Tiny Tapeout IHP 0p4 Datasheet

github.com/TinyTapeout/tinytapeout-ihp-0p4

March 29, 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/ttihp0p4-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]

float_synth

by **Niklaus Leuenberger**

0032 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?*

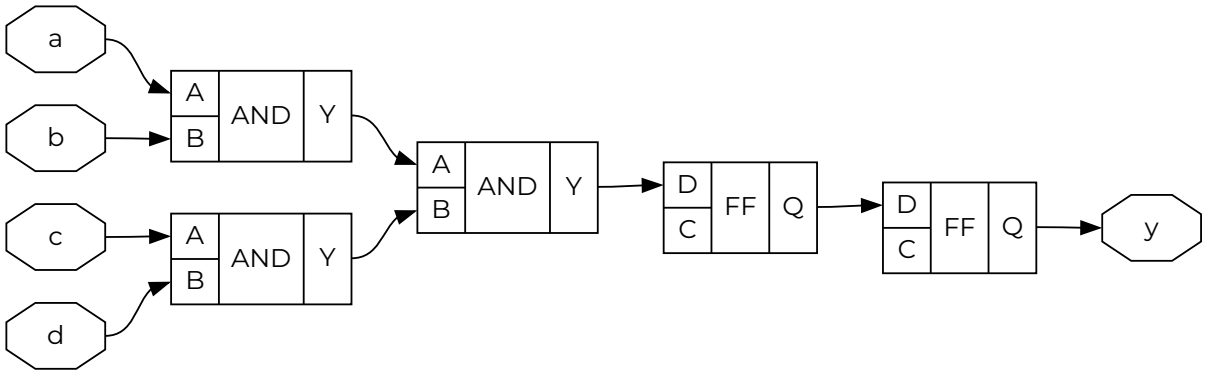
Spoiler: We absolutely can!

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.

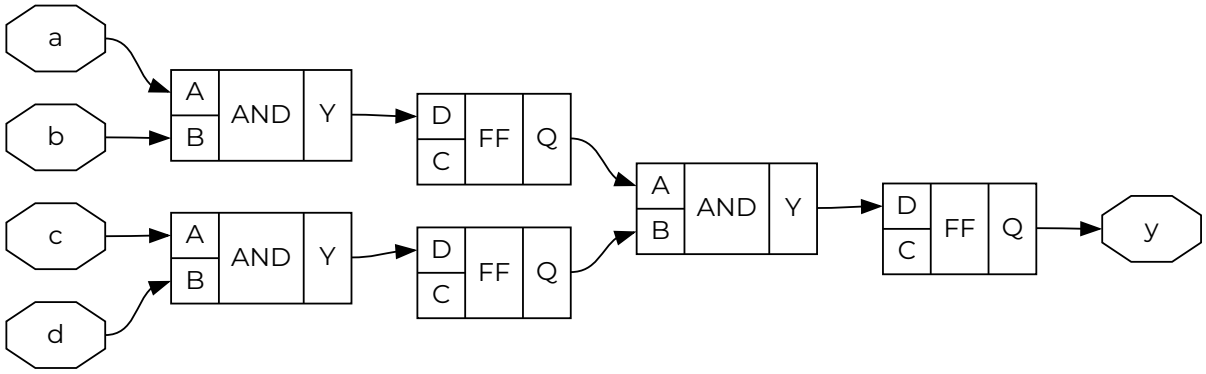
But we are lazy and want to test what the tools can do about it. So we simply add, after our *lazy* design, a few (or many) flip-flops back-to-back like a shift register. That does not change what is computed, but simply introduces latency, i.e. the output is delayed by N clocks. The excellent ABC tool 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 32.1: 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 32.2: 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.

