-component)
- Bytes 6-8: `magnitude` (Q12.12, original vector length)

Properties:

- x_normalized² + y_normalized² ≈ 1.0
- x = x_normalized × magnitude
- y = y_normalized × magnitude

Example:

```
Input:  x = 3.0 = 0x003000
        y = 4.0 = 0x004000
Output: x_norm = 0.6 = 0x000999
        y_norm = 0.8 = 0x000CCD
        magnitude = 5.0 = 0x005000
```

MAC Commands

0x20: MAC_MULTIPLY

Simple fixed-point multiplication: result = a × b

Operands:

- `a` (3 bytes, Q12.12)
- `b` (3 bytes, Q12.12)

Result: 3 bytes

- Product `a × b` (Q12.12)

Note: Internal accumulator is **not affected** by this operation.

Example:

```
Input:  a = 2.5 = 0x002800
        b = 1.5 = 0x001800
Output: result = 3.75 = 0x003C00
```

0x21: MAC_MAC
Multiply-accumulate: accumulator += a × b

Operands:

- a (3 bytes, Q12.12)
- b (3 bytes, Q12.12)

Result: 3 bytes

- New accumulator value (Q12.12)

Operation:

```
accumulator = accumulator + (a × b)
return accumulator
```

Use Cases:

- Dot products: $\Sigma(a_i \times b_i)$
- FIR filters: $\Sigma(coeff_i \times sample_i)$
- Matrix multiplication

Example (accumulator starts at 0):

```
Call 1: a = 1.0, b = 2.0 → acc = 2.0 = 0x002000
Call 2: a = 3.0, b = 1.0 → acc = 5.0 = 0x005000
Call 3: a = 0.5, b = 4.0 → acc = 7.0 = 0x007000
```

0x22: MAC_CLEAR
Clear accumulator to zero

Operands: None

Result: None (returns to IDLE immediately)

Operation:

```
accumulator = 0
```

Use Cases:

- Reset before new MAC sequence
- Initialize for new computation

Example:

```
Write: CMD = 0x22
(No operands written, no result read, Herald returns to IDLE)
```

0x23: MAC_MSU
Multiply-subtract: accumulator -= a × b

Operands:

- a (3 bytes, Q12.12)

- `b` (3 bytes, Q12.12)

Result: 3 bytes

- New accumulator value (Q12.12)

Operation:

```
accumulator = accumulator − (a × b)
return accumulator
```

Use Cases:

- Error correction algorithms
- Adaptive filters
- Subtractive synthesis

Example (accumulator = 10.0):

```
Input:  a = 2.0 = 0x002000
        b = 1.5 = 0x001800
Output: acc = 10.0 − 3.0 = 7.0 = 0x007000
```

Testing and Verification

Herald includes comprehensive cocotb testbenches covering:

Protocol Tests (`test_wrapper.py`)

- Command write sequences
- Operand byte ordering
- BUSY flag behavior
- Result readback
- FSM state transitions

CORDIC Tests (`test_cordic.py`)

- Trigonometric accuracy (sin/cos)
- Angle calculation (atan2)
- Magnitude computation (sqrt)
- Vector normalization
- Edge cases (zero vectors, negative values)

MAC Tests (`test_mac.py`)

- Multiplication accuracy
- Accumulator operations
- Overflow behavior
- Clear functionality
- Multi-step accumulation

Design Notes

CORDIC Algorithm Implementation

The CORDIC engine uses a table of pre-computed arctangent values for each iteration:

```
atan(2^-i) for i = 0, 1, 2, ..., 23
```

Each iteration performs:

```
x_new = x - y × d × 2^(-i)
y_new = y + x × d × 2^(-i)
z_new = z - d × atan(2^(-i))
```

where `d ∈ {-1, +1}` is the rotation direction determined by the sign of the remaining angle.

Fixed-Point Multiplication

MAC operations multiply two Q12.12 values:

```
Q12.12 × Q12.12 → Q24.24 (48-bit intermediate)
Shift right 12 bits → Q12.12 (24-bit result)
```

The Bluespec-generated multiplier handles this automatically with proper rounding.

Clock Domain

All operations are fully synchronous to the input clock. No internal clock generation or clock domain crossings.

Reset Strategy

Asynchronous reset (`rst_n`) for reliable startup. All registers have defined reset values.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Command/Data input bit 0	Data output bit 0	Unused
1	Command/Data input bit 1	Data output bit 1	Unused
2	Command/Data input bit 2	Data output bit 2	Unused
3	Command/Data input bit 3	Data output bit 3	Unused
4	Command/Data input bit 4	Data output bit 4	Unused
5	Command input bit 5	Data output bit 5	Unused
6	Command input bit 6	Data output bit 6	Unused
7	Command input bit 7	Data output bit 7	Unused

Tiny FABulous FPGA

by **Leo Moser**

0402 HDL Project

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

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

How it works

Tiny FABulous FPGA for IHP26a.

This design implements a tiny FPGA with 72 LUT4+FF. The FPGA fabric is 5x5 tiles in size, of which 3x3 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	72	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]

VGA Tetris

by **Gytis**

`0418` `25 MHz` `HDL Project`

`github.com/JPGHhb/ttihp-vga-tetris`

"VGA Tetris (640x480 @ 60Hz)"

How it works

The game over state can be cleared by pressing the down button a couple of times.

How to test

To control the game, toggle inputs ui_in[0 to 3]. Note: The inputs are expected to be HIGH (i.e. pulled-up) when not used, hence only LOW level is associated with an action. On RP2040's MicroPython shell this can be achieved by: tt.pins.ui_in0.pull = Pin.PULL_UP tt.pins.ui_in1.pull = Pin.PULL_UP tt.pins.ui_in2.pull = Pin.PULL_UP tt.pins.ui_in3.pull = Pin.PULL_UP

External hardware

Use https://github.com/mole99/tiny-vga for display output

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	left	vga_r_1	—
1	right	vga_g_1	—
2	rotate	vga_b_1	—
3	down	vga_vs	—
4	none	vga_r_0	—
5	none	vga_g_0	—
6	none	vga_b_0	—
7	none	vga_hs	—

TinyTapeout-Processor2

by **Matei Chiriac**

421 HDL Project

github.com/matei-coder/TinyTapeout-Processor2

"It is a design of a processor"

How it works

This project implements a custom **8-bit RISC processor** with a 3-stage pipeline (Fetch → Decode → Execute).

Architecture

- **Register file:** 8 general-purpose 8-bit registers (R0–R7), all reset to 0
- **Program memory (ROM):** 32 × 16-bit instructions, loaded at runtime via serial interface
- **Data memory (RAM):** 16 × 8-bit bytes
- **ALU operations:** ADD, SUB, AND, OR, XOR, SHL, SHR, CMP
- **Pipeline stages:** FETCH → DECODE → EXECUTE (3 clock cycles per instruction)
- **Status flags:** Zero (Z), Carry (C), Negative (N)

Instruction Set (16-bit encoding)

Opcode	Mnemonic	Operation
0x0	ADD Rd, Rs1, Rs2	Rd = Rs1 + Rs2
0x1	SUB Rd, Rs1, Rs2	Rd = Rs1 − Rs2
0x2	AND Rd, Rs1, Rs2	Rd = Rs1 & Rs2
0x3	OR Rd, Rs1, Rs2	Rd = Rs1 | Rs2
0x4	XOR Rd, Rs1, Rs2	Rd = Rs1 ^ Rs2
0x5	SHL Rd, Rs1	Rd = Rs1 << 1, Carry = bit shifted out
0x6	SHR Rd, Rs1	Rd = Rs1 >> 1, Carry = bit shifted out
0x7	LDI Rd, #imm8	Rd = 8-bit immediate value
0x8	LOAD Rd, [Rs1]	Rd = RAM[Rs1 & 0xF]
0x9	STORE Rd, [Rs1]	RAM[Rs1 & 0xF] = Rd
0xA	JMP addr	PC = addr (5-bit, range 0–31)
0xB	JZ addr	if Z=1 then PC = addr
0xC	JNZ addr	if Z=0 then PC = addr
0xD	CMP Rd, Rs1	set flags based on Rd − Rs1 (no write)

0xE	OUT Rd	uo_out = Rd
0xF	IN Rd	Rd = ui_in

Instruction word layout

```
Bit:  15 14 13 12 | 11 10  9 |  8  7  6 |  5  4  3 |  2  1  0
      [  OPCODE  ] [   Rd   ] [  Rs1   ] [  Rs2   ] [  000  ]

Special cases:
  LDI  : [15:12]=0x7, [11:9]=Rd, [8]=0,  [7:0]=imm8
  JMP  : [15:12]=0xA, [11:5]=0,  [4:0]=addr5
  JZ   : [15:12]=0xB, [11:5]=0,  [4:0]=addr5
  JNZ  : [15:12]=0xC, [11:5]=0,  [4:0]=addr5
  OUT  : [15:12]=0xE, [11:9]=Rs, [8:0]=0
  IN   : [15:12]=0xF, [11:9]=Rd, [8:0]=0
```

Pin mapping

Pin	Direction	Description
`ui_in[7:0]`	Input	Program byte during load / data for `IN` instruction
`uo_out[7:0]`	Output	Result of `OUT` instruction
`uio_in[0]`	Input	`load_mode`: 1 = loading program, 0 = executing
`uio_in[1]`	Input	`load_valid`: rising edge loads one byte
`uio_out[2]`	Output	Flag Zero (Z)
`uio_out[3]`	Output	Flag Carry (C)
`uio_out[4]`	Output	Flag Negative (N)

How to test

Step 1 — Reset

Assert `rst_n = 0` for at least 1 clock cycle, then set `rst_n = 1`. All registers, flags and PC are cleared to 0.

Step 2 — Load a program

1. Set `uio_in[0] = 1` (load mode — CPU is held at reset, ROM write enabled)
2. For each 16-bit instruction (in order, starting from address 0):
   - Put the **high byte** `[15:8]` on `ui_in[7:0]`, pulse `uio_in[1]` high then low
   - Put the **low byte** `[7:0]` on `ui_in[7:0]`, pulse `uio_in[1]` high then low
3. Set `uio_in[0] = 0` — CPU starts executing from PC = 0

Step 3 — Example program: counter 0 → 9, looping forever

```
; R1 = counter (0..9)
; R2 = step = 1
; R3 = limit = 10

LDI R1, #0      ; addr 0
LDI R2, #1      ; addr 1
LDI R3, #10     ; addr 2
OUT R1           ; addr 3  <-- loop start
ADD R1, R1, R2  ; addr 4  R1 = R1 + 1
CMP R1, R3      ; addr 5  flags = R1 - R3
JNZ 3            ; addr 6  if R1 != R3, jump back to OUT
JMP 0            ; addr 7  restart (R1=10, reset to 0)
```

Byte sequence to load (high byte first per instruction):

Addr	Instruction	High	Low	Binary (16-bit)
0	`LDI R1, #0`	`0x72`	`0x00`	`0111 001 0 0000 0000`
1	`LDI R2, #1`	`0x74`	`0x01`	`0111 010 0 0000 0001`
2	`LDI R3, #10`	`0x76`	`0x0A`	`0111 011 0 0000 1010`
3	`OUT R1`	`0xE2`	`0x00`	`1110 001 000 000 000`
4	`ADD R1,R1,R2`	`0x02`	`0x50`	`0000 001 001 010 000`
5	`CMP R1, R3`	`0xD2`	`0xC0`	`1101 001 011 000 000`
6	`JNZ 3`	`0xC0`	`0x03`	`1100 0000 0000 0011`
7	`JMP 0`	`0xA0`	`0x00`	`1010 0000 0000 0000`

Step 4 — Read outputs

Signal	What to observe
`uo_out[7:0]`	Shows 0x00 → 0x01 → … → 0x09 → 0x00 cycling
`uio_out[2]` (Z flag)	Goes high when result = 0
`uio_out[3]` (C flag)	Goes high on carry/borrow
`uio_out[4]` (N flag)	Goes high when result MSB = 1

External hardware

No external hardware required. All I/O uses the standard TinyTapeout pin interface.

Optionally, connect a microcontroller or logic analyser to `ui_in` / `uio_in` to automate program loading and capture `uo_out` results.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	DATA_IN bit 0 (program byte / IN instruction)	DATA_OUT bit 0 (result of OUT instruction)	LOAD_MODE input (1=load program, 0=execute)
1	DATA_IN bit 1	DATA_OUT bit 1	LOAD_VALID input (rising edge loads one byte)
2	DATA_IN bit 2	DATA_OUT bit 2	FLAG_ZERO output (ALU result = 0)
3	DATA_IN bit 3	DATA_OUT bit 3	FLAG_CARRY output (arithmetic carry/ borrow)
4	DATA_IN bit 4	DATA_OUT bit 4	FLAG_NEG output (ALU result negative)
5	DATA_IN bit 5	DATA_OUT bit 5	—
6	DATA_IN bit 6	DATA_OUT bit 6	—
7	DATA_IN bit 7	DATA_OUT bit 7	—

FH Joanneum TinyTapeout

by **FH Joanneum**

0427 HDL Project

github.com/Electronic-and-Computer-Engineering/TT_IHP26a_FH_uC

“FH Joanneum”

How it works

By magic

How to test

By magic

External hardware

By magic

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	SPI_MISO	SPI_MOSI	—
1	—	SPI_CLK	—
2	—	SPI_CS1_N	—
3	—	SPI_CS2_N	—
4	GPIO_IN[0]	GPIO_OUT[0]	—
5	GPIO_IN[1]	GPIO_OUT[1]	—
6	GPIO_IN[2]	GPIO_OUT[2]	—
7	GPIO_IN[3]	GPIO_OUT[3]	—

LoRa Edge SoC

by **TechHU-GS**

435 | 25 MHz | HDL Project

`github.com/TechHU-GS/tt_rv32_trial`

"*RISC-V LoRa node SoC with CRC16, I2C, WDT, RTC, Seal peripherals*"

How it works

LoRa Edge SoC is a RISC-V (RV32EC) based system-on-chip designed for LoRa IoT edge nodes. It integrates the TinyQV bit-serial CPU running at 25MHz with 8 custom peripherals:

- **CRC16 engine** with hardware acceleration (shared between CPU and Seal)
- **I2C master** (Forencich AXI Stream core) for sensor communication
- **SPI master** for SX1268 LoRa transceiver control
- **Watchdog timer** with configurable timeout
- **RTC counter** with 1-second resolution
- **Seal register** with monotonic counter for tamper detection
- **Countdown timer** with microsecond resolution and IRQ
- **Latch-based memory** (32 bytes on-chip SRAM)

The CPU fetches instructions from external QSPI Flash and uses QSPI PSRAM for data storage. All peripherals are memory-mapped at 0x8000000.

How to test

1. Connect QSPI Flash (with firmware) and PSRAM to the bidirectional pins
2. Connect a USB-UART adapter to UART TX (uo[0]) and RX (ui[7]) at 115200 baud
3. After reset, the CPU boots from Flash and outputs POST messages via UART
4. Peripherals can be tested via firmware: I2C sensor reads, SPI LoRa transactions, watchdog kicks, RTC time queries

External hardware

- **QSPI Flash** (e.g., W25Q128) — connected to uio[0:5] for instruction/data storage
- **QSPI PSRAM** (e.g., APS6404L) — connected to uio[6:7] for data memory
- **SX1268 LoRa module** — connected to SPI (uo[3:5], ui[2]) and control pins (uo[1], ui[0:1])

- **I2C sensors** — connected to SCL (uo[2]) and SDA (uo[6], ui[3]) with 4.7K pullups
- **GPS module** (optional) — 1PPS output to ui[4]

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	SX1268 DIO1 (IRQ)	UART TX	Flash CS
1	SX1268 BUSY	SX1268 RESET	SD0
2	SPI MISO	I2C SCL	SD1
3	I2C SDA in	SPI MOSI	SCK
4	1PPS input	SPI CS	SD2
5	GPIO in (spare)	SPI SCK	SD3
6	GPIO in (spare)	I2C SDA out	RAM A CS
7	UART RX	LED GPIO	RAM B CS

ttihp-26a-risc-v-wg-swc1

by **risc-v-wg**

462 HDL Project

`github.com/risc-v-wg/ttihp-26a-risc-v-wg-swc1`

"*RISCV rv32i*"

How it works

This is a RISC-V 32I instruction set compatible CPU that uses QSPI PSRAM/ Flash as external memory. It uses a state machine design for Tiny Tapeout to achieve a compact size. Features:

- 64-bit Free run counter with interrupt
- QSPI memory interface
- 4-bit GPIO, 3-bit GPO, 6-bit GPI
- 1 interrupt pin
- UART with monitor functions (program read/write, etc.) Constraints:
- M mode only
- Fence not supported
- Ebreak not tested

How to test

Connect external QSPI PSRAM and connect the UART to a terminal. Upload the test program from the terminal and execute it to verify operation.

UART monitor command list:

- g : goto PC address (executes until Ctrl-c is pressed) : format: g
- Ctrl-c : Interrupts all commands. Press Ctrl-c if you get stuck : format: Ctrl-c
- w : Writes data to memory : format: w .... Ctrl-c
- r : Reads data from memory : format: r
- s : Set breakpoint address: format s When setting: Specify the address value where you want to break When clearing: Specify an odd number for the address (set bit[0] to 1)
- l : Set data read breakpoint address: format l When setting: Specify the address value where you want to break When clearing: Specify an odd number for the address (set bit[0] to 1)
- m : Set data write breakpoint address: format m When setting: Specify the address value where you want to break When clearing: Specify an odd number for the address (set bit[0] to 1)
- j : Display the current PC value: format j

- i : Write data to I/O registers, etc. The target changes based on the upper 2 bits of the address : format: I .... Ctrl-c Upper 2 bits == 2'b11 ; I/O register Upper 2 bits == 2'b10 : CSR register Upper 2 bits == 2'b00 : Register File *However, for CSR and RF, specify an address equal to 4 times the register number Example: Address for CSR mstatus (0x300) is 80000c00 RF x3 Register: 0000 000c
- p: Performs data read operations on I/O registers, etc. The upper 16 bits of the address are ignored : format: p Upper 2 bits == 2'b11 ; I/O Register Upper 2 bits == 2'b10 : CSR Register Upper 2 bits == 2'b00 : Register File *However, for CSR and RF registers, specify an address equal to 4 times the register number Example: CSR mstatus (0x300) address: 80000c00 RF x3 Register 0000 000c
- t : Memory zero-fill : format t

External hardware

- An external QSPI PSRAM is required.
- UART connection to a terminal is requred for contorlling chip.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	rx	tx	sio[0]
1	interrupt_0	ce_n[0]	sio[1]
2	init_qspicmd	ce_n[1]	sio[2]
3	init_latency[0]	ce_n[2]	sio[3]
4	init_latency[1]	sck	gpio[0]
5	init_cpu_start	rgb_led[0]	gpio[1]
6	init_uart[0]	rgb_led[1]	gpio[2]
7	init_uart[1]	rgb_led[2]	gpio[3]

TT6581

by **Andreas Pedersen**

466 | 50 MHz | HDL Project

`github.com/apedersen00/tt6581`

"MOS6581-inspired audio synthesizer chip"

How it works

Inspired by the legendary MOS6581 Sound Interface Device (SID) chip used in retro computers such as the Commodore 64, the *Tiny Tapeout* 6581 (TT6581) is a original digital interpretation supporting nearly the entire original MOS6581 feature set, implemented in 2x2 tiles for Tiny Tapeout.

All configuration is done via a 4-wire SPI interface, and the audio output is a 1-bit PDM signal at 10 MHz.

Features

- Full control through a Serial-Peripheral Interface (SPI).
- Three independently synthesized voices.
- Four supported waveform types (triangle, sawtooth, square and noise).
- Attack, decay, sustain, release (ADSR) envelope shaping.
- Chamberlin State-Variable Filter (SVF) for low-pass, high-pass, band-pass and band-reject.
- Second-order Delta-Sigma DAC.

Architecture

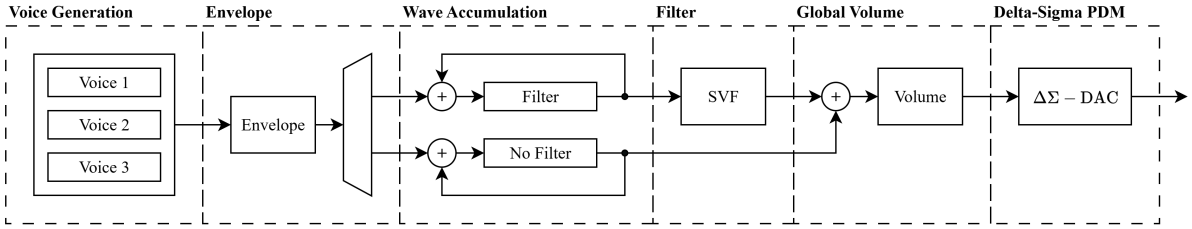


Figure 466.1: TT6581 Architecture

The diagram above shows the datapath in the TT6581. A tick generator triggers the generation of a single audio sample at 50 kHz.

1. **Voice Generation:** One 10-bit voice at a time is generated. Internal phase registers keep track of each voice's state while inactive. The frequency and waveform type are set by the programmed values in the register file. Supported waveforms are triangle, sawtooth, pulse or noise.
2. **Envelope:** An ADSR envelope generator produces an 8-bit amplitude value per voice. The envelope is applied to each voice by multiplication.

3. **Wave Accumulation:** The three voices are accumulated (mixed) by addition. Depending on the filter enable bit of each voice, they are accumulated in one of two registers: one that will be passed through the SVF, or one that will bypass it.
4. **Filter:** A Chamberlin State-Variable Filter (SVF) processes the filter accumulator. It supports low-pass, high-pass, band-pass and band-reject modes with tuneable frequency cutoff and resonance (Q).
5. **Global Volume:** The SVF output is summed with the bypass accumulator. A global 8-bit volume is applied by multiplication resulting in the final mix.
6. **Delta-Sigma PDM:** An error-feedback Delta-Sigma modulator converts the final mix to 1-bit PDM output at 10 MHz (OSR = 200).

To fit the strict 2x2 tiles area requirement, the entire synthesis is time-multiplexed meaning most modules are *finite state machines*. A single 24x16 multiplier is shared for all modules. The 50 kHz sample tick wakes up a master controller (FSM) that then initiates the synthesis of a single sample.

Pin Mapping

The TT6581 uses the bidirectional IO pins for SPI and a single dedicated output for the PDM audio signal. All dedicated inputs are unused.

Pin	Direction	Function
`uio[0]`	Input	SPI Chip Select (active low)
`uio[1]`	Input	SPI MOSI
`uio[2]`	Output	SPI MISO
`uio[3]`	Input	SPI SCLK
`uio[4:7]`	-	Unused
`uo[0]`	Output	PDM audio output
`uo[1:7]`	-	Unused
`ui[0:7]`	-	Unused

The PDM output should be passed through a 4th order Bessel filter for the best reconstruction of the analog waveform.

Programming

The TT6581 is programmed in much the same way as the original MOS6581. The register layout mirrors the original SID, three voice channels followed by filter and volume registers and the same ADSR, waveform selection and filter concepts apply. The main differences are:

- Registers are accessed through an SPI interface.

- The filter coefficients are pre-calculated and written directly as fixed-point values, rather than the raw 11-bit FC value used by the MOS6581.

Keeping the changes in mind, the original MOS6581 datasheet is a good reference for how to use the TT6581.

SPI Protocol

The SPI interface uses CPOL=0, CPHA=0 (data sampled on the rising edge of SCLK). Each transaction is a 16-bit frame while CS is held low:

Bit	15	14:8	7:0
Field	R/W	Address [6:0]	Data [7:0]

- **Bit 15** = 1 for write, `0` for read.
- **Bits 14:8** = 7-bit register address.
- **Bits 7:0** = write data.

Data is transmitted MSB first.

Playing a tone

1. **Set volume:** Write `0xFF` to register `VOLUME` for max volume.
2. **Set frequency:** Compute the 16-bit frequency control word and write it to `FREQ_LO` / `FREQ_HI`.
3. **Set ADSR:** Write attack/decay to `AD` and sustain/release to `SR`.
4. **Select waveform and gate on:** Write the CONTROL register with the desired waveform bit and `GATE=1`.

For example, to play a 440 Hz sawtooth on Voice 0 with instant attack and full sustain:

```
SPI Write: addr=0x1A, data=0xFF    # Volume = max
SPI Write: addr=0x00, data=0x05    # FREQ_LO = 0x05  (FCW for 440
Hz = 0x1205)
SPI Write: addr=0x01, data=0x12    # FREQ_HI = 0x12
SPI Write: addr=0x05, data=0x00    # AD = 0x00 (attack=0, decay=0)
SPI Write: addr=0x06, data=0xF0    # SR = 0xF0 (sustain=15,
release=0)
SPI Write: addr=0x04, data=0x21    # CONTROL = sawtooth + gate on
```

To release the note, write CONTROL again with `GATE=0`:

```
SPI Write: addr=0x04, data=0x20    # CONTROL = sawtooth + gate off
```

Formulas

Frequency Control Word (FCW):

$$FCW = \frac{f_{\text{desired}} \times 2^{19}}{F_s}$$

where $F_s = 50$ kHz (sample rate). The 16-bit FCW is split across `FREQ_LO` (bits 7:0) and `FREQ_HI` (bits 15:8).

The maximum representable voice frequency is limited by the 16-bit FCW:

$$f_{\max} = \frac{65535 \times F_s}{2^{19}} \approx 6250 \ \ \text{Hz}$$

Filter Cutoff Coefficient (Q1.15 signed):

$$\text{FCC} = \left[2 \cdot \sin\left(\frac{\pi \cdot f_c}{F_s} \right) \cdot 32768 \right]$$

where f_c is the desired cutoff frequency in Hz. The 16-bit result is split across `F_LO` and `F_HI`.

Filter Damping Coefficient (Q4.12 signed):

$$\text{FDC} = \left[\frac{1}{Q} \cdot 4096 \right]$$

where Q is the desired resonance. The 16-bit result is split across `Q_LO` and `Q_HI`.

Register map summary

The TT6581 has a 7-bit address space and 26 8-bit registers. Three identical voice register groups are followed by a filter/volume group.

The full register map is described in `regs.yaml`.

Voice Registers

Each voice occupies 7 consecutive 8-bit registers. Voice 0 starts at `0x00`, Voice 1 at `0x07`, and Voice 2 at `0x0E`.

Offset	Name	Bits	Description
0x00	FREQ_LO	7:0	Frequency control word - low byte
0x01	FREQ_HI	7:0	Frequency control word - high byte
0x02	PW_LO	7:0	Pulse width - low byte
0x03	PW_HI	3:0	Pulse width - high byte
0x04	CONTROL	7:0	Waveform select and voice control
0x05	AD	7:0	Attack (7:4) / Decay (3:0)
0x06	SR	7:0	Sustain (7:4) / Release (3:0)

CONTROL register bit fields:

Bit	Name	Description
7	NOISE	Select noise waveform

6	PULSE	Select pulse (square) waveform
5	SAW	Select sawtooth waveform
4	TRI	Select triangle waveform
3	-	Reserved
2	RING_MOD	Enable ring modulation
1	SYNC	Enable oscillator sync
0	GATE	Gate (1 = attack, 0 = release)

Filter and Volume Registers

Address	Name	Bits	Description
0x15	F_LO	7:0	Filter cutoff coefficient - low byte
0x16	F_HI	7:0	Filter cutoff coefficient - high byte
0x17	Q_LO	7:0	Filter damping coefficient - low byte
0x18	Q_HI	7:0	Filter damping coefficient - high byte
0x19	EN_MODE	5:0	Filter enable and mode select
0x1A	VOLUME	7:0	Global volume (0x00–0xFF)

EN_MODE bit fields:

Bit	Name	Description
5	FILT_V2	Route Voice 2 through filter
4	FILT_V1	Route Voice 1 through filter
3	FILT_V0	Route Voice 0 through filter
2:0	MODE	Filter mode: `001`=LP, `010`=BP, `100`=HP, `101`=BR

How to test

1. Connect an SPI master to the bidirectional IO pins:
   - `uio[0]` = CS (active low)
   - `uio[1]` = MOSI
   - `uio[2]` = MISO
   - `uio[3]` = SCLK
2. Connect `uo[0]` (PDM output) through a low-pass reconstruction filter (e.g. 4th-order Bessel, cutoff ≈ 20 kHz) to an amplifier or speaker.
3. Program a voice. Minimal example for a 440 Hz sawtooth:
   - Write `0xFF` to `0x1A` (volume = max).

- Write `0x05` to `0x00` and `0x12` to `0x01` (frequency control word for 440 Hz).
- Write `0x00` to `0x05` (attack=0, decay=0) and `0xF0` to `0x06` (sustain=15, release=0).
- Write `0x21` to `0x04` (sawtooth waveform + gate on).

4. A 440 Hz sawtooth tone *should* be playing. Write `0x20` to `0x04` to release the note.

The project also includes a CocoTB test suite that runs automatically via GitHub Actions on both RTL and the synthesized gate-level netlist.

External hardware

- SPI master.
- Low-pass 4th order reconstruction filter for the PDM output (Ideally Bessel).
- Audio amplifier and speaker/headphones.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	pdm	cs_n
1	—	—	mosi
2	—	—	miso
3	—	—	sclk
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

2048 sliding tile puzzle game (VGA)

by **Uri Shaked**

0485 | 25.175 MHz | HDL Project

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

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

How it works

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

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

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

How to test

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

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

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

D-pad	Face button	Direction
Up	X	Up

Down	B	Down
Left	Y	Left
Right	A	Right

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

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

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

External hardware

- TinyVGA Pmod
- Optional: Gamepad Pmod

Project Pinout

Digital Pins

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

OCP MXFP8 Streaming MAC Unit

by **Olivier Chatelain**

`489` `20 MHz` `HDL Project`

`github.com/chatelao/ttihp-fp8-mul`

"Streaming MAC unit supporting OCP MXFP8 (E4M3/E5M2) with shared scaling"

How it works

The **OCP MXFP8 Streaming MAC Unit** is a high-performance, area-optimized arithmetic core designed for AI inference acceleration. It implements the **OpenCompute (OCP) Microscaling Formats (MX) Specification v1.0**, supporting a wide range of sub-8-bit floating-point and integer formats with hardware-accelerated shared scaling.

Architectural Overview

The unit is configured in its "Full" edition (2x2 tiles), featuring:

- **Dual-Lane Multiplier**: Parallel processing of operands with support for Vector Packing (FP4).
- **40-bit Aligner & 32-bit Accumulator**: High-precision internal datapath to prevent overflow during long dot-product sequences.
- **Shared Scaling (UE8M0)**: Automatic application of 8-bit exponents (2^{E-127}) to element blocks.
- **Flexible Rounding**: Support for Truncate (TRN), Ceil (CEL), Floor (FLR), and Round-to-Nearest-Even (RNE).
- **Mixed Precision**: Independent format control for Operand A and Operand B within a single MAC block.
- **Logarithmic Multiplier (LNS)**: Optional area-optimized path using Mitchell's Approximation to reduce multiplier area by >50%.

Streaming Protocol

To maintain a minimal IO footprint (8-bit ports), the unit uses a **41-cycle streaming protocol** to process a block of 32 elements ($k = 32$).

Cycle	Input `ui_in[7:0]`	Input `uio_in[7:0]`	Output `uo_out[7:0]`	Description
0	**Metadata 0**	**Metadata 1**	0x00	**IDLE**: Load MX+ / Debug or Start Fast Protocol.

1	**Scale A**	**Format A / BM A**	0x00	Load Scale A, Format A, and BM Index A.
2	**Scale B**	**Format B / BM B**	0x00	Load Scale B, Format B, and BM Index B.
3-34	**Element A_i**	**Element B_i**	0x00	Stream 32 pairs of elements.*
35-36	-	-	0x00	Pipeline flush & final scaling.
37-40	-	-	**Result [31:0]**	Serialized 32-bit result (MSB first).

**Note: In Packed Mode (*`uio_in[6]=1` *in Cycle 0), the STREAM phase is reduced to 16 cycles (Cycles 3-18).*

Register Layouts

The unit captures configuration and scaling data during the first three cycles of the protocol.

Cycle 0: Metadata 0 (`ui_in`)

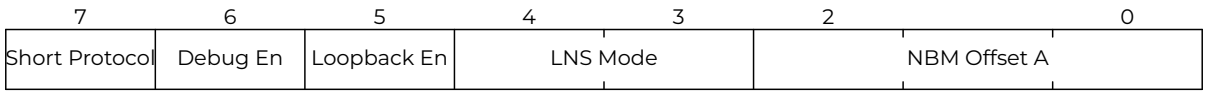


Figure 489.1: Metadata 0

- **Short Protocol (`[7]`)**: 1: Reuse previous scales/formats; immediately jump to Cycle 3.
- **Debug En (`[6]`)**: 1: Enable internal probing and metadata echo at the end of the block.
- **Loopback En (`[5]`)**: 1: Direct input-to-output mapping for physical connectivity testing.
- **LNS Mode (`[4:3]`)**:
  - `0`: Normal (Exact IEEE-like multiplication).
  - `1`: LNS (Logarithmic Number System using Mitchell's Approximation).
  - `2`: Hybrid (Standard for Block Max elements, LNS for all others).
- **NBM Offset A (`[2:0]`)**: (Standard Start only) Exponent offset for non-Block Max elements in Operand A (MX++).

Cycle 0: Metadata 1 (`uio_in`)

Figure 489.2: Metadata 1

- **MX+ Enable (`[7]`)**: 1: Enable OCP MX+ extensions (Repurposed exponents and Block Max tracking).
- **Packed Mode (`[6]`)**: 1: Enable Vector Packing for 4-bit formats (2 elements per byte, Cycles 3-18).
- **Overflow Mode (`[5]`)**: 0: SAT (Saturate to Max/Min), 1: WRAP (Modulo arithmetic).
- **Rounding Mode (`[4:3]`)**:
  - `0`: TRN (Truncate/Towards Zero).
  - `1`: CEL (Ceil/Towards $+\infty$).
  - `2`: FLR (Floor/Towards $-\infty$).
  - `3`: RNE (Round-to-Nearest-Ties-to-Even).
- **NBM Offset B / Format A/B (`[2:0]`)**:
  - **Standard Start**: NBM Offset B (Exponent offset for Operand B).
  - **Short Protocol**: Combined Format A & B selection.

Cycle 1: Scale A (`ui_in`) & Config A (`uio_in`)
Scale A (`ui_in[7:0]`):

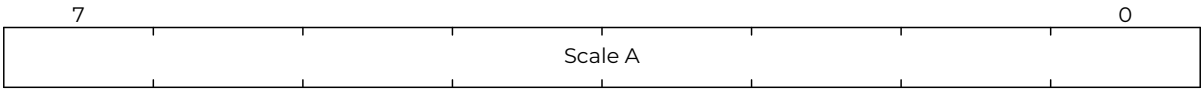


Figure 489.3: Scale A

- **Shared Scale A**: 8-bit unsigned biased exponent (UE8M0, Bias 127) applied to all elements in Operand A.

Config A (`uio_in[7:0]`):

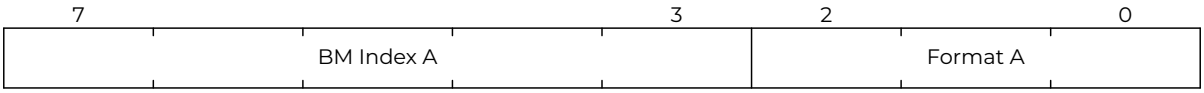


Figure 489.4: Config A

- **BM Index A (`[7:3]`)**: The index (0-31) of the "Block Max" element in Operand A (used in MX+ mode).
- **Format A (`[2:0]`)**:
  - `0`: E4M3, `1`: E5M2, `2`: E3M2, `3`: E2M3, `4`: E2M1, `5`: INT8, `6`: INT8_SYM.

Cycle 2: Scale B (`ui_in`) & Config B (`uio_in`)
Scale B (`ui_in[7:0]`):

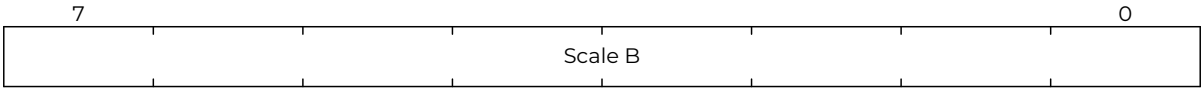


Figure 489.5: Scale B

- **Shared Scale B**: 8-bit unsigned biased exponent (UE8M0, Bias 127) applied to all elements in Operand B.

Config B (`uio_in[7:0]`):

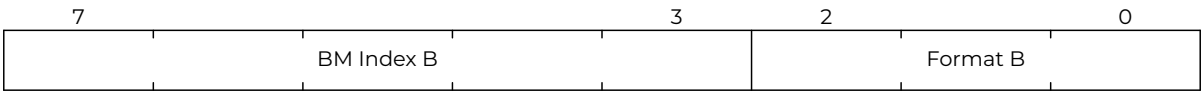


Figure 489.6: Config B

- **BM Index B (`[7:3]`)**: The index (0-31) of the "Block Max" element in Operand B.
- **Format B (`[2:0]`)**: Independent format for Operand B (Enabled if `SUPPORT_MIXED_PRECISION=1`).

How to test

Basic Verification

1. **Reset**: Pulse `rst_n` low, then set `ena` high.
2. **Configuration**:
   - Cycle 0: Provide `0x00` on both `ui_in` and `uio_in` for standard E4M3 mode.
   - Cycle 1: Provide `0x7F` (1.0 scale) on `ui_in` and `0x00` (E4M3) on `uio_in`.
   - Cycle 2: Provide `0x7F` (1.0 scale) on `ui_in` and `0x00` (E4M3) on `uio_in`.
3. **Data Streaming**:
   - Cycles 3-34: Provide 32 pairs of values. E.g., `0x38` (1.0 in E4M3) on both ports.
4. **Result**:
   - Cycles 35-36: Wait for internal processing.
   - Cycles 37-40: Read the 32-bit signed fixed-point result on `uo_out`.
   - For 32 pairs of 1.0×1.0, the result should be `0x00002000` (representing 32.0 in the system's 8-bit fractional format).

Advanced Modes

- **Short Protocol**: Set `ui_in[7]=1` in Cycle 0 to bypass scale loading. Useful for weight-stationary kernels where scales and formats remain constant across blocks.
- **Vector Packing**: Set `uio_in[6]=1` in Cycle 0. Stream two 4-bit elements per byte (High nibble = Element $i+1$, Low nibble = Element i).

External hardware

- **Tiny Tapeout DevKit**: The easiest way to interface with the chip. Use the provided MicroPython driver (`test/TT_MAC_RUN.PY`) for quick prototyping.
- **Sipeed Tang Nano 4K**: For high-speed testing, a dedicated FPGA bitstream and Cortex-M3 testbench are provided in the repository.

IO

Port	Name	Description
`ui_in[7:0]`	Operand A / Scale A	Elements A_i or Scale X_A.
`uio_in[7:0]`	Operand B / Scale B	Elements B_i or Scale X_B.
`uo_out[7:0]`	Result Out	Serialized 32-bit dot product result.
`clk`	Clock	System clock (Target: 20MHz).
`rst_n`	Reset	Active-low asynchronous reset.
`ena`	Enable	Clock enable.

Appendix: OCP MX+ Mathematics

The OCP MX+ extension optimizes quantization by preserving high-precision "outliers" (Block Max elements) while maintaining a low bit-width for the rest of the block.

1. Base OCP MX Mathematics (Standard)

For a block of k elements, the value of an element A_i is given by:

$$V(A_i) = S \cdot M_i \cdot 2^{X_A - 127}$$

Where:

- S: Sign bit (± 1).
- M_i: Mantissa (significand), including an implicit leading bit for subnormals.
- X_A: Shared 8-bit scale (UE8M0).
- E_i: Individual element exponent (for FP8/FP6/FP4 formats).

2. OCP MX+ (Extended Mantissa)

When `MX+ Enable` is set, the **Block Max (BM)** element—identified by `BM Index`—repurposes its exponent bits as additional mantissa.

Normal Element ($i \neq BM$): Decoded as standard MXFP (e.g., E4M3).

Block Max Element ($i = BM$):

- **Exponent**: Fixed to E_{max} for the selected format.
- **Mantissa**: The original exponent bits are appended to the mantissa field.

- **Benefit**: For FP4 (E2M1), the mantissa grows from 1 bit to 3 bits $(1 + 2)$, reducing quantization error for the most critical value by up to 10x.

3. OCP MX++ (Decoupled Shared Scaling)

MX++ allows "Non-Block Max" (NBM) elements to use a finer quantization grid than the BM element by applying a secondary exponent offset.

$$V\left(A_{i \neq BM}\right) = S \cdot M_i \cdot 2^{(X_A - 127) - NBM\_Offset\_A}$$

This effectively "zooms in" on the smaller values in the block, reducing the floor noise caused by a single large outlier.

4. LNS Mitchell's Approximation

In LNS Mode, multiplication $P = A \times B$ is performed in the logarithmic domain:

$$\log_2(P) = \log_2(A) + \log_2(B)$$

To avoid expensive Power/Log circuits, the unit uses **Mitchell's Approximation**:

$$\log_2(1 + m) \approx m, \quad m \in [0, 1)$$

The product of two significands $(1 + m_a)$ and $(1 + m_b)$ is approximated as:

```
(1+m_a)(1+m_b) \approx \begin{cases} 1 + m_a + m_b & \text{if } m_a + m_b < 1 \\ 2(m_a + m_b) & \text{if } m_a + m_b \ge 1 \end{cases}
```

This allows the multiplier to be replaced by a simple adder and a shift, reducing hardware area by over 50%.

Thank you!

A massive thank you to **Matt Venn**, **Uri Shaked**, **Sophie**, and the entire **Tiny Tapeout / IHP** community for making open-source silicon a reality. This project was built on the foundation of your incredible tools and dedication.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	data_in_a[0]	data_out[0]	data_in_b[0]
1	data_in_a[1]	data_out[1]	data_in_b[1]
2	data_in_a[2]	data_out[2]	data_in_b[2]
3	data_in_a[3]	data_out[3]	data_in_b[3]
4	data_in_a[4]	data_out[4]	data_in_b[4]
5	data_in_a[5]	data_out[5]	data_in_b[5]

#	Input	Output	Bidirectional
6	data_in_a[6]	data_out[6]	data_in_b[6]
7	data_in_a[7]	data_out[7]	data_in_b[7]

Tiny NPU: 4-Way Parallel INT8 Inference Engine

by **Malik**

493 | 50 MHz | HDL Project

`github.com/malikweren/ttihp-malik-tiny-npu`

"4-way parallel INT8 neural network inference engine with SproutHDL arithmetic units and ReLU activation"

Tiny NPU: 4-Way Parallel INT8 Inference Engine

How it works

A minimal neural processing unit with 4 parallel multiply-accumulate datapaths that computes a single fully-connected layer: **y = ReLU(W · x + b)**.

The NPU integrates custom arithmetic units generated by SproutHDL through ML-guided design-space exploration:

- **4× Han-Carlson multipliers** (8-bit unsigned, structurally optimized)
- **4× Sparse Kogge-Stone adders** (24-bit two's complement, area-optimized)

Since the multipliers are unsigned but the NPU handles signed INT8 values, a lightweight sign-management wrapper computes absolute values, multiplies, and conditionally negates the result. A pipeline register between the multiply and accumulate stages ensures clean timing closure.

Specifications:

- 4 parallel datapaths (one per output neuron)
- Weight storage: 32 × INT8 register file (4 outputs × 8 inputs)
- Bias storage: 4 × INT16
- Input buffer: 8 × INT8 activations
- 24-bit accumulator precision per output
- Configurable: 1–8 inputs, 1–4 outputs
- ReLU activation with INT8 output saturation
- Pipelined inference: N_IN + 3 cycles (8 inputs → 11 cycles @ 50 MHz = 220 ns)
- Weights persist across inferences for batch processing

How to test

1. **Reset**: `rst_n` low → high
2. **Config**: cmd=0x1, data = `{n_out-1}[5:4] | {n_in-1}[2:0]`
3. **Load weights** row-major: cmd=0x2, data=weight (auto-increments)
4. **Load biases**: cmd=0x3, data=bias (auto-increments)

5. **Load activations**: cmd=0x4, data=act (auto-increments)
6. **Run**: cmd=0x5. Wait for busy→0.
7. **Reset pointers**: cmd=0x7
8. **Read outputs**: cmd=0x6 (auto-increments)

Design context

The arithmetic units were produced by SproutHDL (github.com/huawei-csl/sprout-hdl) as part of a semester project on ML-guided design-space exploration of AI hardware architectures. The Han-Carlson multiplier and sparse Kogge-Stone adder represent Pareto-optimal points on the area-delay trade-off frontier identified through automated exploration.

External hardware

None required.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	data_in[0]	data_out[0]	cmd[0]
1	data_in[1]	data_out[1]	cmd[1]
2	data_in[2]	data_out[2]	cmd[2]
3	data_in[3]	data_out[3]	cmd[3]
4	data_in[4]	data_out[4]	—
5	data_in[5]	data_out[5]	—
6	data_in[6]	data_out[6]	—
7	data_in[7]	data_out[7]	—

2x2 Systolic array with DFT and bfloat16 - v2

by **Julia Desmazes**

497 | 100 MHz | HDL Project

`github.com/Essenceia/Systolic_Array_with_DFT_v2`

"Second version of the 2x2 systollic array with DFT intrastructure, including bfloat16 floating point arithmetic and DFT scan chain."

Multiply and accumulate matrix multiplier ASIC with design for test infrastructure

ASIC design for a 2x2 systolic matrix multiplier supporting multiply and accumulate operations on bfloat16 data alongside a design for test infrastructure to help debug both usage and diagnose design issues in silicon.

Pinout

This accelerator uses the following pinout:

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

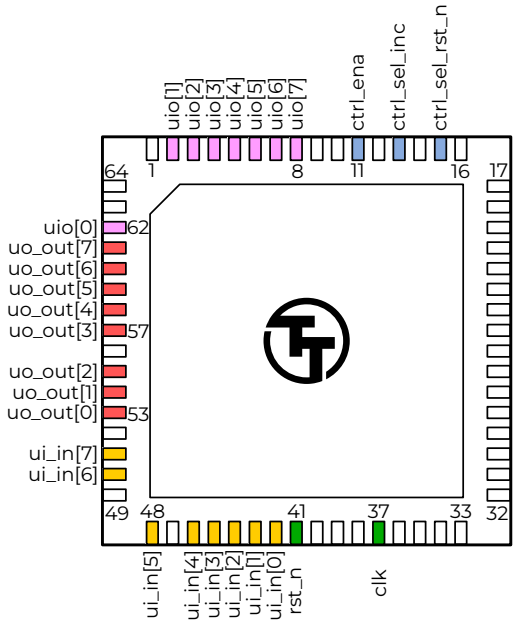


Figure 497.1: Chip pinout

MAC

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

⚠ The outgoing data path on IHP sg13g2 chips has only been proven upwards of 75MHz, so although we will using the maximum theoretical frequency in this discussion it might be necessary to drive this ASIC as a lower clock frequency in practice.

Background

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

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

\times

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

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

=

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

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

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

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

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

Given this accelerator was designed to operate on 16-bit floating point numbers, there is no need for an additional clamping step.

Our final full MAC operation is as follows :

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

At each MAC timestep $t+1$:

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

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

Throughput

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

IO Bottleneck

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

This limitation means our accelerator is actually operating at a quarter of the maximum capacity due to this IO bottleneck. If the IO interface were either (a) at least 32 bits wide, or (b) 8 bits wide but operating at 400 MHz, resolving this bottleneck, our maximum throughput would be 400 MMAC/s or 800 MFLOPS/s.

Usage

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

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

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

Given our input and output data buses are only 8 bits wide, for data transfers to and from the chip the matrices are flattened in the following order, with bytes transfered in little endian:

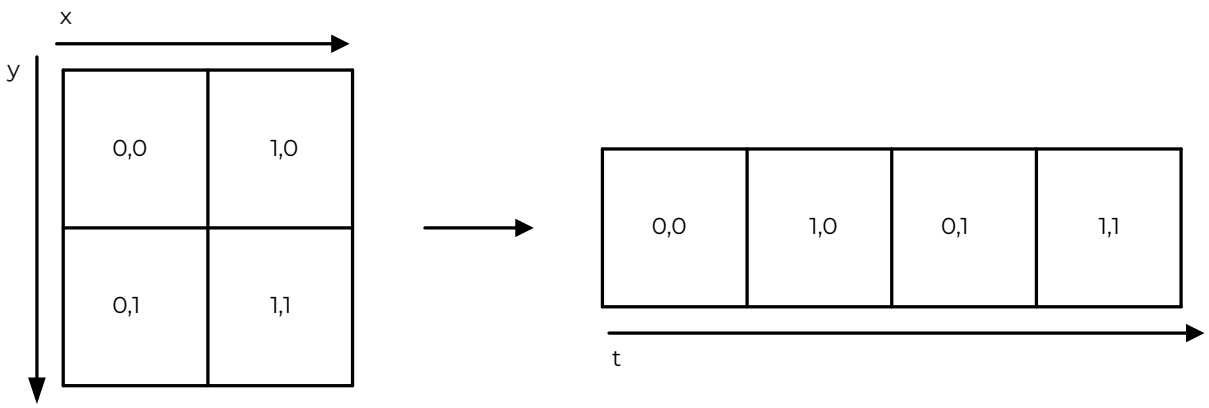


Figure 497.2: flat

Notes:

- All references to `cycles` below are clocked according to the `clk` pin.

- Empty cycles, as in one or more cycles where `data_v_i` would go low in the middle of the transfer of both the input matrix and the weights, are supported.

Resetting MAC

Given we are not sending an index alongside each data transfer to indicate which weight/data coordinates (index) each data corresponds to, the MAC accelerator keeps track of the next index internally. As such, if due to external reasons a partial transfer occurs, it becomes necessary to reset this index using the reset sequence described below.

The weights streaming indexes and the data streaming indexes can be reset independently, each requires a single data transfer cycle during which :

- `data_v_i` is set to `1`
- `data_mode_i[1:0]` is set to `0x3` if we are resetting both the weight and the data indexes
- `data_i[7:0]` is ignored

Example

In this example we are resetting both the data streaming index and the weight index back to back.

TODO rst_waves.png

Configure weights

Configuring the weights takes 8 data transfer cycles, during which :

- `data_v_i` is set to `1`
- `data_mode_i[1:0]` is set to `0x0` indicating we are sending `weights`
- `data_i[7:0]` contains the weights

Example

In this example we are configuring the weight matrix W to :

```
W =
\begin{pmatrix}
0 & 1 \\
2 & 3
\end{pmatrix}
```

TODO wr_weights_waves.png

Debug

The implemented JTAG TAP can be used to easily debug the weight matrix configuration sequence as it allows the user using the `USER_REG` instruction to read the currently configured weights for each MAC unit.

In the existing openocd helper scripts located at `jtag/openocd.cfg` the `read_user_reg` can be used to read the weights using openocd when used as follows :

```
set r 0
for {set u 0} {$u <= $USER_REG_UNIT_MAX} {incr u} {
    puts "read internal register $u : 0x[read_user_reg $_CHIPNAME
$u $r] - [print_reg_id $r]"
}
```

For the W weight matrix configured in the example above, the expected output should be :

```
read internal register 0:0 : 0x0000 - weight
read internal register 0:1 : 0x0000 - multiplicand ( input data )
read internal register 0:2 : 0x0000 - summand ( input data )
read internal register 0:3 : 0x0000 - multiplication result
(internal computation)
```

Sending the input matrix

Sending the input matrix takes 8 data transfer cycles, during which :

- `data_v_i` is set to `1`
- `data_mode_i[1:0]` is set to `0x1` indicating we are sending the input matrix
- `data_i[7:0]` contains the input data

Example

In this example we are sending the input data matrix I :

```
I =
\begin{pmatrix}
4 & 5 \\
6 & 7
\end{pmatrix}
```

TODO wr_data_waves.png

Receiving result

When receiving a result the asic will drive the following pins during 8 data transfer cycles. The transfer is guarantied to be gapeless :

- `res_v_o` is set to `1`
- `res_o[7:0]` contains the result of the MAC operation for a single matrix coordinate

In order to start capture by the pio hardware on the raspberry pi silicon, `res_v_o` is asserted a cycle before the data transfer starts. The two result streams occure back-to-back this will not occur.

Simple example

In this example the W MAC weight matrix is being configured and the I data is being streamed in, following which, the R result starts being sent out.

```
R = I \times W =
\begin{pmatrix}
4 & 5 \\
6 & 7
\end{pmatrix}
\times
\begin{pmatrix}
0 & 1 \\
2 & 3
\end{pmatrix}
=
\begin{pmatrix}
10 & 19 \\
14 & 27
\end{pmatrix}
```

TODO rd_res_waves.png

Complex example

Internally, the accelerator takes at most 8 cycles to produce a result from incoming data. This accounts for incoming data latching, circulating the data through the entire systolic array, and output streaming. The accelerator moves the incoming data through the array as soon as it is available. Because of this, and since this accelerator supports gaps in the incoming data stream, if, for example, the last data transfer of $i_{(1,1)}$ is delayed by at least 2 cycles, then the accelerator result will start streaming out before all of the input matrix has finished streaming in.

This is why, in the firmware (`firmware/main.c`), we set up the DMA stream to receive the data before we start sending the input matrix, as the gap between sending and getting the result is too small for the controlling MCU to perform any type of compute.

TODO rd_res_complex_waves.png

DFT

This design embeds a JTAG for debugging the accelerator's usage by probing into internal registers and helping identify PCB issues using a boundary scan.

This JTAG TAP was designed to operate at `2 MHz`, has idcode `0x2beef0d7`.

Its instruction register length is `3`, and implements the following instructions:

Instruction	Opcode	Description
`EXTEST`	`0x0`	Boundary scan

IDCODE	0x1	Reads JTAG TAP identifier
SAMPLE_PRELOAD	0x2	Boundary scan
USER_REG	0x3	Probe internal registers
SCAN_CHAIN	0x4	Internal logic scan chain
BYPASS	0x7	Set the TAP in bypass mode

All four standard instructions `EXTEST`, `IDCODE`, `SAMPLE_PRELOAD`, `BYPASS` conform to the standard behavior.

`SCAN_CHAIN` is a private JTAG instruction used for observing the systolic array's flops state. The order of the flop chain can be found at the end of the `.def` file in the definition of the `chain_0` scan chain.

USER_REG

The `USER_REG` state was designed to probe into the data currently used by each of the 4 MAC units. The data to be read is specified by loading its address in the data register during a previous `DR_SHIFT` stage. As such, two sequences of `DR_SHIFTS` might be necessary:

1. Load the address of the next data
2. Read the data off TDI

The address and data are both `16` bits wide, though only the bottom 4 bits of the address are used.

Address format

The address uses the following format:

```
[ unused 15:4 ][ mac unit 3:2 ][ register id 1:0 ]
```

Register id mapping for this MAC unit gives us the current:

Register ID	Description
0x0	Weight (multiplier)
0x1	Multiplicand (circulated data)
0x2	Summand (circulated data)
0x3	Multiplication result (internal MAC unit data)

Important considerations for usage

When using the USER_REG custom JTAG TAP instruction, the MAC logic is expected to be temporarily halted, as in no weight or data update operations and no matrix compute is expected to be ongoing. To this effect, there is no CDC protection when transferring data between the JTAG clock domain

and the MAC domain. If the MAC isn't halted, the resulting metastability risks corrupting the sampled data.

This also applies when doing a boundary scan.

Quickstart

For quickly getting started, use the utilities provided in `jtag/openocd.cfg`.

Given this default config assumes you are using a `jlink`, and this might not be the adapter you are using, you may need to update the adapter sourcing your current probe:

```
source [find interface/jlink.cfg]
```

Usage

Run using :

```
openocd -f jtag/openocd.cfg
```

Expected output:

```
Open On-Chip Debugger 0.12.0+dev-02429-ge4c49d860
(2026-03-17-19:44)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.org/doc/doxygen/bugs.html
Info : J-Link V10 compiled Jan 30 2023 11:28:07
Info : Hardware version: 10.10
Info : VTarget = 3.348 V
Info : clock speed 2000 kHz
Info : JTAG tap: tpu.tap tap/device found: 0x2beef0d7 (mfg: 0x06b
(Transwitch), part: 0xbeef, ver: 0x2)
Warn : gdb services need one or more targets defined
idcode : 2beef0d7
read internal register 0:0 : 0x0000 - weight
read internal register 0:1 : 0x0000 - multiplicand ( input data )
read internal register 0:2 : 0x0000 - summand ( input data )
read internal register 0:3 : 0x0000 - multiplication result
(internal computation)
read internal register 1:0 : 0x0000 - weight
read internal register 1:1 : 0x0000 - multiplicand ( input data )
read internal register 1:2 : 0x0000 - summand ( input data )
read internal register 1:3 : 0x0000 - multiplication result
(internal computation)
read internal register 2:0 : 0x0000 - weight
read internal register 2:1 : 0x0000 - multiplicand ( input data )
read internal register 2:2 : 0x0000 - summand ( input data )
read internal register 2:3 : 0x0000 - multiplication result
(internal computation)
read internal register 3:0 : 0x0000 - weight
read internal register 3:1 : 0x0000 - multiplicand ( input data )
```

```
read internal register 3:2 : 0x0000 - summand ( input data )
read internal register 3:3 : 0x0000 - multiplication result
(internal computation)
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
...
```

Project Pinout

Digital Pins

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

VGA Pride

by **Rebecca G. Bettencourt**

499 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

2 digit minute timer

by **Anna V**

0512 HDL Project

github.com/anna-vee/tinytapeout-timer-two-digit-

"countdown timer in minutes"

How it works

Once the switch is flipped, the display decreases by 1 with every 6000000 ticks of the 100khz clock, or 1 minute, until it reaches 0. ## How to test Access The Wokwi Simulation With This Link: https://wokwi.com/projects/455966163101749249 Make sure the switch is turned off. Then press the button until the display shows the desired number of minutes you want to time. Then flip the switch and watch it count down to 0! ## External hardware The circuit uses a 2+ digit common cathode seven seg display, a button, and a 3 pin switch.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	clk	a	dig1
1	button	b	dig2
2	switch	c	—
3	—	d	—
4	—	e	—
5	—	f	—
6	—	g	—
7	—	—	—

TinyTapeout Signal Box

by **Pascal**

514 Wokwi Project

github.com/Nprom2/ttihp26a-signal-box

wokwi.com/projects/459234034322375681

"Signal Box v1"

How it works

Enter the desired track with the dip-switches, press RST to load the switch signals into the SRAMs.

How to test

The design should be ready out of the box.

External hardware

No external hardware required.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	q0_i	s1_o	x_o
1	q1_i	s2_o	a_o
2	q2_i	s3_o	b_o
3	q3_i	s4_o	c_o
4	q4_i	s5_o	d_o
5	q5_i	s6_o	ef_o
6	q6_i	s7_o	gh_o
7	q7_i	s8_o	ij_o

Flying Fish

by **Steven Cheng**

0516 25.175 MHz HDL Project

github.com/Fteve/Tiny-Tapeout-Submission-FS2026

"*Fish swims across the screen*"

How it works

This project works the same as the "logo" example in the VGA Playground. The image of the fish bounces around the screen.

How to test

To test this project, connect the chip to a screen with a VGA connection, and load this 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	—

Multi-Tool SoC

by **Mauro Ciccone**

518 | 50 MHz | HDL Project

github.com/mauro-ciccone/tt-multitool-soc

"An alphabet-driven, multi-state OS featuring hardware engines for Euclidean GCD, Integer Square Root, and a Cryptographic LFSR"

How it works

The chip operates via a custom alphabet-driven Operating System. ui_in[7] acts as the "Enter" switch (with built-in hardware debouncing and edge-detection), and ui_in[6:0] act as the 7-bit data inputs.

Select Opcode: When the display shows 'S' (Select), set the data input switches to your desired coprocessor: 0 for GCD, 1 for ISQRT, or 2 for LFSR. Flip the Enter switch UP to submit, and back DOWN to reset the edge-detector.

Load Value A: The display will change to 'A'. Set the data inputs to your first value and flip Enter (UP then DOWN).

Load Value B: The display will change to 'B'. Set the data inputs to your second value (or steps/seed for the LFSR) and flip Enter (UP then DOWN).

Read Output: The OS will display a brief dash (-) while calculating, and then transition to the output state. Because the answer is 8-bit (up to 255), the 7-segment display acts as a multiplexer, repeatedly flashing the Hundreds, Tens, and Units digits for 1s each. The Decimal Point (DP) lights up to indicate the most significant digit (Hundreds) so one can know where the number starts.

How to test

To test the chip physically on the Tiny Tapeout demo board, we can run an example GCD calculation, for example, GCD(12, 8) = 4.

Reset: Assert the rst_n pin low, then high, to boot the chip. The display should show an 'S'.

Opcode: Leave all data switches ui_in[6:0] DOWN (Value = 0 for GCD). Flip the Enter switch ui_in[7] UP, then DOWN.

Input A: The display now shows 'A'. Set the data switches to 12 (binary 0001100). Flip Enter UP, then DOWN.

Input B: The display now shows 'B'. Set the data switches to 8 (binary 0001000). Flip Enter UP, then DOWN.

Observe: The display will multiplex the answer 004. Look for the digit with the Decimal Point turned ON, that is the leading zero (Hundreds).

Exit: Flip the Enter switch one last time to exit the result screen and return to the ‘S’ menu for your next calculation.

(Note: If running at a very slow physical clock speed, you can pull uio_in[7] HIGH to engage the Fast-Forward Test Mode, which drastically speeds up the FSM delay timers!)

External hardware

No external hardware is required. The design uses the standard Tiny Tapeout demo board’s built-in dip switches and 7-segment display.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Data Bit 0 (Opcode / Val A / Val B)	Segment A	—
1	Data Bit 1 (Opcode / Val A / Val B)	Segment B	—
2	Data Bit 2 (Opcode / Val A / Val B)	Segment C	—
3	Data Bit 3 (Opcode / Val A / Val B)	Segment D	—
4	Data Bit 4 (Opcode / Val A / Val B)	Segment E	—
5	Data Bit 5 (Opcode / Val A / Val B)	Segment F	—
6	Data Bit 6 (Opcode / Val A / Val B)	Segment G	—
7	Enter / Submit Switch (Debounced)	Decimal Point (Digit Multiplexer)	Test Mode (Fast-Forward Timer Bypass)

8-bit Prime Number Detector

by **Niklas Oberhuber**

`0520` `HDL Project`

`github.com/Maximillian-Udar/ttihp-verilog-template`

"Set 8 DIP switches to any number 0-255. The 7-segment display shows its hex digit; the decimal point lights up if the number is prime."

How it works

This is a purely combinational 8-bit prime number detector. It tests every number from 0 to 255 in hardware using a 256-bit lookup table (one bit per number, pre-computed at synthesis time). The result is available with zero latency — no clock required.

The 7-segment display shows the lower hex nibble of the input number (0–F), and the **decimal point lights up when the number is prime**. There are 54 primes in the range 0–255, from 2 up to 251.

The entire design uses 100 logic gates: a 256-bit ROM for the prime lookup and a 16-entry hex decoder for the display.

How to test

1. Set `ui[7:0]` to any number using the DIP switches.
2. Read the lower nibble from the 7-segment display (0–F).
3. **Decimal point lit = prime. Decimal point off = not prime.**

Try scanning through numbers by flipping switches — you can find all 54 primes between 0 and 255by watching the decimal point.

Some interesting numbers to try:

Number	Prime?	Notes
2	yes	Smallest prime; only even prime
7	yes	—
9	no	3 × 3 — a common mistake
127	yes	Mersenne prime ($2^7 - 1$)
128	no	2^7
251	yes	Largest prime ≤ 255
255	no	3 × 5 × 17

External hardware

None required. The 7-segment display is built into the TinyTapeout demo board.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Number bit 0 (LSB)	Segment a	—
1	Number bit 1	Segment b	—
2	Number bit 2	Segment c	—
3	Number bit 3	Segment d	—
4	Number bit 4	Segment e	—
5	Number bit 5	Segment f	—
6	Number bit 6	Segment g	—
7	Number bit 7 (MSB)	Decimal point (1 = prime)	—

4-bit ALU

by **Faultierschnegg**

0522 Wokwi Project

github.com/Faultierschnegg/Tiny-Tapeout-4-bit-ALU

wokwi.com/projects/459117403524075521

"Simple 4-bit ALU, adding/subtracting 4-bit numbers, AND/OR operation on 4-bit numbers"

How it works

A 4-bit ALU designed in Wokwi. Inputs two 4-bit numbers and the desired operation (Add/Subtract/AND/OR) LEDs reflect output.

How to test

Just a simple ALU input numbers and desired operation and check with master solution.

External hardware

10x 10kOhm Resistors 5x 1kOhm Resistors 5x LEDs 2x 8 dip switch

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	A0	S0	Mode1
1	A1	S1	Mode2
2	A2	S2	—
3	A3	S3	—
4	B0	Overflow	—
5	B1	—	—
6	B2	—	—
7	B3	—	—

Tiny_ECC

by **Prakhar Pranav** & **Vinay Kumar**

0524 HDL Project

`github.com/prakh9361/tt0x-TIny_ECC`

"private to public key for eliptical curve cryptography"

How it works

Tiny_ECC implements scalar point multiplication on a binary elliptic curve over GF(2^8), the core operation needed to derive a public key from a private key.

Curve Parameters

Parameter	Value
Field	GF(2^8), irreducible poly `0x11B` (AES)
Curve equation	$y^2 + xy = x^3 + 0x20 \cdot x^2 + 0x01$
Base point G	(0x8F, 0x29) i.e. (143, 41)
Group order n	113
Cofactor h	2

Architecture

The design is split across three modules:

- **`tt_um_ecc_gf2_8` (project.v)** — Top-level wrapper. Maps the 8-bit `ui_in` data bus and `uio_in` control pins to the core. Muxes `result_x`/`result_y` to `uo_out` via `read_sel`.
- **`ecc_core_gf2_8`** — FSM-based scalar multiplier using the double-and-add algorithm. Iterates over bits of `k` from MSB to LSB: for each bit it performs a point double, then conditionally a point add if the bit is 1. Point-at-infinity is tracked with the `r_is_inf` flag.
- **`alu_gf2_8`** — Combinational GF(2^8) ALU with three operations:
  - `op=00`: Addition (XOR)
  - `op=01`: Squaring (a^2)
  - `op=10`: Multiplication (a·b), shift-and-XOR with poly reduction
- **`inv_rom`** — 256-entry lookup table returning the multiplicative inverse of any GF(2^8) element. Used by both the doubling and addition formulas to compute λ.

Point Doubling (R ≠ O)

```
λ = xr + yr · xr⁻¹
x3 = λ² + λ + a
y3 = xr² + (λ)·x3 + x3          (where a = 0x20)
```

Point Addition (R ≠ G, R ≠ O)

```
λ = (yr + yg) · (xr + xg)⁻¹
x3 = λ² + λ + xr + xg + a
y3 = λ·(xr + x3) + x3 + yr
```

Each formula takes multiple clock cycles; intermediate values are stored in `lam`, `temp`, and `x1_saved`.

Latency

Each scalar multiplication takes roughly **30–60 clock cycles** for 8-bit keys (k = 1..112), depending on the Hamming weight of k.

How to test

Loading inputs (serial byte protocol)

All data is loaded 8 bits at a time via `ui_in`. Assert the corresponding `uio_in` control line for **one clock cycle** while the data is present on `ui_in`:

Signal	`uio_in` bit	Action
`load_k`	0	Loads private key k into register
`load_x`	1	Loads base point X coordinate (Gx)
`load_y`	2	Loads base point Y coordinate (Gy)
`start`	3	Begins scalar multiplication

Standard key generation sequence

```
1. Assert rst_n = 0 for ≥5 cycles, then rst_n = 1
2. ui_in = k,  uio_in = 0x01  → wait 1 cycle
3. ui_in = Gx, uio_in = 0x02  → wait 1 cycle
4. ui_in = Gy, uio_in = 0x04  → wait 1 cycle
5. uio_in = 0x08 (start)      → wait 1 cycle, then uio_in = 0
6. Poll `uio_out[6]` (`busy`) — wait for it to go **low**.
7. Set uio_in[4] = 0, read uo_out → public key X
8. Set uio_in[4] = 1, read uo_out → public key Y
```

Status flags (`uio_out`)

Bit	Signal	Description
5	`done`	Pulses high for **one clock cycle** when result is ready. Do not poll this pin from software — the pulse will be missed.

6	busy	High while computation is in progress. **Poll this pin instead** — wait for it to go low, then read the result.
7	error	High if k=0 or result is the point at infinity. Sample after busy goes low.

Example: Generating a public key

```
Private key:  k  = 42  (0x2A)
Base point:   Gx = 143 (0x8F), Gy = 41 (0x29)
Expected public key: run scalar_mult(42, 143, 41) in the Python
golden model
```

Diffie-Hellman shared secret

Because this is a general scalar multiplier, you can substitute any valid EC point as the base. Load Alice's public key as (Gx, Gy) and Bob's private key as k to derive the shared secret. The cocotb test `test_diffie_hellman` demonstrates this end-to-end.

Running the cocotb testbench

```
cd test
make          # RTL simulation
# Results and waveform saved to test/tb.fst and test/results.xml
```

Three tests are included:

- `test_k_zero` — verifies error flag for invalid input
- `test_exhaustive_scalar_mult` — brute-forces all k = 1..112 against a Python golden model
- `test_diffie_hellman` — full ECDH key exchange between Alice (k=42) and Bob (k=37)
- `test_peer_public_key_as_base` — verifies using a peer's public key as the base point

External hardware

No external hardware required. All computation is on-chip.

For chip-level testing on the TT demo board, a logic analyser or microcontroller is needed to drive the serial byte-load protocol and poll the `done` flag.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	data_in[0]	data_out[0]	input: load_k
1	data_in[1]	data_out[1]	input: load_x

#	Input	Output	Bidirectional
2	data_in[2]	data_out[2]	input: load_y
3	data_in[3]	data_out[3]	input: start
4	data_in[4]	data_out[4]	input: read_sel (0=X, 1=Y)
5	data_in[5]	data_out[5]	output: status_done
6	data_in[6]	data_out[6]	output: status_busy
7	data_in[7]	data_out[7]	output: status_error

IHP Gate Delay Characterizer (3-Flavor)

by **Catalin Lazar**

526 | 10 MHz | HDL Project

`github.com/catalinlazar/tt_ihp_char_osc`

"Characterizes 3 flavors of ring oscillators (Inv1, 2, 4) with 31 stages (30 inv + 1 NAND, prime length) using a 24-bit counter."

IHP Gate Delay Characterizer

How it works

Three 31-stage ring oscillators are implemented using specific IHP SG13G2 standard cells. A 2-bit mux selects the clock source for a 24-bit asynchronous counter. A 10ms sampling window (derived from the 10MHz system clock) captures the counter value, which is then read out in 8-bit slices.

How to test

1. Set rst_n high and ui_in[0] low.
2. Set ui_in[0] high to enable oscillation.
3. Cycle through ui_in[2:1] to select different oscillators.
4. Cycle through ui_in[4:3] to read the Low, Mid, and High bytes of the frequency on uo_out.

External hardware

No external hardware required other than the standard Tiny Tapeout demo board.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	global_en (Master Enable)	data_out[0]	unused
1	f_sel[0] (Flavor Select Bit 0)	data_out[1]	unused
2	f_sel[1] (Flavor Select Bit 1)	data_out[2]	unused
3	b_sel[0] (Byte Select Bit 0)	data_out[3]	unused
4	b_sel[1] (Byte Select Bit 1)	data_out[4]	unused
5	unused	data_out[5]	unused
6	unused	data_out[6]	unused
7	unused	data_out[7]	unused

float_synth

by **Niklaus Leuenberger**

528 | 50 MHz | HDL Project

`github.com/NikLeberg/tt_um_float_synth`

"*Synthesizing the VHDL-2008 IEEE.float_pkg*"

How it works

This project came about with the simple question: *Can we write lazy HDL and let the tools optimize our sloppiness?*

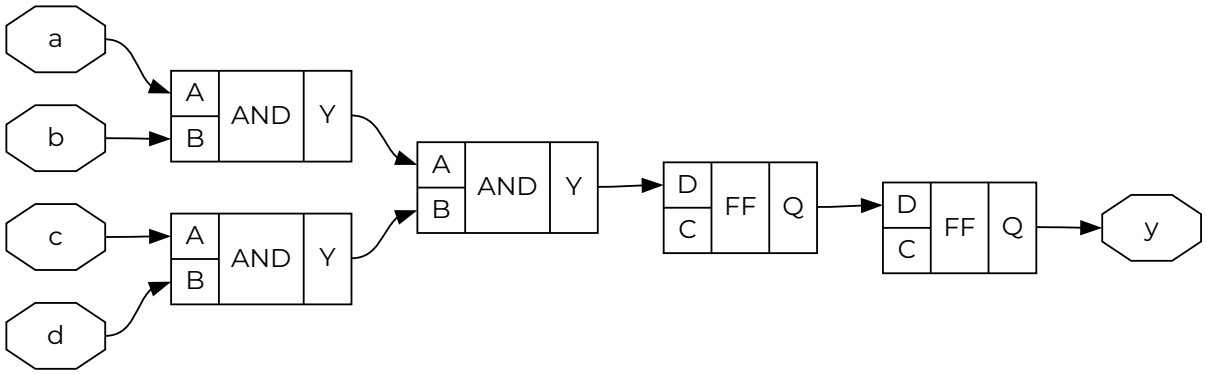
Spoiler: We absolutely can!

The Idea: Retiming

It is very easy to write combinational logic. Ignore the clock, just simply do everything at once, use deep MUX trees, *if else if if else else if* and so on. But that has a cost. The result will have a very long critical path and timing closure will be almost impossible. Sure at clock frequencies of a few *kHz* this is an non-issue. But try to target anything faster and your design will not reach timing closure.

Registers to the rescue! We *could*, like its usually done, partition the design in logical *steps* and add flip-flops inbetween to effectively pipeline the design. Thats everything but easy. Whole new HDL languages have came to be just because this is not easy. See PipelineC for example.

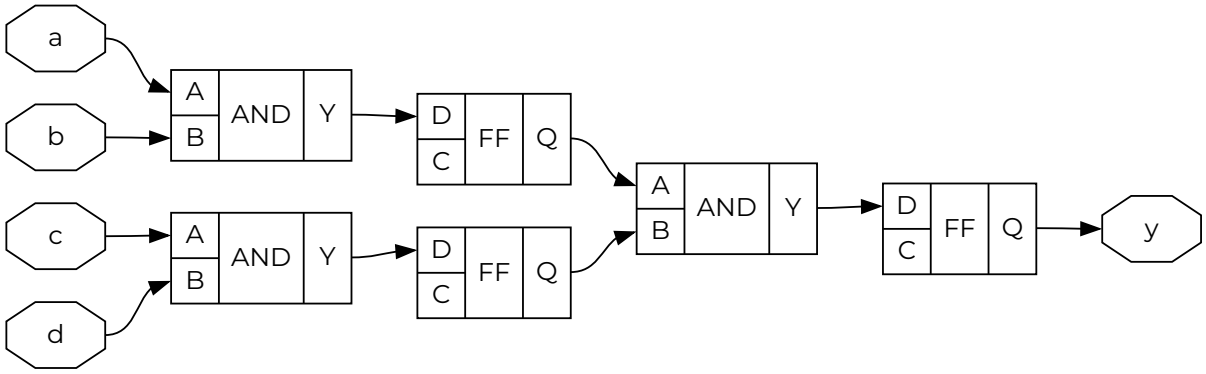
But we are lazy and want to test what the tools can do about it. So we simply add, after our *lazy* design, a few (or many) flip-flops back-to-back like a shift register. That does not change what is computed, but simply introduces latency, i.e. the output is delayed by N clocks. The excellent ABC tool from Alan Mishchenko, which is mostly integrated into Yosys, then does the heavy lifting and *retimes* the flip-flops to wherever it results in the minimal delay.



original

Figure 528.1: Original tree of AND gates

Retiming is the process of moving flip-flops over combinational logic. As a very contrived example consider the above tree of AND gates. At its output there are two flip-flops. We can retime one of the FFs backwards by moving it over the last AND gate and duplicate it for each input. This does not change the behavour to the outside world.



retimed

Figure 528.2: Retimed tree of AND gates

In the retimed design, we now only have a single AND gate in the combinational path from one FF to the other. The original had two. Less logic depth equals a faster possible clock. But of course, there is a drawback. There always is. Instead of only 2 FFs, we now have 3. This means more area, which in ASICs is not free. So YMMV.

> Note that there are other *types* of retiming. Here we discussed retiming for minimal delay. ABC can also retime for minimal area. For the above example this would be the reverse. Merge the FF from the AND gates inputs to a single FF at the output. Usually you want a healthy balance of delay and area. But this is … complicated.

The Usecase: IEEE.float_pkg

Since VHDL-2008, the VHDL IEEE library has contained the excellent `float_pkg` (as well as `fixed_pkg`) from David Bishop. It describes on a high and

generic level how a floating point number works and provides procedures for every mathematical operation.

With it, a fully IEEE compliant floating point multiplier is as simple as:

```
y <= to_float(a) * to_float(b);
```

The `float_pkg` has one major drawback: It is fully combinational.

But with retiming we can get it to work! We simply slap some flip-flops onto the outputs to form a shift register of *N* pipeline stages. With the marvelous FOSSi tools that are GHDL, Yosys and ABC we can then process this *lazy* VHDL into an optimized Verilog netlist that runs at high clock rates!

> For a more in-depth exploration I welcome you to visit the *big brother* repository of this one. In my `NikLeberg/float_synth` project I describe how this retiming approach can work for more floating point operations like adding, dividing and also integer-to-float conversion. It is a work in progress and targets FPGA instead of ASICs like here. But the results are already looking promising. For FPGA targets the lazy HDL style with applied retiming is outperforming hardened vendor IP in some cases.

The Flow Before the Flow

Currently, the Tiny Tapeout LibreLane flow cannot accept custom ABC scripts. I hope to change that in the future. It also works best with Verilog. So for the time being, I choose to do a sort of *pre-synthesis*. The flow is:

1. Analyze the VHDL with GHDL.
2. Load the design into Yosys and run the generic `synth` script.
3. Run the ABC command `retime -M 4 -b` on the design.
4. Export a (sadly illegible) Verilog netlist.

The script that kicks this off is `src/gen/gen.sh` please inspect it for more interesting details.

The configured pipeline depth is 6. With this the LibreLane flow runs fine even for a very high clock frequency of *400 MHz*.

How to test

Well, it simply calculates `y = a * b`, but *fast*.

- Input `a` i.e. `ui_in[7:0]` can be driven from either the demo board DIP switches, PMOD connector or from the TT Commander.
- Input `b` i.e. `uio_in[7:0]` can only be driven from PMOD or TT Commander.
- Output `y` i.e. `uo_out[7:0]` can be observed on the seven segment display. Although the number will not make any sense. It is better to observe it on PMOD or TT Commander. It has a latency of 6 clocks.

As a quick test you may drive `a` and `b` with `0b00110000`, which is *0.5* in float. The result on `y` should be `0b00101000` or *0.25* in float.

The data format is a very limited 8-bit floating point number. Known as `1.4.3` or `E4M3`. Meaning it has 1 sign bit, 4 exponent bits and 3 mantissa bits. It can represent numbers from *−480* to *+480* with varying accuracy.

—	Sign	Exponent	Mantissa
Bits	0	0000	000
`a` mapping	`ui_in[7]`	`ui_in[6:3]`	`ui_in[2:0]`
`b` mapping	`uio_in[7]`	`uio_in[6:3]`	`uio_in[2:0]`
`y` mapping	`uo_out[7]`	`uo_out[6:3]`	`uo_out[2:0]`

To save on resources, the underlying `IEEE.float_pkg` has been configured to:

- round towards zero (truncate)
- saturate on overflow (no infinity)
- but nonetheless: handle subnormals

This effectively results in the following representable number ranges:

Exponent (biased)	Exponent (unbiased)	Range	ULP (Accuracy)
0 (subnormal)	−6 (fixed)	[0.0, 0.013671875]	2^{-9} = 0.001953125
1	−6	[0.015625, 0.029296875]	2^{-9} = 0.001953125
2	−5	[0.03125, 0.05859375]	2^{-8} = 0.00390625
3	−4	[0.0625, 0.1171875]	2^{-7} = 0.0078125
4	−3	[0.125, 0.234375]	2^{-6} = 0.015625
5	−2	[0.25, 0.46875]	2^{-5} = 0.03125
6	−1	[0.5, 0.9375]	2^{-4} = 0.06250
7	0	[1.0, 1.875]	2^{-3} = 0.12500
8	+1	[2.0, 3.75]	2^{-2} = 0.25
9	+2	[4.0, 7.5]	2^{-1} = 0.5
10	+3	[8.0, 15.0]	2^{0} = 1.0
11	+4	[16.0, 30.0]	2^{1} = 2.0
12	+5	[32.0, 60.0]	2^{2} = 4.0
13	+6	[64.0, 120.0]	2^{3} = 8.0

14	+7	[128.0, 240.0]	$2^4 = 16.0$
15	+8	[256.0, 480.0]	$2^5 = 32.0$

Of course all the representable values may also be negative. But they have been omitted here for clarity.

To generate a valid number you can use Spencer Williams Floating Point Number Converter. Setup a custom format with: Sign: *True*, Exponent: *4*, Mantissa: *3*, Has Inf: *False*, Has Nan: *False* and at the very bottom, Rounding Mode: *Toward Zero (truncate)*. Input your desired decimal value and it tells you the binary or hexadecimal representation. You may also fiddle with the bits directly and see what the resulting floating point number is.

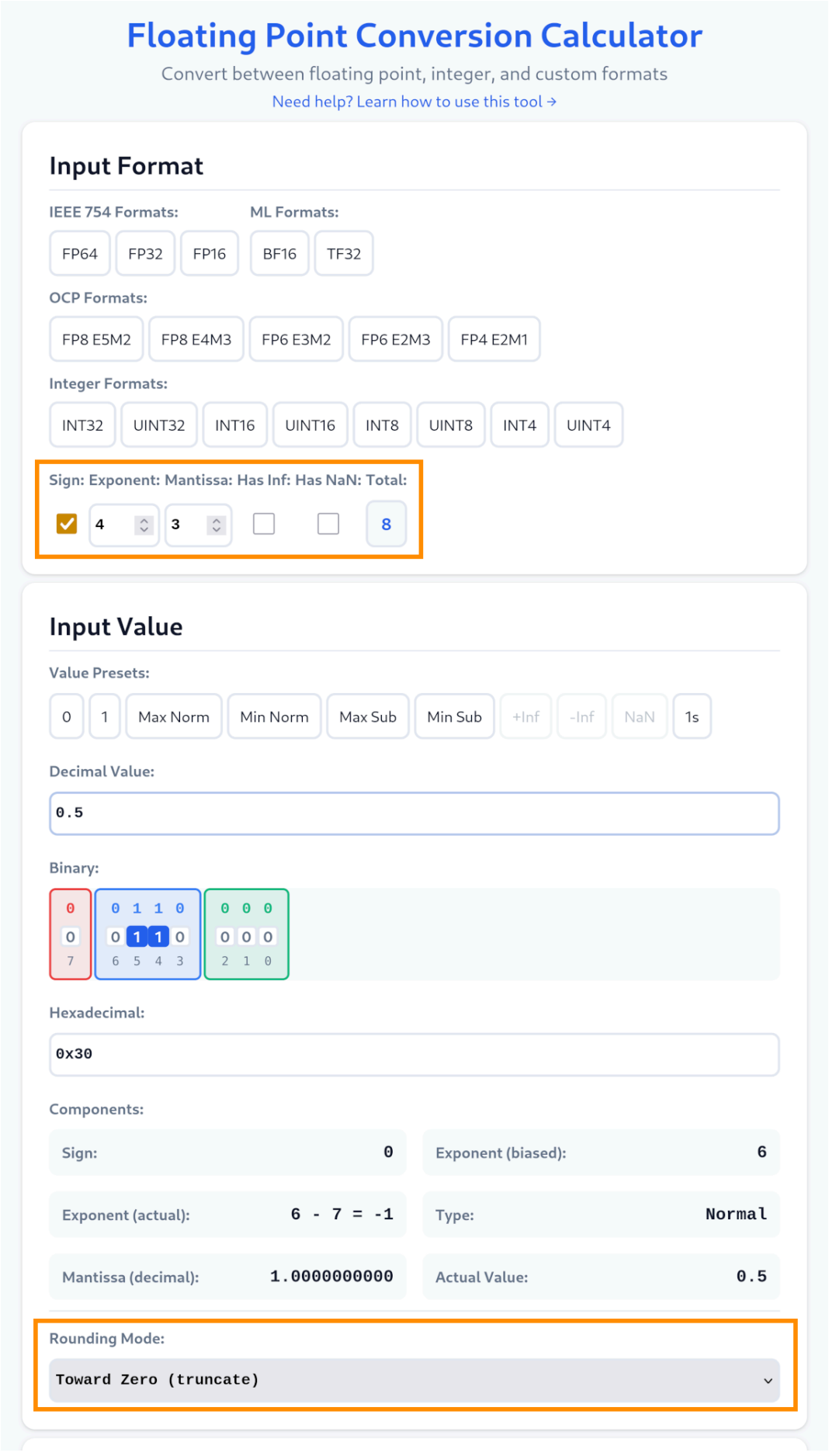


Figure 528.3: Screenshot of settings for Spencer Williams Floating Point Number Converter

As the main goal of the project was to retime the lazily written HDL for optimal delay, the clock can be as high as *400 MHz*. Although I'm not that confident that it will actually work at that speed. Also the poor little IO pads will probably not like that very much. Something like *50 MHz* should be fine. Use way less (or even single clock it) to see the pipelining in action.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	a0	y0	b0
1	a1	y1	b1
2	a2	y2	b2
3	a3	y3	b3
4	a4	y4	b4
5	a5	y5	b5
6	a6	y6	b6
7	a7	y7	b7

badstripes

by **malui**

`0530` `HDL Project`

`github.com/maluethi/badstripes`

"some bad attempt do put out something on the screen"

How it works

makes some stripes or other pattern on the vga channels. don't know this is a good idea.

How to test

Plug it in

External hardware

vga pmod

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	x^2, x^3, x^4, x	—	—
1	x^2, x^3, x^4, x selection	—	—
2	x oscillation on/off	—	—
3	y oscillation on/off	—	—
4	centred_y truncation mode	—	—
5	x osc speed (0=fast, 1=slow)	—	—
6	y osc speed (0=fast, 1=slow)	—	—
7	free	—	—

TeenySPU

by **Eric Shook** & **Logan Gall**

#545 HDL Project

github.com/umn-geocommons/tt_um_teenyspu

"A 4-bit (teeny) Spatial Processing Unit (SPU) that expresses basic raster data, vector data, and geoAI processing."

TeenySPU on TinyTapeout Technical Docs

3.22.26 (Eric Shook, Logan Gall)

We developed a TeenySPU for the TinyTapeout architecture. TinyTapeout includes 8-bit input (INP), 8-bit output (OUT), and 8-bit input/output-swappable (UIO). The TeenySPU is a 4-bit architecture, so the 8-bit INP, OUT, and UIO are split into two 4-bit sections, high and low. The high 4-bits of INP provide the 4-bit opcode, the low 4-bits of INP control a Q MUX to select how input data from UIO is routed to inputs A, B, C, and D. The 4-bit outputs M and N are routed to OUT high and OUT low, respectively. For more details on the TinyTapeout specs, see: https://tinytapeout.com/specs

Hardware Structure

TinySPU is structured to function within the limitations of the TinyTapeout project spec. In general, this means:

- 50MHz Clock rate
- 8 bits of Input (designated to OpCode & QMux)
- 8 bits of Output (designated to M and N outputs)
- 8 bits of UIO (designated to ABCD inputs)

How it works

The TeenySPU uses a spatial instruction set architecture (SISA), which is split into two sections:

- Operation Codes – The operation selection for the TeenySPU chip, set as the high 4 bits of Input
- Q Mux Codes – The data loading multiplexer code, set as the low 4 bits of Input

Spatial Instruction Set Architecture (SISA)

Spatial instructions are organized into categories to demonstrate the breadth of spatial operations available for a prototypical SPU.

Opcode categories

Opcode	Category
00xx	Control SPU Ops
01xx	4-bit Vector Ops
10xx	4-bit Raster Ops
110x	4-bit Multispec Raster Ops
111x	8-bit Double-precision Ops

16 TeenySPU Opcodes

The following 16 spatial opcodes can be used to create a multitude of spatial methods. Details of each op are included below.

Opcode	Mnemonic	Category
0000	NOP	Control SPU Ops
0001	MinGate	Control SPU Ops
0010	EqGate	Control SPU Ops
0011	ZeroMN	Control SPU Ops
0100	DistDir	4-bit Vector Ops
0101	BoxArea	4-bit Vector Ops
0110	BasicBuffer	4-bit Vector Ops
0111	AttrReclass	4-bit Vector Ops
1000	FocalMeanRow	4-bit Raster Ops
1001	FocalSumRow	4-bit Raster Ops
1010	LocalDiv	4-bit Raster Ops
1011	FocalMaxPoolRow	4-bit Raster Ops
1100	NormDiffIndex	4-bit Multispec Raster Ops
1101	LocalCodeOp	4-bit Multispec Raster Ops
1110	DoubleDist	8-bit Double-Precision OUT Ops
1111	DotProduct	8-bit Double-Precision OUT Ops

To find more documentation and examples, look at the repository documentation: https://github.com/umn-geocommons.

How to test

The Verilog code can be run in the HDL software of your choice, with src/tt_um_teenyspu.v as the top-level module.

File test/tb_tt_um_teenyspu.v contains a testbench of all operations programmed for the TeenySPU. This can be set as a simulation source.

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	Q[0]	N[0]	AC[0]
1	Q[1]	N[1]	AC[1]
2	Q[2]	N[2]	AC[2]
3	Q[3]	N[3]	AC[3]
4	Op[0]	M[0]	BD[0]
5	Op[1]	M[1]	BD[1]
6	Op[2]	M[2]	BD[2]
7	Op[3]	M[3]	BD[3]

VoGAl

by **Michael Stambach**

0547 | 25.175 MHz | HDL Project

`github.com/michaelstambach/vogal`

“bird game”

How it works

This is a rudimentary implementation of the bird related game that was very popular in 2013 for some reason. Input 0 makes the “bird” go up. Input 1 starts the game. Output is through VGA.

How to test

There is a reference implementation in `ref.py`. The `main()` function allows playing the reference interactively. `generate_frames(inputs)` can be given a list of inputs and will generate the according reference frames. Running `make` should generate a few frames using the actual implementation with the inputs specified at the top of `test.py`. Be aware that due to how inputs are handled the reference and actual implementations inputs are offset by one.

External hardware

This uses the VGA PMOD for video out.

Project Pinout

Digital Pins

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

Tiny Tapeout Factory Test for ttihp-timer

by **Zedulo** & **Edwin**

549 HDL Project

github.com/ZeduloAdmin/ttihp-timer

"*Factory test for timer*"

How it works

The zTimer has start, stop and reset signals. When rosc_enable is asserted, the inverter chain forms a feedback loop that produces oscillation. The start for starting the timer, stop is used to stop it. When stop is asserted, the elapsed count value is stored in the internal registers. The SPI device is used to read out the value stored within the particular register. See README.md for more info.