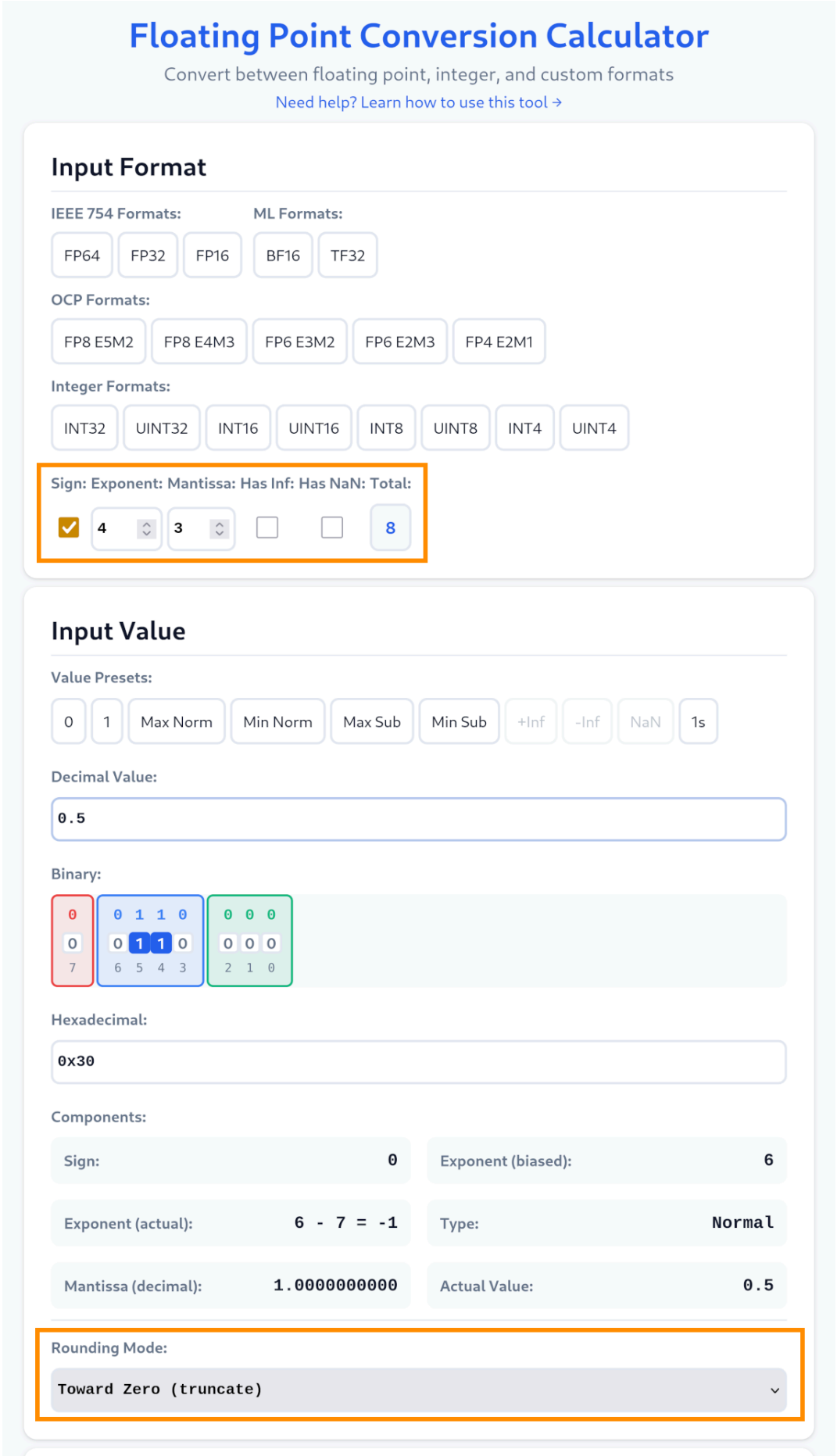


Figure 32.3: Screenshot of settings for Spencer Williams Floating Point Number Converter

As the main goal of the project was to retime the lazily written HDL for optimal delay, the clock can be as high as *555 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

Simon's Caterpillar

by **htfab**

0034 | 50 kHz | HDL Project

`github.com/htfab/ttihp0p4-caterpillar`

"Port of Caterpillar Logic to Simon Says PMOD"

How it works

Simon's Caterpillar is a re-implementation of the game Caterpillar Logic by Fuks Michael targeting Tiny Tapeout with the Simon Says PMOD.

The game consists of 20 levels. Each level has a secret rule that is valid for certain sequences of colors. For instance, if the rule is "contains exactly two yellow tokens" then blue-yellow-green-yellow is a valid sequence and yellow-red-blue is an invalid one.

A new level starts in exploration mode. You can ask an unlimited number of questions where you learn whether a particular sequence is valid or not. Once you know the rule you can activate challenge mode. Now the roles are reversed and the game asks you 15 questions. If you can answer all of them correctly, you advance to the next level.

How to test

Set the clock to 50 kHz. Activate and reset the project. The 7-segment display should indicate level 1 and only the blue led should light up. You are in exploration mode.