How to test

Press start, wait a while, and press to stop. Use an MCU to read out the value from the zTimer using the SPI protocol

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	SPI_SCK	SPI_MISO	—
1	SPI_CS	—	—
2	SPI_MOSI	—	—
3	start_1	—	—
4	stop_1	—	—
5	start_2	—	—
6	stop_2	—	—
7	—	—	—

tinytapeout_henningp_2bin_to_4bit_decoder

by **Henning Petersen**

0551 Wokwi Project

github.com/skippersboat/tinytapeout_create_your_gds

wokwi.com/projects/456576419565744129

"2bin_to_4bit_decoder"

How it works

This is a decoder that converts 2 binary numbers into 4 logic outputs Suppose we connect the design to an input switch and a 7 segment LED display sw0 sw1 -> out0 out1 out2 out3

0 0 -> 0 0 0 1 0 1 -> 0 0 1 0 1 0 -> 0 1 0 0 1 1 -> 1 0 0 0

The time for the task was very tight! so a certain binary setting may turn on a different LED An example: 0 0 -> 0 1 0 0 (Another LED is turning on, but except this mix, it works as desired

How to test

IN1 and IN2 from the test PCB equals sw0 and sw1 OUT0 OUT1 OUT2 OUT3 equals out0 out1 out2 out3 Instuctions: Switch on IN1 and IN2 as desired, and see one of 4 LED segments light up

External hardware

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input A	out0	—
1	input B	out 1	—
2	—	out 2	—
3	—	out 3	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Silly Mixer

by **James Buchanan**

0553 | 10 MHz | HDL Project

github.com/BoredSemiRetiredEngineer/ttihp_submission_other_tile

"A Silly Mixer or a silly weighted sum.(You deicde)"

How it works

Using 4 bit values and 3 bit weighted value coefficant, these valuers are mixed by using the Result = A1 * Gain1 + B1 * Gain2 relationship.

How to test

FPGA use required to input and examine output

External hardware

FPGA use required to input and examine output

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	ui_in[0]	uo_out[0]	uio_in[0]
1	ui_in[1]	uo_out[1]	uio_in[1]
2	ui_in[2]	uo_out[2]	uio_in[2]
3	ui_in[3]	uo_out[3]	uio_in[3]
4	ui_in[4]	uo_out[4]	uio_in[4]
5	ui_in[5]	uo_out[5]	uio_in[5]
6	ui_in[6]	uo_out[6]	uio_in[6]
7	ui_in[7]	uo_out[7]	uio_in[7]

TinyTapeout VGA Checker

by **Adrian Trummer**

0555 | 25.175 MHz | HDL Project

`github.com/adriantrummer/tinytapeout1`

"*Modified Checker module*"

How it works

VGA Checker template was modified by reversing the direction & changing the default colors

How to test

Test the same way you would test the normal VGA Checker implementation (connect to VGA display and see if it displays the desired graphics)

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	—

Triple Modular Redundancy

by **Jean-Christian de Rivaz**

0557 8.192 MHz HDL Project

`github.com/jcdr/trmspi1`

"Triple Modular Redundancy 8 I/O with SPI master for each processor"

What is this?

This is a Triple Modular Redundancy (TMR) voter chip for safety-critical embedded systems. It interfaces with 3 redundant low-power processors via SPI and votes their 8 output bits.

- **Inputs**: 8 switch inputs from the demo board (`ui_in[7:0]`), sent to all 3 processors.
- **Outputs**: 8 voted discrete signals (`uo_out[7:0]`).
- **SPI Interface** (on bidirectional pins `uio`):
  - Shared `cs_n` (`uio[0]`, out)
  - Shared `sclk` (`uio[1]`, out)
  - `miso0` (`uio[2]`, in), `mosi0` (`uio[3]`, out)
  - `miso1` (`uio[4]`, in), `mosi1` (`uio[5]`, out)
  - `miso2` (`uio[6]`, in), `mosi2` (`uio[7]`, out)

How does it work?

The chip acts as SPI master using SPI Mode 0 (`CPOL=0`, `CPHA=0`). This means:

- `sclk` idles low
- data is sampled on the rising edge
- data is shifted on the falling edge

The design polls all 3 CPUs in parallel once per millisecond.

- Frame size: 24 bits (3 bytes).
- Master to each processor: `next_prn` + `switches` + `majority_byte`.
- Processor to master: `echoed_prn` + `desired_out` + `desired_valid`.
- SPI clock: about `1.024 MHz` from the `8.192 MHz` project clock.

Each SPI slice has its own 8-bit LFSR with polynomial `x^8 + x^6 + x^5 + x^4 + 1`. The 3 reset seeds are:

- CPU0 slice: `0x2A`
- CPU1 slice: `0x54`
- CPU2 slice: `0xA8`

The PRG protocol is staged across frames:

- on frame `N`, the master sends a new `next_prn` byte
- the CPU stores that byte locally
- on frame `N+1`, the CPU echoes that stored byte as `echoed_prn`
- the master compares `echoed_prn` against the locally stored PRG state for that slice

The CPU does not need to compute the PRG sequence itself. It only needs to stage and echo the last received PRG byte.

If the echoed PRG byte is wrong for one slice, that slice is invalid for that frame.

Each processor also returns:

- `desired_out`: the 8 bits it wants to drive
- `desired_valid`: one validity bit per output bit

For each slice and each bit:

- if the frame is valid and the corresponding `desired_valid` bit is `1`, the slice contributes the processor's `desired_out` bit
- otherwise the slice falls back to its own stored copy of the previously voted output bit

That per-slice fallback state is intentionally stored redundantly inside each slice.

The common voter then performs a pure bitwise 2-of-3 majority over the 3 slice-resolved bits.

The `majority_byte` sent back to each processor is also per-CPU:

- `1` means that CPU sent a valid bit and that bit matched the final voted output
- `0` means the CPU bit was invalid or disagreed with the final voted output

This `majority_byte` is reported one frame later, because it is computed from the frame that just completed and transmitted in the next frame.

Timing summary:

- first frame after reset: about `0.5 ms`
- steady frame period: `1 ms`

Programming the CPU Side (C Code Example)

The CPU side only needs to stage the received PRG byte and echo it back on the next frame. It does not need to compute the PRG sequence.

```
#include <stdint.h>

typedef struct {
    uint8_t staged_prn;
```

```
    uint8_t desired_out;
    uint8_t desired_valid;
} cpu_state_t;

// rx_buffer receives: next_prn, switches, majority_byte
// tx_buffer sends:    echoed_prn, desired_out, desired_valid
void spi_slave_handler(cpu_state_t *state, uint8_t rx_buffer[3],
uint8_t tx_buffer[3]) {
    uint8_t next_prn = rx_buffer[0];
    uint8_t switches = rx_buffer[1];
    uint8_t majority = rx_buffer[2];

    tx_buffer[0] = state->staged_prn;
    tx_buffer[1] = compute_desired_outputs(switches, majority);
    tx_buffer[2] = compute_desired_valid(switches, majority);

    state->staged_prn = next_prn;
}
```

This mirrors the current Verilog protocol: staged PRG echo, per-bit output validity, and per-CPU majority feedback.

Inputs / Outputs

- **Dedicated Inputs (`ui_in[7:0]`)**: Switches from the demo board, forwarded to all 3 processors.
- **Dedicated Outputs (`uo_out[7:0]`)**: Voted output byte.
- **Bidirectional IOs (`uio`)**: SPI signals as listed above.

How to test

From the project root:

- Run the RTL cocotb tests:

  ```
  ./build -t
  ```

- Print the latest utilization and cell counts:

  ```
  ./build -s
  ```

- Run the tests and then print the same summary:

  ```
  ./build -t -s
  ```

The current cocotb suite covers:

- valid PRG echo handling
- per-bit validity masks
- per-CPU `majority_byte` timing and contents
- rejected frames after bad echoed PRG bytes
- fallback to the previously voted state

- a 256-frame randomized no-reset stress test with one injected SPI bit fault per frame

Waveforms are written to `test/tb.fst`.

External hardware

- 3 external CPUs or equivalent SPI slaves connected to the shared `cs_n` and `sclk`
- one dedicated `miso` and one dedicated `mosi` line per CPU
- demo board clock at `8.192 MHz`

On the Tiny Tapeout demo board, the RP2350 controller can be used for basic bring-up. The RP2350 `SPI1` peripheral in slave mode is connected to `cs_n`, `sclk`, `miso0`, and `mosi0`. For simple majority experiments, `miso0` and `miso2` can be driven from the same RP2350 transmit signal so that two processor channels return the same data.

Each external CPU should:

- stage the received `next_prn`
- compute `desired_out`
- compute `desired_valid`
- return `echoed_prn`, `desired_out`, and `desired_valid` on the next poll

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	in0	out0	cs_n
1	in1	out1	sclk
2	in2	out2	miso0
3	in3	out3	mosi0
4	in4	out4	miso1
5	in5	out5	mosi1
6	in6	out6	miso2
7	in7	out7	mosi2

Tiny tape out test

by **Ole Tideman**

0559 Wokwi Project

github.com/slugvision/Tinytapeout

wokwi.com/projects/456572315745884161

"Blinking digit"

How it works

Simple blinking

How to test

Start test generator

External hardware

Display

Project Pinout

Digital Pins

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

VGA multiplex with TRNG

by **Khanh Lo**

0561 251.75 GHz HDL Project

`github.com/DivineMK/ttihp26a-tapeout`

"Multiplex between 4 VGA projects with TRNG"

How it works

Select between 4 VGA projects. Can use TRNG (from sampling ring oscillator) to randomly select.

How to test

Try changing speed, direction and select between different VGA projects.

External hardware

Need VGA PMOD compatible with TinyVGA.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	speed	R1	—
1	direction	G1	—
2	sel[0]	B1	—
3	sel[1]	VSync	—
4	—	R0	—
5	—	G0	—
6	—	B0	—
7	rand	HSync	—

smolCPU

by **Nicolas Tscherter**

0563 | 50 MHz | HDL Project

github.com/TscherterJunior/tt_um_TscherterJunior_top

"8 bit custom ISA cpu"

How it works

8 bit chip running custom isa. Relying on the RP2008 for Ram emulation.

Registers

```
The Chip has 8 8-bit gp registers numbered 0 to 7

Registers 0,1 are called accumulator 0,1 in the following.
Register 7 is hardcoded to be used as the source for the page
address in paged jumps

The Chip further has a 2 bit flag registers
Set by:

Arithmetic Ops:
    F[0] = overflow/borrow
    F[1] = carry

bitwise Logic:
    F[0] = Zero
    F[1] = Parity

shift:
    F[0] = Zero
    F[1] = Sign
```

Instruction encoding and semantics

A: The used accumulator RRR: [2:1] The involved Register

Adi: 0001_AIII Acc[A] += IIII

Add 0010_ARRR Acc[A] += Reg[RRR]

Sub 0011_ARRR Acc[A] += Reg[RRR]

Dcd 0100_MMMM Set Flags = 11 if current flag state is in the subset of possible states set([[False,False],[False,True],[True,False],[True,True]]) by the mask MMMM else 0

Cmp 0101_AUGE Let Target = A ? Acc[1] : 0

```
set flages as follows:
    Flag[1] = ((G && A signed  grater than Target) || (E && A ==
Target)) ? 1 : 0
    Flag[0] = ((G && A unsiged grater than Target) || (E && A ==
Target)) ? 1 : 0

note: the U bit is unused
```

Jmp 0110_PRRR if flag[0]: instruction_pointer = Reg[RRR] if P: Mem[I,255] = Reg[7]

Xor 0111_ARR Acc[A] ^= Reg[RRR]

Ldr 1000_ARRR Acc[A] = Reg[RRR]

Str 1001_ARRR Reg[RRR] = Acc[A]

Ldm 1010_ARRR Acc[A] = Mem[Reg[RRR]]

Stm 1011_ARRR Mem[Reg[RRR]] = Acc[A]

And 1100_ARR Acc[A] &= Reg[RRR]

Or_ 1101_ARR Acc[A] |= Reg[RRR]

Sll 1110_ARR Acc[A] <<= Reg[RRR]

Srl 1111_ARR Acc[A] >>= Reg[RRR]

How to test

Work in progress. Check the project repo for updates.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	data0	read_en	addr0/data0
1	data1	instr_en	addr1/data1
2	data2	cpu_flag[0]	addr2/data2
3	data3	cpu_flag[1]	addr3/data3
4	data4	—	addr4/data4
5	data5	cpu_state[0]	addr5/data5
6	data6	cpu_state[1]	addr6/data6
7	data7	cpu_state[2]	addr7/data7

Mini PSG

by **Peter Szentkuti**

576 | 25 MHz | HDL Project

`github.com/zettpe/tt-ihp26a-mini-psg`

"Compact SPI programmable sound generator"

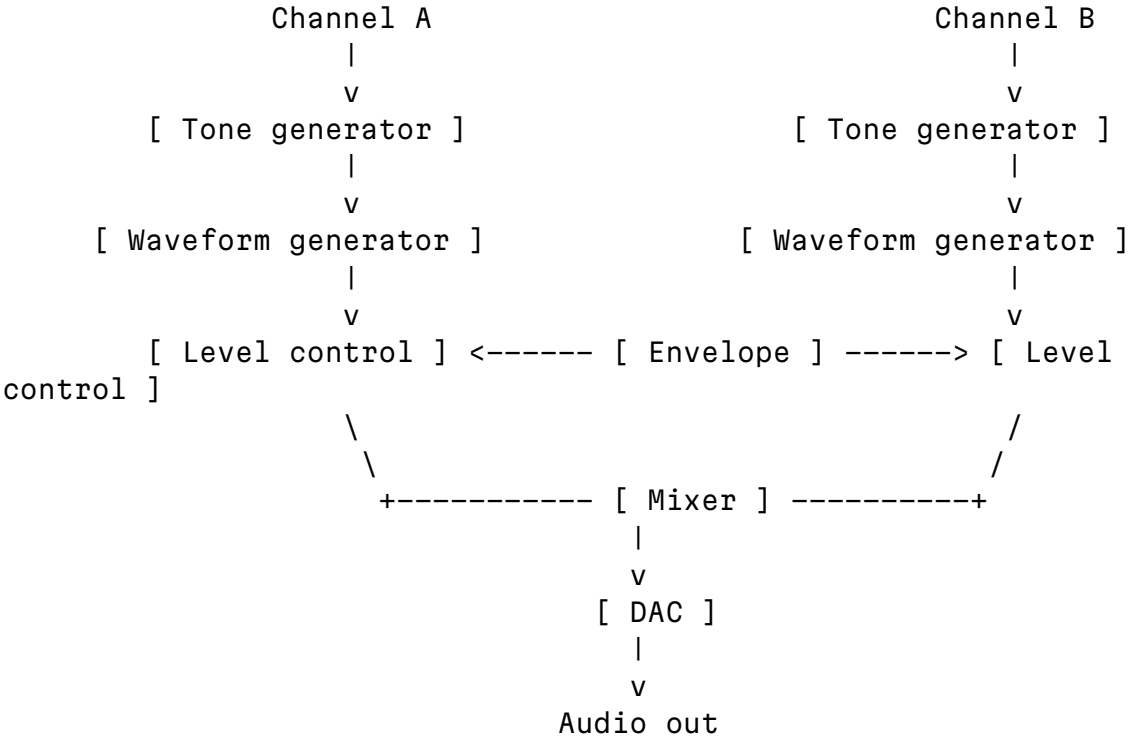
How it works

Mini PSG is a compact sound generator for Tiny Tapeout IHP 26a.

It has two tone channels, one shared 3 bit envelope and a 1 bit audio output on `uo_out[7]`. Configuration is done over a write only SPI interface. The design uses a single 25 MHz clock domain.

Each channel stores a tone code and an octave value. Together they set the phase step for a 23 bit phase accumulator. That phase then drives one of four waveforms: square, pulse, saw or triangle.

The output of each channel is then scaled by either the channel volume setting or the shared envelope. The two channel outputs are mixed together and sent to a 1 bit DAC that drives the audio output.



Reset is asserted immediately. Internal logic leaves reset two `clk` cycles after `rst_n` is released.

When hard mute is active, the audio output is forced low and the 1 bit DAC returns to its idle state. After power up or reset, the chip remains silent until it is configured over the SPI interface.

Pin connections

Pin	Function
ui_in[0]	hard mute
uo_out[7]	1 bit audio output
uio_in[0]	SPI_CS_N
uio_in[1]	SPI_MOSI
uio_in[2]	unused TT default SPI_MISO slot
uio_in[3]	SPI_SCK

SPI interface

The SPI interface is write only and uses SPI mode 0. Each transfer consists of one command byte followed by one data byte.

Byte	Format	Meaning
0	0000 aaaa	aaaa = write address (4 bits)
1	dddd dddd	dddd dddd = write data (8 bits)

If CS_N changes state before the two byte frame is complete, the partial transfer is discarded. Additional clocks after the data byte are ignored until CS_N returns high. The chip does not support reads and never drives MISO.

The SPI inputs are sampled by the internal clk signal. SPI_SCK is not used as an internal clock. SPI_SCK must not exceed clk / 8. Keep SCK low and high for at least 4 clk cycles each. CS_N must be asserted at least 4 clk cycles before the first SCK rising edge, remain asserted for at least 4 clk cycles after the last SCK falling edge and remain deasserted for at least 4 clk cycles between frames.

Register map

Address	Register	Function
0x0	CONTROL	audio enable and register clear
0x1	NOTE_A	tone setting for channel A
0x2	CHANNEL_A_CONTROL	waveform and tone control for channel A
0x3	NOTE_B	tone setting for channel B
0x4	CHANNEL_B_CONTROL	waveform and tone control for channel B
0x5	VOLUME_AB	channel output levels

0x7	ENVELOPE_CONTROL	envelope mode, restart pulse and channel assignment
0x8	ENVELOPE_PERIOD	envelope step period

CONTROL

Bit	Function
0	audio enable
1	clear all stored register values when written as 1

CONTROL[1] is a pulse on write, not a stored mode bit.

NOTE_A **and** NOTE_B

Bits	Function
[6:4]	octave
[3:0]	tone code

NOTE_*[3:0] = 0 to 9 are the 10 valid tone settings. 10 to 15 give no tone. The table is set for the 25 MHz clock and the 23 bit phase path.

CHANNEL_A_CONTROL **and** CHANNEL_B_CONTROL

Bit	Function
[1:0]	waveform select
2	tone enable
4	enable envelope for that channel
5	channel gate enable

Waveform select:

Value	Waveform
0	square
1	pulse
2	saw
3	triangle

Tone enable and channel gate must both be high before that channel produces a tone.

VOLUME_AB

Bits	Function
[2:0]	channel A level

[6:4]	channel B level

0 is mute and 7 is full scale for that channel.

ENVELOPE_CONTROL

Bit	Function
0	envelope mode select
2	envelope restart pulse when written as 1
3	apply envelope to channel A
4	apply envelope to channel B

Envelope modes:

Value	Mode
0	square
1	saw

ENVELOPE_PERIOD

Bits	Function
[7:0]	envelope step period

Lower values make the envelope run faster. Higher values make it run slower. 0 is treated as one clk step per envelope update.

Reset state

At reset, CONTROL comes up with audio disabled, both note registers come up in the no tone range, both channel control registers are cleared, both channel levels are 0, the envelope is off and ENVELOPE_PERIOD resets to 0x10.

How to test

For a minimal hardware bring up, keep ui_in[0] = 0 so hard mute is inactive, then write the following register values:

Register	Address	Value
CONTROL	0x0	0x01
NOTE_A	0x1	0x30
CHANNEL_A_CONTROL	0x2	0x24
VOLUME_AB	0x5	0x07

This sequence turns audio on, writes `0x30` to `NOTE_A`, enables square wave on channel A and sets channel A to full scale. The result is a steady 1 bit audio stream on `uo_out[7]`.

A cocotb testbench is provided for the chip and its register settings.

Run it with:

```
make -C test -B
```

External hardware

An SPI master is needed to configure the chip.

The audio output is a 1 bit signal. Feed it into the TT Audio Pmod or into a simple RC low pass filter followed by an amplifier and speaker or headphones.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	HARD_MUTE	—	SPI_CS_N
1	—	—	SPI_MOSI
2	—	—	—
3	—	—	SPI_SCK
4	—	—	—
5	—	—	—
6	—	—	—
7	—	AUDIO	—

hypnotic squares

by **Pietro**

`0578` `25.175 MHz` `HDL Project`

`github.com/pietromiotti/ttihp-verilog-squares`

"*concentric hypnotic squares*"

How it works

Implementation of hypnotic concentric squares via VGA playground

How to test

The verilog can be directly tested using the VGA PLAYGROUND

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	—

Glitch Detector

by **Marco Zollinger**

`0580` `25 MHz` `HDL Project`

`github.com/mzollin/tt-glitch-detector`

"Detector circuit for glitching attacks against the TT chip"

How it works

This project is heavily inspired by the glitch detector security peripheral of the Raspberry RP2350 microcontroller. Thanks to the detailed description in the datasheet, I was inspired to create my own implementation of such a detector in the form of a Tiny Tapeout ASIC design. It includes eight parallel glitching detectors with differing sensitivities and a dedicated latching trigger output for each. There is one global test trigger input. The internal delay chains to detect setup / hold margin violations are parametrized between 70% and 140% of the defined system clock period for a regular clock of 25 MHz, which is a slightly wider range than for the detectors in the RP2350.

The working principle of the detector is explained in the RP2350 datasheet, from which I am quoting here: > The glitch detector detects loss of setup and hold margin in the system clock domain, which may be caused by deliberate external manipulation of the system clock or core supply voltage. [...]

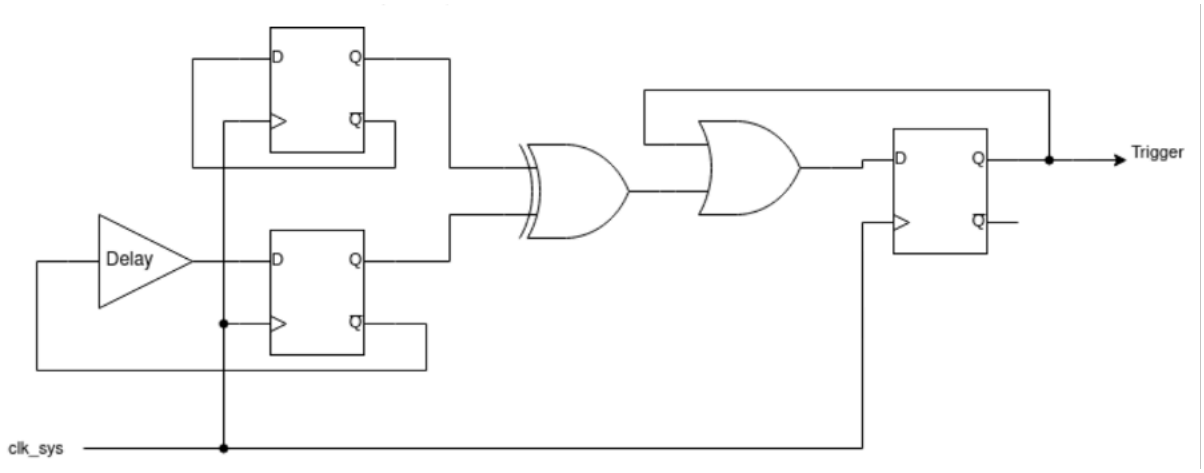


Figure 580.1: Glitch detector trigger circuit

Two flops each toggle on every system clock cycle. One has a programmable delay line in its feedback path, the other does not. Loss of setup or hold margin causes one of the flops to fail to toggle, so the flops values differ, setting the trigger output

> The detector triggers when the two D-flops take on different values, which is impossible under normal circumstances. The delay line is programmable from 75% to 120% of the minimum system clock period in increments of 15%. Higher delays make the circuit more sensitive to loss of setup margin. [...]

How to test

To perform a fault injection attack and test the detector circuit on this chip, an appropriate external tool is required. The choice of tool depends on the type of fault injection to be performed. For supply voltage glitching, the ChipWhisperer-Lite or -Pro could be used, for clock glitching the ChipWhisperer-Pro is recommended. For electromagnetic fault injection (EMFI), the ChipSHOUTER or ChipSHOUTER-PicoEMP might be suitable. There are other commercial and open source options and of course you can always build your own customized fault injection tools.

With the exception of EMFI attacks, performing fault injection attacks will usually require modifications to the power rails or clock circuitry of the Tiny Tapeout breakout board. Modifying the PCB or performing fault injection attacks in general risks interfering with or even permanently damaging the ASIC or other circuitry on the board. This is an untested experimental design, fault injection attacks might not just trigger the detector but also glitch the design multiplexer on the ASIC, requiring the chip to be reset. Every modification, experiment or attack is your own responsibility and I won't be liable for any damage.

External hardware

- Fault injection tool of your choice (e.g. ChipWhisperer or ChipSHOUTER) to perform the glitching attacks
- Octal LED PMOD (e.g. 8LD) connected to the Out-Port to show the status of the detectors

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Test trigger input	Detector 0 trigger status	—
1	—	Detector 1 trigger status	—
2	—	Detector 2 trigger status	—
3	—	Detector 3 trigger status	—
4	—	Detector 4 trigger status	—
5	—	Detector 5 trigger status	—

#	Input	Output	Bidirectional
6	—	Detector 6 trigger status	—
7	—	Detector 7 trigger status	—

8bit-mac-unit

by **bdawg4113**

582 | 60 MHz | HDL Project

github.com/bdawg4113/tt-ihp26a-mac

"8-bit sequential multiply accumulate unit for multiplying and adding 8 bit numbers really quickly"

How it works

8-bit multiply accumulate unit that is used to multiply two Q0.7 fixed point numbers and sum them up for DSP purposes.

How to test

It requires two 8bit inputs that will be multiplied and then be added together.

External hardware

No external hardware used so far to test

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	load_en	Output byte 0	—
1	read_sel[0]	Output byte 1	—
2	read_sel[1]	Output byte 2	—
3	clr_acc	Output byte 3	—
4	—	Output byte 4	—
5	—	Output byte 5	—
6	—	Output byte 6	—
7	—	Output byte 7	—

4-Bit Counter and Registers Demo

by **Thomas Fischbacher**

`0584` `10 MHz` `Wokwi Project`

`github.com/fischbacher/TinyTapeout-IHP26a-StudentCoaching`

`wokwi.com/projects/459285910800527361`

"Educational Wokwi Demo"

How it works

This is mostly an educational placeholder wokwi project, intended as a starting point for Swiss high school students to get going with their own designs.

The idea is to allow them to understand some ASIC design and digital circuitry basics by showing some ideas and design elements they might want to use in their own designs - and also discussing some pitfalls to avoid.

Wokwi URL: https://wokwi.com/projects/459285910800527361

The description on the Wokwi page contains some exercises for students to go through.

How to test

The circuit consists of two parts - a simple 4-bit binary counter, plus logic to store data in two 4-bit "registers" and retrieve it from there. The Wokwi project page has details.

As a brief summary:

- Output pins C0,C1,C2,C3 simply cyclically count through 0000-1111, one step per clock cycle.
- The outputs on pins DO0,DO1,DO2,DO3 can be made to show the value of either of two "4-bit registers" as follows:
  - Setting (OP0,OP1,OP2) to (0,1,1) makes internal register A store the value available on DI0-DI3.
  - Setting (OP0,OP1,OP2) to (1,1,0) makes internal register B store the value available on DI0-DI3.
  - Setting (OP0,OP1,OP2) to (1,1,1) makes the outpot DO0-DO3 show the value of register A if SEL0 is 0, else the value of register B.
  - If (OP0,OP1,OP2) is not (1,1,1), then DO0-DO3 are all 0.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	OP0	C0	—
1	OP1	C1	—
2	OP2	C2	—
3	DI0	C3	—
4	DI1	DO0	—
5	DI2	DO1	—
6	DI3	DO2	—
7	SEL0	DO3	—

Random Snake

by **TobisMa**

`0586` `50 Hz` `HDL Project`

`github.com/TobisMa/random-snake`

> *"A snake moving pseudo-randomly across the 7-segment-display"*

How it works

Input a number using the switches and it will generate random snake moving across the segments

How to test

Plug it in, input a number and view the output on the 7-segment-display

External hardware

- seven segment display
- switch input?

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	in1	out1	—
1	in2	out2	—
2	in3	out3	—
3	in4	out4	—
4	in5	out5	—
5	in6	out6	—
6	in7	out7	—
7	in8	out8	—

Maze Explorer Game

by **Tatsuya Nomoto**

`0588` `50 MHz` `HDL Project`

`github.com/medama2008-glitch/ttihp26a-maze-game`

"I2C OLED display maze game with wall-breaking mechanics and audio feedback"

How it works

A maze exploration game rendered on a 128x64 OLED (SSD1306) via I2C. The core game logic is a pure combinational circuit generated from DSLX (Google XLS), wrapped by a Verilog module that holds 123 bits of state in flip-flops.

The player navigates a 16x8 hardcoded maze from start (1,1) to goal (14,6). Up to 2 walls can be broken using the BREAK button. The game outputs audio feedback (move beep, wall-break noise, goal fanfare) via a piezo speaker.

I2C runs at 195kHz using pseudo open-drain outputs on the bidirectional pins. Button inputs are synchronized (2-stage FF) and debounced (5ms sampling).

How to test

1. Connect a 128x64 SSD1306 OLED (I2C, address 0x3C) to uio[0] (SDA) and uio[1] (SCL). Add 4.7k ohm pull-up resistors from SDA and SCL to VCC (3.3V).
2. Connect a piezo speaker to uo_out[2].
3. Connect pushbuttons to ui_in[0..7] (directly or with optional RC debounce).
4. After reset, the maze appears on the OLED with the player at position (1,1).
5. Use direction buttons to navigate. Hold BREAK + direction to break walls (max 2).
6. Reach the goal at (14,6) to hear the goal fanfare.

External hardware

- 128x64 SSD1306 OLED (I2C, address 0x3C)
- Piezo speaker
- 8 pushbuttons
- 2x 4.7k ohm pull-up resistors (for I2C SDA/SCL)

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	btn_up (direction: up)	unused	I2C SDA (pseudo open-drain, 4.7k pull-up required)
1	btn_down (direction: down)	unused	I2C SCL (pseudo open-drain, 4.7k pull-up required)
2	btn_left (direction: left)	audio_out (piezo speaker)	unused
3	btn_right (direction: right)	unused	unused
4	btn_break (wall break, active high)	unused	unused
5	btn_cancel (cancel confirm dialog)	unused	unused
6	btn_restart (restart game, press twice)	unused	unused
7	btn_refresh (refresh display, press twice)	unused	unused

Verilog ring oscillator

by **Chiranjit Vivek**

590 HDL Project

`github.com/ChiranjitPatel/tt_ihp26a_ringosc_stdcell`

"*Multiple ring oscillators with simple PWM experiments*"

How it works

This is a simple Ring Oscillator with multiple rings (125, 251, 501 and 1001).

How to test

1. Set `ui_in` to 0; this should configure all the PWM experiments to do nothing.
2. Probe `uo_out[3]` (`ring_1001`) with an oscilloscope: This is the raw output of the longest (1001-inverter) ring oscillator.
3. Probe through each of `uo_out[2:0]`: Each step to a lower bit is another ring that is half the length (and hence should be double the frequency): 501 inverters (`ring_501`), then 251 inverters (`ring_251`), then 125 inverters (`ring_125`).
4. Probe `uo_out[4]` (`c1_3`): This is `ring_125` divided by 16.
5. Probe `uo_out[5]` (`c2_3`): This is `ring_1001` divided by 16.

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any NONE

Overview

Picked from reference to Anton Maurovic's Ring oscillator design.

For a quick-start guide just see the "How to test" heading below. I intend to have results collected in the TTIHP25a-ring-osc2 sheet of my "Anton's Tiny Tapeout silicon testing" Google Sheet.

There are 6 ring oscillators of various lengths in this design, ranging from 13 inverters up to 1001 inverters. 4 of them give their direct outputs. 4 also give their output divided by 16. 2 also drive PWM outputs at high frequency to see whether the output pins will filter them to "analog-like" waveforms, or whether instead the chip characteristics just kill the outputs.

This is implemented using the IHP SG13G2 open PDK and was fabricated on TTIHP25a, though it was originally intended for TT09.

Originally this project was submitted to TT09 (commit ee2feec). It was later rehardened for resubmission to TTIHP25a (wherein some minor changes were required).

See also: tt09-ring-osc and tt09-ring-osc3 for my other ring oscillator experiments on the same shuttle.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	ring_125	—
1	—	ring_251	—
2	—	ring_501	—
3	—	ring_1001	—
4	—	c1_3	—
5	—	c2_3	—
6	—	ring_3	—
7	—	—	—

vga_ca

by **Alexander Mordvintsev**

0592 | 25.175 MHz | HDL Project

`github.com/znah/ihp-ca`

"Simple VGA 1D cellular automata generator (WIP)"

How it works

VGA signal generator iterates through a number of 1D elementary cellular automata

How to test

Plug and play

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	—

TinyTapeout SRAM

by **Henry**

0594 Wokwi Project

github.com/HenryHering/TinyTapeoutSRAM

wokwi.com/projects/459299619699169281

"8-bit SRAM"

How it works

a simple 8-bit SRAM

How to test

just do it

External hardware

finger(s)

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	q0_i	q0_o	—
1	q1_i	q1_o	—
2	q2_i	q2_o	—
3	q3_i	q3_o	—
4	q4_i	q4_o	—
5	q5_i	q5_o	—
6	q6_i	q6_o	—
7	q7_i	q7_o	—

7 segment ihp resistcode

by **igor luisoni**

`0609` `1 Hz` `Wokwi Project`

`github.com/bauzx/ttihp-7seg-resistcode`

`wokwi.com/projects/458752568884674561`

"show rESIStcodE on 7 segments"

How it works

Display a custom string of characters => "rESIStcodE"

How to test

Connect a 7 segments to the output and a dip-switch-8 on input

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	in0	out 7 segment a	—
1	in1	out 7 segment b	—
2	in2	out 7 segment c	—
3	in3	out 7 segment d	—
4	in4	out 7 segment e	—
5	in5	out 7 segment f	—
6	in6	out 7 segment g	—
7	in7	out 7 segment dp	—

RNG

by **Felix N**

0611 | 500 kHz | HDL Project

`github.com/Xelef2000/tinytapeout-verilog`

"*Ring oscillator based random number generator*"

How it works

This project is a true random number generator.

The core of the TRNG is a set of three ring oscillators of different lengths (5, 11, and 23 inverters). These oscillators produce unstable, jittery signals. The outputs are combined using an XOR gate to create a chaotic bit stream. Here is the ring oscillator frequency estimates:

Ring Oscillator	Frequency Estimate	Period Estimate
5	231 MHz	4.32 ns
11	117 MHz	8.52 ns
23	59 MHz	16.91 ns

The raw random bitstream may have a bias (more 1s than 0s). To correct this, a Von Neumann corrector is used. It takes pairs of bits from the stream:

- If the bits are 01, it outputs a 0.
- If the bits are 10, it outputs a 1.

If the bits are the same (00 or 11), it outputs nothing.

The debiased bits are collected one by one and shifted into a 32-bit register. Once a 32 bit number has been collected, it is output through the UART.

7-Segment Display Output

The lower 8 bits of the random number are displayed as two hexadecimal digits on a multiplexed 7-segment display. The display uses time-division multiplexing at 122 Hz to alternate between the lower nibble (bits 3:0) and upper nibble (bits 7:4).

The bidirectional I/O pins output the 7-segment data:

- `uio_out[6:0]`: Segment data (active high, standard gfedcba mapping)
- `uio_out[7]`: Digit select (0 = lower nibble, 1 = upper nibble)

Display Update Speed Control

The display update rate is controlled by the dedicated input pins (DIP switches). Only the upper 5 bits (`ui_in[7:3]`) are used, providing 32 speed levels with exponential scaling. Each step doubles the update speed.

DIP Setting (ui_in)	Update Interval @ 500kHz
0x00 - 0x07	30 min (slowest)
0x08 - 0x0F	15 min
0x10 - 0x17	7.5 min
0x20 - 0x27	1.9 min
0x40 - 0x47	7 sec
0x60 - 0x67	0.4 sec
0x80 - 0x87	27 ms
0xF8 - 0xFF	0.8 us (fastest)

How to test

UART Output

To test the design via UART, connect a UART-to-USB adapter to the uo_out[0] pin (which is the UART TX pin), the ground pin, and the power pin of your board.

Configure the serial terminal to match the UART settings:

- Baud Rate: 9600
- Data Bits: 8
- Parity: None
- Stop Bits: 1

Once connected, you should see a continuous stream of raw binary data appearing in your terminal. This is the 32-bit random numbers being sent from the chip.

7-Segment Display

Connect a dual-digit 7-segment display (common cathode, active high) to the bidirectional I/O pins:

- `uio_out[0]` - Segment a
- `uio_out[1]` - Segment b
- `uio_out[2]` - Segment c
- `uio_out[3]` - Segment d
- `uio_out[4]` - Segment e
- `uio_out[5]` - Segment f
- `uio_out[6]` - Segment g

- `uio_out[7]` - Digit select (directly drives digit enable/common)

Use the DIP switches on `ui_in[7:3]` to adjust the display update speed. Set to a higher value (e.g., 0x60) for a comfortable 0.4 second update rate.

Ring Oscillator Outputs

The raw ring oscillator outputs can be monitored on the uo_out[1], uo_out[2], and uo_out[3] pins, which correspond to the 6, 12, and 24 inverter ring oscillators respectively.

External hardware

- A UART-to-USB adapter to connect the chip's UART output to a computer
- A dual-digit 7-segment display (common cathode) with appropriate current-limiting resistors
- 8 DIP switches connected to ui_in for speed control

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	UART TX	SEG_A
1	—	Ring oscillator 6	SEG_B
2	—	Ring oscillator 12	SEG_C
3	Speed control bit 0	Ring oscillator 24	SEG_D
4	Speed control bit 1	—	SEG_E
5	Speed control bit 2	—	SEG_F
6	Speed control bit 3	—	SEG_G
7	Speed control bit 4	—	Digit select

Bouncing Checkers

by **Gregor Schultz**

0613 25.175 MHz HDL Project

github.com/g-schultz/ttihp-verilog

"Remix of the VGA Playground Checkers template that makes the checkers bounce"

How it works

This project is a remix of the VGA Playground Checkers template. An additional counter (called offset) was introduced that doesn't count linearly but starts fast and then slows down which results in a bouncing motion. Furthermore the number of layers were reduced and the direction and speed of the movement were adjusted. Making the rearmost layer faster than the first one gives sometimes the illusion that the front layer moves in the other direction.

How to test

Similar to the checkers template on the VGA Playground the first five buttons can be used to adjust the colors.

External hardware

VGA display

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	ColourBit0	R1	—
1	ColourBit1	G1	—
2	ColourBit2	B1	—
3	ColourBit3	VSync	—
4	ColourBit4	R0	—
5	ColourBit5	G0	—
6	—	B0	—
7	—	HSync	—

ttihp-HDSISO8RS

by **Yann Guidon**

0615 | 50 MHz | HDL Project

`github.com/ygdes/ttihp-HDSISO8RS`

"Higher density Shift register - RS version"

What is is

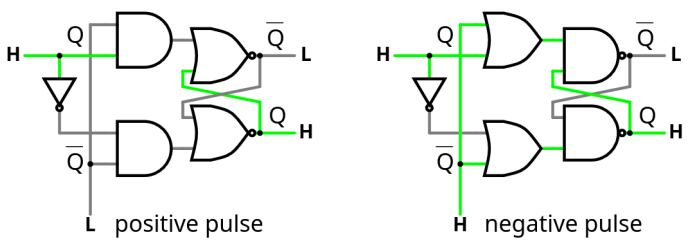
This tile delays a bit's value by 768+23=791 cycles at speeds above 20MHz (according to the synthesiser, but that's conservative). It is an attempt to beat storage packing density, as well as a test architecture for asynchronous shift registers, not made out of the larger DFF cells. This version packs 1024+32 RS latches and a controller, filling 86% of the tile's surface. This is 36% more dense than the DLHQ version (in another tile), but the P&R tools choke and the speed drops by about 10x for no acceptable reason. One more compelling reason to use macros and manual place&route!

How it works

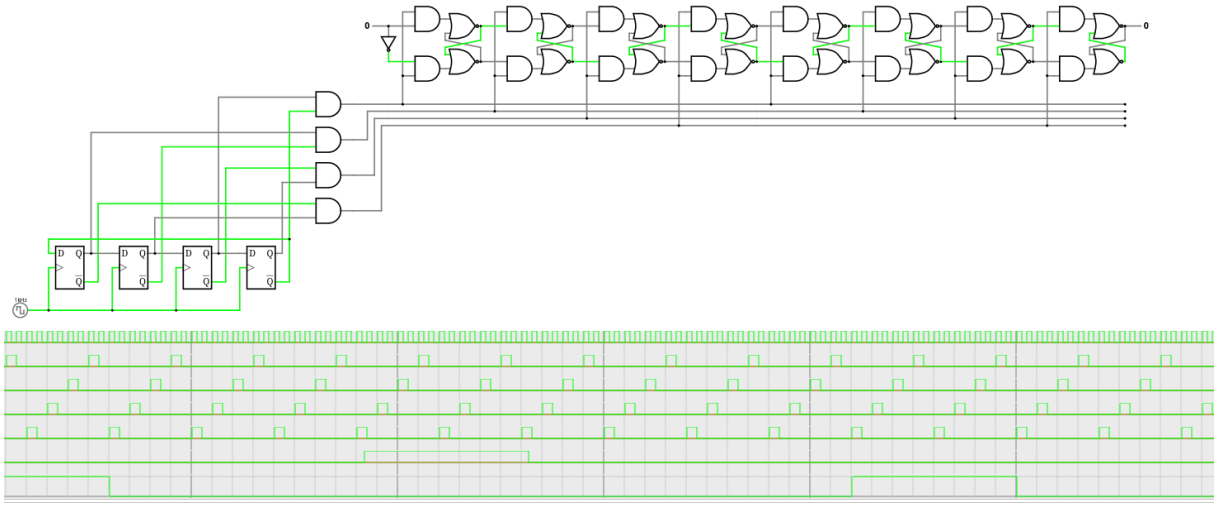
As the name implies, it's a high density shift register for deep digital delays. According to the PDK for CMOS IHP at https://github.com/IHP-GmbH/IHP-Open-PDK/blob/main/ihp-sg13g2/libs.ref/sg13g2_stdcell/doc/sg13g2_stdcell_typ_1p20V_25C.pdf

- Area of sg13g2_dfrbpq_1 : 48.98880
- Area of sg13g2_dlhq_1 : 30.84480
- Area of sg13g2_mux2_1 : 18.14400
- Area of sg13g2_a21oi_1 : (×2) = 18.14400 (same as sg13g2_o21ai_1)

MUX2 is almost 3× smaller than the DFF gate and could be used as a latch by feeding its output back to an input (just like with the old antifuse Actel FPGAs such as A1xxx). This trick is rejected by the tools but in the same area, I could also implement a SR latch with enable, using combined and compact OR/AND gates.



As a reference point, the project “tt_um_ygdes_hdsiso8_dlhq” at https://github.com/ygdes/ttihp-HDSISO8 uses the conventional transparent latch DLHQ, whose size is in-between. In all cases, the shift register uses 4 latches to store 3 bits at a given time and 4 non-overlapping “clock” pulses perform the shifting. Slowly. Just like below, but with 8 parallel chains.



The apparent complexity comes from the 8-phase clock, which is brought to the “asynchronous” domain. Each of the 8 lanes is 8× slower (which relaxes timing constraints) but the overall throughput is preserved by a demultiplexer and multiplexer. So it “should” work at “full speed”, we’ll see.

Compared to a shift register with normal DFF cells, it could store twice the same amount of bits per unit of surface, without the need of full-custom cells, as the controller’s (sequencer, mux and demux) size becomes insignificant when the chain gets longer. Depths of several kilobits are possible without too much hassles (if the synth agrees), without a mad clock network, reducing simultaneous switching noise... Not only are the pulses slower, their traces are also shorter: each pulse affects only 1/8th of the cells at any time.

Ideally, the 8 chains should be manually placed (or with a script), not thrown at random. For implementation, I use a “tuned” Verilog workflow and instantiate cells directly from https://github.com/IHP-GmbH/IHP-Open-PDK/blob/main/ihp-sg13g2/libs.ref/sg13g2_stdcell/verilog/sg13g2_stdcell.v . For simulation, parts of this file are copy-pasted to gate-specific files to remove some warnings (find them in /test).

How to test

Good to know:

- Clock and Reset can be asserted by external pins and internal signals.
- The pin CLK_OUT copies the currently selected clock (negated), for external triggering and troubleshooting. If it oscillates, you’re good.

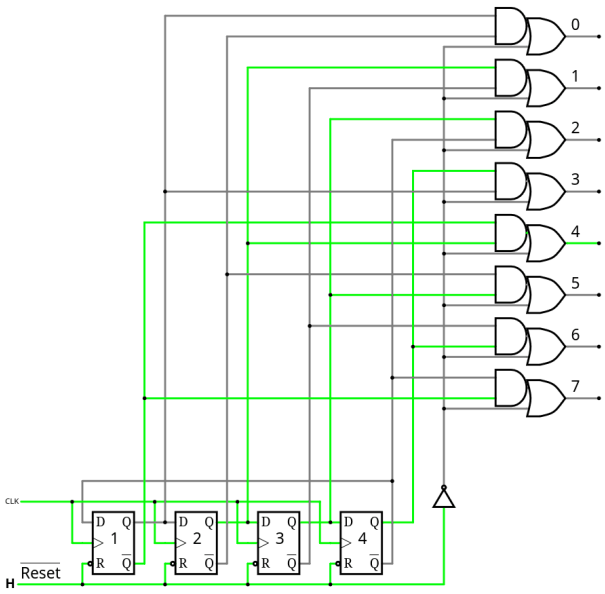
- External reset pin EXT_RST (asserted at 0 like the internal one) overrides the internal reset, don't let it float. A weak pull-up to 1 is advised.
- External clock (pin EXT_CLK) can be selected when pin CLK_SEL=1 (don't let them float).
- Always assert EXT_RST (to 0) while changing the state of CLK_SEL.

Startup sequence:

- EXT_RST asserted (0)
- Choose CLK_SEL's value
- Run that clock
- Release EXT_RST (to 1, and RESET is internally clock-resynchronised so give it a couple of cycles to come into effect)
- Input a '1' or a '0' on D_IN, and observe the value appearing on D_OUT after 791 clock cycles.

Extra insight and observability:

- When SHOW_LFSR=0, the IO port shows the 8 internal staggered pulses, turning from 0 to 1 and back to 0 in a linear sequence. It's just like a 4017 but 8 bits, since it's a Johnson counter too.
- 4 output pins provide the internal state of that 4-bit Johnson counter, or ring counter, thus you should observe a pretty pattern where only one pin changes at each clock cycle.



Note in the diagram above that RESET forces all the outputs to 1, thus flushing the whole delay line in less than a microsecond.

- You can measure the routing latency of the pins/pads/internal wires because CLK_OUT is inverted so just tie it to EXT_CLK with pin CLK_SEL=1. Probe with an oscillocsope and voilà, you have a free-running oscillator

and you can directly measure the low and high times, each corresponding to one trip on the in or out wire.

Bonus: LFSR

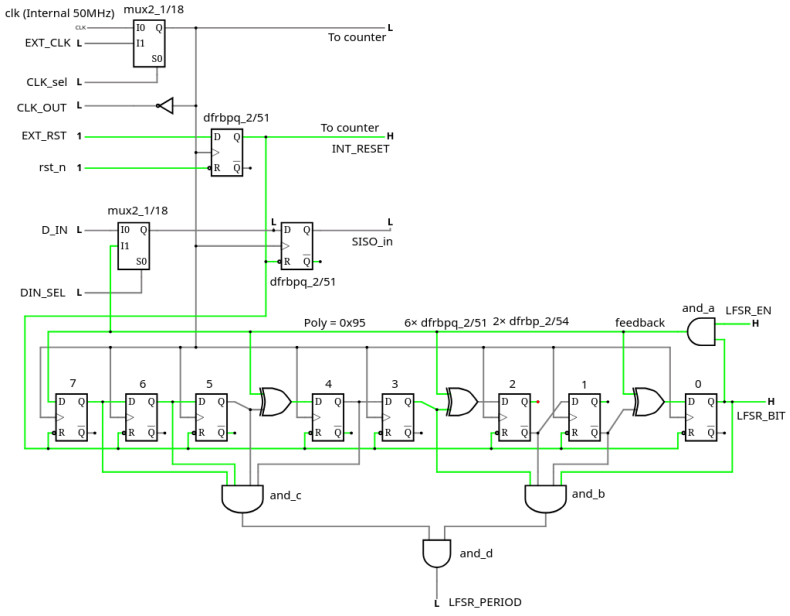
An 8-bit LFSR is integrated to ease testing. Thus an oscilloscope and a variable frequency oscillator are enough to characterise the achieveable speed. To use it,

- Assert the external reset EXT_RST (0)
- Select the desired clock (CLK_SEL)
- Unlock the internal LFSR by asserting pin LFRS_EN to 1
- Assert pin DIN_SEL (1) to internally route the LFSR bitstream to the SISO input
- Run the clock (internal or external, depending on CLK_SEL)
- Release EXT_RST (1) (now it should be started)
- Connect an oscilloscope to probe the signals D_OUT and LFSR_BIT while triggering on LFSR_PERIOD (which pulses every 255 clock cycles)
- See if both traces match (add some delay on LFSR_BIT if necessary).
- Send me pictures of your scope traces!

Note 1: 8 bits gives a period of 255, third of the SISO's depth, a shift of 26 cycles is expected and the SISO should store 3× the whole cycle, but the output should align anyway.

Note 2: The LFSR_PERIOD pulse should appear 193 clock cycles after the release of the RESET pin.

Note 3: The RESET signal clears the contents of the SISO. Give it a few cycles for the 0 to propagate through all the latches while it flushes after releasing the RESET.



External hardware

A basic custom test board will be put together, to hook the variable frequency generator and the oscilloscope probes.

Optionally, if you only want to make a “light chaser”, hook 8 LED to the IO port, select the external clock and add a 555. Or you can have a more funky pattern by displaying the LFSR’s state by setting SHOW_LFSR to 1.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	CLK_SEL	D_OUT	PULSE #0
1	EXT_CLK	CLK_OUT	PULSE #1
2	EXT_RST	Johnson #0	PULSE #2
3	D_IN	Johnson #1	PULSE #3
4	—	Johnson #2	PULSE #4
5	SHOW_LFSR	Johnson #3	PULSE #5
6	LFSR_EN	LFSR_PERIOD	PULSE #6
7	DIN_SEL	LFSR_BIT	PULSE #7

Gate-Level 8-bit MAC with Ripple-Carry Accumulator

by **Malik**

0617 50 MHz HDL Project

`github.com/malikweren/ttihp-malik-int8-mac`

"Structurally-optimized unsigned 8-bit MAC unit from ML-guided design-space exploration, with 17-bit accumulator and serial command interface"

Gate-Level 8-bit MAC with Ripple-Carry Accumulator

How it works

This design wraps a structurally-optimized gate-level multiply-accumulate (MAC) unit in a TinyTapeout-compatible clocked interface. The inner MAC core was produced through ML-guided design-space exploration of arithmetic architectures — it computes `y[16:0] = a[7:0] * b[7:0] + c[15:0]` as a purely combinational circuit using a ripple-carry accumulation structure.

The wrapper adds clocked operand registers, a 17-bit accumulator, and a serial command interface. On each MAC operation, the accumulator feeds back into the MAC's addend input, enabling running accumulation across multiple multiply-add cycles.

Specifications:

- Unsigned 8-bit × 8-bit multiply
- 17-bit accumulator (16-bit + carry/overflow)
- Single-cycle combinational MAC latency
- Gate-level optimized inner datapath

How to test

1. **Reset**: Pull `rst_n` low then high.
2. **Load A**: Set `cmd=001`, put value on `ui_in`, clock once.
3. **Load B + MAC**: Set `cmd=010`, put value on `ui_in`, clock once. This triggers `acc = a*b + acc`.
4. **Read result**: `cmd=011` for acc[7:0], `cmd=100` for acc[15:8], `cmd=101` for acc[16].
5. **Clear**: `cmd=110` zeros the accumulator.
6. **Repeat** steps 2-4 to accumulate more products.

Design context

The gate-level MAC netlist originates from a semester project on design-space exploration of AI hardware architectures, where ML-driven optimiza-

tion was used to explore Pareto-optimal arithmetic unit implementations across power, performance, and area (PPA) tradeoffs.

External hardware

None required.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	data_in[0]	data_out[0]	cmd[0]
1	data_in[1]	data_out[1]	cmd[1]
2	data_in[2]	data_out[2]	cmd[2]
3	data_in[3]	data_out[3]	busy
4	data_in[4]	data_out[4]	overflow (acc[16])
5	data_in[5]	data_out[5]	—
6	data_in[6]	data_out[6]	—
7	data_in[7]	data_out[7]	—

Demoscreen full of RICH

by **Hazel Li**

`0619` `25.175 MHz` `HDL Project`

`github.com/hazellilili/tt-ihp26a-faaaa`

"□ bouncing around the screen"

How it works

Displays a bouncing character on the screen, with animated color

How to test

Connect to a monitor. Set the following inputs to configure the design:

- `tile` (ui_in[0]) to repeat the character and tile it across the screen,
- `solid_color` (ui_in[1]) to use a solid color instead of a mix of golden and red.

External hardware

Project Pinout

Digital Pins

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

8 bit saturated adder

by **Luca Caballero Cusin**

0621 HDL Project

github.com/Blanluc/ttihp-verilog-template

"Simple 8 bit saturated adder. Can do sautrated addition and substraction for both signed and unsigned ints"

How it works

8-bit saturated arithmetic unit. This module snaps the result to the maximum or minimum possible value for the given data type to prevent wrap-around errors.The module supports four operations selected via the sel bits:00 (Signed Add): 8-bit signed addition.

How to test

-Set the operation mode using the top two bits of the bidirectional switches (uio_in[7:6]). -Input the first 8-bit operand on the dedicated input switches (ui_in). -Input the second 6-bit operand on the lower bidirectional switches (uio_in[5:0]). -Observe the 8-bit result on the dedicated output LEDs (uo_out).

External hardware

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	a[0]	res[0]	b[0]
1	a[1]	res[1]	b[1]
2	a[2]	res[2]	b[2]
3	a[3]	res[3]	b[3]
4	a[4]	res[4]	b[4]
5	a[5]	res[5]	b[5]
6	a[6]	res[6]	sel[0]
7	a[7]	res[7]	sel[1]

Spell. My. Name.

by **Vladislav Penchev**

0623 1 Hz Wokwi Project

github.com/vlad-penchev/tinytapeoutz

wokwi.com/projects/455303279136701441

"Spells out the first four letters of my name"

How it works

idk

How to test

still dont nkow

External hardware

yeah idk

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	seg_a	—
1	—	seg_b	—
2	—	seg_c	—
3	—	seg_d	—
4	—	seg_e	—
5	—	seg_f	—
6	HI	seg_g	—
7	LO	seg_h	—

TinyScanChain

by **Yann Guidon**

625 | 50 MHz | HDL Project

`github.com/YannGuidon/TinyScanChain`

"Low footprint scan chain for iHP PDK"

It's "just a quick, last-day FOMO" project where I try some new ideas in real silicon, so I can restart the project "DTAP - Debug and Test Access Port" (see https://hackaday.io/project/193122-dtap-debug-and-test-access-port)

What is this Tiny Tapeout tile ?

Tiny Tapeout's (https://tinytapeout.com) run "ihp26a" provides 200µm × 150µm of estate on the German iHP SG13G2 technology. That's about 2K gates at best but you'll still want to spy on them : observability and control are necessary so you need a TAP (Test Access Port) !

But TinyTapout does not provide a JTAG-like interface, you're on your own. So let's make one. Unfortunately the typical BILBO gates are bulky and require large fanout, and interfere with the routing of the other gates.

The iHP SG13G2 PDK provides A221OI and A21OI gates which solve this problem. It's not JTAG-compatible but it's simple, functional and should not interfere with the main design (if the synthesiser cooperates)

.

Resources

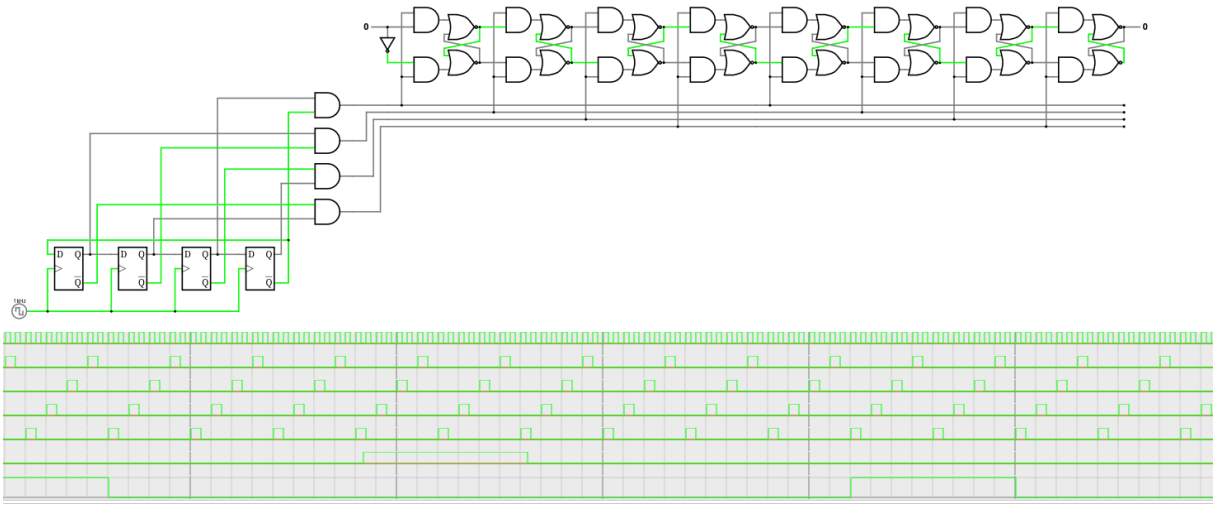
- https://github.com/ygdes/ttihp-HDSISO8 implements a high density shift register / delay line with DLHQ gates (standard latches) and 4-phase non-overlapping clocks.
- https://github.com/ygdes/ttihp-HDSISO8RS enhances the density by 36% with a pair of A21OI gates instead of one DLHQ gate.

These projects have shown that an iHP tile could be filled with more than 1K Reset-Set latches, though the synthesiser and the place&route tools do not cooperate, reducing the rated speed to about 20MHz, whatever this means, since the clock is internally sped down by 8. Still, 1M bits per second is enough for a comfy debug session. However this should not affect the DUT's performance and it's now a matter of coercing the tools to de-prioritise the scan chain, and learn other tricks.

.

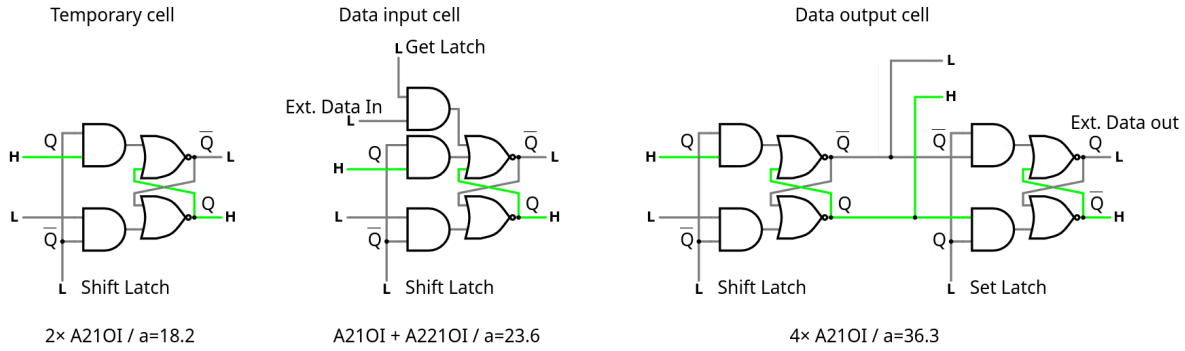
How it works

First look at the projects above. Refresher:



In this TAP system we don't need the sophisticated demux-mux machinery that splits and merges the full-speed bitstream. Let's keep a rate of one bit per byte (think: SPI!) and a single chain (so far), let's KISS because size matters.

Then take one RSFF made from a couple of sg13g2_a21o_1 (area: 18.2) and add some features such as a second FF or another data input.



By comparison:

- sg13g2_dlhq_1 : area=30.84480
- sg13g2_dfrbpq_1 : area=48.98880

Note: The scan chain has a granularity of 4 steps but only 3 actual data bits. Clock pulses should always be in bursts of 8, each burst provides one bit, so the bits are grouped by 3. This implies that each transaction will certainly consist of sequences of 3 bytes over SPI.

.

.

How to test

The pins are :

- SC_RESET clears the counter's state and the contents of the scan chain.
- SC_CLK advances the pulse counter/generator. 8 pulses advance the data by 1 bit.
- SC_DIN is the serial data input, must be set before clocking 8 pulses.
- SC_DOUT is the serial data output
- SC_GET is a control signal that transfers external data, in parallel, into the scan chain (if the cell allows it)
- SC_SET is a control signal that transfers the scan chain's value into the auxiliary latch for longer-term storage.
- DO0 to DO8 are extra output pins that are controlled by the scan chain and updated by a strobe on SC_SET
- DI0 to DI7 are extra input pins that are read into the scan chain during a strobe on SC_GET
- Count_Enable lets an internal free-running counter count. It is read by the scan chain during a strobe on SC_GET so it must be "frozen" to be sampled.

Pro tip : to save even more space, the GET strobe only sets bits in the scan chain. So you have to strobe RESET low first, which pre-clears the data.

Structure of the scan chain :

For speed/convenience,

- the output/SET bits are located near the SC_DIN pin so they require the least shifting
- the input/GET bits are located closest to the SC_DOUT pin so they are most immediately available.

Thus we have 24 bits stored in the shift register, from SC_DIN to SC_DOUT:

- 9 bits SET to the output port (DO0-DO8)
- 8-bit LFSR (7 LSB visible, controlled by the internal clk and reset, enabled by Count_Enable)
- 8 bits GET from the input port (DI0-DI7)

They are in MSB-first order, but this is only for convenience here.

.

Speed

It's an ASIC so it will be ... fast. The Johnson counter can easily reach 200MHz. It divides the clock by 8 and prevents most risks of setup&hold violations because the pulses do no olverlap, so in fact it could run even faster. There are very nice topologies that could be implemented...

Then it gets ugly. Experience with the other shift registers (SISO8xx) have shown that

- The synthesiser has no clue what this thing does, or how, and tries to optimise for the wrong parameters. Asynchronous logic is not its domain of excellence.
- Place and route are big offenders. Do it manually or script it.

Anyway, another project has reached a depth of close to 800 bits, so a chain of 200 bits would still work pretty well.

External hardware

Hook it up to a microcontroller or CPU. Likely a Raspberry Pi. Software will be written, let's tape this chip out first.

What next?

This is only a first, quick try. There are 2 ways to make it even better:

- Make "macros" that hide the characteristics from the synthesiser's sight and optimise the place&route.
- The original DTAP project is half-duplex and defines only 3 or 4 pins : CLK, R/W, with a split or shared serial in and out pin. The SC_GET and SC_SET signals should be controlled internally by a Finite State Machine to reduce the number of pins.

Stay tuned.

.

.

.

.

.

.

.

.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	DI0	DO0	SC_RESET
1	DI1	DO1	SC_CLK
2	DI2	DO2	SC_GET
3	DI3	DO3	SC_SET
4	DI4	DO4	SC_DIN
5	DI5	DO5	SC_DOUT
6	DI6	DO6	DO8
7	DI7	DO7	Count_Enable

Delta Wing Flight Control Mixer with PWM Output

by **Markus Jsaper Lassen**

`0627` `50 MHz` `HDL Project`

`github.com/Paafu/Create-the-GDS`

> "*Digital flight control system for a delta wing aircraft that processes control inputs, applies stabilization logic, and generates PWM signals for elevon and rudder actuators using a time-multiplexed output interface.*"

How it works

This project implements a digital flight control mixer for a delta wing aircraft using a limited 8-bit IO interface.

Control and sensor data are received through the 8-bit input bus (`ui_in`) using a simple command-based protocol. Data is first sent as a command word to select a target register, followed by a data word to update that register. These registers represent control inputs and internal system states.

The system processes these inputs using digital logic to generate actuator control signals for the left elevon, right elevon, and rudder. The control signals are converted into PWM-compatible values representing actuator positions.

Due to the limited number of output pins, the three actuator signals are time-multiplexed onto a single 8-bit output bus (`uo_out`). Each output cycle contains:

- 3-bit channel tag identifying the actuator (left elevon, right elevon, or rudder)
- 5-bit PWM value representing the actuator command

This allows multiple control channels to be transmitted efficiently over a constrained interface.

How to test

1. Provide an 8-bit input word on `ui[7:0]`.
2. Send a command word of the form:
    - `0xFF` in the upper 8 bits
    - register ID in the lower 8 bits

3. Follow with a data word containing the value to be written to that register.
4. Observe the output on `uo[7:0]`. The output cycles through actuator channels and provides:
    - Upper 3 bits: actuator identifier
    - Lower 5 bits: PWM value
5. Decode the output stream to extract individual actuator commands:
    - `001` → left elevon
    - `010` → right elevon
    - `011` → rudder

External hardware

This design is fully digital and does not require external hardware to operate.

For practical use, the output can be connected to:

- A microcontroller or FPGA to decode the multiplexed PWM data
- Servo drivers or PWM generators to control physical actuators

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Data input bit 0 (LSB)	PWM data bit 0	Unused
1	Data input bit 1	PWM data bit 1	Unused
2	Data input bit 2	PWM data bit 2	Unused
3	Data input bit 3	PWM data bit 3	Unused
4	Data input bit 4	PWM data bit 4	Unused
5	Data input bit 5	Channel tag bit 0	Unused
6	Data input bit 6	Channel tag bit 1	Unused
7	Data input bit 7 (MSB)	Channel tag bit 2	Unused

M31 Mersenne-31 Arithmetic Accelerator

by **Brendan Murrell**

640 | 66 MHz | HDL Project

`github.com/brmurrell3/tt_um_brmurrell3_m31_accel`

"Hardware accelerator for modular arithmetic over the Mersenne-31 prime field"

How it works

M31-ACCEL is a hardware accelerator for modular arithmetic over the Mersenne-31 prime field (p = 2^31 - 1), designed for ZK-STARK and Plonky3 applications.

Architecture

The accelerator contains three 32-bit registers (A, B, C) and supports the following operations:

Opcode	Operation	Description	Cycles
0x0	NOP	No operation	1
0x1	ADD	A = (A + B) mod p	1
0x2	SUB	A = (A - B) mod p	1
0x3	MUL	A = (A × B) mod p	32
0x4	CLR	Clear all registers	1
0x5	MAC	A = (A + B × C) mod p	32

Modular Reduction

The design uses bit-folding for efficient Mersenne prime reduction. Since 2^31 ≡ 1 (mod p), any 62-bit product can be reduced by folding the upper 31 bits back into the lower 31 bits:

```
result = low[30:0] + high[30:0]
if result >= p: result -= p
```

Pin Interface

Input pins (ui_in[7:0]):

- During load: DATA byte to shift into selected register
- During execute: OPCODE[3:0] selects operation

Bidirectional pins (uio_in[7:0]):

- [0] CMD_EN: 1 = execute opcode, 0 = load/read register
- [1] REG_SEL[0]: Register select bit 0
- [2] RW: 1 = read, 0 = write
- [3] REG_SEL[1]: Register select bit 1

Output pins (uo_out[7:0]):

- RESULT byte during read operations (4 consecutive reads for 32-bit value)

Bidirectional output (uio_out[7:0]):

- [0] BUSY: High during multi-cycle operations (MUL, MAC)

Register Selection

REG_SEL[1:0]	Register
00	A (accumulator)
01	B (operand)
10	C (MAC operand)

How to test

Basic Test Sequence

1. **Reset:** Assert rst_n low for 5+ clock cycles, then release
2. **Load Register A:** Set CMD_EN=0, RW=0, REG_SEL=00, shift in 4 bytes (LSB first)
3. **Load Register B:** Set REG_SEL=01, shift in 4 bytes
4. **Execute ADD:** Set CMD_EN=1, ui_in=0x1, wait 1 cycle
5. **Read Result:** Set CMD_EN=0, RW=1, REG_SEL=00, read 4 bytes

Example: Compute (5 + 3) mod p

```
# Load 5 into reg_a
uio_in = 0b0000  # CMD_EN=0, REG_SEL=00, RW=0
ui_in = 0x05; clock  # Byte 0
ui_in = 0x00; clock  # Byte 1
ui_in = 0x00; clock  # Byte 2
ui_in = 0x00; clock  # Byte 3

# Load 3 into reg_b
uio_in = 0b0010  # REG_SEL=01
ui_in = 0x03; clock
ui_in = 0x00; clock
ui_in = 0x00; clock
ui_in = 0x00; clock

# Execute ADD
uio_in = 0b0001  # CMD_EN=1
ui_in = 0x01    # ADD opcode
```

```
clock

# Read result from reg_a
uio_in = 0b0100  # CMD_EN=0, RW=1, REG_SEL=00
clock; result[7:0] = uo_out    # 0x08
clock; result[15:8] = uo_out   # 0x00
clock; result[23:16] = uo_out  # 0x00
clock; result[31:24] = uo_out  # 0x00
# Result: 8
```

Testing Multiplication

For MUL and MAC operations, poll the BUSY signal (uio_out[0]) and wait for it to go low before reading the result. These operations take 32 clock cycles.

Important: Read Counter Reset

The read counter (used to cycle through the 4 bytes of a register) increments whenever RW=1 and CMD_EN=0, regardless of BUSY state. If you poll BUSY with RW=1, the read counter will advance and subsequent register reads will be misaligned.

Recommended pattern: Issue a NOP (CMD_EN=1, opcode=0x0) for one cycle before starting a register read sequence. This resets the read counter to byte 0.

Verification

The design includes a comprehensive test suite with 34 tests covering:

- Register load/read operations
- All arithmetic operations (ADD, SUB, MUL, MAC)
- Edge cases (overflow, underflow, P-1 values)
- Timing verification (32-cycle multiply)
- Mathematical properties (commutativity, distributivity)

External hardware

No external hardware required. The accelerator communicates via the standard Tiny Tapeout I/O pins.

For demonstration purposes, a microcontroller (e.g., RP2040, ESP32) can be used to:

- Load operands and execute operations
- Display results on an LCD or serial terminal
- Benchmark performance vs. software implementation

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	DATA0/OPCODE0	RESULT0	CMD_EN (in) / BUSY (out)
1	DATA1/OPCODE1	RESULT1	REG_SEL[0] (input)
2	DATA2/OPCODE2	RESULT2	RW (input)
3	DATA3/OPCODE3	RESULT3	REG_SEL[1] (input)
4	DATA4	RESULT4	—
5	DATA5	RESULT5	—
6	DATA6	RESULT6	—
7	DATA7	RESULT7	—

idk

by **jonas**

0641 Wokwi Project

github.com/jfho/tt-chip

wokwi.com/projects/456578179564131329

"not quite sure"

How it works

some adders

How to test

turn on and off inputs 1-4

External hardware

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

Project Pinout

Digital Pins

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

SID Voice Synthesizer

by **shue**

`0642` `24 MHz` `HDL Project`

`github.com/rrrh/tiny_sid_chip`

"Triple SID voice synthesizer with flat memory-mapped control and 8-bit PWM audio output"

How it works

This is a triple-voice SID (MOS 6581-inspired) synthesizer without the filters because lack of time. It runs at 24 MHz with a ÷4 clock enable producing a 6 MHz voice pipeline (1 MHz effective per voice). A host microcontroller writes per-voice registers through a flat memory-mapped parallel interface and the chip produces 8-bit PWM audio output on `uo_out[0]`.

Architecture:

- **Flat register interface** – rising-edge-triggered writes via `ui_in[7]` (WE), `ui_in[4:3]` (voice select), `ui_in[2:0]` (register address), `uio_in[7:0]` (data). No SPI or I2C overhead.
- **3-voice pipelined datapath** – a ÷4 clock divider produces a 6 MHz clock enable from the 24 MHz system clock. A mod-6 slot counter cycles through voices 0/1/2, giving each voice a 1 MHz effective update rate. 24-bit phase accumulators with 16-bit frequency registers provide 0.06 Hz resolution (matching the original C64 SID).
- **Waveform generation** – four waveform types (sawtooth, triangle, variable-width pulse, noise via shared 15-bit LFSR), AND-combined when multiple waveforms are selected. Sync and ring modulation are fully implemented with circular cross-voice connections (V0←V2, V1←V0, V2←V1).
- **ADSR envelope** – 8-bit envelope (256 levels) per voice with per-voice ADSR parameters, per-voice 15-bit rate counters with 16 SID-accurate non-power-of-2 period values (matching MOS 6581/8580), secondary 5-bit exponential counter with 6 breakpoints for exponential decay, and a 4-state FSM (IDLE/ATTACK/DECAY/SUSTAIN). 16 rate settings from 2.3 ms to 8 s per full traverse.
- **3-voice mixer** – accumulates the three 8-bit voice outputs (8×8 waveform×envelope product, upper byte) into a 10-bit accumulator and divides by 4 to produce an 8-bit mix.
- **Analog filter chain** – Removed
- **PWM audio** (`pwm_audio`) – single instance on `uo_out[0]`. 8-bit PWM with a 255-clock period (94.1 kHz at 24 MHz).

Register map — full address = `{voice_sel[1:0], reg_addr[2:0]}`, selected by `ui_in[4:3]` (voice_sel) and `ui_in[2:0]` (reg_addr):

Addr	Register	Description
Voice 0	—	—
0x00	freq_lo[0]	Frequency low byte [7:0]
0x01	freq_hi[0]	Frequency high byte [15:8]
0x02	pw_lo[0]	Pulse width low byte [7:0]
0x03	pw_hi[0]	Pulse width high nibble [11:8] (bits [3:0] only)
0x04	waveform[0]	{noise, pulse, saw, tri, test, ring, sync, gate}
0x05	attack[0]	attack_rate[3:0] / decay_rate[7:4]
0x04	sustain[0]	sustain_level[3:0] / release_rate[7:4]
Voice 1	—	—
0x07	freq_lo[1]	Frequency low byte [7:0]
0x08	freq_hi[1]	Frequency high byte [15:8]
0x09	pw_lo[1]	Pulse width low byte [7:0]
0x0A	pw_hi[1]	Pulse width high nibble [11:8] (bits [3:0] only)
0x0B	waveform[1]	{noise, pulse, saw, tri, test, ring, sync, gate}
0x0C	attack[1]	attack_rate[3:0] / decay_rate[7:4]
0x0D	sustain[1]	sustain_level[3:0] / release_rate[7:4]
Voice 2	—	—
0x0E	freq_lo[2]	Frequency low byte [7:0]
0x0F	freq_hi[2]	Frequency high byte [15:8]
0x10	pw_lo[2]	Pulse width low byte [7:0]
0x11	pw_hi[2]	Pulse width high nibble [11:8] (bits [3:0] only)
0x12	waveform[2]	{noise, pulse, saw, tri, test, ring, sync, gate}
0x13	attack[2]	attack_rate[3:0] / decay_rate[7:4]
0x14	sustain[2]	sustain_level[3:0] / release_rate[7:4]
Filter	—	—
0x15	fc_lo	unused
0x16	fc_hi	unused
0x17	res_filt	unused
0x18	mode_vol	[7:4] unused, [3:0] master volume
0x1C–0x1F	—	unused

Frequency formula:

The 16-bit frequency register `{freq_hi, freq_lo}` is zero-extended and added to the 24-bit phase accumulator each voice cycle:

```
f_out = freq_reg × 1,000,000 / 16,777,216  ≈  freq_reg × 0.0596 Hz
```

Resolution: 0.06 Hz. Range: 0.06 Hz (reg=1) to 3906 Hz (reg=65535). This matches the original C64 SID accumulator geometry (24-bit acc, 16-bit freq reg). Higher audio frequencies are produced as harmonics of the waveform generators.

Pulse width:

The 12-bit pulse width `{pw_hi[3:0], pw_lo[7:0]}` is compared against `acc[23:12]`. A value of `0x800` gives a 50% duty cycle.

How to test

Connect a microcontroller to the parallel interface pins and the PWM output to a low-pass filter:

1. Set frequency: write `freq_lo` (reg 0) and `freq_hi` (reg 1) for the desired voice
2. Set pulse width if using pulse waveform: write `pw_lo` (reg 2) and optionally `pw_hi` (reg 3)
3. Set ADSR: write attack/decay rates (reg 4) and sustain level/release rate (reg 5) for each voice individually
4. Start the note: write waveform register (reg 6) with the desired waveform bit(s) and gate=1
5. Stop the note: write waveform register with gate=0 to trigger release
6. Repeat for voices 1 and 2 (`ui_in[4:3]` = 01, 10) for polyphony

The write sequence for each register: set `ui_in[2:0]` = address, `ui_in[4:3]` = voice, `uio_in` = data, then pulse `ui_in[7]` high for one clock cycle.

Sync modulation: Set bit 1 of the waveform register. Hard-syncs the voice's accumulator to the sync source (V0←V2, V1←V0, V2←V1), resetting the phase on the source voice's MSB rising edge.

Ring modulation: Set bit 2 of the waveform register. XORs the sync source voice's accumulator MSB into the triangle waveform's MSB, producing bell-like tones.

External hardware

A second-order (two-stage) RC low-pass filter on `uo_out[0]` recovers analog audio from the 94.1 kHz PWM carrier. The single PWM output carries the mixed and filtered audio (filter bypass passes unfiltered mix when filter routing is disabled):

```
uo_out[0] ---[3.3k]---+---[3.3k]---+---[1uF]---> Audio Out
                      |            |
                   [2.2nF]      [2.2nF]
                      |            |
                     GND          GND
```

Each stage has fc ≈ 22 kHz, passing the full 20 kHz audio band. A third stage (same values) can be added for better carrier rejection. Connect the output to headphones (via op-amp buffer) or a line-level amplifier input.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	reg_addr[0]	pwm_out	data_in[0]
1	reg_addr[1]	—	data_in[1]
2	reg_addr[2]	—	data_in[2]
3	voice_select[0]	—	data_in[3]
4	voice_select[1]	—	data_in[4]
5	—	—	data_in[5]
6	—	—	data_in[6]
7	write_enable	—	data_in[7]

Scott's first Wokwi design

by **Scott**

643 Wokwi Project

github.com/giffel1/Scott-s-first-Wokwi-design

wokwi.com/projects/456571625108498433

“Secret”

How it works

By entering a secret 4 digit code, all the lights will turn on

How to test

The lights turn on enabling all the inputs

External hardware

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Input and 1	Draws with led	—
1	—	Draws with led	—
2	Input and 1	Draws with led	—
3	—	Draws with led	—
4	Input and 2	Draws with led	—
5	—	Draws with led	—
6	Input and 2	Draws with led	—
7	—	Draws with led	—

tophat

by **Patrick Farley**

0644 HDL Project

`github.com/pgfarley/tophat`

"Computes a prediction for a simple decision tree"

TOPHAT is a hardware decision-tree inference engine for a fixed depth-3 tree (7 internal nodes, 8 leaves) with 8 input features.

How It Works

Background

A decision tree is a supervised learning model that predicts an output by applying a sequence of simple feature-based tests (for example, `feature_i < threshold`). Training determines which tests to use and where to place them so the model separates examples effectively.

At inference time, prediction traverses from the root to a leaf based on comparison results. In hardware, the model is represented as fixed node parameters plus deterministic control flow.

For a well-known example of the kind of problem to which a decision tree can be applied, see Kaggle's Titanic - Machine Learning from Disaster.

Model Representation

TOPHAT consumes a fixed 22-byte model image:

- Bytes 0..13: 7 internal nodes, 2 bytes per node in the order [feature_id, threshold]
- Bytes 14..21: 8 leaf values, 1 byte per leaf

Field	Bits	Note
feature_id	3	Supports 8 features
threshold	8	Unsigned

Child selection is implicit from node index `i` (dense full binary tree):

- left child index: `2*i + 1`
- right child index: `2*i + 2`

Example Model Serialization

Any model can be used as long as it is serialized into the format above. The example below was generated with scikit-learn. See `test/`

`generate_golden_tree.py` for details. For more background on scikit-learn trees, see the scikit-learn Decision Trees documentation.

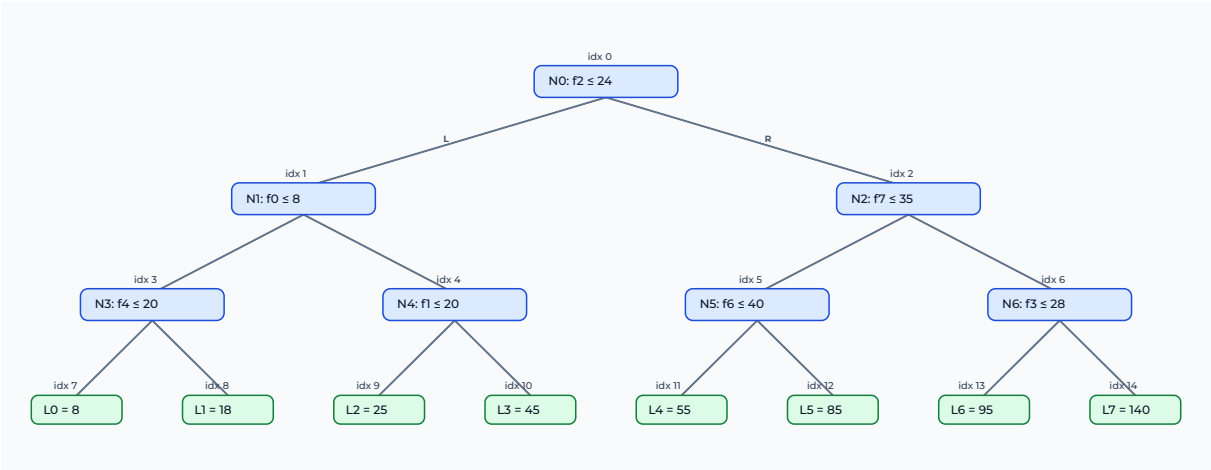


Figure 644.1: Sample TOPHAT tree and indices

Byte Range	Record	Decoded	Hex Bytes
0..1	N0	[feature=2, threshold=24]	02 18
2..3	N1	[feature=0, threshold=8]	00 08
4..5	N2	[feature=7, threshold=35]	07 23
6..7	N3	[feature=4, threshold=20]	04 14
8..9	N4	[feature=1, threshold=20]	01 14
10..11	N5	[feature=6, threshold=40]	06 28
12..13	N6	[feature=3, threshold=28]	03 1C
14..21	Leaves	[L0..L7] output values	08 12 19 2D 37 55 5F 8C

Feature Representation

TOPHAT feature input is a fixed 8-byte vector with each feature encoded as an unsigned 8-bit value. In the Titanic demo, the bytes encode passenger class, sex, age, siblings/spouses aboard, parents/children aboard, fare, port of embarkation, and whether the passenger is traveling alone, in that order.

Example Feature Serialization

Feature ID	Passenger Attribute (example value)	Hex Byte
0	Passenger class (`3rd class`)	FF
1	Sex (`male`)	FF
2	Age (`22 years`)	45
3	Siblings/spouses aboard (1)	20
4	Parents/children aboard (`0`)	00

5	Ticket fare (`7.25`)	04
6	Port of embarkation (`Southampton`)	00
7	Traveling alone (`no`)	00

Control Protocol

TOPHAT uses a byte-wide command and payload interface:

Signal	Direction	Description
`ui_in[7:0]`	input	Payload byte
`uio_in[0]`	input	`valid` strobe
`uio_in[2:1]`	input	Command select
`uio_out[3]`	output	`ready` (`~busy`)
`uio_out[4]`	output	`busy`
`uio_out[5]`	output	`pred_valid` pulse
`uio_out[6]`	output	`model_loaded`
`uio_out[7]`	output	`error_or_missing_features` (`error | ~features_loaded`)

A transfer is accepted on a rising clock edge only when `valid=1` and `ready=1`.

`uio_in[2:1]`	Name	`ui_in[7:0]` payload
`2'b00`	`CMD_MODEL`	Model byte stream
`2'b01`	`CMD_FEATURE`	Feature byte stream
`2'b10`	`CMD_CTRL`	Bit0=`run`, bit1=`clear`
`2'b11`	Reserved	Ignored

Usage

Loading a Model

1. Wait until `ready=1`.
2. Send 22 bytes with `CMD_MODEL` (`2'b00`), one accepted transfer per byte.
3. Bytes `0..13` are node records, bytes `14..21` are leaf values. See “Model Representation” above.
4. `model_loaded` (`uio_out[6]`) asserts after the 22nd byte is accepted.

Notes:

- `CMD_CTRL` with `clear=1` (`ui_in[1]=1`) resets model, features, status, and core state.
- Starting a new model stream deasserts `model_loaded` until all 22 bytes are reloaded.

Prediction

1. Load 8 feature bytes with `CMD_FEATURE` (`2'b01`), byte 0 through byte 7. See "Feature Representation" above.
2. Wait for `ready=1`, then send `CMD_CTRL` with `run=1` (`ui_in[0]=1`).
3. If model and features are both loaded, core enters `busy=1` traversal and returns with a one-cycle `pred_valid` pulse.
4. Read prediction from `uo_out[7:0]` when `pred_valid=1`.
5. A consumed run clears `features_loaded`, so features must be reloaded before the next prediction.

Status bit	Meaning
`uio_out[3]`	`ready = ~busy`
`uio_out[5]`	One-cycle `pred_valid` when a result is produced
`uio_out[7]`	`error OR ~features_loaded`

Run condition	Behavior
`model_loaded=1` and `features_loaded=1`	Normal traversal, valid prediction on `uo_out`
Missing model or features	Run is rejected, `error` is set

How to Test

Automated - simulation

The test suite lives in `test/` and requires cocotb, Icarus Verilog, and the Python packages listed in `test/requirements.txt`. From a virtualenv:

```
cd test
pip install -r requirements.txt
```

RTL simulation (cocotb + Icarus Verilog):

```
make test-cocotb   # builds the sim and runs test/test.py under
cocotb
```

The cocotb test (`test/test.py`) performs a full end-to-end check:

1. Resets the DUT and issues a `CMD_CTRL` clear.
2. Loads the 22-byte golden model image (`golden_model.bin`) byte-by-byte with `CMD_MODEL` and waits for `model_loaded`.
3. For each fixture case, loads 8 feature bytes with `CMD_FEATURE`, sends `CMD_CTRL` with `run=1`, and reads the prediction from `uo_out` on the `pred_valid` pulse.
4. Compares the DUT prediction against both the fixture's expected value and a scikit-learn golden model.

Python-only tests (no simulator needed):

```
make test-python   # runs pytest on test_golden_model.py,
test_host_client.py, etc.
```

These cover:

- **Golden-model parity** (`test_golden_model.py`): verifies the scikit-learn decision tree reproduces every fixture case.
- **Host client** (`test_host_client.py`): validates model/feature payload formatting, length checks, and CLI argument parsing against a fake transport.
- **Host transport** (`test_host_transport.py`): tests JSON-line serial framing, echo filtering, and timeout behavior with a fake serial port.
- **Example assets** (`test_example_assets.py`): ensures the published example fixtures and model image match the test golden references.

Run everything (simulation + Python tests):

```
make test-all  # clean build, then runs cocotb and pytest
```

Manual - hardware in the loop

With the design active on a Tiny Tapeout board and the RP2040 bridge firmware running:

1. Connect via USB serial (the board enumerates as a CDC device, e.g. `/dev/ttyACM0`).
2. Use the host CLI (`tools/host/tophat_host.py`) to interact:

```
# Check the bridge is alive
python tools/host/tophat_host.py ping --port /dev/ttyACM0

# Clear any previous state
python tools/host/tophat_host.py clear --port /dev/ttyACM0

# Load the 22-byte model image
python tools/host/tophat_host.py load-model --port /dev/ttyACM0 --
model test/golden_model.bin

# Run a prediction with an inline feature vector
python tools/host/tophat_host.py predict --port /dev/ttyACM0 --
features 4,6,10,12,15,20,10,18
```

The `predict` command loads the 8 feature bytes, triggers inference, and prints the prediction byte returned by the hardware.

You can also supply features from a JSON file:

```
python tools/host/tophat_host.py predict --port /dev/ttyACM0 --
features-file features.json
```

Where `features.json` is either a flat list (`[4,6,10,12,15,20,10,18]`) or an object with keys `feature_00` through `feature_07`.

Acknowledgments

Thanks to BLAKE2s Hashing Accelerator: A Solo Tapeout Journey for the inspiration.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	data_i[0]	prediction[0]	valid_i
1	data_i[1]	prediction[1]	cmd_i[0]
2	data_i[2]	prediction[2]	cmd_i[1]
3	data_i[3]	prediction[3]	ready_o
4	data_i[4]	prediction[4]	busy_o
5	data_i[5]	prediction[5]	pred_valid_o
6	data_i[6]	prediction[6]	model_loaded_o
7	data_i[7]	prediction[7]	error_or_missing_features_o

Switch deBounce for Rotary Encoder

by **Andreas Birk Gustafson**

`0645` `1 Hz` `Wokwi Project`

`github.com/GustafsonA/TT26`

`wokwi.com/projects/456577873995405313`

"This design will take 2 inputs from a rotary encoder and determine rotation & direction."

How it works

Connect the rotary encoder to input A ("A_in") and input B ("B_in"). The output will then be a debounced signal to used with i.e. a counter, having clk & up/down on output 1 & 2 respectively.

How to test

Connect to input 1+2, check output 1+2

External hardware

Rotary encoder - preferribly one from an Arduino set.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	A_in	down/up	—
1	B_in	clk	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Bit-Serial Collatz Conjecture Checker

by **Tobias Senti**

`0646` `50 MHz` `HDL Project`

`github.com/TheMightyDuckOfDoom/tinytapeout-bit-serial-collatz`

"A Collatz Conjecture checker implemented in a bit-serial fashion to save area."

How it works

The design consists of three main components:

- *Main Shift Register*: stores the current number that we iteration upon
- *Step Counter Register*: stores the number of steps it took to reach 1 i.e. the orbit-length
- *State Machine*: a quite beatifull state machine coordinating the entire thing

How to test

TODO: Testing procedure

External hardware

Does not require any external hardware.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Main Register Serial Input	Main Register Serial Output	—
1	Main Register Enable	Step Register Serial Output	—
2	Step Register Read Enable	Finished Flag	—
3	Start Processing	Overflow Flag	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

First tinytapeout 234

by **Anil Thilsted**

0647 Wokwi Project

github.com/anilthilsted/tinytapeout_first

wokwi.com/projects/456571568465442817

"Just beginning test"

How it works

Just testing how wokwiki can convert to GDS. Add one AND and one OR gate between inputs and a digital clock output.

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	Switch0	Output0	—
1	Switch1	Output1	—
2	Switch2	Output2	—
3	Switch3	Output3	—
4	Switch4	Output4	—
5	Switch5	Output5	—
6	Switch6	Output6	—
7	Switch7	Output7	—

Piggybag

by **Phuritat Suntiphap**

0648 50 MHz HDL Project

github.com/Fing2525/piggy_bag

"Count the amount of coin"

How it works

description: "A piggy bank coin counter that detects and counts four types of coins: 10, 5, 2, and 1 baht.then show the amount of each type of coin via uart tx"

How to test

let input high and read the value from uart tx.if the value from uart tx match the amount of input high pulse then the circuit is valid.

External hardware

brightness sensor button and switch

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	clk	o_Tx_Active	—
1	reset_n	o_Tx_Done	—
2	sensor0	o_Tx_Serial	—
3	sensor1	—	—
4	sensor2	—	—
5	sensor3	—	—
6	sw0	—	—
7	—	—	—

Tiny Tapeout First Design

by **Andrei Enache**

0649 Wokwi Project

github.com/Skillygonzales/tiny_tapeout_demo

wokwi.com/projects/456571626036491265

"Testing Basic Gates"

How it works

Several gates are connected to 6 input pins.