Exploration mode

A sequence of up to 7 colors can be typed into the buffer with short presses of the buttons. The leds indicate the sequence status in real time:

- red: sequence is invalid
- green: sequence is valid
- blue: buffer is empty
- yellow: buffer is full

(The empty sequence is neither valid nor invalid.)

Further operations are available as long button presses or a combination of two buttons:

- long-press red: clear buffer
- long-press yellow: erase last color from buffer ("backspace")
- long-press blue: show buffer contents (as a series of led flashes)

- long-press green: activate challenge mode
- short-press green & yellow: show a random valid sequence (and load into buffer)
- short-press red & blue: show a random invalid sequence (and load into buffer)
- short-press blue & yellow: switch to next level
- short-press red & green: switch to previous level
- short-press green & blue: toggle sound

Challenge mode

A sequence of up to 6 colors is shown as a series of led flashes. Press the green or red button to mark it as valid or invalid respectively.

Each correct answer adds a notch (turns on a new segment on the 7-segment display). After the 15th one the next level is loaded. An incorrect answer switches back to exploration mode.

Other keys and combinations:

- short-press or long-press blue: repeat the current question
- short-press red & yellow: switch back to exploration mode
- short-press blue & yellow: add a notch
- short-press red & green: remove a notch
- short-press green & blue: toggle sound

External hardware

Simon Says PMOD

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	red button	red led	segment A
1	green button	green led	segment B
2	blue button	yellow led	segment C
3	yellow button	blue led	segment D
4	display polarity	speaker	segment E
5	—	digit 1	segment F
6	—	digit 2	segment G
7	—	—	—

OCP MXFP8 Streaming MAC Unit

by **Olivier Chatelain**

0038 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`)

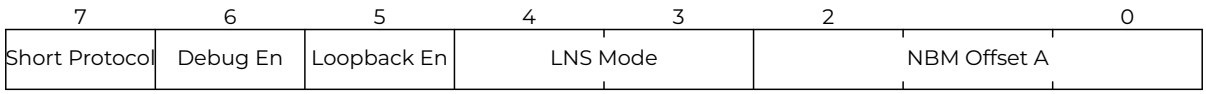


Figure 38.1: Metadata 0

- **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 38.2: 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]`):

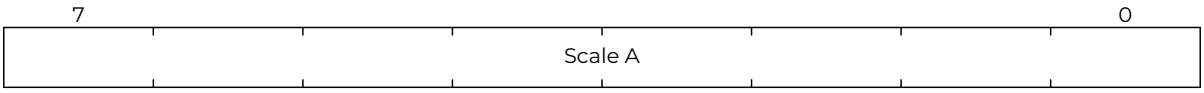


Figure 38.3: Scale A

- **Shared Scale A**: 8-bit unsigned biased exponent (UE8M0, Bias 127) applied to all elements in Operand A.

Config A (`uio_in[7:0]`):

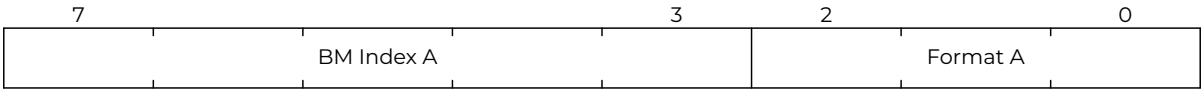


Figure 38.4: Config A

- **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]`):

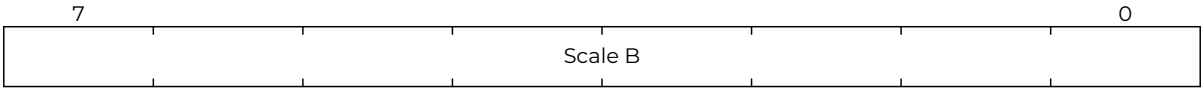


Figure 38.5: Scale B

- **Shared Scale B**: 8-bit unsigned biased exponent (UE8M0, Bias 127) applied to all elements in Operand B.

Config B (`uio_in[7:0]`):

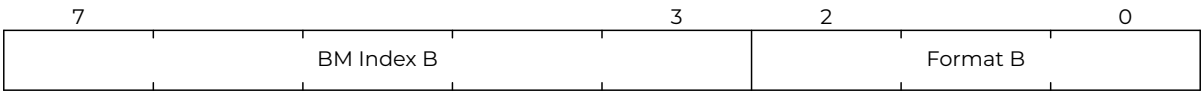


Figure 38.6: Config B

- **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]

USB CDC (Serial) Device

by **Uri Shaked**

0042 | 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	—

TinyMOA-IHP0P4-16x16

by **Ezra Wolf**

0075 50 MHz HDL Project

github.com/EzraWolf/TinyMOA-IHP0P4-16x16

"16x16 digital compute-in-memory (DCIM) core on an experimental IHP SG13G2 CMOS5L shuttle."