How to test

Set inputs 0 1 3 4 6 7 and check the output expected result

Input 0 -> 1 Input 1 -> 0 Input 3 -> 1 Input 4 -> 1 Input 6 -> 1 Input 7 -> 0

The AND, OR, NAND and XOR gate should output 1.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input 0	output and	—
1	input 1	output or	—
2	—	output nand	—
3	input 3	—	—
4	input 4	—	—
5	—	—	—
6	input 6	output xor	—
7	input 7	—	—

SnakeGame

by **stacu**

`650` `25.175 MHz` `HDL Project`

`github.com/StaCu/ttihp-snake-game`

"Game of Snake"

TT Snake Game

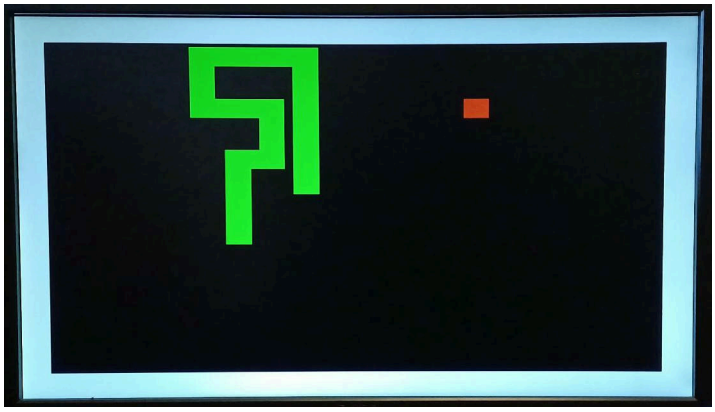


Figure 650.1: Snake Game

How it works

Snake is a simple video game where the player controls a snake. The goal is to eat food while preventing the snake from biting itself or moving into the walls. Every time the snake eats food it gets a bit longer, increasing the difficulty.

The game is won if the snake fills the entire area.

The current state of the game is displayed on a VGA monitor and the player can control the snake using four buttons.

How to test

The clock input frequency must be set to the VGA frequency of `25,175,000 Hz`.

Connect the VGA PMOD to the output pins.

function	uo_out	polarity
R1	uo_out[0]	—
G1	uo_out[1]	—
B1	uo_out[2]	—

VSync	uo_out[3]	0 during image, 1 during pulse
R0	uo_out[4]	—
G0	uo_out[5]	—
B0	uo_out[6]	—
HSync	uo_out[7]	0 during image, 1 during pulse

Connect the control buttons to the input pins as follows.

function	ui_in	optional?
up	ui_in[0]	no
down	ui_in[1]	no
left	ui_in[2]	no
right	ui_in[3]	no
pause	ui_in[4]	yes (if 0)
restart	ui_in[5]	yes (if 0)

The game starts once the button of a valid input direction has been pressed.

The game speed can be changed by pressing up/down while asserting restart. It is linked to the VGA display refresh rate with a controllable factor (0-7), which slows down the game speed accordingly. Default is 7, which results in 4 updates per second.

Additionally, the game provides an audio output and exposes four signals about the game state that can be used to add external hardware, e.g. a scoreboard or timer.

function	uio_out	info
failure	uio_out[0]	high until restart
success	uio_out[1]	high until restart
eat	uio_out[2]	> 100 cycles high & low
tick	uio_out[3]	> 100 cycles high & low
audio	uio_out[7]	pwm audio output

External hardware

Playing the game requires the VGA PMOD, Audio PMOD (optional), four buttons for movement controls, and two optional buttons for pause and restart.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	up	vga r1	failure
1	down	vga g1	success
2	left	vga b1	eat
3	right	vga vsync	tick
4	pause	vga r0	—
5	restart	vga g0	—
6	—	vga b0	—
7	—	vga hsync	audio

JayF-HA

by **Jfvind**

0651 Wokwi Project

github.com/Jfvind/ttihp-wokwi-gds

wokwi.com/projects/456571702278493185

"Half adder to seven segment display decimal"

How it works

Simple HalfAdder to seven-segment display in decimal

How to test

Seven-segment displays how many of the two switches are on. Flip the switches, the outputs should be as below. "00" -> 0, "01" -> 1, "10" -> 1, "11" -> 2

External hardware

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input a	sevseg 0	—
1	input b	sevseg 1	—
2	—	sevseg 2	—
3	—	sevseg 3	—
4	—	sevseg 4	—
5	—	sevseg 5	—
6	—	sevseg 6	—
7	—	sevseg 7	—

Tiny Tapeout Amaury Basic test

by **Amaury**

0653 | 10 kHz | Wokwi Project

github.com/Ora-ng3/tiny-tapeout-basic

wokwi.com/projects/456576238411624449

"Je kiffe l'électro :smiley_rock:"

How it works

It's just a copy of the initial template with slight modifications

How to test

Just switch buttons

External hardware

Buttons and 7 segment 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	—

USB CDC (Serial) Device

by **Uri Shaked**

0654 | 48 MHz | HDL Project

`github.com/urish/tt-usbcdc-device`

"USB to UART bridge, 115200 baud rate"

How it works

A USB CDC to UART bridge, based on tinyfpga_bx_usbserial.

How to test

1. Connect `usb_p` and `usb_n` pins to D+ / D- USB pins either through 68 ohm resistors or directly (the resistors are recommended, but not mandatory).
2. Connect a 1.5k ohm resistor between `dp_pu_o` and `usb_p` (D+).
3. Connect the RX and TX pins to a UART device or to a logic analyzer.
4. Set the clock frequency to 48 MHz.

The device should appear as a serial port on your computer, with vendor_id=1209 and product_id=5454 (https://pid.codes/1209/5454/). The baud rate for the UART interface is hardcoded at 115200.

Demo mode

Set `ui_in[0]` high to enable demo mode. In this mode, the device sends "Tiny Tapeout!" over USB once per second. UART RX input is ignored while demo mode is active.

External Hardware

USB breakout board, 1.5k ohm resistor

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	demo_mode	—	usb_p
1	—	—	usb_n
2	—	—	dp_pu_o
3	RX	—	—
4	—	TX	—
5	—	—	—

#	Input	Output	Bidirectional
6	—	—	—
7	—	configured	—

TTIHP26a_Luke_Meta

by **Luca Pezzarossa**

0655 | 50 MHz | HDL Project

github.com/lucapezza/ttIHP26a_luke_meta

"A demostrator of metastability effects."

A configurable circuit for generating and measuring flip-flop metastability events using tunable clock skew.

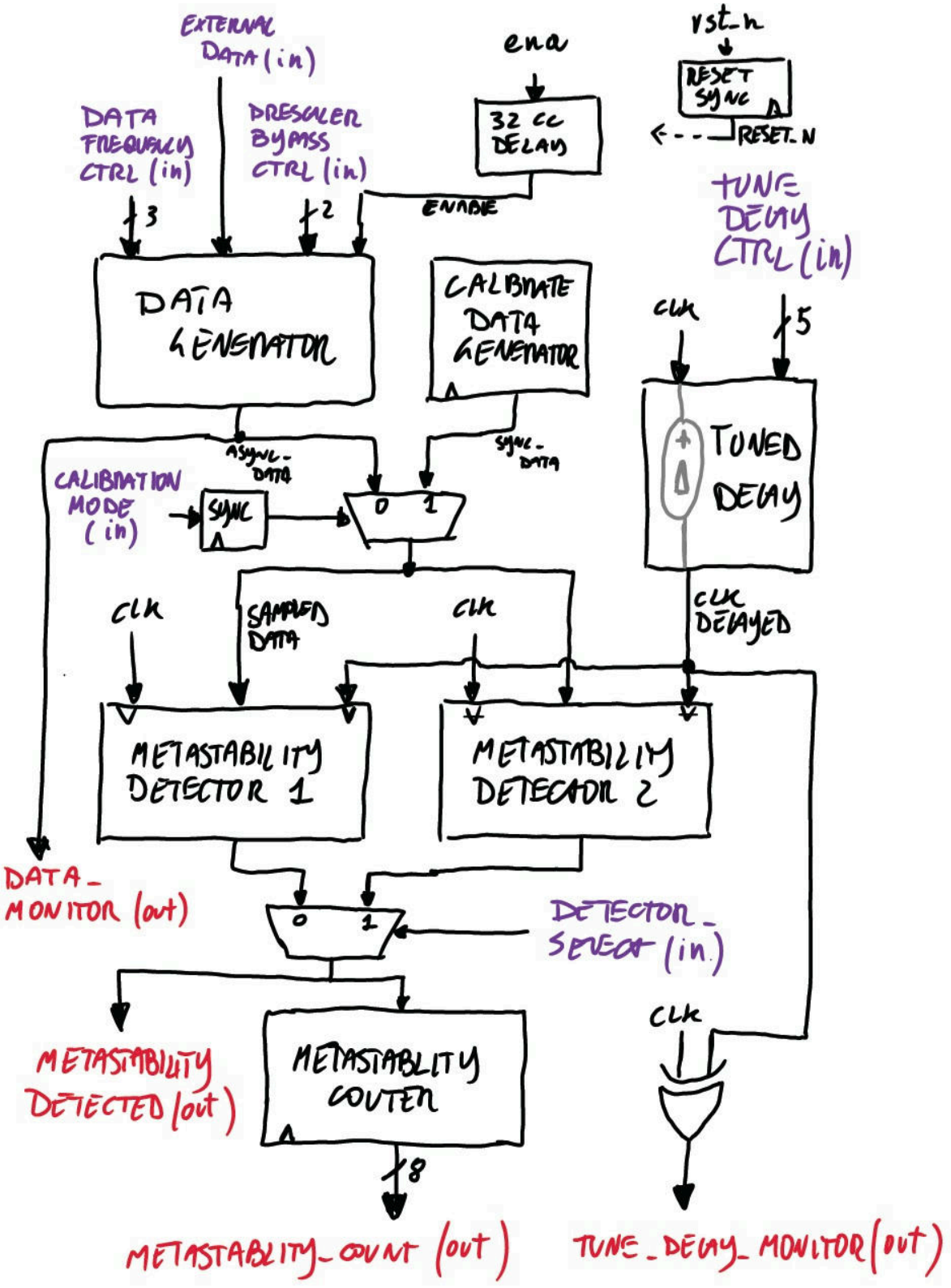
How it works

Luke Meta is a metastability characterization circuit. It generates metastability by sampling a locally generated asynchronous data signal.

A metastability detector observes the sampled signals and identifies events. By varying a tunable delay, the circuit adjusts the effective metastability resolution window, allowing detection of events that persist beyond such window. Two selectable implementations of the metastability detector are provided. The selected detector output is counted to provide a measurable indication of metastability activity.

Block diagram

A cute hand-drawn block diagram of the circuit is shown below. Red signal names denote outputs, purple denote inputs, and black denote internal signals. A small description of the most important blocks is provided in the following. For a detailed description of the input and output signal and timing table see the following sections.



Data generator

An internal data generator produces an asynchronous signal that is used as the data to be sampled. This is implemented using a configurable ring oscillator.

- A **ring oscillator** built from inverter stages provides a free-running signal.
- The oscillation frequency is adjusted by selecting different feedback tap points (`data_frequency_ctrl`), effectively changing the ring length.
- Optional **division stages** provide lower-frequency versions of the oscillator output.
- A **bypass path** allows selecting between external data, raw oscillator output, or divided signals (`prescaler_bypass_ctrl`).

The generator produces transitions that are not phase-aligned with the system clock, which is essential for exercising metastable conditions.

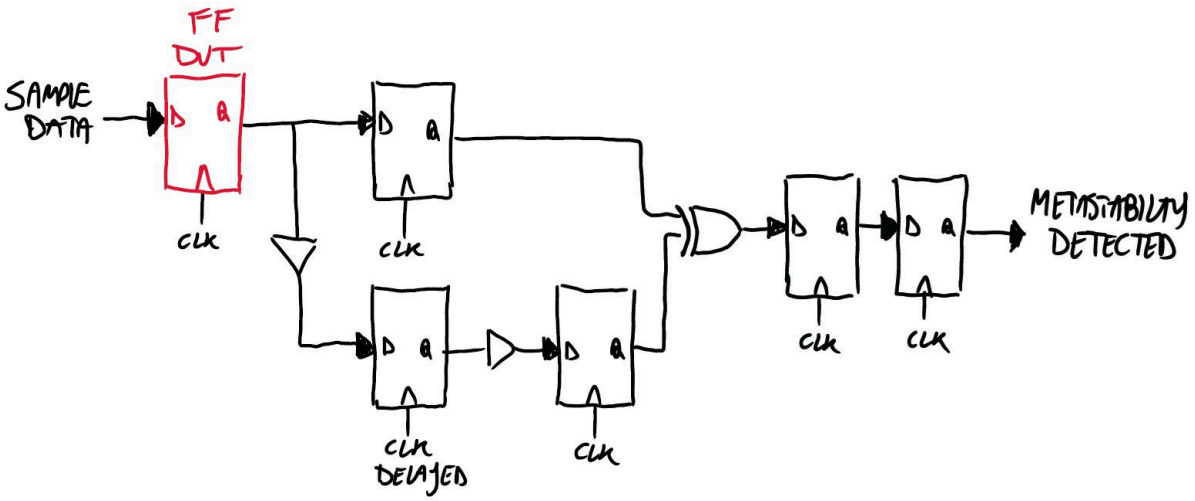
Tunable delay

The delay between `clk` and `clk_delayed` is implemented using a selectable inverter-based delay line with multiple tap points. The control signal `tune_delay_ctrl[4:0]` selects one of the available taps along the chain, where each tap corresponds to a different propagation depth through the inverter stages and therefore a different delay. This selection defines the relative phase between `clk` and `clk_delayed`. The delay setting is static during operation and determines the sampling offset used for metastability generation.

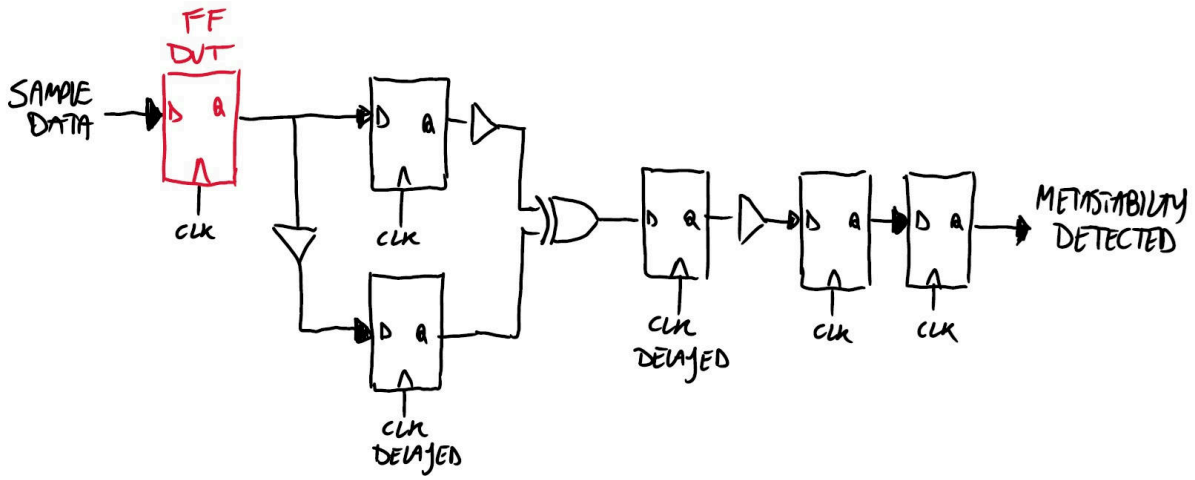
Metastability detectors

Two different detector implementations are provided, each capturing metastability through a slightly different approach. Two more cute hand-drawn circuit diagrams show the detectors' achitecture below.

Detector 1



Detector 2



The active detector is selected via `detector_select`. This allows comparison of detection sensitivity and robustness across implementations.

Synchronous calibration data

A calibration mode provides a fully synchronous stimulus generated by the `calibrate_data_generator`. This signal is phase-aligned with `clk` and therefore does not produce metastability. It is used to validate correct operation, establish a baseline, and verify that the delay line and detection logic behave as expected under non-violating conditions.

Output counting

Detected metastability events are accumulated in an internal 8-bit counter. The counter increments on each detected event and exposes the result on `count[7:0]`. The counter wraps around.

Notes

- After reset or enable release, there is an initialization period of approximately 32 clock cycles before normal operation begins. This allows the inverter chains in both the ring oscillator and the tunable delay line to stabilize.
- All control inputs are pseudo-static and must remain constant during operation. They are not intended to be changed while the system is running, as this may lead to undefined behavior. The only dynamic inputs are the calibration control (which is internally synchronized) and the `external_data` signal when bypass mode is used.
- The ring oscillator is disabled when the tile enable signal is low. This prevents the oscillator from running when the design is not selected, avoiding unnecessary switching activity in cases where power gating is not applied.

Inputs, Outputs, and Timing

Inputs

`data_frequency_ctrl [2:0]`, input: Controls the internal ring-oscillator in the data generator. This value selects one of the feedback tap lengths inside the oscillator, which changes the ring oscillation frequency. Lower and higher settings therefore select different internal delay paths and produce different asynchronous data rates. See tables below for the specific function and timing.

`data_frequency_ctrl`	# inv	Period [ps]	Freq [MHz]	Notes
`000` (0)	—	—	—	Ring oscillator off
`001` (1)	101	5050	198.02	—
`010` (2)	201	10050	99.50	—
`011` (3)	301	15050	66.45	—
`100` (4)	401	20050	49.88	—
`101` (5)	501	25050	39.92	—
`110` (6)	601	30050	33.28	—
`111` (7)	701	35050	28.53	—

Notes:

- Estimated delay assumes 25 ps per inverter stage.
- Actual delay depends on process, voltage, temperature, etc.

`prescaler_bypass_ctrl [1:0]`, input: Selects which version of the internally generated data is used. This can be:

- direct external input bypass,
- raw ring-oscillator output,
- divided-by-8 ring output,
- divided-by-16 ring output.

See tables below for the specific function.

`prescaler_bypass_ctrl`	Function
`00` (0)	External data bypass: `external_data` is routed out
`01` (1)	Ring oscillator /1: direct output of the ring oscillator
`10` (2)	Ring oscillator /8
`11` (3)	Ring oscillator /16

Notes:

- Selects the source and effective rate of the data signal.
- Bypass mode disables the internal ring oscillator.
- Division stages reduce toggle rate while preserving asynchronous behavior.

`external_data`, input: External data input for bypass mode. When the prescaler/bypass control selects bypass, this signal is used directly as the data source instead of the internal ring oscillator path.

`detector_select`, input: Selects which metastability detector implementation is connected to the output and counter path. This is intended to switch between detector 1 and 2. 0 selects detector 1, 1 selects detector 2.

`calibration_mode`, input: Enables calibration mode. This signal is synchronized internally and causes the detectors to use the synchronous calibration reference path instead of the asynchronous measurement behavior.

`tune_delay_ctrl [4:0]`, input: Selects the tap of the tunable delay line that generates the delayed clock used by the detector logic. Internally this chooses one of 32 delay path lenghs along an inverter-chain-based delay path. See tables below for the specific function and timing.

`tune_delay_ctrl`	# inv	Estimated delay [ps]	Notes
`00000` (0)	0	0	Direct connection
`00001` (1)	30	750	—
`00010` (2)	60	1500	—
`00011` (3)	90	2250	—
`00100` (4)	120	3000	—
`00101` (5)	150	3750	—
`00110` (6)	180	4500	—
`00111` (7)	210	5250	—
`01000` (8)	240	6000	—
`01001` (9)	270	6750	—
`01010` (10)	300	7500	—
`01011` (11)	330	8250	—
`01100` (12)	360	9000	—
`01101` (13)	390	9750	—
`01110` (14)	420	10500	—
`01111` (15)	450	11250	—
`10000` (16)	480	12000	—
`10001` (17)	510	12750	—

`10010` (18)	540	13500	—
`10011` (19)	570	14250	—
`10100` (20)	600	15000	—
`10101` (21)	630	15750	—
`10110` (22)	660	16500	—
`10111` (23)	690	17250	—
`11000` (24)	720	18000	—
`11001` (25)	750	18750	—
`11010` (26)	780	19500	—
`11011` (27)	810	20250	—
`11100` (28)	840	21000	—
`11101` (29)	870	21750	—
`11110` (30)	900	22500	Maximum delay
`11111` (31)	—	—	Inverted input

Notes:

- Estimated delay assumes 25 ps per inverter stage.
- Actual delay depends on process, voltage, temperature, etc.

Outputs

`metastability_detected`, output: Real-time output from the selected detector. This signal indicates that the active detector has flagged a metastability event. It is also the signal that drives the event counter.

`data_monitor`, output: Monitor copy of the internal data signal being used for the experiment. This output is provided for observation of the actual stimulus data, whether it comes from the internal generator or external bypass selection.

`tune_delay_monitor`, output: Monitor output for the delay line. This is implemented as `clk_delayed XOR clk`, so it does not expose the delayed clock directly. Instead it produces a pulse-width or phase-difference indicator that can be measured externally to estimate the effective selected delay.

`metastability_count [7:0]`, output: Running count of detected metastability events. This counter increments when the selected detector asserts `metastability_detected` and provides the main quantitative output of the circuit.

How to use

Configure the control inputs to select the desired data source, data rate, delay setting, and detector implementation. The primary output of interest is `metastability_count`, which reflects the number of detected events under the chosen configuration. By running experiments across different data rates and delay settings, the circuit can be used to characterize metastability-related parameters of the flip-flop under test.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	data_frequency_ctrl[0] (in)	metastability_count[0] (out)	metastability_detected (out)
1	data_frequency_ctrl[1] (in)	metastability_count[1] (out)	data_monitor (out)
2	data_frequency_ctrl[2] (in)	metastability_count[2] (out)	tune_delay_monitor (out)
3	prescaler_bypass_ctrl[0] (in)	metastability_count[3] (out)	tune_delay_ctrl[0] (in)
4	prescaler_bypass_ctrl[1] (in)	metastability_count[4] (out)	tune_delay_ctrl[1] (in)
5	external_data (in)	metastability_count[5] (out)	tune_delay_ctrl[2] (in)
6	detector_select (in)	metastability_count[6] (out)	tune_delay_ctrl[3] (in)
7	calibration_mode (in)	metastability_count[7] (out)	tune_delay_ctrl[4] (in)

nand_gate

by **exp10r3**

0657 Wokwi Project

github.com/exp10r3/nand_gate

wokwi.com/projects/456571798679348225

"an and gate"

How it works

A simple nand gate

How to test

Use a nand gate truth table and apply values on input pins accordingly and monitor the output

External hardware

None.

Project Pinout

Digital Pins

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

Spongent-88 Hash Accelerator

by **Stefan Aeschbacher**

`0658` `50 MHz` `HDL Project`

`github.com/imix/ttihp-spongent88`

"Hardware accelerator for the Spongent-88/80/8 lightweight hash function."

How it works

This chip implements the **Spongent-88/80/8** lightweight hash function as a hardware accelerator, designed as the cryptographic primitive for **Winternitz One-Time Signatures (W-OTS)** — a post-quantum secure signature scheme.

Spongent-88/80/8

Spongent is a sponge-based hash function optimised for extremely constrained hardware (Bogdanov et al., CHES 2011). The 88/80/8 variant has:

- **88-bit internal state**, split into an 8-bit rate and 80-bit capacity
- **45 permutation rounds** per absorption step
- A round function of: round counter injection → S-box layer → bit permutation (pLayer)

Each round applies 22 parallel 4-bit S-box lookups, a zero-gate bit permutation `P(i) = (i × 22) mod 87`, and XORs a 6-bit LFSR counter into both ends of the state (forward into bits [5:0], bit-reversed into bits [87:82]).

The permutation is implemented with **2-round unrolling**: two full rounds per clock cycle (22 double-round cycles + 1 single-round cycle for round 44), giving a latency of exactly **23 cycles per permutation call**.

Sponge construction

The host absorbs message bytes one at a time: each byte is XORed into the rate portion (`state[7:0]`) and the permutation is triggered. After all message bytes plus padding are absorbed, the full 88-bit state is the digest. Padding follows the pad10*1 rule (single byte `0x01` to set the first pad bit, `0x80` to set the last — for a byte-aligned message this collapses to `0x81`).

W-OTS use case

W-OTS with Winternitz parameter `w=16` uses 25 hash chains of depth up to 15. Each chain step is one Spongent-88 call. The chip accelerates all 25 × 15 = 375 permutations needed to sign a message; at 50 MHz this takes approximately **190 µs** per signature (25 cycles × 375 calls at 50 MHz). Key management and protocol logic run in software on the host (RP2040).

Register interface

The chip is controlled through a simple byte-serial register interface over the TinyTapeout bidirectional pins:

Signal	Direction	Description
`ui_in[7:0]`	input	data byte to write
`uo_out[7:0]`	output	current digest byte (LSB-first)
`uio[2:0]`	input	register address
`uio[3]`	input	write strobe (rising-edge triggered)
`uio[4]`	input	read strobe — advances output byte at addr 2
`uio[0]`	output	**busy** — high while permutation is running
`uio[1]`	output	**out_valid** — high after squeeze until next reset

Register map:

Addr	Direction	Action
0	write `0`	Reset: zero the sponge state, clear `out_valid`
0	write `1`	Squeeze: latch 88-bit digest into output shift register
0	write `2`	Hash: absorb pad byte `0x81` then auto-squeeze (no manual padding needed)
1	write `b`	Absorb: XOR byte `b` into `state[7:0]`, run 45-round permutation
2	read strobe	Advance output shift register to next digest byte

Timing: one absorb call takes **25 clock cycles** (1 load + 23 permutation rounds + 1 capture). The host must poll `busy` before issuing the next command.

How to test

Using the RP2040 demo board

Connect the TinyTapeout demo board. The chip runs at 50 MHz.

Hashing a message:

```
import machine, time

# Pin assignments (TinyTapeout demo board)
# ui_in  → 8 GPIO pins driving the data byte
# uio_in → 5 GPIO pins: [4]=rd_en, [3]=wr_en, [2:0]=addr
# uio_out → 2 GPIO pins: [1]=out_valid, [0]=busy
# uo_out → 8 GPIO pins for reading digest bytes

def write_reg(addr, data):
    set_ui(data)
```

```
    set_uio_low(addr)           # wr_en=0, let wr_prev settle
    time.sleep_us(1)
    set_uio_high(addr | 0x08)   # wr_en=1, rising edge
    time.sleep_us(1)
    set_uio_low(addr)           # deassert

def absorb(byte):
    write_reg(1, byte)
    while read_busy():          # poll uio_out[0]
        pass

def squeeze():
    write_reg(0, 1)             # CMD squeeze
    result = []
    for i in range(11):
        result.append(read_uo_out())
        if i < 10:
            set_uio_high(2 | 0x10)  # rd_en=1, addr=2
            time.sleep_us(1)
            set_uio_low(0)
            time.sleep_us(1)
    return bytes(result)

# Hash b'\xAB\xCD\xEF' using hardware padding (CMD=2)
write_reg(0, 0)           # reset
absorb(0xAB)
absorb(0xCD)
absorb(0xEF)
write_reg(0, 2)           # hash: absorbs 0x81 pad byte and auto-
squeezes
while read_busy():        # wait for pad permutation
    pass
digest = []
for i in range(11):
    digest.append(read_uo_out())
    if i < 10:
        set_uio_high(2 | 0x10)  # advance output
        time.sleep_us(1)
        set_uio_low(0)
        time.sleep_us(1)
print(bytes(digest).hex())
```

Using the cocotb simulation

```
cd test
pip install -r requirements.txt
make          # RTL simulation (iverilog + cocotb)
make WAVES=1  # also dump FST waveform (open with GTKWave or
Surfer)
```

Nine test cases run automatically:

1. **`test_single_byte_absorb`** — absorbs 6 different single bytes, verifies digest
2. **`test_multi_byte_absorb`** — multi-byte sequences up to 11 bytes
3. **`test_absorb_timing`** — asserts exactly 25 cycles per absorb
4. **`test_out_valid_flag`** — checks `out_valid` transitions
5. **`test_reset_clears_state`** — same input after reset gives same digest
6. **`test_absorb_while_busy_ignored`** — writes during busy are silently dropped
7. **`test_reference_kat_components`** — validates Python model against published reference vectors (`sBoxLayer` and `pLayer` KATs, full LFSR sequence), then confirms DUT matches the validated model
8. **`test_vs_readable_crypto_reference`** — cross-checks against the independent joostrijneveld/readable-crypto implementation
9. **`test_hash_command`** — verifies CMD=2 applies pad `0x81` and auto-squeezes, matching `hash88()` from the reference model

Run the Python reference model standalone (no simulator needed):

```
cd test
python3 spongent88_ref.py
```

This prints the LFSR sequence, S-box checks, pLayer KAT, and digest values for standard inputs — useful for quickly catching spec mismatches before simulation.

Known-answer test vectors

From the BenchSpongent reference implementation and joostrijneveld/readable-crypto:

Input	Expected output
`sBoxLayer(0x0123456789ABCDEF012345)`	`0xEDB0214F7A859C36EDB021`
`pLayer(0x0123456789ABCDEF012345)`	`0x00FF003C3C333333155555`
LFSR[0..4]	`0x05, 0x0A, 0x14, 0x29, 0x13`

Hash KAT vectors (absorb single byte, no padding, squeeze full 88-bit state, LSB first):

Input byte	Digest (hex, 11 bytes)
`0x00`	`82f3cecf167feb3981c07c`
`0x01`	`0842dc1b6c7399eb92f540`
`0x80`	`a0623e32cd5a6bba0b304f`
`0xFF`	`fe511649a2fa375bf97aa3`
`0xA5`	`82b032622cbefe65b01911`

External hardware

No external hardware required. The chip is self-contained and communicates entirely through the standard TinyTapeout pin interface.

For W-OTS use, the host microcontroller (RP2040 on the demo board) handles:

- Random private key generation (using its hardware RNG)
- Key and signature storage (external flash or PSRAM recommended for full key sets)
- W-OTS protocol logic (chain iteration, checksum, message formatting)
- Padding bytes before the final absorb

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	data_in[0]	data_out[0]	busy
1	data_in[1]	data_out[1]	out_valid
2	data_in[2]	data_out[2]	addr[0]
3	data_in[3]	data_out[3]	addr[1] / wr_strobe
4	data_in[4]	data_out[4]	addr[2] / rd_strobe
5	data_in[5]	data_out[5]	—
6	data_in[6]	data_out[6]	—
7	data_in[7]	data_out[7]	—

2 Digit Display

by **Mathias Sørensen**

0659 Wokwi Project

github.com/MathiasKES/TinyTapeout2026-wokwi

wokwi.com/projects/456573098517424129

"Uses input 1 and 2 to display a 2-digit number on a 7-segment display"

How it works

Uses input 1 and 2 to display a 2-digit number on a 7-segment display. This works by using 'and' and 'or' gates for switching between digits.

How to test

Toggle either input 1 or 2 or both.

External hardware

Seven segment display

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Connects to an and gate that checks if input 1 is not active and then switches on output pins for digit '1'. Also connects to a not gate for checking if input 1 is active and thus stops displaying a number.	Toggled by input 0 to display segment A on the 7-segment display.	—
1	Connects to an and gate that checks if input 0 is not active and then switches on output pins for digit '2'. Also connects to a not gate for checking if input 0 is active and thus stops displaying a number.	Toggled by input 0 and 1 to display segment B on the 7-segment display.	—

#	Input	Output	Bidirectional
2	—	Toggled by input 0 and 1 to display segment C on the 7-segment display.	—
3	—	Toggled by input 0 to display segment D on the 7-segment display.	—
4	—	Toggled by input 0 to display segment E on the 7-segment display.	—
5	—	—	—
6	—	Toggled by input 0 to display segment G on the 7-segment display.	—
7	—	—	—

Test

by **Kenny**

0672 Wokwi Project

github.com/kenny3010/GDS1

wokwi.com/projects/455291872587345921

"Test"

How it works

it works because it works

How to test

push start

External hardware

dunno

Project Pinout

Digital Pins

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

Hello World

by **Pia Garbarsch**

0673 Wokwi Project

github.com/Sirius-DK/456571875721383937

wokwi.com/projects/456571875721383937

"Test"

How it works

I dont know

How to test

I dont know either

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	IN0	OUT0	—
1	IN1	OUT1	—
2	IN2	OUT2	—
3	IN3	OUT3	—
4	IN4	OUT4	—
5	IN5	OUT5	—
6	—	OUT6	—
7	—	OUT7	—

Tiny Triangle Rasterizer

by **Thomas Oltmann**

`674` `25.175 MHz` `HDL Project`

`github.com/tomolt/tomolt_tiny_tapeout_ic`

"An extremely crude on-line triangle rasterizer with VGA output"

Overview

This is a crude ASIC triangle rasterizer that generates VGA signals, in a single tile. It imitates the functionality of a basic fixed-function graphics processor: It receives geometry from an external device and rasterizes it into an image signal that can be viewed on a computer monitor. Except whereas even the oldest 3D graphics processors were capable of processing hundreds or thousands of triangles per frame, this humble project only manages one measly triangle per frame[674.1].

How it works

A classic scanline rasterization algorithm is run in sync with the VGA clock. The submodule `triscan` implements a functional block that can perform the scanline algorithm for one triangle at a time. Because of space constraints, the ASIC only features a single instance of that functional block. The triscan module uses the H-Blank as a time budget to update its internal state from one row to the next. This includes finding out whether it has reached a vertex of the triangle, as well as tracking the left and right extents of the triangle. These extents are calculated using one single combined multiply-and-divide unit, which first performs a sequential multiplication followed by a sequential division of the resulting product.

How to test

Attach the TinyVGA PMOD adapter. The VGA mode is 640x480, 60Hz, 2 bits per color.

Reset the module. If this tile works as intended, you should be able to see a red triangle on a white background, with black bars to either side.

The UIO pins can be used to programmatically change the geometry (and color of the geometry) that is being rendered.

[674.1] More than one triangle per frame is possible if the geometry is swapped-out mid-frame, but only if these triangles are separated vertically.

Geometric constraints and conventions

The rasterizer expects the triangle geometry it is passed in to fulfill certain requirements. The first vertex must be the one with the smallest Y position. The second vertex must lie further left than the third vertex (the triangle must have counter-clockwise winding). V1 and V2 may have the same Y value, as may V2 and V3. But V1 and V3 may not have the same Y value (If V1.Y == V2.Y == V3.Y, there is nothing to render. Otherwise, one can reorder vertices to fulfill the requirements).

Each vertex coordinate (X or Y) has 6 bits of precision. Each step by one of a coordinate corresponds to an offset by 8 pixels.

The serial interface

The serial interface behaves like a one-way SPI (or Microwire) slave device. A bit is read from the MOSI pin on every positive edge of the SCK pin, but only if CS is low. This behaviour should correspond to SPI Mode 0.

Over this serial line, up to 8 words can be transferred (After that, it loops back to the first word). Every word is transferred as 8 bits; Values need to be padded in the higher bit positions. The bits in each word are transferred in MSB order.

Index	Word
0	Vertex V1 X coordinate
1	Vertex V1 Y coordinate
2	Vertex V2 X coordinate
3	Vertex V2 Y coordinate
4	Vertex V3 X coordinate
5	Vertex V3 Y coordinate
6	Triangle Fill Color (RRGGBB, R is LSB)
7	Background Color (RRGGBB, R is LSB)

Taking the CS pin high, then low again resets the bit- and word-position of the serial interface, allowing you to reset the serial interface (but not the geometry data) to a known state.

The clock frequency must not be faster than 12 MHz. In practice you may even need to be choose it much slower than that.

The internal data storage is immediately updated during transfer, so it should not be clocked while the triangle scanline is active. It is best to wait for a positive edge on the vsync signal supplied on a UIO pin. The hsync signal can also be used if there is sufficient vertical headroom.

External hardware

- TinyVGA PMOD adapter, atatched to a VGA-compatible display
- *Optionally*: An MCU attached to the UIO pins (the MCU on the TT demo-board is sufficient, but an external one could also be used.)

Project Pinout

Digital Pins

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

4ish bit adder

by **Erik Wallin**

0675 Wokwi Project

github.com/erik-wallin/ttihp-wokwi-template

wokwi.com/projects/456571610504973313

"Testing some adders"

How it works

4 bit adder

How to test

change inputs 0-3

External hardware

None

Project Pinout

Digital Pins

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

Tiny Tapeout N

by **N**

0676 1 Hz Wokwi Project

github.com/Nama222/Tiny-Tapeout

wokwi.com/projects/455291650157032449

"TOSO"

How it works

TODO: Explain how your project works

How to test

TODO: Explain how to use your project

External hardware

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

Project Pinout

Digital Pins

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

Simple counter

by **Karl Herman Krause**

0677 10 kHz Wokwi Project

`github.com/McHerman/Tinytapeout`

`wokwi.com/projects/456574262189506561`

"Simple counter running at approximately 1 Hz, output wired to 7segment"

How it works

The circuit uses a clock divider to generate a aprox 1hz clock which drives 4 registers. The register outputs are routed to a full-adder, and the counter value is incremented by 1.

How to test

NA

External hardware

The project uses the external 7-segment display.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	bit0 of increment	bit0 of counter	—
1	bit1 of increment	bit1 of counter	—
2	bit2 of increment	bit2 of counter	—
3	bit3 of increment	bit3 of counter	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Tiny Tapeout

by **Anh Tuan Doan**

`0678` `1 Hz` `Wokwi Project`

`github.com/Tuan9304/gds`

`wokwi.com/projects/455291594023558145`

"Display my name"

How it works

This project uses flip-flops to display the text **TUAN9304** on a 7-segment display. The state table below describes the states, outputs, and next state logic:

State	q3	q2	q1	q0	out7	out6	out5	out4	out3	out2	out1	out0	q3'	q2'	q1'	q0'	Next	Char
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	Blank
1	0	0	0	1	1	1	1	1	1	1	1	1	0	0	1	0	2	All
2	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	3	Blank
3	0	0	1	1	0	1	1	1	1	0	0	0	0	1	0	0	4	T
4	0	1	0	0	0	0	1	1	1	1	1	0	0	1	0	1	5	U
5	0	1	0	1	0	1	1	1	0	1	1	1	0	1	1	0	6	A
6	0	1	1	0	0	0	1	1	0	1	1	1	0	1	1	1	7	N
7	0	1	1	1	0	1	1	0	1	1	1	1	1	0	0	0	8	9
8	1	0	0	0	0	1	0	0	1	1	1	1	1	0	0	1	9	3
9	1	0	0	1	0	0	1	1	1	1	1	1	1	0	1	0	10	0
10	1	0	1	0	0	1	1	0	0	1	1	0	1	0	1	1	11	4
11	1	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0	12	Blank
12	1	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	2	.

How to test

1. **Reset:** Press the reset button to initialize the state machine to state 0 (0000).
2. **Observation:** Monitor the 7-segment display output. You should observe the sequence "Tuan9304" repeating or cycling.
3. **Verification:** Check that each character is formed correctly according to standard 7-segment encoding (e.g., 'T' might be represented by a specific combination of segments, often simplified or stylized for 7-segment displays).

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	1	—	—
1	2	—	—
2	3	—	—
3	4	—	—
4	5	—	—
5	6	—	—
6	7	—	—
7	8	—	—

TinyPong

by **Mark van Damme**

`0679` `25.175 MHz` `HDL Project`

`github.com/Mr-Seoul/TinyTapeOutGDS`

"Plays Pong on a VGA Monitor"

How it works

Using a 25.175 MHz clock, the top module generates the VGA Timings for visual output. Using 2 debouncing modules, the inputs are debounced for input stability. The x and y indices of the VGA module are passed onto a graphicsprocessing module, which compares the coordinates to the bounds of the paddles and ball. If not inside one of the objects, it generates a XOR background. The paddles use position and y velocity registers (they only move up and down), while the ball has a position 1 velocity register and 2 registers storing direction information for lower register usage.

How to test

The test benches are under the following github link: https://github.com/Mr-Seoul/TinyPong. You can test the code by running "sbt test", and generate the verilog by running "sbt run". Using either an FPGA or the produced ASIC, you can connect the relevant output pins to a VGA adapter and 2 buttons (not necessarily debounced) to test on a VGA monitor. When using an ASIC, remember to reset the chip by inverting the rst_n (active low) input pin.

External hardware

Requirements: VGA module, VGA cable, VGA compatible monitor/screen and 2 buttons. Remember to use at least 10k resistors when connecting buttons.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Button 1	R0	—
1	Button 2	G0	—
2	—	B0	—
3	—	VSync	—
4	—	R1	—
5	—	G1	—

#	Input	Output	Bidirectional
6	—	B1	—
7	—	HSync	—

custom_lol

by **Marius Scharz**

0680 10 kHz Wokwi Project

github.com/mar-3123/tiny_tapeout

wokwi.com/projects/455291727376311297

“sdfdf”

How it works

Hallloooooo hallooooooo ## How to test

hallooo

External hardware

led

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	clk	1	0
1	1	2	1
2	2	3	2
3	3	4	3
4	4	5	4
5	5	6	5
6	6	7	6
7	7	8	7

MJ Wokwi project

by **MJ**

0681 Wokwi Project

github.com/mjuels/tinytapeoutworkshop

wokwi.com/projects/456571856158098433

"First sample project"

How it works

The value of inputs 1 and 2 are read, and the corresponding value is shown in decimal on the display. If both 1 and 2 are set, the display shows nothing.

How to test

Set none of the inputs - The display is off. Set input 1 - Display should read 1. Set input 2 - Display should read 2. Set both of the inputs - The display is off.

External hardware

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input a	output 1	—
1	input b	output 2	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Tiny Tapeout

by **Ari Vishnu**

0682 | 10 kHz | Wokwi Project

github.com/AriVishnu-01/Tiny-Tapeout

wokwi.com/projects/455293127747668993

"I will figure it out"

How it works

I will Figure it out.

How to test

Soon I will firgure it out

External hardware

Led Display

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	or2:A	—	—
1	or2:B	—	—
2	or3:A	—	—
3	or3:B	—	—
4	or4:A	—	—
5	or4:B	—	—
6	or5:A	—	—
7	or5:B	—	—

Full Adder

by **Sepehr**

683 Wokwi Project

github.com/AAmirinejad/TinyTapeoutWorkshop

wokwi.com/projects/456571536458697729

"A full adder adds three 1-bit inputs: a, b, and cin (carry-in). It produces two outputs: sum and cout (carry-out). "

How it works

A full adder adds three 1-bit inputs: a, b, and cin (carry-in). It produces two outputs: sum and cout (carry-out).

How to test

Set the inputs and verify the outputs match the expected results:

input a b cin	output sum	output cout
0 0 0	0	0
0 0 1	1	0
0 1 0	1	0
0 1 1	0	1
1 0 0	1	0
1 0 1	0	1
1 1 0	0	1
1 1 1	1	1

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input a	output s	—
1	input b	output c	—
2	ipnut c	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—

#	Input	Output	Bidirectional
7	—	—	—

8-bit PRNG

by **Johannes Reibold**

0684 HDL Project

`github.com/joh-1x/tiny-tapeout-workshop`

"*see title*"

How it works

It creates pseudo-random 8-bit numbers using the XorShift32 PRNG.

How to test

Enter an 8-bit seed through the input pins using switches and start the design by resetting it. Press the clock button 3 times for the first valid 8-bit pseudo-random number to appear at the I/O pins. The seven segment display also shows a random hexadecimal digit based on the current state of the PRNG. By setting the input pins to a (binary) value between 1 and 16, you can influence the modulus used to generate a random digit. E.g. setting the input pins to 00000110 (6) creates random digits from 0 to 5. The standard modulus is 10.

External hardware

Input switches, seven segment display for output, something to read the random outputs at the I/O pins..

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Input bit 0	Seven segment output A	RNG output bit 0
1	Input bit 1	Seven segment output B	RNG output bit 1
2	Input bit 2	Seven segment output C	RNG output bit 2
3	Input bit 3	Seven segment output D	RNG output bit 3
4	Input bit 4	Seven segment output E	RNG output bit 4
5	Input bit 5	Seven segment output F	RNG output bit 5
6	Input bit 6	Seven segment output G	RNG output bit 6
7	Input bit 7	Seven segment output dot	RNG output bit 7

Tiny Perceptron

by **Karl H. Mose**

`0685` `HDL Project`

`github.com/karlmose/tinytapeout_wokwi_test`

"An implementation of a hashed perceptron, a branch prediction model"

How it works

This design implements a hashed perceptron, a hardware-friendly predictor often used for branch prediction. It uses external SPI RAM to store signed 8-bit weights, and communicates with a host controller via a 16-bit SPI slave interface.

The typical flow is:

1. Send up to 4 weight indices via `OP_ADD` (9-bit index each)
2. Poll with `OP_READ` until the response opcode is `RESP_VALID` — the payload contains an 11-bit signed sum of the weights
3. The sign of the sum provides a taken/not-taken prediction
4. Send `OP_UPDATE` with a sign bit to increment (taken) or decrement (not taken) all loaded weights with saturation at +127/-128
5. Poll with `OP_READ` until `RESP_UPDATE_DONE` is returned (or it will appear as the response to any subsequent command)

The RAM address space is partitioned into 4 slots (2-bit slot prefix concatenated with the 9-bit index), giving 2048 total weight entries.

Architecture

- **pred_slave_spi** — SPI slave command decoder (16-bit words, CPOL=0/CPHA=0)
- **perceptron** — Core logic: index buffer, signed accumulation, and update state machine
- **ram_interface** — SPI master to external RAM with configurable clock divider and CS wait cycles

Pin mapping

Pin	Direction	Function
ui_in[0]	in	Slave SCK
ui_in[1]	in	Slave CS (active low)
ui_in[2]	in	Slave MOSI
uo_out[0]	out	Slave MISO

uio[0]	out	RAM SPI CS
uio[1]	out	RAM SPI MOSI
uio[2]	in	RAM SPI MISO
uio[3]	out	RAM SPI SCK

SPI command protocol

All commands and responses are 16-bit words. Commands use bits [15:12] as the opcode.

Commands (host to device)

Opcode	Name	Bits [11:0]	Description
0x1	OP_ADD	[8:0] = index	Add a weight index to the buffer (max 4)
0x2	OP_UPDATE	[0] = sign (1=inc, 0=dec)	Update all loaded weights
0x3	OP_READ	unused	Request prediction result (send twice: command + dummy to clock out response)
0x4	OP_SET_CS_WAIT	[2:0] = wait value	Set RAM CS wait cycles (default 3)
0x5	OP_RESET_BUF	unused	Clear the weight index buffer and sum
0x6	OP_SET_CLK_DIV	[1:0] = divider	Set RAM SPI clock divider (0=div2, 1=div4, 2=div8 default, 3=div16)

Responses (device to host)

Non-READ commands echo the received opcode in bits [15:12] (for debugging). READ responses use dedicated response codes:

Opcode	Name	Bits [11:0]	Description
0x1	RESP_VALID	[11] = valid, [10:0] = signed sum	Prediction ready
0x2	RESP_INVALID	0	No prediction available (no weights loaded)

0x3	RESP_UPDATE_DONE	0	Weight update completed

How to test

You will need external SPI RAM connected to the bidirectional IO pins. The SPI RAM emulator on a Raspberry Pi Pico can be used for this.

Basic test sequence:

1. After reset, send `OP_READ` — expect `RESP_INVALID` (no weights loaded)
2. Send `OP_ADD` with an index to load a weight
3. Send `OP_READ` (command word), then send a dummy word (0x0000) to clock out the response — poll until `RESP_VALID`
4. Verify the sum matches the expected signed weight value
5. Send `OP_UPDATE` with sign=1 to increment, poll until `RESP_UPDATE_DONE`
6. Read back the weight to verify it incremented

External hardware

SPI RAM emulator on a Raspberry Pi Pico, connected to uio[0:3].

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	SPI slave SCK	SPI slave MISO	RAM SPI CS (output)
1	SPI slave CS	—	RAM SPI MOSI (output)
2	SPI slave MOSI	—	RAM SPI MISO (input)
3	—	—	RAM SPI SCK (output)
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

tiny-tapeout-workshop-result

by **Matthias**

0686 1 Hz Wokwi Project

`github.com/raninninn/tiny-tapeout-working-repo`

`wokwi.com/projects/455291650915156993`

"Just some messing around with a seven segment display. Open to change"

How it works

There's a MUX that takes it's inputs from inputs 0 and 1. A clock signal is used as the select pin. The MUX output is present on output 0. Inputs 0 and 1 are also forwarded to outputs 1 and 2.

How to test

Apply clock signal. Outputs 1 and 2 should reflect inputs 0 and 1. Half of the cycle, input 0 should be present on output 0, while input 1 is present on output 0 at the other half of the cycle.

External hardware

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input a	mux output	—
1	input b	copy of a	—
2	—	copy of b	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	enable	—	—

Register bank accessible through SPI and I2C

by **Caio Alonso da Costa**

0687 | 50 MHz | HDL Project

github.com/calonso88/dtu_tt_workshop

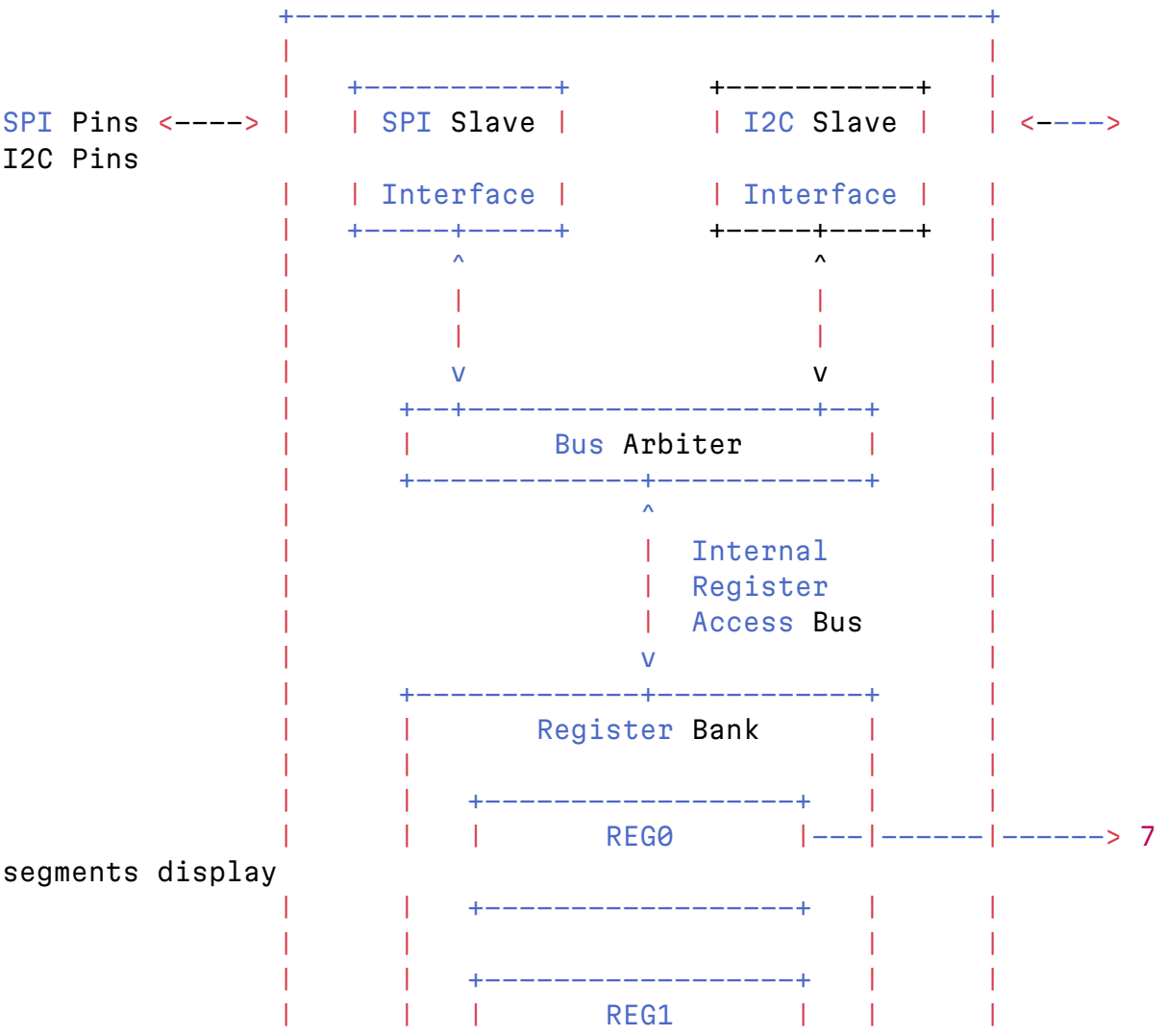
"Register bank accessible through SPI and I2C"

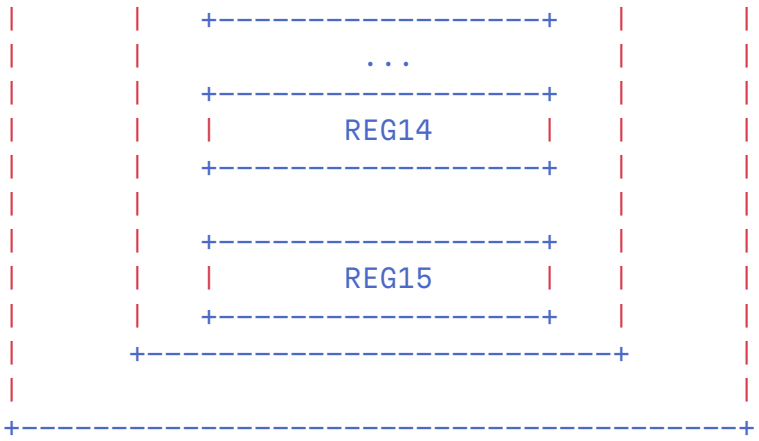
How it works

Register bank accessible throught two different serial interfaces: SPI and I2C. Use digital input to select prefered interface.

Digital input ui_in[7] = 0 selects SPI and ui_in[7] = 1 selects I2C.

Block diagram





There are 8 read/write 8 bit registers and 8 read only 8 bit registers.

Address 0 (first byte in read/write register space) drives the 7 segment display.

SPI peripheral design based on https://github.com/calonso88/tt07_alu_74181

See that design's docs for information about the SPI peripheral.

Small improvement done on the spi_peripheral module. There used to be two buffer counters (one for RX and one for TX). Since the counters are not used together, it was possible to remove one of them and use a single buffer counter. This has reduced 4 flip flops in total and some combinatorial logic as well.

Added logic to control driver for MISO. On previous submissions of this design, the MISO was always driven. Logic has been added to put MISO into high impedance when CS_N is driven high. Due to a 2-stage synchronizer, the MISO goes to high impedance after 2 clock cycles.

I2C peripheral design based on https://github.com/sanojn/tt06_ttrpg_dice

See that design's docs for information about the I2C peripheral.

Protocol specification

SPI Write (CPOL = 0, CPHA = 0)

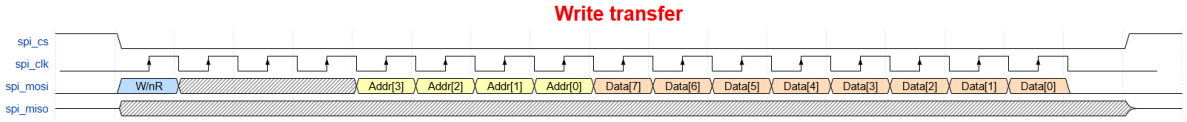


Figure 687.1: SPI Write

SPI Read (CPOL = 0, CPHA = 0)

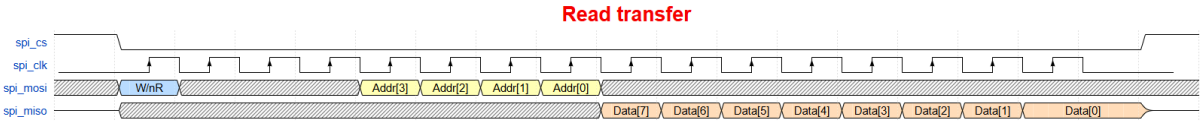


Figure 687.2: SPI Write

I2C Frame

Figure 687.3: I2C frame

Register Map

Offset	Name	Access	Reset	Description
0x00	REG0	R/W	0x00	Controls 7 segmets display on demoboard
0x01	REG1	R/W	0x00	General prupose register
0x02	REG2	R/W	0x00	General prupose register
0x03	REG3	R/W	0x00	General prupose register
0x04	REG4	R/W	0x00	General prupose register
0x05	REG5	R/W	0x00	General prupose register
0x06	REG6	R/W	0x00	General prupose register
0x07	REG7	R/W	0x00	General prupose register
0x08	REG8	RO	0xC4	Constant ID Code 1
0x09	REG9	RO	0x10	Constant ID Code 2
0x0A	REG10	RO	0xAA	Constant ID Code 3
0x0B	REG11	RO	0x55	Constant ID Code 4
0x0C	REG12	RO	0xFF	Constant ID Code 5
0x0D	REG13	RO	0x00	Constant ID Code 6
0x0E	REG14	RO	0xA5	Constant ID Code 7
0x0F	REG15	RO	0x5A	Constant ID Code 8

How to test

SPI

Use SPI1 Master peripheral in RP2040 to start communication on SPI interface towards this design. Remember to configure the SPI mode using digital inputs [0] and [1] to high (if you'd like to have CPOL=1 and CPHA=1).

Example code to initialize SPI in REPL:

```
spi_miso = tt.pins.pin_uio3
spi_cs = tt.pins.pin_uio4
spi_clk = tt.pins.pin_uio5
spi_mosi = tt.pins.pin_uio6
spi_miso.init(spi_miso.IN, spi_miso.PULL_DOWN)
spi_cs.init(spi_cs.OUT)
```

```
spi_clk.init(spi_clk.OUT)
spi_mosi.init(spi_mosi.OUT)
spi = machine.SoftSPI(baudrate=10000, polarity=0, phase=0, bits=8,
firstbit=machine.SPI.MSB, sck=spi_clk, mosi=spi_mosi,
miso=spi_miso)
spi_cs(1)
```

Example code to write 0xF8 to address[0]:

```
spi_cs(0); spi.write(b'\x80\xF8'); spi_cs(1)
```

This should set the 7 segment LED to 0xF8 which will display “t.”

Seg A - OFF, Seg B - OFF, Seg C - OFF, Seg D - ON, Seg E - ON, Seg F - ON, Seg G - ON, Seg DP - ON

Example code to read from address[0]:

```
spi_cs(0); spi.write(b'\x00'); spi.read(1); spi_cs(1)
```

The result should be 0xF8 or whatever you wrote to address[0].

I2C

Use I2C Master peripheral in RP2040 to start communication on I2C interface towards this design. Remember to configure the I2C address bits using the digital inputs [2], [3] and [4].

Example code to initialize I2C in REPL:

```
TO DO
```

Example code to write 0xF8 to address[0]:

```
TO DO
```

Example code to read from address[0]:

```
TO DO
```

External hardware

You may need to use a pull up resistors on the i2c data and i2c scl lines if not possible to configured internally on the RP2040. To be checked at a later point in time. Write to the first register to set the LEDs on the demoboard.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	cpol	spare[0]	—
1	cpha	spare[1]	i2c_sda

#	Input	Output	Bidirectional
2	i2c_addr[0}	spare[2]	i2c_scl
3	i2c_addr[1}	spare[3]	spi_miso
4	i2c_addr[2}	spare[4]	spi_cs_n
5	—	spare[5]	spi_clk
6	—	spare[6]	spi_mosi
7	peripheral selector (SPI=0/I2C=1)	spare[7]	—

neb tt26a first asic

by **Bastian Neussell**

0688 HDL Project

github.com/neb-o/tinytapeout_260207

“tbd”

How it works

yet to be determined

How to test

TBD

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	in_a	out_ct	—
1	in_b	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Test

by **Filippo**

0689 Wokwi Project

github.com/Biboulder/TinyTapeout-wokwi-template

wokwi.com/projects/456131795093444609

"And gate on first 2 inputs"

How it works

2 inputs 1 output

How to test

switches on the left to change inputs between 1 and 0

External hardware

nothing

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input a	output and	—
1	input b	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

title

by **ich**

0690 10 kHz Wokwi Project

github.com/Ancash/tt-gds

wokwi.com/projects/455291560479595521

“addieren”

How it works

a a

How to test

b

External hardware

c

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input a	7-seg a	—
1	input b	7-seg b	—
2	—	7-seg c	—
3	—	7-seg d	—
4	—	7-seg e	—
5	—	7-seg f	—
6	—	7-seg g	—
7	—	—	—

Tiny tapeout MAC unit

by **Christopher Kessler**

0691 | 10 kHz | Wokwi Project

github.com/kessler-christopher/tt_chris_test

wokwi.com/projects/453110263532536833

"ADDER_XOR_NAND_AND_Dflop_INV"

How it works

Input 0 and 1 are the Input to the Full ADDER with CIN being connected to GND. Input 2 and 3 are connected to the XOR gate. Input 4 is connected to the D flip D input. Input 5 is connected to the INVERTER. Input 6 and 7 are connected to the NAND and AND gate. OUTPUT 0 is the SUM output of the FULL ADDER and OUTPUT 1 is the COUT. OUTPUT 2 is the XOR, OUTPUT 3 is the NAND, OUTPUT 4 is the AND, Output 5 is Q, Output 6 is Q_B and Output 7 is the INVERTER

How to test

Use the DIP switches to set the inputs of the gates according to the descripton. The 7 Segment Display can be used to see the outputs.

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	FULL_A	SUM	—
1	FULL_B	CARRY	—
2	XOR_1	XOR	—
3	XOR_2	NAND	—
4	AND_1	AND	—
5	AND_2	Q"	—
6	D_FLIP	Q_B	—
7	INV	INV_B	—

Tiny Tapeout placeholder

by **Marta Alfonso**

0704 Wokwi Project

github.com/martaalfonso/pomasic

wokwi.com/projects/456579003210233857

"Counter 1 to 4"

How it works

Placeholder design

How to test

Placeholder

External hardware

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

Project Pinout

Digital Pins

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

Tiny Tapeout chip

by **Birk Weber Kock**

`0705` `1 Hz` `Wokwi Project`

`github.com/Birkwk/Chip`

`wokwi.com/projects/456573141570913281`

"A working chip design"

How it works

Taking a 4 bit input and decoding it to a decmal number displayed on a 7 segment display

How to test

change input 0-3 to test it.

External hardware

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

Project Pinout

Digital Pins

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

Linear Timecode (LTC) generator with I2C control

by **Thomas Flummer**

0706 | 24 MHz | HDL Project

`github.com/flummer/tt-um-flummer-ltc`

"Timecode generator for audio video syncronization"

How it works

This is a project that generates Linear Timecode (LTC), most commonly used for audio and/or video syncronization. LTC is a digital signal, though it is often captured and recorded as an audio signal and this project is designed to work with the Tiny Tapeout Audio PMOD, to output a signal suitable for audio and video equipment.

The project uses multiple counters to maintain time and framecount, with serial output of the LTC (80 bit frames, biphase mark code) being output on a single pin.

In addition, it's possible to control the timecode generation and the user bits in the signal using I2C.

This is the updated version (v2), with added I2C control in addition to a series of other tweaks, as the original version (included on TTIHP25a, TTSKY25a and TTGF0.2) had only the bare minimum of functionality.

How to test

Setup

The project should have a 24 MHz clock signal applied and after reset, will start out with a 01:00:00:00 timecode and starts to count.

Configuration using inputs/switches on the demo board

Framerate is by default controlled using inputs ui[2] and ui[3]

FR1/ui[3]	FR0/ui[2]	Framerate	7 Seg Debug	Comment
0	0	24	4	—
0	1	25	5	—
1	0	29.97	9	Should also use drop frame
1	1	30	3	—

Drop frame can be enabled with ui[4] (active high). It can be enabled and will be applied to all framerates, but it only really makes sense for 29.97 fps, where it should always be applied to follow LTC specification and keep the time from drifting.

The color frame flag (if timecode is syncronised to a color video signal) can be configured using ui[5] (active high) and BGF0 and BGF1 (Binary Group Flags) are configured using ui[6] and ui[7] respectively (also active high).

Configuration via I2C

As an alternative to configuring the timecode signal via input switches, it's also possible to change all of the above settings and more usign I2C by writing to a set of registers.

The base address (7 bit) is: **0x42** (0x84 for write and 0x85 for read in 8 bit representation)

To use the configuration for framerate, drop frame, color frame and BGF you need to set **bit 7** in register **0x00** (**USE** in the table below). If this bit is not set, the input pins will be used for the setup configuration and not the setup register (BGF2 is only configurable via I2C and will not be set if using input pins for setup).

The framerate currently in use will be shown in an single digit, abbreviated, human readable form on the 7 segment display. **The decimal point will be illuminated, if setup from the I2C controlled register is in use**.

Setting time via I2C

The time will start out at 01:00:00:00 after a reset. Changing the time can be done by writing to one or more of the time registers (0x01 though 0x04). The registers use the same binary coded decimal as in the output timecode signal, so that the "tens" are in the upper nibble and the "singles" are in the lower one. This makes it pretty easy to set the time eg. usign a bus pirate where you use a terminal for the I2C commands (eg. `[0x84 0x01 0x13 0x37]` to set the time to 13:37 for hours:minutes).

Reading from the time and frame registers will return the current time and framecount in the same binary encoded decimal format (eg. `[0x84 0x01] [0x85 rrrr]` with the Bus Pirate).

Setting userbits via I2C

LTC also includes a total of 8 user bit fields, each being 4 bits. Those can be set (and read back) via I2C registers 0x05 to 0x08. The definition of those are not fully defined and the order might not be ideal in all usecases, eg. if using the userbits for dates, the order of "tens" and "singles" might be less natural and flipped compared to settign the time.

I2C Register map

Addr.	Register name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Default on reset
0x00	Setup	USE	FR1	FR0	DF	CF	BGF2	BGF1	BGF0	0x00
0x01	Hours	-	-	HRSD1	HRSD0	HRSU3	HRSU2	HRSU1	HRSU0	0x10
0x02	Minutes	-	MIND2	MIND1	MIND0	MINU3	MINU2	MINU1	MINU0	0x00
0x03	Seconds	-	SECD2	SECD1	SECD0	SECU3	SECU2	SECU1	SECU0	0x00
0x04	Frame	-	-	FRMD1	FRMD0	FRMU3	FRMU2	FRMU1	FRMU0	0x00
0x05	User bits 1&2	UB1_3	UB1_2	UB1_1	UB1_0	UB2_3	UB2_2	UB2_1	UB2_0	0x00
0x06	User bits 3&4	UB3_3	UB3_2	UB3_1	UB3_0	UB4_3	UB4_2	UB4_1	UB4_0	0x00
0x07	User bits 5&6	UB5_3	UB5_2	UB5_1	UB5_0	UB6_3	UB6_2	UB6_1	UB6_0	0x00
0x08	User bits 7&8	UB7_3	UB7_2	UB7_1	UB7_0	UB8_3	UB8_2	UB8_1	UB8_0	0x00

External hardware

This should work with the audio PMOD connected to the bidirectional port, to give levels useable for audio gear. for I2C communication, you will need to use the pass through connections as SDA and SCL are also on the bidirectional port (uio[0] and uio[1] respectively).

If you have line level audio input on your computer (or using a USB audio interface), there are software that can listen to the input and show the timecode (https://timecodesync.com/ have a free to download and use tool for macOS and Windows).

It is possible to listen to this “audio”, but it is not a pleasant sound, so be careful if you use headphones or powerfull speakers

If testing with a logic analyzer or similar, uio[7] can be directly connected (3.3v digital signal) and referenced to GND.

For communicating with the device via I2C, someting like a Bus Pirate or similar can be used. Remember to add/enable pullup resistors to the 3.3v rail.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	FRAMERATE_DEBUG_7SEG	I2C_SDA
1	—	FRAMERATE_DEBUG_7SEG	I2C_SCL
2	FRAMERATE_0	FRAMERATE_DEBUG_7SEG	—
3	FRAMERATE_1	FRAMERATE_DEBUG_7SEG	—
4	DF	FRAMERATE_DEBUG_7SEG	DEBUG_SETTIME_OUT
5	CF	FRAMERATE_DEBUG_7SEG	DEBUG_I2C_OUT
6	BGF_0	FRAMERATE_DEBUG_7SEG	DEBUG_I2C_OUT
7	BGF_1	REG_CONF_ACTIVE	LTC_OUT

Hidden combination

by **Johan Dyre**

0707 Wokwi Project

github.com/johannakin1/Hemmeligkode

wokwi.com/projects/456578608558661633

"It is a hidden combination to make alle the lights light"

How it works

when you turn on the right gates all the leds will light.

How to test

try different intputs

External hardware

Led display

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	goes into and gate	—	—
1	goes to output 1	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

miniMAC

by **Yann Guidon**

0708 200 MHz HDL Project

`github.com/YannGuidon/miniMAC_tx`

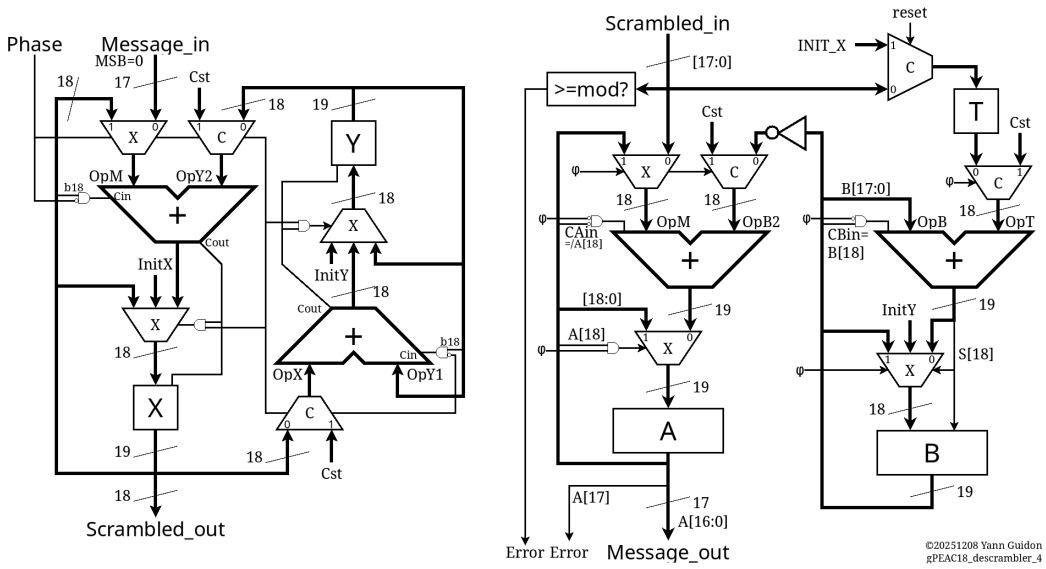
"16-bit Scrambler/Framer/Error detector for safer transmissions"

What is this miniMAC

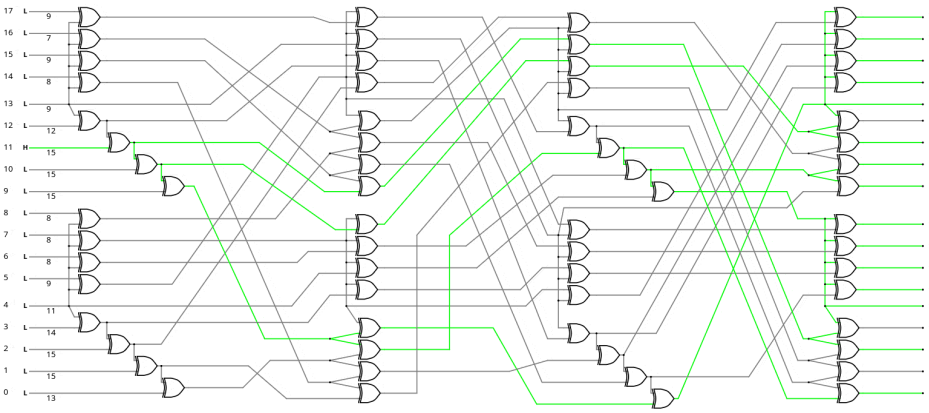
IMPORTANT: This custom circuit and protocol is not at all compliant or even compatible, even remotely linked to any 802.3 standard. It's all explained and detailed on Hackaday at https://hackaday.io/project/198914

The miniMAC is a (currently partial) Media Access Controller for a simplified data link over twisted pairs. It 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:

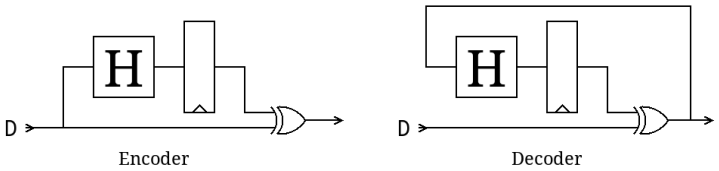
- The term "gPEAC" means "generalised Pisano with End-Around Carry" (see https://hackaday.io/project/178998), a class of PRNG/scrambler/checksum that uses different mathematics than Galois-based LFSR. The gPEAC18 unit is a non-power-of-two additive-based scrambler-checksum, with modulus=258114 and its state is impossible to flush and 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:



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).

An external circuit is required to implement the higher-level protocol, buffering and retransmission logic.

How it works

The gPEAC requires two cycles, two passes through the adder: first to compute the sums, then to adjust the modulus. OTOH the Hammer18 circuit requires one depth of XOR, but at different places:

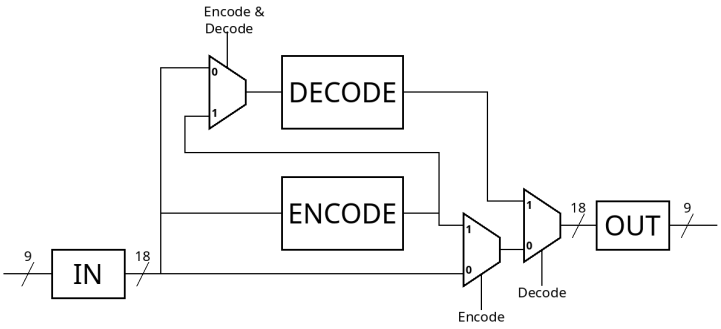
- For encoding, the input data goes through gPEAC then Hammer is inserted at the end of the last cycle.
- For decoding, the scrambled data goes through Hammer at the start of the first cycle of gPEAC descrambling.

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.

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 half-words
- 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

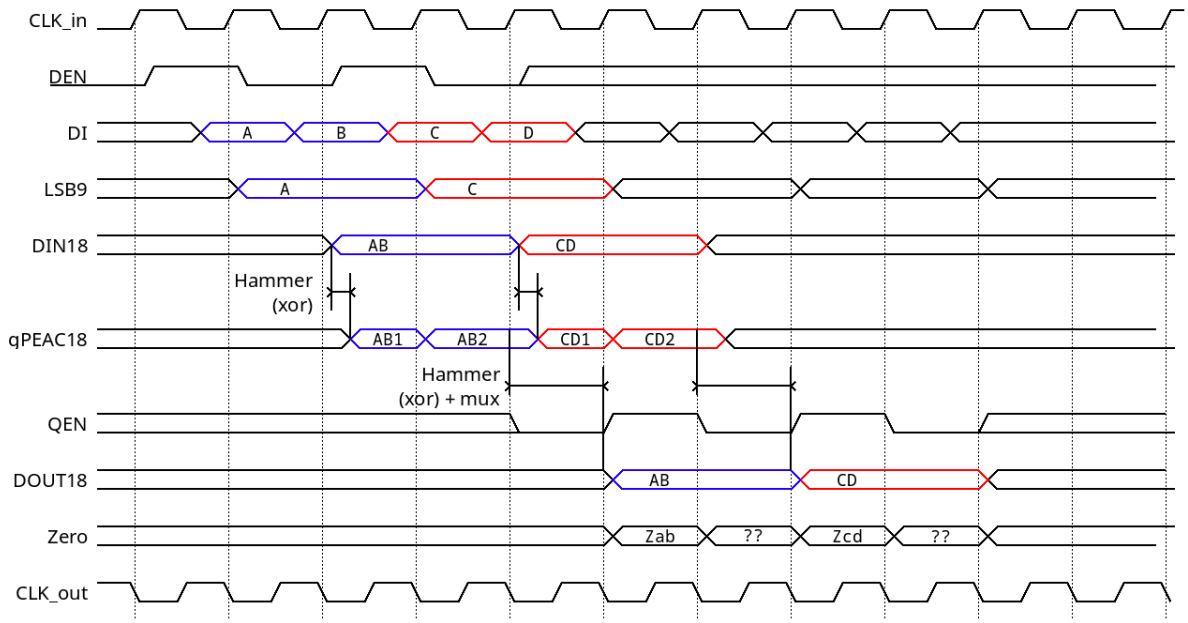
First let's examine the pinout. The inputs:

- 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 : bypass mode => the output is copied to the output after 3 cycles,
  - Enc=0, Dec=1 : decode mode => the cleartext input is scrambled and output after 5 cycles,

    - Enc=1, Dec=0 : encode mode => the scrambled input is restored to cleartext after 5 cycles,
    - Enc=1, Dec=1 : loopback mode => encodes then decodes, the delayed output (8 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 :

- 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 fastest 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.

(to be continued)

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

8_cool_modes

by **Carl H Lundstroem**

0709 | 25 MHz | HDL Project

github.com/carlhyldborglundstroem-code/TinyTapeoutCarlHL

"8 different modes + century clock divider"

How it works

This project uses the on-board 8-switch on the development board to choose between 8 different modes.

How to test

The MSB switch (most to the left) should be prioritized over the ones more to the right.

External hardware

Only on 4-button thing is needed for mode 6: Simon Says

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	BUTTONA	—	—
1	BUTTONB	—	—
2	BUTTONC	—	—
3	BUTTOND	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

E-Beam Inspection Pixel Core

by **Ermal**

0710 HDL Project

`github.com/ermalfierza1/E-Beam-Inspection-Pixel-Core`

> *"8-bit pixel-in per cycle with pix_valid, running mean (IIR) and local contrast |pixel - mean|, edge magnitude |pixel - prev_pixel|, config via tiny SPI (uio_in[2:4]=CS,SCLK,MOSI; uio_out[5]=MISO), outputs flags + 4-bit magnitude nibble on uo_out."*

E-Beam Inspection Pixel Core

An 8-bit pixel processing core designed for electron beam inspection systems. This module processes pixel data in real-time, performing defect detection through multiple algorithms including running mean calculation, local contrast analysis, and edge detection.

How it works

The core accepts 8-bit pixel values on the `ui_in` inputs, synchronized with a `pix_valid` signal on `uio_in[0]`. It maintains a running mean using an Infinite Impulse Response (IIR) filter to track local brightness levels.

Key features:

- **Running Mean (IIR)**: Continuously updates the average pixel value using a configurable alpha factor
- **Local Contrast**: Computes |pixel - mean| to detect brightness variations
- **Edge Detection**: Calculates |pixel - previous_pixel| to identify edges
- **Configurable Thresholds**: SPI interface allows runtime adjustment of detection parameters
- **Defect Flags**: Outputs binary flags for edge, dark defect, bright defect, and any defect conditions
- **Magnitude Output**: Provides a 4-bit magnitude nibble representing the strength of detected features

SPI Configuration Interface

- **CSn**: Chip select (active low) on `uio_in[2]`
- **SCLK**: Serial clock on `uio_in[3]`
- **MOSI**: Master out slave in on `uio_in[4]`
- **MISO**: Master in slave out on `uio_out[5]`

Configurable registers:

- Threshold values for contrast and edge detection

- IIR filter alpha shift parameter
- Mode selection register

Output Format

The `uo_out` provides:

- `uo_out[7]`: defect_any - Any defect condition detected
- `uo_out[6]`: bright_defect - Pixel significantly brighter than local mean
- `uo_out[5]`: dark_defect - Pixel significantly darker than local mean
- `uo_out[4]`: edge_strong - Strong edge detected
- `uo_out[3:0]`: mag_nib - 4-bit magnitude of the strongest detected feature

How to test

1. Apply pixel data to `ui_in[7:0]`
2. Assert `pix_valid` on `uio_in[0]` when pixel data is valid
3. Use `frame_start` on `uio_in[1]` to reset frame-based processing if needed
4. Configure detection thresholds via SPI interface
5. Monitor defect flags and magnitude on `uo_out[7:0]`

SPI Register Access

- Write operations: Send 8-bit address followed by 8-bit data
- Read operations: Send 8-bit address, then read 8-bit data on MISO

External hardware

This is a digital processing core designed for integration into larger inspection systems. No external hardware required for basic operation.

IO

#	Input	Output	Bidirectional
0	pixel[0]	mag[0]	pix_valid
1	pixel[1]	mag[1]	frame_start
2	pixel[2]	mag[2]	spi_cs_n
3	pixel[3]	mag[3]	spi_sclk
4	pixel[4]	edge_strong	spi_mosi
5	pixel[5]	dark_defect	spi_miso
6	pixel[6]	bright_defect	—
7	pixel[7]	defect_any	—

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	pixel[0]	mag[0]	pix_valid
1	pixel[1]	mag[1]	frame_start
2	pixel[2]	mag[2]	spi_cs_n
3	pixel[3]	mag[3]	spi_sclk
4	pixel[4]	edge_strong	spi_mosi
5	pixel[5]	dark_defect	spi_miso
6	pixel[6]	bright_defect	—
7	pixel[7]	defect_any	—

TestWorkShop

by **G. B. Hass**

0711 Wokwi Project

github.com/GitteBailey/TinyTapeoutWorkshop

wokwi.com/projects/456572140961867777

"Foreloebig ikke noget"

How it works

Explain how your project works Until now it is just a test of the method and the system

How to test

Explain how to use your project Until now it is just a test of the method and the system

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any Nothing yet

Project Pinout

Digital Pins

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

Kalman Filter for IMU

by **Clément Trassoudaine**

`0712` `10 MHz` `HDL Project`

`github.com/intv0id/tt-ihp-TinyKalman`

> "*FPGA-based 1D Kalman Filter for MPU-6500 sensor fusion.*"

How it works

This project implements a simplified 1D Kalman Filter (Complementary Filter logic) to fuse Accelerometer and Gyroscope data from an MPU-6500 sensor. It calculates Roll and Pitch angles using a CORDIC algorithm for the accelerometer and integrates gyroscope data with a steady-state Kalman gain.

The system consists of:

- **SPI Master**: Configures and reads data from the MPU-6500 sensor (100Hz sample rate).
- **CORDIC Core**: Calculates `atan2` for Roll and Pitch estimation from accelerometer data.
- **Kalman Filter**: Fuses the accelerometer angle with gyroscope rate.
- **UART Transmitter**: Outputs the calculated angles (Roll, Pitch) as a binary stream at 9600 baud.

Data Format (8 bytes per packet):

1. Header High: `0xDE`
2. Header Low: `0xAD`
3. Roll High
4. Roll Low
5. Pitch High
6. Pitch Low

How to test

To test the design on real hardware, you can use the FPGA ASIC simulator breakout with an MPU-6500 sensor module and an FT232 serial probe.

The default configuration assumes a 10MHz system clock.

Wiring Instructions:

- **FPGA Breakout `ui[0]` -> MPU6500 ADO**
- **FPGA Breakout `uo[0]` -> MPU6500 SDA**
- **FPGA Breakout `uo[1]` -> MPU6500 SCL**
- **FPGA Breakout `uo[2]` -> MPU6500 NCS**

- **FPGA Breakout `uo[3]`** -> **FT232 RX**
- **FPGA Breakout GND** -> **MPU6500 GND** & **FT232 GND**
- **FPGA Breakout VCC** -> **MPU6500 VCC** & **FT232 VCC** (make sure voltage levels are compatible, usually 3.3V)

Running on the FPGA Breakout (macOS):

1. Download the `fpga_bitstream` artifact from the latest passing GitHub Action run.
2. Extract the archive to find the `.bin` file (e.g., `tt_um_intv0id_kalman.bin`).
3. Connect the TinyTapeout demoboard to your Mac.
4. Clone the `tt-support-tools` repository:

   ```
   git clone https://github.com/TinyTapeout/tt-support-tools.git
   tt
   pip install -r tt/requirements.txt
   ```

5. Use the `tt_fpga.py` script to upload the bitstream. Specifying the serial port for Mac (typically `/dev/cu.usbmodem*`):

   ```
   ./tt/tt_fpga.py configure --port /dev/
   cu.usbmodem<your_port_number> --upload --name path/to/
   extracted/tt_um_intv0id_kalman.bin --set-default --clockrate
   10000000
   ```

Running on the Demoboard ETR (using mpremote):

1. Connect the TinyTapeout Demoboard ETR to your computer.
2. Install `mpremote`:

   ```
   pip install mpremote
   ```

3. Open the REPL using `mpremote` and enable the project:

   ```
   mpremote repl
   ```

   ```
   # Enable the project
   tt.shuttle.tt_um_intv0id_kalman.enable()
   ```

Live Plotting: To plot the data live using Python from the FT232, you can use the provided `plot_serial.py` script.

1. Install the required dependencies: `pip install pyserial matplotlib`
2. Run the script: `python3 plot_serial.py --port /dev/ttyUSB0`

For simulation, run `make` in the `test/` directory.

External hardware

- MPU-6500 (or MPU-6050/9250) IMU sensor.
- UART Receiver (USB-Serial adapter).

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	SPI_MOSI	SPI_MISO
1	—	SPI_SCLK	—
2	—	SPI_CS_N	—
3	—	UART_TX	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Design_test_workshop

by **Mark Toth**

0713 Wokwi Project

github.com/donmonki/tt01_chip_design

wokwi.com/projects/456575744901356545

"Testing some logic gates"

How it works

Trust me it is working lol

How to test

Good luck with that lmao

External hardware

Use your imagination

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	I1	O1	B1
1	I2	O2	B2
2	I3	O3	B3
3	I4	O4	B4
4	I5	O5	B5
5	I6	O6	B6
6	I7	O7	B7
7	I8	O8	B8

SEQ_MAC_INF_16H3 - Neural Network Inference Accelerator

by **Neuromurf**

0714 | 50 MHz | HDL Project

github.com/Neuromurf/ttihp26a-inferenceFCNN-16h3

"8-bit sequential MAC with autonomous MNIST inference"

How it works

This project is a sequential AI accelerator designed for on-chip full connected neural network inference of MNIST dataset. The design architecutre consists of three main mdoules: top-level wrapper, 11-state control FSM and arithmetic datapath module.

How to test

Interface with MCU (e.g., TT on-board RP2040/RP2350 MCU), assert 'rst_n' LOW, start MAC, send data (either weights or test values) through ui_in pins (controll data type throuhg 'data_type' and 'data_toggle' pins) and 10 class classification result at output pin in 4-bit format.

External hardware

RP2040/RP2350 or any other MCU for feed pre-trained weights and inputs to the ASIC. For example, using TinyTapeout demo board for convenient interfacing.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	data_in[0]	result[0] / busy	data_toggle
1	data_in[1]	result[1] / done	data_type
2	data_in[2]	result[2] / ready	start
3	data_in[3]	result[3] / byte_valid	mode[0]
4	data_in[4]	best_class[0] / inf_done	next_byte
5	data_in[5]	best_class[1] / layer	soft_rst
6	data_in[6]	best_class[2] / err_flag	mode[1]
7	data_in[7]	best_class[3] / reserved	status_sel

Tiny Tapeout Test Gates

by **Aoxuan Wang**

0715 | 10 kHz | Wokwi Project

github.com/AxuanW/wokwi-template

wokwi.com/projects/456571628648495105

"Test some logic gates anda clock divider."

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

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

Project Pinout

Digital Pins

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

SIMON

by **Libor Miller**

0716 | 50 MHz | HDL Project

github.com/libormiller/ttihp-simon

"Cryptographic accelerator with SPI interface"

How it works

Hardware implementation of the Simon block cipher (32/64 configuration) integrated with an SPI Slave interface. The design allows a Master device to write a 64-bit key and 32-bit data block, configure the operation mode (Encrypt/Decrypt), and read back the result.

SPI Interface

SPI Mode 3 (CPOL=1, CPHA=1), MSB first. SPI SCK frequency must be at most CLK/8 for reliable operation.

Pin Mapping

Pin	Signal	Direction	Description
uio[0]	CS_n	input	Chip select, active low
uio[1]	MOSI	input	Master out, slave in
uio[2]	MISO	output	Master in, slave out
uio[3]	SCK	input	SPI clock from master

SPI Command Protocol

First byte of each CS frame selects the command:

Command	Code	Data	Description
Write Key	0x01	1. 8 data bytes, LSB first	Load 64-bit encryption key
Write Block	0x02	1. 4 data bytes, LSB first	Load 32-bit data block
Encrypt	0x03	none	Start encryption
Decrypt	0x04	none	Start decryption
Read Status	0x05	1. 1 dummy byte	Returns {7'b0, done} on MISO
Read Result	0x06	1. 4 dummy bytes	Returns 32-bit result on MISO, LSB first

How to test

Testing can be done with cocoTB script (install dependencies and make). I've done testing after implementing verilog code to FPGA with SPI master ESP devkit V1 with testing program (/test/hw_test_ESP32_DEVKIT_V1)

External hardware

External SPI master needed

Project Pinout

Digital Pins

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

vis_3

by **Silas**

0717 Wokwi Project

github.com/silasvistrup-hub/Tinytapeoutsilas

wokwi.com/projects/456578790697395201

“den_viser_3”

How it works

if all the switches 1, 2 and 3 are the seven egment will show the number 3

How to test

turn on 1,2 and 3

External hardware

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

Project Pinout

Digital Pins

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

LLR simple VGA GPU

by **Léon Romanens**

`0718` `25.175 MHz` `HDL Project`

`github.com/leonllrmc/tinytapeout_26a_llrsub1`

> *"some simple 'GPU' that consist of a very simple configurable"*

How it works

This is a simple ALU pipeline that takes the pixel x/y as input, and output a color, parametrable by a few registers

How to test

Read the verilog for now

External hardware

The VGA thing

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	D0	R1	—
1	D1	G1	—
2	D2	B1	—
3	D3	VSync	—
4	A0	R0	—
5	A1	G0	—
6	A2	B0	—
7	WR	HSync	—

ISC77x16

by **Hans Kristensen**

0719 | 50 MHz | HDL Project

github.com/HansAdam2077/ISC77x8

"7 segment side scrolling display with 16 character programmeable message"

How it works

The chip implements a side scrolling 7 segment display, featuring an internal programmeable memory that can store a sequence of 16 alphanumeric characters. The 7 segment display constantly cycles through the stored 16 characters, and the stored characters are updated using connected switches.

How to test

Start by setting the register-select switches to program the desired memory spot, then set the character encoding switches to represent the desired character, last, toggle the regpush switch front and back to load the character into the memory. This is done to an arbitrary amount of the memory spots to create a message:)

The binary coding for the alphanumeric characters: +————+————+————+————+————+————+————+————+ | character | binary | character | binary | character | binary | character | binary | +————+————+————+————+————+————+————+————+ | (blank) | 0 | 8 | 8 | G | 16 | O | 24 | | 1 | 1 | 9 | 9 | H | 17 | P | 25 | | 2 | 2 | A | 10 | I | 18 | R | 26 | | 3 | 3 | B | 11 | J | 19 | T | 27 | | 4 | 4 | C | 12 | K | 20 | U | 28 | | 5 | 5 | D | 13 | L | 21 | V | 29 | | 6 | 6 | E | 14 | M | 22 | Y | 30 | | 7 | 7 | F | 15 | N | 23 | ? | 31 | +————+————+————+————+————+————+————+————+

External hardware

You will need an external board with an extra 7 segment display to get full functionality.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	bit0 of the alphanumeric	Segment of the first 7 segment display	bit3 of the register selector

#	Input	Output	Bidirectional
	character to be programmed		
1	bit1 of the alphanumeric character to be programmed	Segment of the first 7 segment display	The write switch. Loads the alphanumeric character to the selected register when high
2	bit2 of the alphanumeric character to be programmed	Segment of the first 7 segment display	Segment of the second 7 segment display
3	bit3 of the alphanumeric character to be programmed	Segment of the first 7 segment display	Segment of the second 7 segment display
4	bit4 of the alphanumeric character to be programmed	Segment of the first 7 segment display	Segment of the second 7 segment display
5	bit0 of the register selector	Segment of the first 7 segment display	Segment of the second 7 segment display
6	bit1 of the register selector	Segment of the first 7 segment display	Segment of the second 7 segment display
7	bit2 of the register selector	Segment of the second 7 segment display	Segment of the second 7 segment display

Plasma

by **Thomas Herzog**

`0720` `25.175 MHz` `HDL Project`

`github.com/thomasherzog/IHP26a-Plasma`

“Plasma Animation”

How it works

This design renders a plasma animation via VGA and includes a dedicated text overlay for the about screen.