How it works

Digital compute-in-memory (DCIM) 16x16 array core.

How to test

Documentation in progress.

External hardware

Documentation in progress.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	data_in[0]	result[0]	wen (in)
1	data_in[1]	result[1]	execute (in)
2	data_in[2]	result[2]	read_next (in)
3	data_in[3]	result[3]	acc_clear (in)
4	data_in[4]	result[4]	col_sel[0] (out)
5	data_in[5]	result[5]	col_sel[1] (out)
6	data_in[6]	result[6] (zero)	col_sel[2] (out)
7	data_in[7]	result[7] (zero)	done (out)

Pattern-Guided Arithmetic Optimizations with MLIR per-op

by **L. Ledoux**

0098 500 kHz HDL Project

`github.com/Bynaryman/ttihp04_ieee754_perop`

"Streaming byte-fed IEEE754 per-op seq+comb accumulation core generated from MLIR for-loop."

How it works

`tt_um_lledoux_s3fdp_seqcomb` wraps a generated seq+comb arithmetic core built from MLIR with Emeraude + CIRCT.

The flow detects a loop pattern in MLIR (`scf.for` + `memref.load/store` + `arith.mulf/addf`) and emits a specialized S3FDP accumulator instead of generic floating-point datapath logic.

Canonical loop shape:

```
scf.for %k = %c0 to %c2 step %c1 {
  %x = memref.load %a[%k] : memref<2xf32>
  %y = memref.load %b[%k] : memref<2xf32>
  %acc = memref.load %c[%c0] : memref<2xf32>
  %m = arith.mulf %x, %y : f32
  %s = arith.addf %acc, %m : f32
  memref.store %s, %c[%c0] : memref<2xf32>
}
```

Arithmetic model (S3FDP)

S3FDP is used as a truncated Kulisch-like fixed-point accumulation strategy:

- multiply input terms,
- accumulate in a constrained fixed-point-like internal format,
- convert back to `f32`.

Specialization used in this project:

- `ovf=2`
- `msb=4`
- `lsb=-6`
- `chunk_size=16`

Llama / PyTorch source example

The source pattern used in experiments:

```python
class LlamaFfnSublayer(nn.Module):
    """Llama FFN sublayer using SwiGLU (SiLU-gated linear unit)."""

    def __init__(self, dim: int = 512, hidden_dim: int | None =
None, multiple_of: int = 256):
        super().__init__()
        if hidden_dim is None:
            hidden_dim = 4 * dim
            hidden_dim = int(2 * hidden_dim / 3)
            hidden_dim = multiple_of * ((hidden_dim + multiple_of -
1) // multiple_of)
        self.w_gate = nn.Linear(dim, hidden_dim, bias=False)
        self.w_up = nn.Linear(dim, hidden_dim, bias=False)
        self.w_down = nn.Linear(hidden_dim, dim, bias=False)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        gate = F.silu(self.w_gate(x))
        up = self.w_up(x)
        return self.w_down(gate * up)
```

High-level linalg excerpt:

```
...
%8 = linalg.generic
  {ins(%7: tensor<1x2x16xf32>) outs(%5: tensor<1x2x16xf32>) {
    %19 = arith.negf %in : f32
    %20 = math.exp %19 : f32
    %21 = arith.addf %20, %cst_1 : f32
    %22 = arith.divf %cst_1, %21 : f32
    linalg.yield %22 : f32
  }} -> tensor<1x2x16xf32>

%9 = linalg.generic
  {ins(%8, %7: tensor<1x2x16xf32>, tensor<1x2x16xf32>)
   outs(%5: tensor<1x2x16xf32>)} {
    %19 = arith.mulf %in, %in_7 : f32
    linalg.yield %19 : f32
  } -> tensor<1x2x16xf32>
...
```

Internal IR snapshots

Comb-specialized stage (`generated/ir-stages/20-flopoco-comb.mlir`):

```
%s3fdp_accum.r = hw.instance "s3fdp_accum"
@s3fdp_accum_core_wE8_wF23_cs16(
  clk: %3: !seq.clock, reset: %arg1: i1, x: %1: i32, y: %2: i32
) -> (r: i32)
```