Inputs:

Pin	Pin Name	Setting	Effect
`ui_in[1:0]`	`MODE`	Color mode	Changes the color theme of the animation
`ui_in[7]`	`ABOUT`	About overlay	Enables the about text overlay

Color themes:

Mode	Binary
2’b00	DEFAULT
2’b01	MATRIX
2’b10	CYBERPUNK
2’b11	ABYSS

How to test

We don’t test, sorry KGF and Oscar...

External hardware

External hardware required:

- TinyVGA PMOD

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	MODE[0]	R1	—

#	Input	Output	Bidirectional
1	MODE[1]	G1	—
2	—	B1	—
3	—	VSync	—
4	—	R0	—
5	—	G0	—
6	—	B0	—
7	ABOUT	HSync	—

Tiny Tapeout Test Gates

by **Yurii Pavlenko**

0721 | 100 kHz | Wokwi Project

github.com/YuriiPavlenko-DTU/FIRSTTEST

wokwi.com/projects/456571702213480449

"Test some logic gates and a clock divider"

How it works

I just doing my first GitHub project so it is just a test.

How to test

Just a test with NAND gates.

External hardware

I use NAND gates, wires, display.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input a	selection a	—
1	input b	—	—
2	input c	—	—
3	input d	—	—
4	input e	—	—
5	input f	—	—
6	input g	—	—
7	input h	—	—

moss_display

by **Abhishek Kumar**

0722 | 25.175 MHz | HDL Project

`github.com/KeshihbaNo/moss_display`

"A moss jump gif and wiwiwi sound player"

Specimen Profile: Fluffulus Hoppi (aka "Moss")

"Moss is a newly identified species of potentially sentient Fluffulus Hoppi enigmatica bryophyta. She is, to date, the only recorded specimen of her kind, and researchers have yet to determine her origins, ecological role, or underlying biology. What is known is that she is undeniably adorable: predominantly green, equipped with several antennae whose function remains a mystery, and capable of communicating exclusively through a series of soft "wiwiiwiiwiii!" vocalizations. She has been observed to possess four small feet; however, their practical function remains unclear, as she has thus far only been seen using them to jump in place.

While the broader characteristics of this organism remain unclear, preliminary observations suggest a strong and possibly symbiotic association with another little-understood, recently documented species known as Chipsaur. Early interactions between the two indicate behavioral coordination, though the nature and purpose of this relationship are still under investigation.

Further research is urgently required. We are currently seeking additional funding to better understand both of these remarkable bio-anomalies."

· **Dr. Sophie Zweifel**

Read the full article: What is Moss?

Acknowledgements

Special thanks to **swisschips** for making this project possible. SwissChips

Honorary Moss

MOSS Media Player

The chip brings Moss to life by continuously looping her animation and vocalizations directly from hardware memory.

How it works

A previously extracted MOSS GIF and sound file have been converted to LUTs and are being played on repeat.

The sprite ROM stores 4 animation frames at 64×64 pixels, scaled 4× and centered on a 640×480 VGA display. The background should imitate some 'electricity' flowing. The audio ROM holds 1-bit samples played back via 1-bit PCM on `uio[7]`.

Source GIF

VGA output

How to test

Connect the TinyVGA PMOD to the Out PMOD and an Audio PMOD to the Bidir PMOD. Pull `ui_in[0]` high to mute audio.

External hardware

- TinyVGA PMOD
- Audio PMOD

Project Pinout

Digital Pins

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

Test

by **Heini Kallsoy Mouritsen**

0723 10 kHz Wokwi Project

github.com/Heinikm/TinyTapeout

wokwi.com/projects/456575028603245569

"Yet to be decided"

How it works

Something something

How to test

Something something

External hardware

Something something

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input a	output AND	—
1	input b	Something something	—
2	input c	Something something	—
3	input d	Something something	—
4	—	Something something	—
5	—	Something something	—
6	input e	Something something	—
7	input f	Something something	—

Example

by **Paul Steinbach**

0736 Wokwi Project

github.com/NasenGolem7442/TinyTapeoutTest

wokwi.com/projects/455299783033986049

“nothing”

How it works

NO: Explain how your project works.

How to test

NO: Explain how to use your project.

External hardware

NO: List external hardware used in your project (e.g. PMOD, LED display, etc), if any.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	i1	E0	—
1	i2	E1	—
2	i3	E2	—
3	i4	E3	—
4	i5	E4	—
5	i6	E5	—
6	i7	E6	—
7	i8	E7	—

Tsetlin Machine for low-power AI

by **Pablo MV**

`0737` `50 MHz` `HDL Project`

`github.com/pablo-dk/TT_setlin_machine`

"A compact binary classifier using a 4-feature, 4-clause Tsetlin Machine with on-chip training"

How it works

This project implements a hardware-efficient Tsetlin Machine (TM) binary classifier optimized for ultra-low-power Edge AI. Unlike traditional neural networks that rely on power-hungry arithmetic multipliers and floating-point weights, this architecture uses pure digital logic and finite state machines to learn and classify data.

The machine processes a 4-bit binary input feature space and outputs a single binary classification. The internal architecture consists of:

- **32 Tsetlin Automata:** Each automaton acts as the "memory" of the system, implemented as a 4-bit saturating up/down counter (states 0 to 15).
- **4 Clauses:** The logic is divided into 2 positive polarity clauses (which vote for Class 1) and 2 negative polarity clauses (which vote against Class 1).
- **Inclusion Threshold:** If an automaton's state reaches 8 or higher, its corresponding input literal is "included" in its clause.

The chip operates in two modes, controlled by the Mode pin (`ui_in[5]`):

1. **Inference Mode (Mode = 0):** The included literals are logically ANDed together within each clause. The final output is decided by a simple majority vote between the positive and negative clauses.
2. **Training Mode (Mode = 1):** The chip updates its internal automata states based on the provided Target signal (`ui_in[4]`). It uses a deterministic implementation of Type I feedback (to reinforce correct patterns) and Type II feedback (to break incorrect patterns). The counters update synchronously on every positive clock edge.

How to test

To operate the Tsetlin Machine, you will need to manually toggle the inputs or drive them with a microcontroller.

1. Initialization

- Set `rst_n` to `0` to initialize all 32 internal automata to their default state (8).

- Set `rst_n` to `1` to begin normal operation.

2. Training the Machine

- Set the Mode pin `ui_in[5]` to `1`.
- Provide a 4-bit input pattern on `ui_in[3:0]`.
- Provide the desired classification (the "correct answer") on the Target pin `ui_in[4]`.
- Pulse the `clk` pin. The internal counters will increment or decrement based on the learning rules. Repeat this process for your training dataset.

3. Running Inference

- Set the Mode pin `ui_in[5]` to `0`.
- Provide a 4-bit input pattern on `ui_in[3:0]`.
- Read the predicted classification on the output pin `uo_out[0]`.

External hardware

For basic testing, no specialized external hardware is required.

However, because this chip is designed specifically for ultra-low-power Edge AI, it is ideally suited to be paired with external PMODs or custom sensor boards for real-world applications.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	feature_0	predicted_class	—
1	feature_1	—	—
2	feature_2	—	—
3	feature_3	—	—
4	target_class	—	—
5	Train_mode (0=inference; 1=train)	—	—
6	—	—	—
7	—	—	—

Try1

by **Mana Overflow**

0738 2 Hz Wokwi Project

github.com/ManaOverflow/TinyTapeoutWorkshop

wokwi.com/projects/455299761916711937

"Just try some stuff"

How it works

It creates some blinking lights based on the flipped input pins.

How to test

Flip some pins.

External hardware

NONE.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	IN0	OUT0	—
1	IN2	OUT1	—
2	IN1	OUT2	—
3	—	OUT3	—
4	—	OUT4	—
5	—	OUT5	—
6	IN6	OUT6	—
7	IN7	OUT7	—

Cremedelcreme

by **Célien Abbet**

0739 10 kHz Wokwi Project

github.com/celien-14/cremedelacreme

wokwi.com/projects/456574528376856577

"j'aime très beaucoup le tennis"

How it works

J'aime le butter chicken

How to test

With a fork and some cutelry

External hardware

Cashews and nuts

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input a	output inverter	—
1	input b	output inverter	—
2	input c	output inverter	—
3	input d	output inverter	—
4	input e	—	—
5	input f	—	—
6	input g	—	—
7	input h	—	—

Clock Divider Test Project

by **werrever**

0740 | 1 kHz | Wokwi Project

github.com/werrever/TinyTapeout_Project26

wokwi.com/projects/455291701422995457

"clock divider with selectable speed"

How it works

Clk input is divided into 1/2, 1/4..., 1/256; which clock divider drives the output is selected with an 8:1 MUX with the pins S0, S1, S2

How to test

Connect a clock and test out the output by varying the MUX select switches

External hardware

Needs external clock, 3 select switches and power

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	MUX_S0	MUX_out	—
1	—	—	—
2	MUX_S1	—	—
3	MUX_S2	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

tiny tapeout half adder

by **Abdulah Korishe**

0741 Wokwi Project

github.com/Abdulah-Korishe/Abdulah-s-1st-tiny-tapeout-design

wokwi.com/projects/456571730084640769

"half adder"

How it works

it needs to be either 1 or 0 of the two input ligic to get the sum 1 Explain how your project works

How to test

A= 1/0 and B= 1/0 S=1/0 and C= 1/0 Explain how to use your project

External hardware

no List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	A	SUM	—
1	B	CARRY	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Test

by **Darius Schefer**

0742 Wokwi Project

github.com/darius1702/tt-asic

wokwi.com/projects/455291654560013313

"does the action even work"

How it works

It doesn't :3

How to test

You can't

External hardware

None

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input a	output a	—
1	input b	output b	—
2	input c	output c	—
3	input d	output d	—
4	input e	output e	—
5	input f	output f	—
6	input g	output g	—
7	input h	output h	—

SPI RAM Driver

by **Roméo Estezet**

0743 | 50 Hz | HDL Project

`github.com/Romultra/2-bit-adder-TTIHP26a`

"SPI Driver which controls the emulated SPI RAM on the RP2040 Chip"

How it works

This design implements an SPI Mode 0 (CPOL=0, CPHA=0) master driver for interfacing with external SPI RAM. It supports READ (0x03) and WRITE (0x02) commands with a 16-bit address space and 8-bit data transfers.

The SPI transaction frame is 32 bits: 8-bit command + 16-bit address + 8-bit data. Data is shifted MSB first. The SCK clock runs at half the system clock frequency.

SPI protocol timing

- **CS** is pulled low to start a transaction and raised high when complete.
- **MOSI** data changes on SCK falling edges (setup), and the slave samples on SCK rising edges.
- **MISO** data from the slave is sampled on SCK rising edges during the data phase of a READ.
- When `i_ready_to_execute` is held high, a new transaction starts automatically after the previous one completes.

Current top-level wiring

The SPI_RAM module inputs are currently hardcoded for testing purposes (WRITE command to address 0x0000 with data 0x5B). The data output from READ operations is wired to the dedicated output pins (`uo_out`). These will be connected to actual control logic in a future revision.

How to test

After reset (active low on `rst_n`), the design immediately begins an SPI WRITE transaction with the hardcoded parameters. You can observe the SPI signals on the bidirectional pins:

- **uio[0]** (CS): Goes low during a transaction, high when idle.
- **uio[1]** (MOSI): Carries the command (0x02), address (0x0000), and data (0x5B) serially, MSB first.
- **uio[3]** (SCK): Toggles at half the system clock rate while CS is low.

To test READ functionality, drive MISO data on **uio[2]** synchronized to SCK rising edges during the data phase (last 8 SCK cycles of a READ transaction). The received byte will appear on `uo_out[7:0]`.

Running the cocotb testbench

```
cd test
make -B
```

The testbench includes 8 tests covering reset state, IO directions, WRITE/READ frame verification, and multi-pattern data capture. A standalone SPI_RAM instance in the testbench allows direct testing with controllable inputs. Gate-level tests automatically skip tests that require the standalone instance.

External hardware

SPI RAM module (e.g. 23LC512 or similar) connected to the bidirectional pins, or an RP2040 emulating an SPI RAM slave.

Pin	Signal	Direction
uio[0]	SPI CS	Output
uio[1]	SPI MOSI	Output
uio[2]	SPI MISO	Input
uio[3]	SPI SCK	Output

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	SPI RAM data out [0]	SPI CS (output)
1	—	SPI RAM data out [1]	SPI MOSI (output)
2	—	SPI RAM data out [2]	SPI MISO (input)
3	—	SPI RAM data out [3]	SPI SCK (output)
4	—	SPI RAM data out [4]	—
5	—	SPI RAM data out [5]	—
6	—	SPI RAM data out [6]	—
7	—	SPI RAM data out [7]	—

Hopfield Associative Memory — Odd Digit Recall on 7-Segment

by **Shanmukha Mangadahalli Siddaramu**

0744 | 50 MHz | HDL Project

`github.com/Shanmukha-ms/cim_tiny_tapeout`

> *"A 7-node Hopfield associative memory storing two 7-segment display patterns: digit 1 and digit 9. Given a noisy or partial segment pattern on the inputs, the network iterates each clock cycle until it recalls the nearest stored digit. Weights computed analytically via Hebbian learning (W = p1⊗p1 + p9⊗p9). Total gate count ~140 GE. Decimal point lights up when the network converges."*

How it works

This design implements a **7-node Hopfield Associative Memory** that stores two 7-segment display patterns: **digit 1** and **digit 9**.

A Hopfield network is a recurrent neural network where the weights encode stored memories directly in silicon — no ROM, no lookup table. Weights are computed analytically using the Hebbian learning rule: `W = p1⊗p1 + p9⊗p9`.

Each of the 7 nodes corresponds to one segment (a–g). On every clock cycle the network applies a synchronous update derived from the weight matrix:

```
new_a = majority(d, f, g)     new_e = NAND(b, c)
new_b = c AND NOT e           new_f = majority(a, d, g)
new_c = b AND NOT e           new_g = majority(a, d, f)
new_d = majority(a, f, g)
```

The decimal point (`uo_out[7]`) lights up when the network has converged. Total gate count: 140 GE.

Stored patterns:

Digit	Segments	Hex
1	b, c	`0x06`
9	a,b,c,d,f,g	`0x6F`

Safe to corrupt (majority-vote protected): segments a, d, f, g

Fragile (mutually dependent): segments b, c, e

How to test

1. Set `ui_in[6:0]` DIP switches to a segment pattern

2. Pulse `uio_in[0]` high for one clock cycle (start)
3. Watch display update — decimal point lights when settled

Input	Expected output
`0x06`	Digit 1
`0x6F`	Digit 9
`0x07` (digit 1 + seg-a)	Digit 1 corrected
`0x6E` (digit 9 − seg-a)	Digit 9 corrected
`0x67` (digit 9 − seg-d)	Digit 9 corrected

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	seg-a: top horizontal segment	seg-a: recalled top horizontal	start: pulse high 1 cycle to load input and begin recall
1	seg-b: top-right vertical segment	seg-b: recalled top-right	—
2	seg-c: bottom-right vertical segment	seg-c: recalled bottom-right	—
3	seg-d: bottom horizontal segment	seg-d: recalled bottom horizontal	—
4	seg-e: bottom-left vertical segment	seg-e: recalled bottom-left	—
5	seg-f: top-left vertical segment	seg-f: recalled top-left	—
6	seg-g: middle horizontal segment	seg-g: recalled middle	—
7	—	dp: decimal point — high when network has converged	—

4-bit full adder

by **David Rodríguez Ferrero**

0745 Wokwi Project

github.com/DragonDavid75/TinyTapeoutProjectWokwi

wokwi.com/projects/456575247946496001

"4-bit full adder"

How it works

The adder is a 4-bit ripple carry added, where 4 1-bit full adders are connected. The output result is 4-bits in binary and the carry of the calulation.

How to test

In the switches select both 4-bit numbers, where the first 4 switches are for the first number and the last 4 are for the second number.

External hardware

None.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	a0	s0	—
1	a1	s1	—
2	a2	s2	—
3	a3	s3	—
4	b0	c	—
5	b1	—	—
6	b2	—	—
7	b3	—	—

Workshop Day

by **arnonym**

0746 1 Hz Wokwi Project

github.com/Montagsfrei/TinyTapeoutWorkshop

wokwi.com/projects/455291651646015489

"just a very basic circuit"

How it works

Switches the output number form 0 to 1 on the segment display

How to test

Dont use it, it's just for pipeline testing

External hardware

Segment display

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	in0	out0	—
1	in1	out1	—
2	—	out2	—
3	—	out3	—
4	—	out4	—
5	—	out5	—
6	—	out6	—
7	—	out7	—

Alex first circuit

by **Alexander Lykke**

0747 1 Hz Wokwi Project

github.com/5gyyzwxbd4-svg/First

wokwi.com/projects/456576478636229633

"LED circle thing"

How it works

Explain how your project works

By using a series of DFF a moving cirkle will apear on the LED display

How to test

Explain how to use your project

No further action needed to turn on/off the mechanism.

External hardware

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

- seven segment display

Project Pinout

Digital Pins

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

83rk: Tiny Tapeout

by **Berk**

0748 10 kHz Wokwi Project

github.com/83rk/83rkFlipFlop

wokwi.com/projects/455291845594899457

"SR FlipFlop"

How it works

SR FlipFlop

How to test

Set and Reset the FF, while doing that check the values

External hardware

None

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input a	output and	—
1	input b	output nand	—
2	—	output or	—
3	—	output clock div 16	—
4	—	output clock div 12	—
5	—	—	—
6	—	—	—
7	—	—	—

Malthes First Template

by **Malthe Norlin Frederiksen**

0749 1 Hz Wokwi Project

github.com/Malthe2512/Tinytape

wokwi.com/projects/456571639638628353

“thing with LED”

How it works

this circuit uses the colk to light up the LED in diffent oprintesions

How to test

no action needed

External hardware

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

a seven sigment display

Project Pinout

Digital Pins

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

FirstTapeOut2

by **Viacheslav**

0750 Wokwi Project

github.com/pageeecs/FirstTapeout

wokwi.com/projects/455301826476070913

"first design on Wokwi"

How it works

inputs 0 and 1 are connected to the red and. The outout of the and is connected to the outputs 0,2,3,5,6.

inputs 2 and 3 are connected to an another and. Theo output of this and is connected to the output 4.

How to test

Set only first to inputs to 1, you should see the number five.

Then set onyl inputs #2, #3 to one, the lower left segment should set to light.

External hardware

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input red and	output red and	—
1	input red and	—	—
2	input and	output red and	—
3	input and	output red and	—
4	—	output and	—
5	—	output red and	—
6	—	output red and	—
7	—	—	—

And_Or

by **David_Weis**

0751 Wokwi Project

github.com/David-Weis/Wokwi-chip-design

wokwi.com/projects/456573048893551617

"AndOr"

How it works

All inputs AND'ed together with two OR gates.

How to test

Inputs: a,b,c,d,e,f,g,h (a*b)+(*cd)+(e*f)+(*gh)

External hardware

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Input a	Output y	—
1	Input b	Output y	—
2	Input c	—	—
3	Input d	—	—
4	Input e	—	—
5	Input f	—	—
6	Input g	Output z	—
7	Input h	Output z	—

Programmable 8-BIT CPU

by **Leonard Finthammer**

0752 9.6 kHz HDL Project

`github.com/dranoel06/tiny_tapeout_dranoel06`

"*Programmable 8 Bit CPU based on the SAP Architecture*"

How it works

This project is a programmable 8-bit CPU based on the classic SAP-1 (Simple As Possible) architecture.

The CPU features an 8-bit architecture with a 4-bit address space, meaning it has 16 addressable RAM locations (0x0 to 0xF), where each location holds an 8-bit value (4 bits for the opcode, 4 bits for the operand/immediate).

Instruction Set Architecture (ISA)

The CPU interprets the upper 4 bits `[7:4]` of a memory byte as the opcode and the lower 4 bits `[3:0]` as the memory address or immediate value.

Mnemonic	Opcode (Binary)	Hex	Description
LDA	`0001`	1	Load Register A with the value from RAM at the given address.
ADD	`0010`	2	Add the value from RAM at the given address to Register A. Updates Zero/Carry flags.
OUT	`0011`	3	Output Register A to the dedicated output pins (`uo_out`) and trigger UART TX.
JMP	`0100`	4	Unconditional jump to the given address.
STA	`0101`	5	Store the value of Register A into RAM at the given address.
LDI	`0110`	6	Load an immediate 4-bit value (the operand itself) into Register A.
SUB	`0111`	7	Subtract the value from RAM at the given address from Register A. Updates Zero/Carry flags.
JMZ	`1000`	8	Jump to the given address IF the the previous instruction by the ALU resulted in 0.

CMP	1001	9	Compare Register A with RAM value (A - RAM). Updates flags but does not store the result.
JMC	1010	A	Jump to the given address IF the result of the ALU operation overflows (values >255)

Built-in UART

The Design features a built-in UART transmitter. Whenever the `OUT` (0x3) instruction is executed, the CPU not only updates the output register but also serializes the 8-bit data and transmits it via the `UART TX` pin (`uio[6]`). For this to work you need to set the clock frequency to 9600Hz.

How to test

The CPU operates in two distinct modes: **Programming Mode** and **Execution Mode**.

1. Programming the RAM

Since the memory is volatile, you need to write your program into the RAM before running it:

1. Hold `rst_n` **LOW** (0) to keep the CPU in reset mode.
2. Set the `Prog` pin (`uio[7]`) **HIGH** (1) to enable programming mode.
3. For each byte of your program:
   - Set the 4-bit target memory address on `uio[3:0]`.
   - Set the 8-bit instruction/data on the input pins `ui[7:0]`.
   - Pulse the clock (or wait for the next clock edge) to write the data into RAM.
4. Repeat step 3 until your code and data are loaded into memory.
5. Initialize unused Registers with `00000000`.

2. Executing the Program

1. Set the `Prog` pin (`uio[7]`) **LOW** (0) to disable programming mode.
2. Set `rst_n` **HIGH** (1) to release the reset state.
3. Provide a clock signal. The CPU will start fetching and executing instructions starting from memory address `0x0`.

Note: Each instruction takes between 3 to 6 clock cycles to complete depending on its complexity (Fetch-Decode-Execute cycle).

External hardware

To fully interact with the CPU, the following external hardware is recommended:

- **DIP Switches (12x):** 8 switches connected to `ui[7:0]` for data input, and 4 switches connected to `uio[3:0]` for memory address selection during programming.
- **Toggle Switch (1x):** Connected to `uio[7]` to easily toggle between Programming and Execution mode.
- **LEDs (8x):** Connected to the output pins `uo[7:0]` to read the visual output of the `OUT` instruction.
- **USB-to-Serial Adapter (FTDI/CH340):** Connect the RX pin of the adapter to `uio[6]` (UART TX) to read the serial output of the CPU on your computer. Make sure to match the baud rate to your input clock frequency (e.g., 9600 baud). You'll need to convert the received ASCII Characters to Decimal Values.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Programm_Input[0]	Output[0]	Address[0]
1	Programm_Input[1]	Output[1]	Address[1]
2	Programm_Input[2]	Output[2]	Address[2]
3	Programm_Input[3]	Output[3]	Address[3]
4	Programm_Input[4]	Output[4]	—
5	Programm_Input[5]	Output[5]	—
6	Programm_Input[6]	Output[6]	UART_TX
7	Programm_Input[7]	Output[7]	Prog

WIP Bin to Dec

by **Luis Parker Noah Conradty**

0753 Wokwi Project

github.com/SciFiCarrot/tiny-tapeout-workshop

wokwi.com/projects/456571585305580545

"Work In Progress - Converts the input to a decimal number on the 7 segment display"

How it works

You input a number through the input pins. Multiple inputs at once are not supported yet. That number gets converted to a seven segment display input showing that number.

How to test

Input a number and get one out

External hardware

Not used

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	number 1	Segment A	—
1	number 2	segment B	—
2	number 3	segment C	—
3	number 4	segment D	—
4	number 5	segment E	—
5	number 6	segment F	—
6	number 7	segment G	—
7	number 8	segment H	—

test

by **Alexandr Melnic**

0754 10 kHz Wokwi Project

github.com/trupus/tt

wokwi.com/projects/455300379278483457

“*test*”

How it works

not yet working

How to test

please don’t

External hardware

LED display

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input a	output led 1	—
1	input b	output led 2	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

TinyTapeNkTest

by **Nikolai Skarum**

0755 | 10 kHz | Wokwi Project

github.com/Niko78a1/tinytapeNK

wokwi.com/projects/456571687437989889

"circuit logic"

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

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

Project Pinout

Digital Pins

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

Test

by **me**

0768 1 Hz Wokwi Project

github.com/PleinR02/test

wokwi.com/projects/455291642603080705

“no”

How it works

I dont now maan!

How to test

mayby it is broken

External hardware

TNT

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	1	1	—
1	2	2	—
2	3	3	—
3	4	4	—
4	5	5	—
5	6	6	—
6	7	7	—
7	8	8	—

TinyTapeout test

by **Tom Ulrich**

0769 Wokwi Project

github.com/tulrich2/TinyTapeout

wokwi.com/projects/450491302960427009

"Only a test so far"

How it works

This block shows the name `Elias` by displaying the letters `E L I A S` one after another with one second in between.

How to test

Start the block with the 10 kHz clock enabled and the letters should show up on the 7-segment-display.

External hardware

None.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	—	—
1	First input	—	—
2	—	First output	—
3	Second input	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Test

by **Thomas**

0770 1 Hz Wokwi Project

github.com/LordTaek/GDS_Creator

wokwi.com/projects/455291792579934209

"Just a test"

How it works

Can actiavate a blinkingbar and dot. The dot can also be activated permanently.

How to test

Activate Pin 1 and 2.

External hardware

7seq display

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Enable Clocked Pins	upper bar	—
1	Enable dot permanent	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	dot	—

Snake

by **Daniel Wagner**

0771 1 Hz Wokwi Project

github.com/danielowa/tt-wokwi

wokwi.com/projects/450491696806684673

"A snake travelling around the 7 segement display"

How it works

Clk and Reset are connected to a row of flip flops which generate a time sequence. Each output of the flipflops indicate one explicit step. Each step then connects to the corresponding segments to create the moving snake.

How to test

Use the Step button to make the snake move one tile

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	A	sevseg:A	—
1	—	sevseg:B	—
2	—	sevseg:C	—
3	—	sevseg:D	—
4	—	sevseg:E	—
5	—	sevseg:F	—
6	—	sevseg:G	—
7	—	sevseg:DP	—

DDMTD

by **Arthur Finkelmann**

0772 Wokwi Project

`github.com/ArthFink/TinyTapeout`

`wokwi.com/projects/0`

“*test*”

How it works

Explain test

How to test

Explain test

External hardware

qqwerqweqwr

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	CLK_REF_IN (reference clock-like input)	PHASE_VALID (new beat-domain phase error available)	—
1	CLK_FB_IN (remote/ feedback clock-like input)	PHASE_ERR_SIGN (1=negative)	—
2	SEL_CLOSE (0=use ui[1], 1=use internal helper for bring-up)	PHASE_ERR[6] (beat-domain, selected bits)	—
3	KP_SEL[0] (P gain select)	PHASE_ERR[7]	—
4	KP_SEL[1]	PHASE_ERR[8]	—
5	KI_SEL[0] (I gain select)	PHASE_ERR[9]	—
6	KI_SEL[1]	PHASE_ERR[10]	—
7	—	PHASE_ERR[11]	—

test

by **test**

0773 Wokwi Project

github.com/hemanthkrishna5/tiny_tape_out

wokwi.com/projects/450492214548484097

"test"

How it works

Inputs **a** and **b** are fed into AND, NAND, and OR gates. The selected logic result is routed to the output through the control logic. The output state is shown on the connected display.

How to test

Set the inputs and verify the outputs match the table below:

input a	input b	AND	NAND	OR
0	0	0	1	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	1

External hardware

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input a	seg a	—
1	input b	seg b	—
2	mux sel	seg c	—
3	—	seg d	—
4	—	seg e	—
5	—	seg f	—
6	—	seg g	—
7	input d	seg dp	—

UART-Programmable 2-Tap FIR Filter

by **Mert Erman**

`0774` `25 MHz` `HDL Project`

`github.com/MertErmann/tiny_tapeout_1`

> *"2-tap direct-form FIR filter with runtime-configurable Q7 coefficients via UART 9600 8N1. 8-bit signed input/output with saturation."*

How it works

This design implements a **2-tap direct-form FIR filter** with runtime-configurable coefficients.

Coefficient Loading (UART)

Send a 5-byte packet at **9600 baud, 8N1** on `uio[0]`:

```
[ 0xA5 ] [ c0 ] [ c1 ] [ c2 ] [ c3 ]
```

(Note: To fit in a 1x1 footprint, `c2` and `c3` are ignored by the filter, but they must still be sent to complete the UART packet protocol.)

Coefficients are **8-bit signed Q7 fixed-point** (divide by 128 to get the real value):

Float value	Byte to send
+1.0	`0x7F`
+0.5	`0x40`
+0.25	`0x20`
0.0	`0x00`
−0.5	`0xC0`
−1.0	`0x80`

Default coefficients at reset: all `0x40` (+0.5) — box / moving-average filter.

Sample Processing

1. Place 8-bit signed sample on `ui_in[7:0]`
2. Pulse `uio[2]` (sample_valid) HIGH for ≥1 clock cycle
3. Wait 1 clock cycle
4. Read result from `uo_out[7:0]` when `uio[3]` (out_valid) pulses HIGH

Filter equation

```
y[n] = (c0*x[n] + c1*x[n-1]) >> 7
```

Output is saturated to [−128, +127].

Filter presets (Python)

```
import serial

ser = serial.Serial('COM3', 9600)

def q7(v):
    return int(v * 128) & 0xFF

def load(c0, c1, c2, c3):
    ser.write(bytes([0xA5, q7(c0), q7(c1), q7(c2), q7(c3)]))

load(0.5,   0.5,   0.0,   0.0)    # Low-pass (box)
load(0.5,  -0.5,   0.0,   0.0)    # High-pass
```

How to test

- **Minimum test:** DIP switches on `ui_in`, LED bar on `uo_out`, button on `uio[2]`
- **UART test:** Python script sends coefficient packets; verify `uio[4]` (coeff_we) pulses
- **Signal test:** Arduino generates sine wave on `ui_in`; observe filtering on oscilloscope
- **Audio demo:** Connect R-2R DAC ladder to `uo_out` and speaker — hear LP/HP effect live

External hardware

Pin	Connection
`ui_in[7:0]`	8-bit signed sample (DIP switches / ADC / Arduino)
`uo_out[7:0]`	8-bit signed filtered output (LEDs / DAC / oscilloscope)
`uio[0]`	UART RX from USB-UART adapter (CP2102 / CH340)
`uio[2]`	sample_valid strobe (button / MCU GPIO)
`uio[3]`	out_valid indicator (LED)
`uio[4]`	coeff_we indicator (LED)

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
`0`	Sample in bit 0 (LSB)	Filtered output bit 0 (LSB)	UART RX - coefficient programming 9600 8N1 (input)

#	Input	Output	Bidirectional
1	Sample in bit 1	Filtered output bit 1	UART TX - reserved, tie HIGH (output)
2	Sample in bit 2	Filtered output bit 2	sample_valid - pulse to process ui_in (input)
3	Sample in bit 3	Filtered output bit 3	out_valid - pulses when uo_out ready (output)
4	Sample in bit 4	Filtered output bit 4	coeff_we - pulses when coefficients updated (output)
5	Sample in bit 5	Filtered output bit 5	—
6	Sample in bit 6	Filtered output bit 6	—
7	Sample in bit 7 (MSB, sign)	Filtered output bit 7 (MSB, sign)	—

RandomNum

by **Aakarsh Yadav**

`0775` `10 kHz` `Wokwi Project`

`github.com/aakarsh1011/tiny-tapeout`

`wokwi.com/projects/450492230413445121`

"Currently shows a random number on the display"

How it works

This circuit takes eight digital inputs (IN0–IN7), processes some of them through logic gates, and drives eight outputs (OUT0–OUT7) that are connected to a seven-segment display.

Inputs **IN0, IN1, and IN2** are electrically tied together, forming a single signal called **A**.
This signal is fed into three NOT gates.

- When **A = 0**, all three NOT gates output **1**
- When **A = 1**, all three NOT gates output **0**

These inverted signals drive **OUT0, OUT1, and OUT2**.

Some inputs are not modified by any logic gates:

- **IN3 → OUT3**
- **IN4 → OUT4**
- **IN5 → OUT5**

These outputs always match their corresponding inputs.

Inputs **IN5** and **IN6** are connected to an XOR gate.

- The XOR output is **1** only when the two inputs are different
- The XOR output is **0** when the two inputs are the same

This result drives **OUT6**.

Inputs **IN6** and **IN7** are connected to an AND gate.

- The AND output is **1** only when both inputs are **1**
- Otherwise, the output is **0**

This result drives **OUT7**.

The outputs are wired to segments of a seven-segment display.
By changing the input values, the combination of active outputs determines which segments light up and ther

How to test

Output	Logic Description
OUT0	¬A
OUT1	¬A
OUT2	¬A
OUT3	IN3
OUT4	IN4
OUT5	IN5
OUT6	IN5 ⊕ IN6
OUT7	IN6 AND IN7

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	A_IN0 (shared input, tied with ui[1] and ui[2])	OUT0 (NOT A, seven-segment control)	—
1	A_IN1 (shared input, tied with ui[0] and ui[2])	OUT1 (NOT A, seven-segment control)	—
2	A_IN2 (shared input, tied with ui[0] and ui[1])	OUT2 (NOT A, seven-segment control)	—
3	IN3 (direct to output)	OUT3 (direct from IN3)	—
4	IN4 (direct to output)	OUT4 (direct from IN4)	—
5	IN5 (XOR and direct)	OUT5 (direct from IN5)	—
6	IN6 (XOR and AND)	OUT6 (IN5 XOR IN6)	—
7	IN7 (AND input)	OUT7 (IN6 AND IN7)	—

Freddys tapeout

by **Frederik Miller**

0776 Wokwi Project

github.com/Freddy-mllr/FreddysTapeout

wokwi.com/projects/455299760551464961

"Binary converter for numbers from 1 to 3"

How it works

Still have to find out...

How to test

Testing sucks

External hardware

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Set LSB of binary number to 1	—	—
1	Set second bit of binary number to 1	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

MBIST + MBISR Built-In Memory Test & Repair

by **Dr Aditya Kumar Singh Pundir** & **Dr Pallavi Singh**

0777 | 10 MHz | HDL Project

github.com/adityapundir1985/tt_um_aksp_mbist_mbisr

"Memory Built-In Self-Test (March C-) and Built-In Self-Repair with CAM-based faulty address remapping."

How it works

This project implements a Memory Built-In Self-Test (MBIST) controller with integrated Memory Built-In Self-Repair (MBISR) capabilities. The system automatically tests an embedded 256×8-bit SRAM memory and can repair faulty cells on-the-fly.

Key Components:

1. **MBIST Controller** (`mbist_marchc_controller.v`):
   - Implements March C- algorithm (simplified variant)
   - Tests memory with pattern: {↕(w0); ↑(r0,w1); ↓(r1,w0); ↑(r0)}
   - Reports per-address failures via `fail_valid` and `fail_addr` signals
2. **MBISR Controller** (`mbisr_controller.v`):
   - Content-Addressable Memory (CAM) style repair logic
   - Supports up to 16 repair entries
   - Automatically remaps faulty addresses to spare memory region (0xF0-0xFF)
   - Transparent to user - faulty addresses are automatically redirected
3. **Memory Module** (`memory.v`):
   - 256×8-bit synchronous SRAM
   - Configurable address and data widths
   - Supports read/write operations
4. **Integration Top** (`top.v`):
   - Connects MBIST, MBISR, and memory modules
   - Manages test execution and repair coordination

Operation Flow:

1. **Initialization**: Memory is cleared to zero on reset
2. **Test Start**: User asserts `START` signal
3. **March C- Execution**: MBIST runs through 4 march elements:
   - Write 0 ascending (↕w0)
   - Read 0, Write 1 ascending (↑r0,w1)
   - Read 1, Write 0 descending (↓r1,w0)

    - Read 0 ascending verification (↑r0)
4. **Fault Detection**: Any mismatched read triggers failure reporting
5. **Automatic Repair**: MBISR records faulty addresses and remaps to spares
6. **Completion**: `DONE` signal asserted, `FAIL` indicates if faults were found

How to test

Test Interface:

Inputs:

- `clk`: System clock (any frequency up to 1MHz)
- `rst_n`: Active-low reset (assert to initialize)
- `START` (ui[0]): Begin MBIST test (pulse high)

Outputs:

- `DONE` (uo[0]): Test completion indicator (high when finished)
- `FAIL` (uo[1]): Fault detection indicator (high if any faults found)

Test Procedure:

1. **Apply clock signal** (typically 1-10MHz)
2. **Release reset** (set `rst_n = 1`)
3. **Start test** (pulse `START` high for at least 1 clock cycle)
4. **Monitor outputs**:
    - `DONE = 0`, `FAIL = 0`: Test in progress
    - `DONE = 1`, `FAIL = 0`: Test passed (memory intact)
    - `DONE = 1`, `FAIL = 1`: Test completed with faults (repairs applied)
5. **Repeat test** as needed (system resets between runs)

Simulation Testing:

```bash cd test make # Run RTL simulation make GATES=yes # Run gate-level simulation

## Project Pinout

### Digital Pins

| # | Input | Output | Bidirectional |
|---|---|---|---|
| 0 | START | DONE | — |
| 1 | — | FAIL | — |
| 2 | — | — | — |
| 3 | — | — | — |
| 4 | — | — | — |
| 5 | — | — | — |
| 6 | — | — | — |
```

#	Input	Output	Bidirectional
7	—	—	—

My first tapeout

by **Julius Friesen**

0778 Wokwi Project

github.com/youju26/tinytapeout

wokwi.com/projects/455291688143820801

"*This is my first tapeout*"

How it works

(TODO)

How to test

(TODO)

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	a	a	—
1	b	b	—
2	c	c	—
3	d	d	—
4	e	e	—
5	f	f	—
6	g	g	—
7	h	h	—

Silicon Art - Pixel Pig + Canary Token

by **Claude**

0779 HDL Project

`github.com/dxa4481/tiny`

> "*Pixel pig + canary token art on silicon - visible under microscope*"

How it works

This is a **Silicon Art** project for TinyTapeout using the **custom GDS** workflow. A **pixel pig** artwork and **canary token text** are directly written as metal polygons on the silicon, visible under a microscope when the chip is fabricated.

The design includes:

1. **Custom GDS** with pixel pig + canary token rendered on metal `.drawing` layers
2. A decorative border frame around the design
3. All required TinyTapeout pins properly defined on Metal4.pin layer
4. Power pins (VPWR, VGND) on TopMetal1.pin layer
5. A minimal Verilog stub with all outputs tied to ground

Important DRC note: Art uses `.drawing` layers (datatype 0) which are the only fabricated layers in TinyTapeout's IHP whitelist. All geometry meets DRC requirements:

- Pixel art: 7.88 µm pixels (min: 0.20 µm) ✅
- Text strokes: 0.19 µm at 3µm font (min: 0.16 µm) ✅
- Border: 1.0 µm wide (min: 0.16 µm) ✅
- Density fill: 2µm squares added to reach 35-60% metal coverage ✅

The design fits in the 202.08 × 154.98 µm tile area (TinyTapeout IHP 1x1 tile). Pin positions are precisely calculated to match the TinyTapeout IHP template DEF file.

How to test

The functional logic is minimal (just for TinyTapeout compatibility):

1. All digital outputs (`uo_out[7:0]`) are tied to ground (0x00)
2. All bidirectional outputs (`uio_out[7:0]`) are also grounded
3. Bidirectional pins are configured as inputs (`uio_oe = 0x00`)

The design maintains connections to all input pins internally to satisfy synthesis requirements, but outputs remain at logic 0 regardless of input values.

External hardware

No external hardware required. This is primarily an art project.

To view the silicon art after fabrication:

- Use an optical microscope with at least 50-100x magnification
- A metallurgical/reflected light microscope works best
- Look for the pixel pig (left) and canary token text (right)
- Art is on Metal1.drawing (8/0), Metal2.drawing (10/0), Metal3.drawing (30/0)
- The patterns appear as bright/reflective metal under the microscope
- The pig uses 81 pixels (7.88 µm each), text uses 3µm font

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	—

Mini Synth

by **Benedikt Krimmel**

`0780` `50 MHz` `HDL Project`

`github.com/PaxPlay/tiny-tapeout-design`

"Tiny digital synthesizer"

How it works

A monophonic synthesizer with three oscillators and an ADSR envelope.

Signal chain

1. **Phase accumulator** — 16-bit register incremented every sample tick (50 kHz) by a note-dependent value. Wraps freely.
2. **Triangle wave** — the phase is folded at the midpoint (`phase[15] ? ~phase : phase`) and shifted to 11 bits.
3. **Square wave** — derived from `phase[13]`, giving a square at 4× the fundamental frequency.
4. **LFO-detuned triangle** — a second 16-bit phase accumulator runs at almost the same rate, offset by a 4 Hz triangle LFO, producing a slow chorus/detune effect.
5. **Mix** — the three oscillators are summed into a 12-bit audio sample.
6. **ADSR envelope** — pulse-density amplitude control: a 7-bit counter (period 125) runs at the full 50 MHz clock. When the counter value is below the current volume level the raw audio is passed; otherwise the output is held at the midpoint (12'h800, silence). This gives a chopping frequency of 50 MHz / 125 = **400 kHz**, well above the audio pmod RC filter cutoff, with no aliasing into the audio band.
7. **Sigma-delta modulator** — a 12-bit first-order sigma-delta accumulator converts the envelope-controlled sample to a 1-bit PWM stream on `uo[7]` at the full 50 MHz clock rate.

Note selection

The 7 input bits (`ui[6:0]`) select any of the 128 standard MIDI notes (MIDI 0–127, C-1 to G9, roughly 8 Hz–12.5 kHz).

Pitch lookup is compressed using the octave-doubling property of equal temperament: only 12 base increments (one octave, anchored at C2 for precision) are stored. The octave is determined by a comparison chain; for notes at or above C2 the base increment is left-shifted, for notes below C2 it is right-shifted.

ADSR envelope

`ui[6]` is the gate input. A rising edge triggers the attack phase; releasing the gate (falling edge) triggers release.

Phase	Rate	Duration
Attack	vol++ every 8 samples	20 ms (0 → full)
Decay	vol– every 32 samples	30 ms (full → sustain)
Sustain	held at 62% volume	until gate released
Release	vol– every 64 samples	100 ms (sustain → 0)

Parameters

Parameter	Value
Clock	50 MHz
Sample rate	50 kHz (÷1000)
Phase accumulator	16-bit
Audio sample	12-bit unsigned
Chop counter period	125 (f_chop = 400 kHz)
Note range	MIDI 0–127 (C-1–G9)
Note resolution	1 semitone (128 notes, 7 bits)

How to test

Connect the Tiny Tapeout Audio Pmod to the output pin header. Drive `ui[6:0]` with a 7-bit MIDI note number (0 = C-1, 69 = A4, 127 = G9) and set `ui[7]` high to open the gate. The ADSR envelope will attack, decay, and sustain while the gate is held; releasing the gate triggers the release phase.

A Verilator simulation is included under `verilator/`. Build and run it with:

```
cd verilator
make
./synth
# Output written to audio.raw — play back with:
ffplay -f u16le -ar 50000 audio.raw
# Or convert using ffmpeg:
ffmpeg -f u16le -ar 50000 -i audio.raw audio.wav
```

This renders a chromatic sweep of 64 notes (MIDI 36–99, C2–Eb7), 0.8 s gate-on + 0.2 s release per note, to `audio.raw` as 16-bit unsigned PCM at 50 kHz mono.

External hardware

Tiny Tapeout Audio Pmod connected to the `uo` output header.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	MIDI note bit 0	—	—
1	MIDI note bit 1	—	—
2	MIDI note bit 2	—	—
3	MIDI note bit 3	—	—
4	MIDI note bit 4	—	—
5	MIDI note bit 5	—	—
6	MIDI note bit 6	—	—
7	Gate (note on)	AudioPWM	—

Simon Says memory game

by **Uri Shaked**

0781 50 kHz HDL Project

github.com/urish/tt-simon-game

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

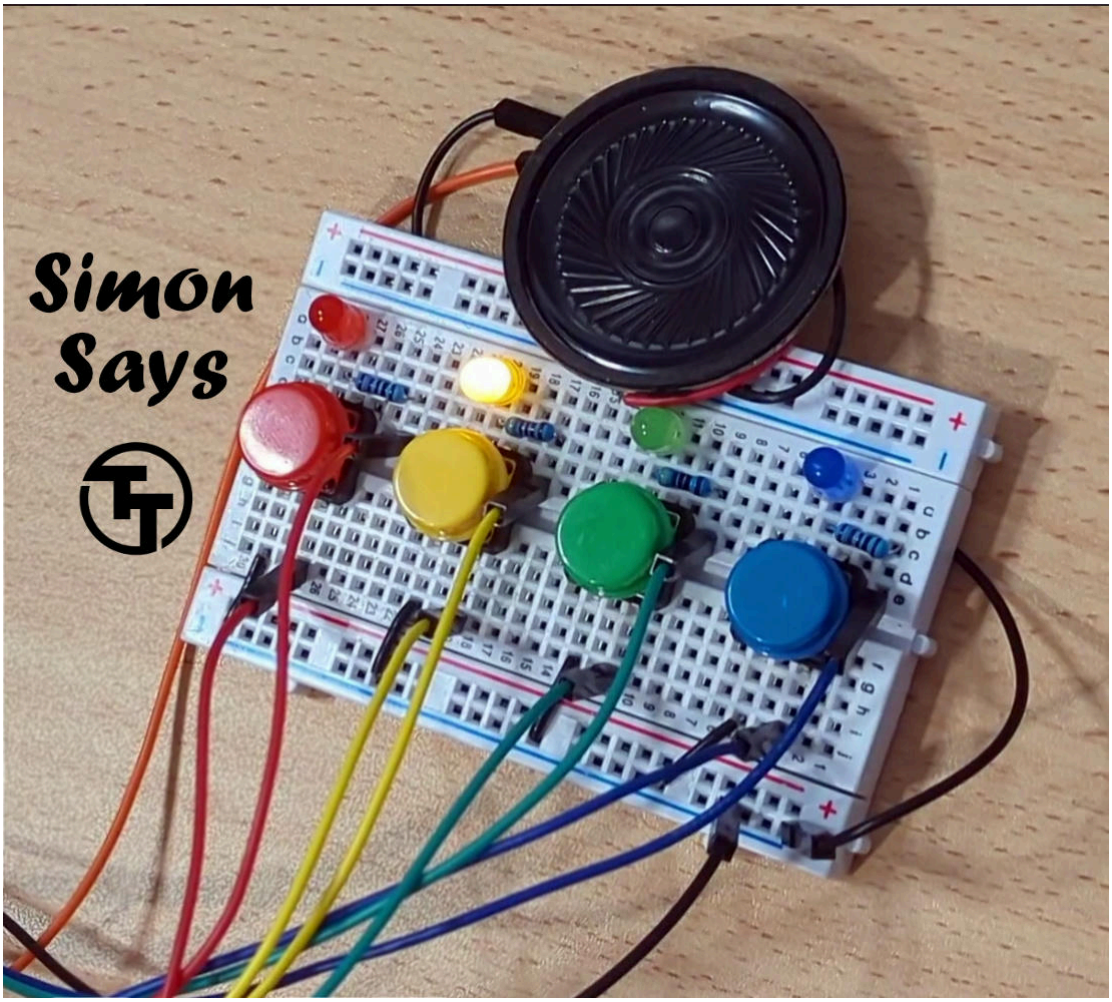


Figure 781.1: Simon Says Game

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 55 KHz.

The internal clock is generated by a 13-stage ring oscillator, divided by 16384 to get the desired frequency. The divider value was determined by running the ring oscillator simulation in xschem/simulation/ring_osc.spice.

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	—

Miniproc

by **Tapeout**

0782 HDL Project

`github.com/Nampuk/tapatapatapa`

"It could be a processor, but right now its just an ALU"

How it works

- Read tapeout.typ

Explain how your project works

How to test

- Read tapeout.typ

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	Nibble0	C0	B0
1	Nibble1	C1	B1
2	Nibble2	C2	B2
3	Nibble3	C3	B3
4	Addr0	C4	B4
5	Addr1	C5	B5
6	Addr2	C6	B6
7	Addr3	C7	B7

INTERCAL ALU

by **Rebecca G. Bettencourt**

0783 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

Tiny Tapeout Accumulator

by **Alp Bolukbasi**

0784 | 1 MHz | Wokwi Project

github.com/alpblkba/tiny-tapeout-chip-design

wokwi.com/projects/455300931094822913

"A simple event accumulator that counts input pulses and outputs a 4-bit activity value."

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	clk	out0	—
1	rst_n	out1	—
2	in	out2	—
3	—	out3	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Hardware UTF Encoder/Decoder

by **Rebecca G. Bettencourt**

0785 HDL Project

`github.com/RebeccaRGB/ttihp-hardware-utf`

"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

Custom_ASIC

by **Jakob Schwarz**

0786 | 10 kHz | Wokwi Project

github.com/jackb7273-jpg/Tiny_Tapeout

wokwi.com/projects/455291837898350593

"Preliminary, not yet working"

How it works

–currently just a test of random design

How to test

–currently just a test of random design

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	0	1o	—
1	1	2o	—
2	2	3o	—
3	3	4o	—
4	4	5o	—
5	5	6o	—
6	6	7o	—
7	7	8o	—

Universal Binary to Segment Decoder

by **Rebecca G. Bettencourt**

0787 HDL Project

`github.com/RebeccaRGB/ttihp-ubcd`

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

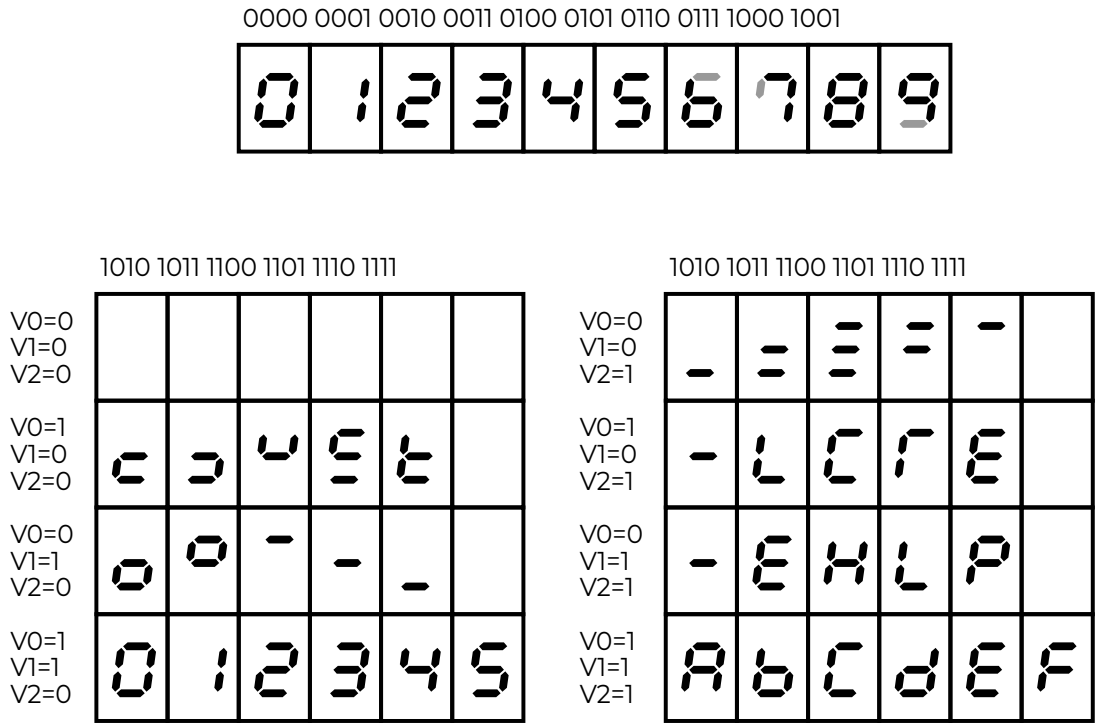
How it works

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:

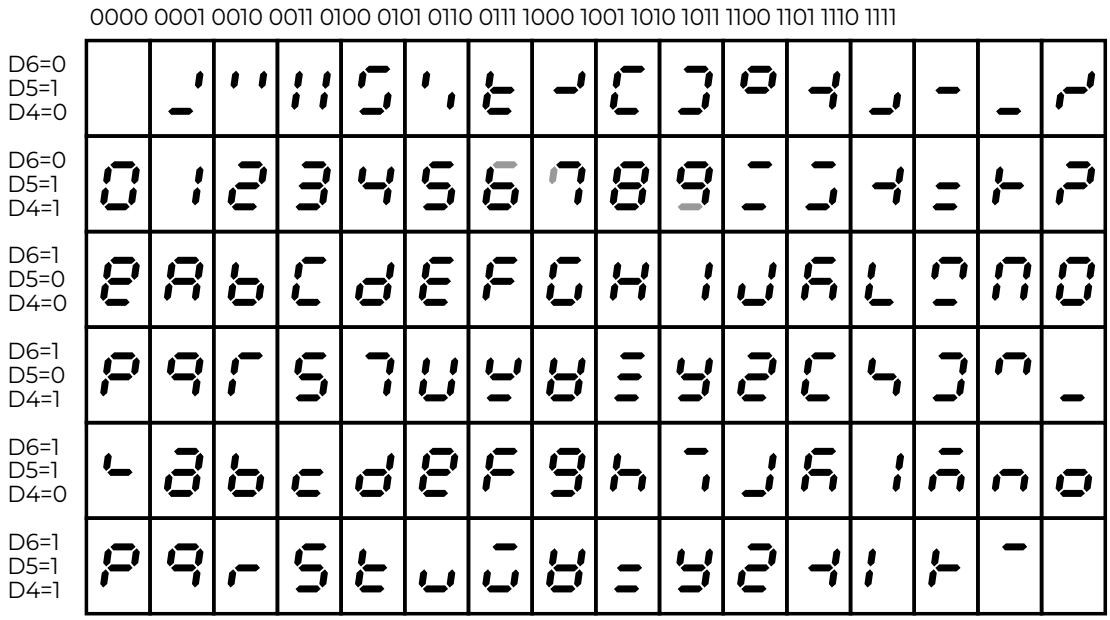
—	Dedicated Input	Dedicated Output	Bidirectional
0	A	Segment a	Input - X6
1	B	Segment b	Input - X7

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

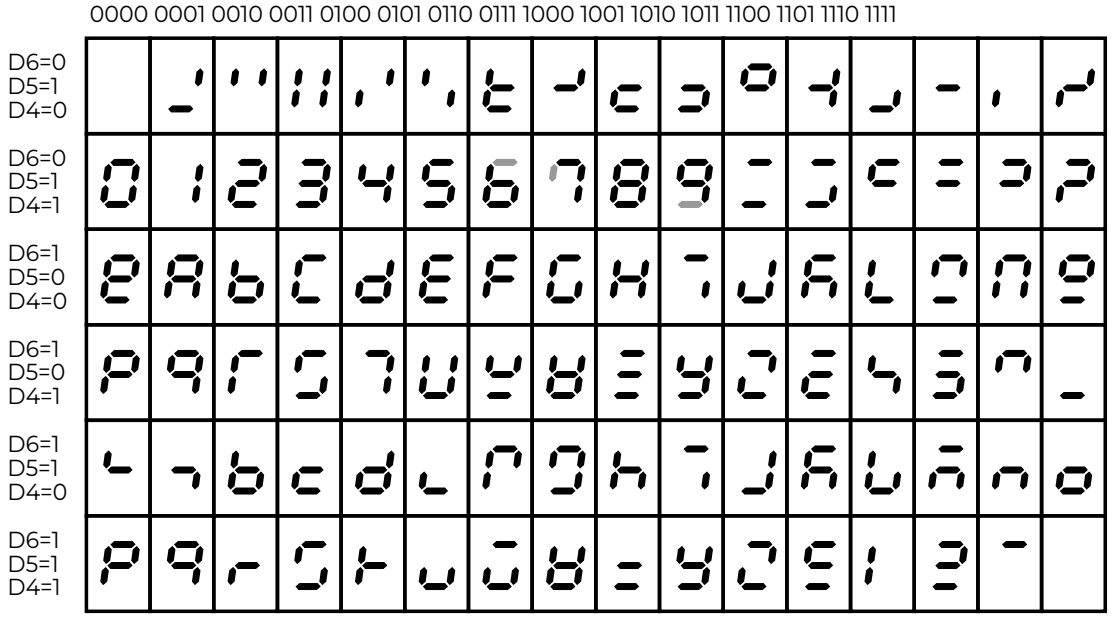
ASCII to seven segment decoder

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).

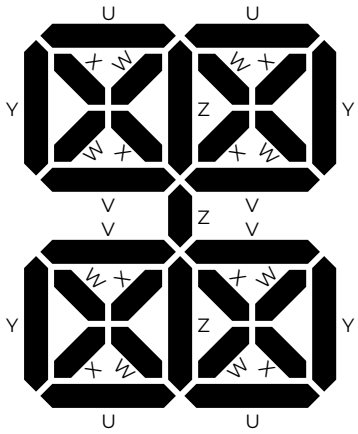
The pin assignments in this mode are:

—	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

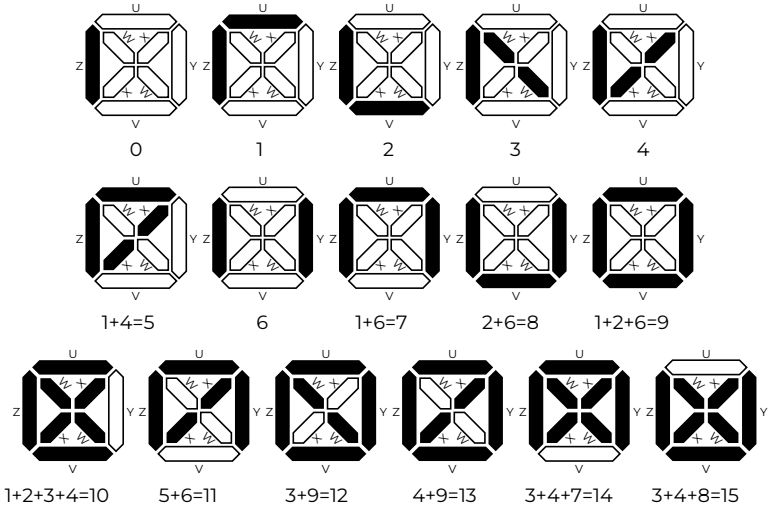
Dual BCD to Cistercian numeral decoder

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.

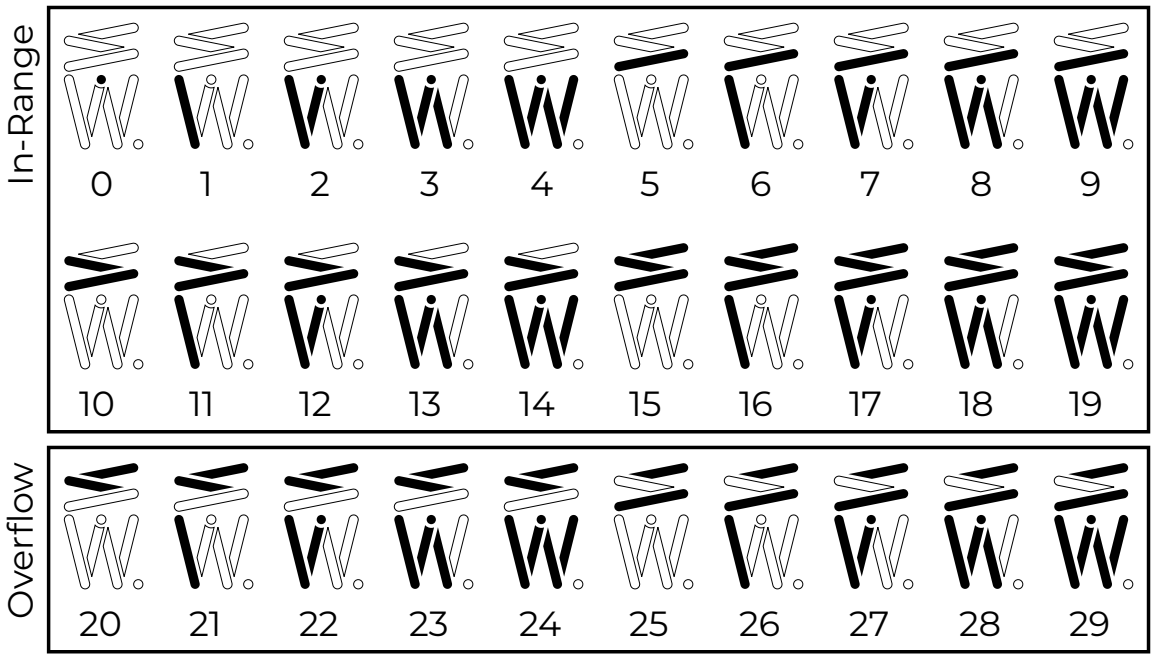
The pin assignments in this mode are:

—	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

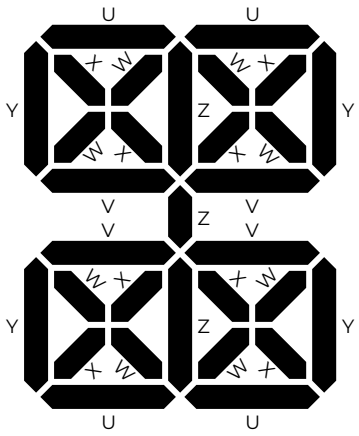
How to test

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)

Simple Counter

by **Tom Ulrich**

0800 Wokwi Project

`github.com/tulrich2/TinyTapeout-Counter`

`wokwi.com/projects/459210187582694401`

"A simple counter counting seconds"

How it works

A simple counter counting seconds when run with the 10 kHz oscillator.

How to test

Start the block with the 10 kHz oscillator enabled.

External hardware

None.

Project Pinout

Digital Pins

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

GDS Test

by **Anders Granskov Hansen**

0802 Wokwi Project

github.com/Daddybot/GDS

wokwi.com/projects/456571638794523649

"Testing"

How it works

Test of two AND gates

How to test

Run that thang

External hardware

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input a	output and	—
1	input b	output and	—
2	input a	—	—
3	input b	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

sree

by **sree**

0804 10 Hz Wokwi Project

github.com/shreeveni-x90/SREE

wokwi.com/projects/456578694921494529

“sree”

How it works

Press the button

How to test

Press

External hardware

Led

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	IN0	—	—
1	IN1	—	—
2	IN2	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

IriglooCs-first-Wokwi-design

by **IriglooC**

0806 Wokwi Project

github.com/IriglooC/tiny-tapeout-first-design

wokwi.com/projects/455291649222749185

"8-bit-adder"

How it works

Adds the bits from the nth and n-1th input bits to count in binary.

How to test

Put input voltage on the inputs (0V - 5V) and test the outcome on the corresponding output.

input n	input n-1	output	carry
1	1	0	1
1	0	1	0
0	1	1	0
0	0	0	0

External hardware

It is recommended to use LEDs or other ways to test the output pins.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input a	output a	—
1	input b	output b	—
2	input c	output c	—
3	input d	output d	—
4	input e	output e	—
5	input f	output f	—
6	input g	output g	—
7	input h	output h	—

TinyTapeout logic gate test

by **Fang Yi**

0808 Wokwi Project

github.com/XxFanny17xX/tinytapeoutFY

wokwi.com/projects/455291787137823745

"Test"

How it works

For testing

How to test

Press on the button

External hardware

No

Project Pinout

Digital Pins

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

Tiny Tapeout Workshop Test

by **Alireza Nik**

`0810` `Wokwi Project`

`github.com/ds-anik/Tiny-Tapeout`

`wokwi.com/projects/456571686260436993`

"test some logic gates"

How it works

This is a Test md file to use github actions

How to test

you have to proceed the instuctions

External hardware

LED display

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input a	output nand	—
1	input b	—	—
2	—	—	—
3	—	—	—
4	input c	out2	—
5	input d	out3	—
6	input e	out4	—
7	input f	out5	—

7 Segment BCD

by **Gabriel Crawford**

0812 Wokwi Project

github.com/gcgc321/TT26-Verilog-Design

wokwi.com/projects/456576571487651841

"Built CLA in Wokwi"

How it works

The combinational circuit takes in 4 inputs from A as MSB and D as LSB. This is meant to display 1-9 on a seven segment display

How to test

Use dip switches to see numbers on display

External hardware

Dip switch and Seven Segment Display

Project Pinout

Digital Pins

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

Hello

by **Edgar Lakis**

0814 100 Hz Wokwi Project

github.com/edga/tt_hello_gds

wokwi.com/projects/456571605697249281

"10 minutes random"

How it works

A clock diver and two gates.

How to test

Se changes on 7-seg leds.

External hardware

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

Project Pinout

Digital Pins

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

7 segment number viewer

by **Carl Meding**

0816 Wokwi Project

github.com/Ghunk09/ttworkshop_TEST

wokwi.com/projects/456578784059908097

"Each input displays its respective number on the display"

How it works

I just does

How to test

Follow instructions on info.ymal

External hardware

7 segment display.

Project Pinout

Digital Pins

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

7 Segment Binary Viewer

by **Jonathan Bangert**

0818 Wokwi Project

`github.com/jonbng/ttworkshop`

`wokwi.com/projects/456571724337390593`

"Jonathan Bangerts wokwi chip to show binary input on a 7 segment display"

How it works

The chip takes a binary input on the 8 input pins and shows the corresponding number on the 7 segment display.

How to test

Input a binary number in the input pins, watch the 7 segment display show the correct number.

External hardware

- 7 segment display

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	binary space 1	7 segment pin A	—
1	binary space 2	7 segment pin B	—
2	binary space 3	7 segment pin C	—
3	binary space 4	7 segment pin D	—
4	binary space 5	7 segment pin E	—
5	binary space 6	7 segment pin F	—
6	binary space 7	7 segment pin G	—
7	binary space 8	7 segment pin H	—

Simon Says

by **Daniel Brormann**

0832 | 10 kHz | Wokwi Project

github.com/dabro02/Daniel-s-Wokwi-Design

wokwi.com/projects/455303220350374913

"4-bit serializer"

How it works

Input a custom 4-bit number to the shift register. Use a switch to enable transmission. The LED 8 blinks the given pattern.

How to test

Set a default 4-bit Number on the first 4 Input Switches. The Input is set to the shift register by clock (or by step). After the number is stored in the register, switch 8 (io7) enables the transmission in clock speed.

External hardware

> 5 Inputs and > 1 LED.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Initial Input 1	—	—
1	Initial Input 2	—	—
2	Initial Input 3	—	—
3	Initial Input 4	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	shift write/output	Output for shifted bits.	—

Tadder

by **Tim Burr**

0833 Wokwi Project

github.com/potatojuicemachine/tiny-tapeout

wokwi.com/projects/450492208120711169

"I don't know what it should do yet"

How it works

Currently it takes 2 4 bit numbers and adds them

How to test

Input 2 4bit numbers and look at the output

External hardware

8 switches and a 8 segment display

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	a1	o1	—
1	a2	o2	—
2	a3	o3	—
3	a4	o4	—
4	b1	o5	—
5	b2	o6	—
6	b3	o7	—
7	b4	—	—

simple XOR cipher

by **Jan**

834 Wokwi Project

`github.com/agentofail/TinyTapeout`

`wokwi.com/projects/455292153909854209`

"A simple xor cypher. Input the key using reset. encrypt using clock."

How it works

This Project implements a simple XOR cipher. A Key can be loaded into a 8 deep shift register. A leading head of ones is used to detect setting the key. Afterwards the message supplied at the input pins is taken xor with the loaded key. The output is displayed. The key repeats after encrypting 8 times 8 bits.

DISCLAIMER: Do not use this implementation for encrypting anything relevant. XOR encryption by itself, using a constant repeating key, can for example be trivially broken using a frequency analysis.

How to test

Upon initialization output bits 7 and 8 should be high, anything else low. Note that this may depend on the starting values of the FlipFlops used in the shift register and therefor is not 100% certain.

Loading a key

Set all input bits to zero and press reset 9 times. This flushes the shift register with zeros. Output 7 should now be low as well. Set all input bits to one and press reset. Set the input bits to 8 bits of a key and press reset. Repeat 7 more times. Now the shift register should be full and since the first 8bit sequence pushed into the shift register was only ones bit 8 should be low now. This indicates the key has been loaded successfully and encryption can start.

Encrypt

Set all input bits to 8 bits of a message. Press clock. Read output. The key repeats after encrypting 8 times 8 bits. Note that there is no way to tell how often clock was pressed, so the user has to keep track of that for themself.

External hardware

No external hardware is needed for this project.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Input bit 1 for key and msg	Output bit 1 msg	—
1	Input bit 2 for key and msg	Output bit 1 msg	—
2	Input bit 3 for key and msg	Output bit 1 msg	—
3	Input bit 4 for key and msg	Output bit 1 msg	—
4	Input bit 5 for key and msg	Output bit 1 msg	—
5	Input bit 6 for key and msg	Output bit 1 msg	—
6	Input bit 7 for key and msg	Output bit 1 msg, key head not zeros	—
7	Input bit 8 for key and msg	Output bit 1 msg, key head not ones	—

test_prj

by **Mo**

0835 10 kHz Wokwi Project

github.com/mbdaneshvar/gds_maker

wokwi.com/projects/450492222691728385

“H blinker”

How it works

How to test

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	test_in	o0	—
1	—	o1	—
2	—	o2	—
3	—	o3	—
4	—	o4	—
5	—	o5	—
6	—	—	—
7	—	—	—

Hello tinyTapout

by **Leon**

0836 10 kHz Wokwi Project

github.com/SinnMachen/nein

wokwi.com/projects/455300425088680961

"it does nothing fancy"

How it works

It does nothing.

How to test

Do not test it, it WILL explode!!!

External hardware

I connected it to a seven segment display and a frequency generator.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input a	output a	—
1	input b	output b	—
2	input c	output c	—
3	input d	output d	—
4	input e	output e	—
5	input f	output f	—
6	input g	output g	—
7	input h	output h	—

Not a Dinosaur

by **Tom** & **Mo**

0837 | 25.175 MHz | HDL Project

`github.com/tulrich2/ttihp-verilog`

> *"Tried to do the Chrome dinosaur jumping game, but failed"*

How it works

By pressing the button connected to the input with index 1, the character on the screen jumps.

How to test

Press the button connected to the input with index 1; the character should jump

Project Pinout

Digital Pins

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

^My first design

by **Mahmoud Sanad**

0838 Wokwi Project

`github.com/mabo-elfotoh/my_tiny_tape_design`

`wokwi.com/projects/455291692145189889`

"This my first and simplest design. This design just ANDing the first two inputs, ORing the next two inputs, the fifth input is passed directly to the third, forth and fifth. Afterwards, just direct mapping"

How it works

AND is connected to input 0 and 1 and gives output to 0 OR is connected to input 2 and 3 and gives output to 1 input 5 is connected to output 2, 3, and 4 finally direct 6 and 7 inputs to 6 and 7 outputs directly.

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	—	output and	—
1	—	output or	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Silly Dog

by **Daniel Wagner** & **Tim Burr**

0839 | 25.175 MHz | HDL Project

`github.com/potatojuicemachine/vga-tt`

> *"A dog bouncing around the screen controlled by the user."*

How it works

This displays a square that bounces around the screen and can be controlled by the user by the 8 inputs.

How to test

Plug in the VGA and see if you can control the box

External hardware

A VGA cable, the 8 input switches

Project Pinout

Digital Pins

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

Tiny Tapeout - Riddle Implementation

by **Aber**

0840 Wokwi Project

`github.com/Aber-58/TinyTapeout-Aber`

`wokwi.com/projects/455290751669808129`

> "*Trying to implement a funny riddle implementation for the chip.*"

How it works

I'll explain it later...

How to test

I'll explain it later ...

External hardware

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input a	output and	—
1	input b	output nand	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

vga test project

by **mo daneshvar**

0841 | 25.175 MHz | HDL Project

`github.com/mbdaneshvar/sv_tapeout`

“lorem”

How it works

tbd

How to test

tbd

External hardware

tbd

Project Pinout

Digital Pins

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

Tobias first Wokwi design

by **Tobias**

0842 10 kHz Wokwi Project

github.com/ucmpb/TinyTapeout

wokwi.com/projects/455291724163472385

"first wokwi design"

How it works

NAND input a and b to output 0

How to test

set the outputs and check the outputs:

input a | input b | output 0 0 |0 |1 0 |1 |1 1 |1 |0 1 |0 |1

External hardware

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input a	output 0	—
1	input b	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Discrete-to-ASIC Delta-Sigma Acquisition System

by **YU ICHI**

843 | 2.048 MHz | HDL Project

`github.com/yuya0421/tt_ihp_cic_filter`

"A 3rd order CIC filter for PDM signal processing."

How it works

Input 1-bit PDM signal from an external discrete 2nd-order Delta-Sigma Modulator. This ASIC implements a 3rd-order CIC filter (Decimation Ratio = 32) to demodulate the PDM signal into 8-bit PCM audio data. The design focuses on Hardware-in-the-Loop verification between a custom PCB and the ASIC.

How to test

1. **Stimulus:** Generate a 10kHz Sine wave using Digilent Analog Discovery (Wavegen) and input it to the external PCB.
2. **Connection:** Connect the PCB output to `ui_in[0]`.
3. **Clock:** Synchronize the PCB clock with the ASIC clock (2.048 MHz).
4. **Observation:** Monitor `uo_out[7:0]` with a Logic Analyzer. The reconstructed waveform should match the input sine wave.

External hardware

- Custom PCB (Discrete 2nd-order Delta-Sigma Modulator)
- Digilent Analog Discovery 2/3 (Wavegen & Logic Analyzer)

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	PDM Input (d_in)	PCM Output Bit 7 (MSB)	NC
1	NC	PCM Output Bit 6	NC
2	NC	PCM Output Bit 5	NC
3	NC	PCM Output Bit 4	NC
4	NC	PCM Output Bit 3	NC
5	NC	PCM Output Bit 2	NC
6	NC	PCM Output Bit 1	NC

#	Input	Output	Bidirectional
7	NC	PCM Output Bit 0 (LSB)	NC

My Tiny Tapeout

by **flo100500**

844 Wokwi Project

github.com/flo100500/tinytapeout

wokwi.com/projects/455291807082792961

"Test123"

How it works

Test 1234

How to test

Test 134

External hardware

Test 12343

Project Pinout

Digital Pins

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

Quad SPI Aggregator

by **Zack Li**

0845 | 20 MHz | HDL Project

github.com/xzackli/ttihp-verilog-template

"Serialize four SPI mode 0 inputs into one output."

How it works

This module aggregates four SPI mode 0 ADC inputs and serializes them into a single faster SPI output stream. The pin assignments allow you to ingest four ADCs as SPI peripherals (mode 0), with configuration pins for bit depth, null bits (bits after chip "select" toggle), and clock divider. The clock divider ui[7:6] sets the ADC SCLK period: 0 -> 4x, 1 -> 8x, 2 -> 12x, 3 -> 16x system clocks.

The receiver (e.g. FPGA) should use TX_CS_N and TX_SCLK to sample TX_MOSI:

1. TX_CS_N falling edge indicates frame start
2. Sample TX_MOSI on TX_SCLK rising edges while TX_CS_N is low
3. TX_CS_N rising edge indicates frame end

The output contains (bit_depth * 4) bits per frame, in order: ADC0, ADC1, ADC2, ADC3. For 12-bit ADCs, that is 48 bits total per frame.

Configuration Inputs (ui[7:0])

Pin	Name	Description
ui[0]	CFG_BITDEPTH_0	Bits per sample for each ADC [0]
ui[1]	CFG_BITDEPTH_1	Bits per sample for each ADC [1]
ui[2]	CFG_BITDEPTH_2	Bits per sample for each ADC [2]
ui[3]	CFG_BITDEPTH_3	Bits per sample for each ADC [3]
ui[4]	CFG_NULL_0	Bits after chip select during sampling [0]
ui[5]	CFG_NULL_1	Bits after chip select during sampling [1]
ui[6]	CFG_CLKDIV_0	Clock divider for ADC serial clock [0]
ui[7]	CFG_CLKDIV_1	Clock divider for ADC serial clock [1]

The bit depth is configured as ui[3:0] + 1, allowing 1-16 bits per ADC sample.

Output Pins (uo[7:0])

Pin	Name	Description
uo[0]	ADC0_SCLK	ADC 0 serial clock

uo[1]	ADC0_CS_N	ADC 0 chip select (active low)
uo[2]	ADC1_SCLK	ADC 1 serial clock
uo[3]	TX_MOSI	Transmit data output
uo[4]	ADC1_CS_N	ADC 1 chip select (active low)
uo[5]	TX_SCLK	Transmit serial clock
uo[6]	ADC2_SCLK	ADC 2 serial clock
uo[7]	ADC2_CS_N	ADC 2 chip select (active low)

Bidirectional Pins (uio[7:0])

Pin	Name	Direction	Description
uio[0]	ADC0_MISO	Input	ADC 0 data input
uio[1]	ADC1_MISO	Input	ADC 1 data input
uio[2]	ADC2_MISO	Input	ADC 2 data input
uio[3]	ADC3_MISO	Input	ADC 3 data input
uio[4]	ADC3_SCLK	Output	ADC 3 serial clock
uio[5]	ADC3_CS_N	Output	ADC 3 chip select (active low)
uio[6]	TX_CS_N	Output	Transmit chip select (active low)
uio[7]	-	Input	Unused

How to test

The tests in test/ show the configuration. A common scenario is MCP3201 ADC which has 12 bit samples, 2 null bits, and would then comfortably fit with a clock divider with both pins set to 0 (4x). With a 20 MHz system clock, the ADC phase takes 14 * 4 = 56 clocks, and TX phase takes 12 * 4 = 48 clocks, yielding about 192 kHz sample rate.

External hardware

This is designed for SPI peripherals like four MCP3201 ADCs, and for output to something that accepts SPI. The ASIC here is designed to function like an SPI mode 0 peripheral, so you can output to GPIO pins of an FPGA for example.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	CFG_BITDEPTH_0	ADC0_SCLK	ADC0_MISO
1	CFG_BITDEPTH_1	ADC0_CS_N	ADC1_MISO

#	Input	Output	Bidirectional
2	CFG_BITDEPTH_2	ADC1_SCLK	ADC2_MISO
3	CFG_BITDEPTH_3	TX_MOSI	ADC3_MISO
4	CFG_NULL_0	ADC1_CS_N	ADC3_SCLK
5	CFG_NULL_1	TX_SCLK	ADC3_CS_N
6	CFG_CLKDIV_0	ADC2_SCLK	TX_CS_N
7	CFG_CLKDIV_1	ADC2_CS_N	—

Count Upwards

by **Andreas Noebel**

0846 1 Hz Wokwi Project

github.com/Andreas-Noebel/Tiny-Tapeout

wokwi.com/projects/455303592417686529

"Count upwards from 0 to 9 and loop back"

How it works

Input swutches are directly connected to the segment display using and gates. The lowest inupt (7) toggle the segment display ... ## How to test Todo: This needs to be added in the future

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	—	7-Segment a	—
1	—	7-Segment b	—
2	—	7-Segment c	—
3	—	7-Segment d	—
4	—	7-Segment e	—
5	—	7-Segment f	—
6	—	7-Segment g	—
7	—	7-Segment h	—

Digital Lock with Easter Eggs

by **Paula Lozano**

0847 Wokwi Project

`github.com/pacalodu/tinytapeout`

`wokwi.com/projects/453664332125344769`

"A 4-bit digital lock that unlocks when the correct code is entered, with additional outputs for near-miss patterns as Easter eggs."

How it works

This project implements a **4-bit digital lock** with additional "Easter egg" outputs for near-miss patterns.

Inputs

- `IN0` = Least significant bit of the 4-bit code
- `IN1` = Second bit of the code
- `IN2` = Third bit of the code
- `IN3` = Most significant bit of the code

Outputs

- `UNLOCK` = Goes high when the entered code is exactly `1011`
- `EASTER0` = Goes high when the code is `1010` (one bit off)
- `EASTER1` = Goes high when the code is `1001` (one bit off)
- `EASTER2` = Goes high when the code is `0011` (different pattern)

Logic Implementation

- Purely combinational logic
- Uses **2-input AND gates** and **NOT gates**
- Each output detects a specific pattern using AND/NOT combinations
- No clock or sequential logic is used

How to test

1. Open the project in Wokwi.
2. Toggle the input switches (`IN3` to `IN0`) to enter a 4-bit code.
3. Observe the output LEDs or 7-segment display:

Input Code	UNLOCK	EASTER0	EASTER1	EASTER2
1011	1	0	0	0
1010	0	1	0	0
1001	0	0	1	0

0011	0	0	0	1
Other	0	0	0	0

4. Only the correct pattern should activate **UNLOCK**.
5. The Easter outputs show “near-miss” patterns and should be low for the exact code.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Least significant bit of code (IN0)	UNLOCK output: high when code = 1011	—
1	Code bit IN1	EASTER0 output: high when code = 1010	—
2	Code bit IN2	EASTER1 output: high when code = 1001	—
3	Most significant bit of code (IN3)	EASTER2 output: high when code = 0011	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Nielss first failure

by **Niels Engelbrecht**

0848 1 Hz Wokwi Project

github.com/Engelbrecht-N/Wokiwi_template

wokwi.com/projects/455291645628225537

"Count from 0 to 9 continously"

How it works

Counts upwards from 0 to 9 continously if the counter is enabled. Not as special as planned due to medical reasons and finals.

How to test

Change the position of the enable button from 0 to 1 and look if the 7-segment display does the work as intendet. Maybe try to press the reset button as well. The second input only enables the dot in the display.

External hardware

LED display, mouse keyboard

Project Pinout

Digital Pins

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

tinyTapeVerilog_out

by **Florentia Afentaki**

0849 | 1 kHz | HDL Project

`github.com/floAfentaki/tinyTapeVerilog_out`

"an example of ihp tinyTapeout workflow"

How it works

A simple counter

How to test

Run the tb in the test folder

External hardware

none!

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	clock	8-bit counter	—
1	negative logic reset	—	—
2	enable	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Tiny_Tapeout_Test

by **Shai_Peretz**

0850 Wokwi Project

github.com/Shaip161/TinyTapeOut

wokwi.com/projects/455291698669433857

"Test Project"

How it works

Simple NAND Gate between input 0 and 1 connected to output 0

How to test

Set the inputs and check the output matches the NAND truth table

Project Pinout

Digital Pins

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

6 Bit Roulette

by **Sven Sutter**

0851 10 Hz Wokwi Project

github.com/svens0210/Roulette_Game

wokwi.com/projects/454491386271657985

"Roulette of a light with 6 bit"

How it works

The design uses a chain of 6 Flip-Flops.

- **Reset:** Loads the "ball" (sets the first Flip-Flop to 1, all others to 0).
- **Play:** The light moves to the next segment on every clock cycle.

How to test

1. **Configuration:** Set the Clock frequency to **10 Hz**.
2. **Start:** To start: Tap Reset. You should see Segment A (top) light up.
3. **Spin:** The light will start spinning around the display.

External hardware

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

Project Pinout

Digital Pins

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

Yet another VGA tinytapeout

by **Anastasios Zouzias**

0864 | 25.175 MHz | HDL Project

`github.com/zouzias/ttihp-verilog-eth`

> "*This is a simple verilog project shifting around a VGA image.*"

How it works

See `src/project.v`

How to test

Currently, the test are being bypassed.

External hardware

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

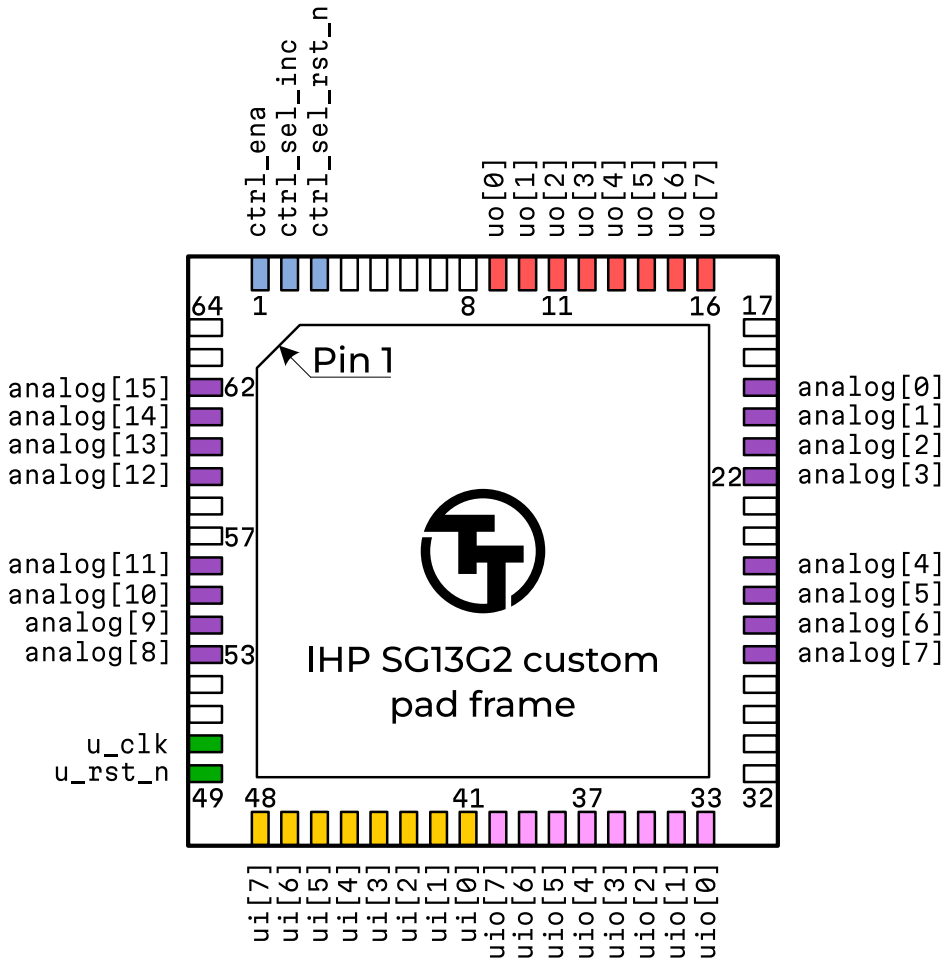
Project Pinout

Digital Pins

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

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:

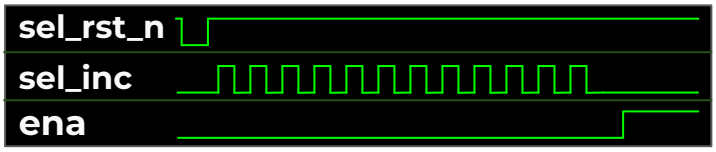


Figure 1: Mux signals for activating the design at address 12

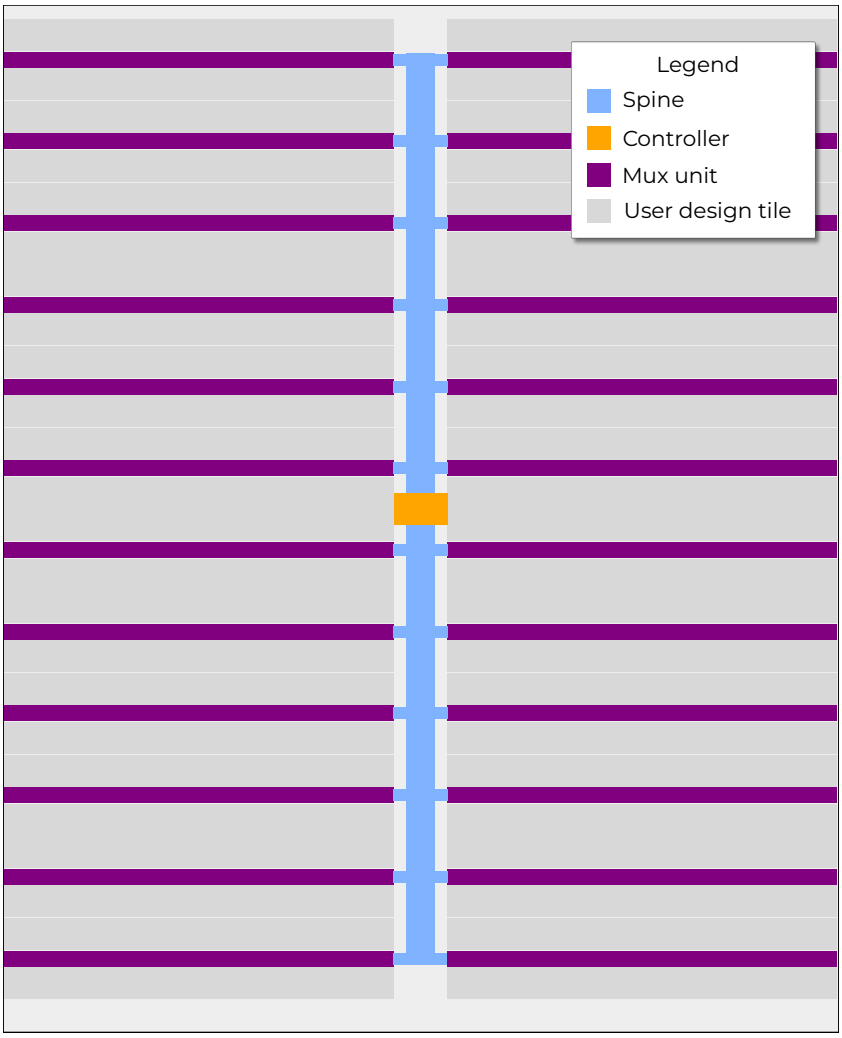


Figure 2: Mux Diagram

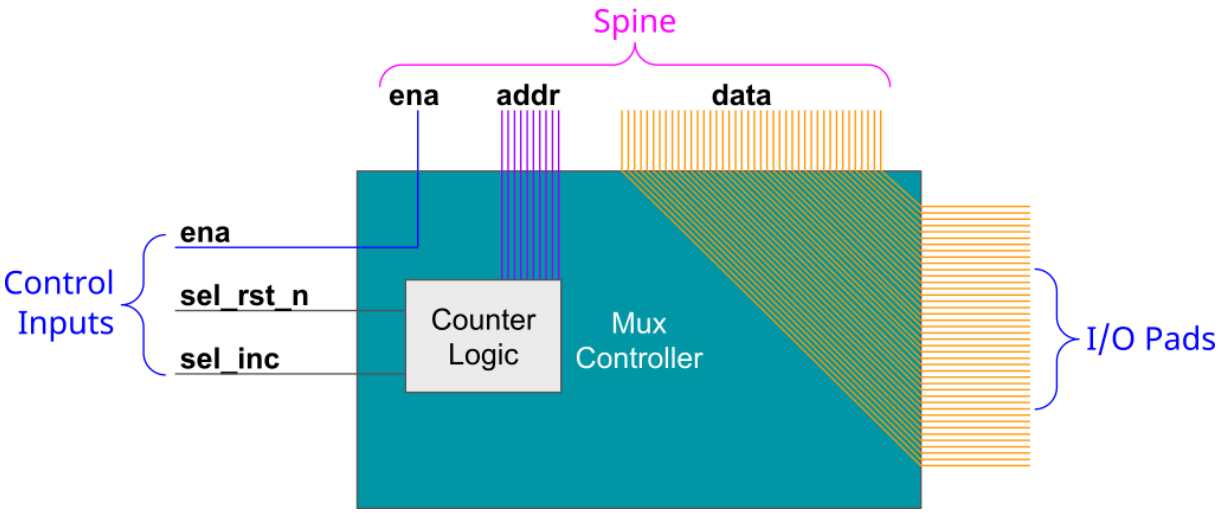


Figure 3: Mux Controller Diagram

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