Seq/HW aggregate stage (`generated/ir-stages/60-hw-aggregate.mlir`):

```
%27 = seq.clock_gate %26, %3
%s3fdp_accum.r = hw.instance "s3fdp_accum"
@s3fdp_accum_core_wE8_wF23_cs16(
  clk: %27: !seq.clock, reset: %reset: i1, x: %18: i32, y: %25: i32
) -> (r: i32)
seq.write %c_mem[%false] %s3fdp_accum.r wren %3 {latency = 1 :
i64} : !seq.hlmem<2xi32>
```

SV-lowered stage (`generated/ir-stages/90-hw-to-sv.mlir`):

```
%c_mem = sv.reg : !hw.inout<uarray<2xi32>>
sv.alwaysff(posedge %clk_0) {
  sv.if %4 {
    %33 = sv.array_index_inout %c_mem[%false] : !
hw.inout<uarray<2xi32>>, i1
    sv.passign %33, %s3fdp_accum.r : i32
  }
}
```

Interface protocol

Input stream on `ui_in[7:0]`:

- 20-byte frame, little-endian
- `a[0..1]` IEEE754 `f32` (8 bytes)
- `b[0..1]` IEEE754 `f32` (8 bytes)
- `c0` IEEE754 `f32` seed (4 bytes)

Execution:

- hold core reset during load,
- release reset after byte 20,
- wait 3 cycles,
- output one 32-bit result on `uo_out[7:0]` as 4 little-endian bytes.

Frame slot: 27 cycles (`20 load + 3 run + 4 output`).

`uio` pins are unused.

Generation and test

Generate core + IR stages:

```
./scripts/generate_s3fdp_core.sh
```

Run simulation:

```
cd test
make clean
make -B
```

Waveform screenshot:

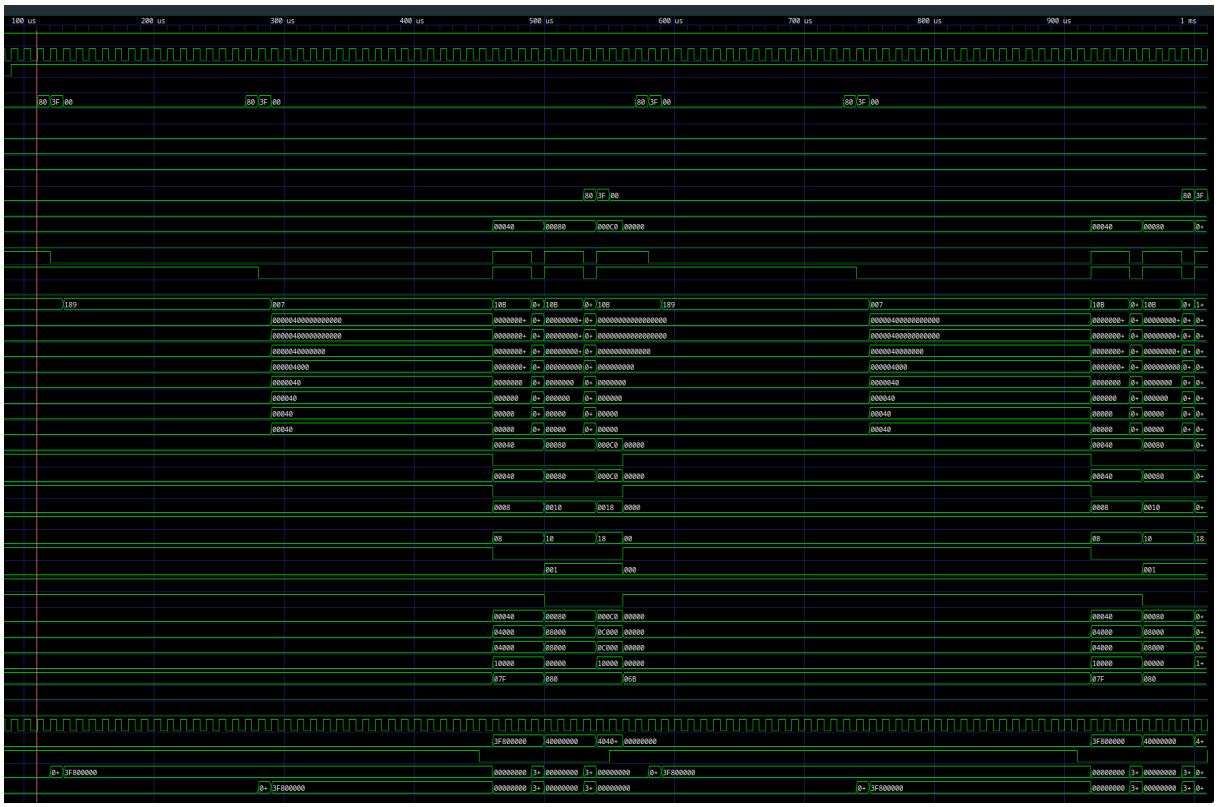


Figure 98.1: Waveform screenshot

External hardware

No external hardware is required.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	in_byte[0]	out_byte[0]	unused
1	in_byte[1]	out_byte[1]	unused
2	in_byte[2]	out_byte[2]	unused
3	in_byte[3]	out_byte[3]	unused
4	in_byte[4]	out_byte[4]	unused
5	in_byte[5]	out_byte[5]	unused
6	in_byte[6]	out_byte[6]	unused
7	in_byte[7]	out_byte[7]	unused

SnakeGame

by **stacu**

`0102` `25.175 MHz` `HDL Project`

`github.com/StaCu/ttihp-snake-game`

"Game of Snake"

TT Snake Game

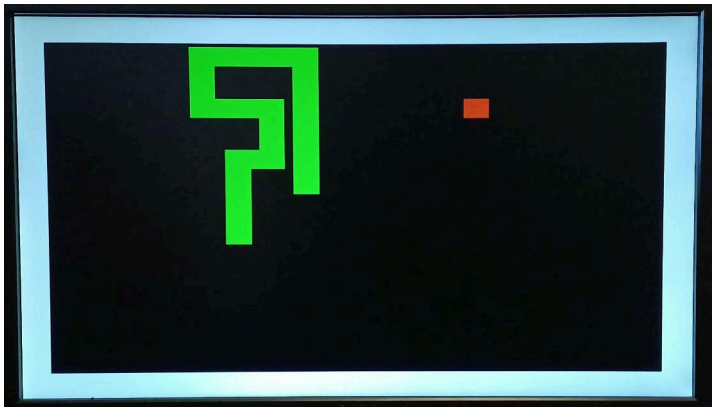


Figure 102.1: Snake Game

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

Spongent-88 Hash Accelerator

by **Stefan Aeschbacher**

`0106` `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]	—

raybox-zero TTIHP0p4 edition

by **algofoogle (Anton Maurovic)**

`0235` `25.175 MHz` `HDL Project`

`github.com/algofoogle/ttihp0p4-raybox-zero`

"TTIHP0p4 v1.8-dev submission of 'simple VGA ray caster game demo'"

Figure 235.1: TTGF0p2 raybox-zero showing 3D views including textures and doors

How it works

This is an experimental GF180 (gf180mcuD Open PDK) updated submission of ttcad25a-raybox-zero (an updated version of tt07-raybox-zero, from the raybox-zero project).

This project is a framebuffer-less VGA display generator (i.e. it is 'racing the beam') that produces a simple implementation of a "3D"-like ray casting game engine... just the graphics part of it. It is inspired by Wolfenstein 3D, using a map that is a grid of wall blocks, with basic texture mapping.

This version features textured walls (internally-generated or from off-chip QSPI memory), optional doors (sliding panels), flat-coloured floor and ceiling, and a variety of other rendering "hacks" for other simple visual effects. No sprites yet, sorry. Maybe that will come in a future version.

The 'player' POV ("point of view") registers allow the player position, facing X/Y vector, and viewplane X/Y vector in one go, and (along with other visual effects registers) are controlled by a single SPI interface.

NOTE: To optimise the design and make it work without a framebuffer, this renders what is effectively a portrait view, rotated. A portrait monitor (i.e. one rotated 90 degrees anti-clockwise) will display this like the conventional first-person shooter view, but it could still be used in a conventional landscape orientation if you imagine it is for a game where you have a first-person perspective of a flat 2D platformer, endless runner, “Descent-style” game, whatever.

TBC. Please contact me if you want to know something in particular and I’ll put it into the documentation!

How to test

TBC. Please contact me if you want to know something in particular and I’ll put it into the documentation!

Supply a clock in the range of 21-31.5MHz; 25.175MHz is ideal because this is meant to be “standard” VGA 640x480@59.94Hz, and note that the design may not be stable above that.

Start with `gen_texb` set low, to use internally-generated textures. You can optionally attach an external QSPI memory (`tex_...`) for texture data instead, and then set `gen_texb` high to use it.

`tex_pmod_type` should be set to 0 when using Leo Moser’s Tiny QSPI PMOD, or 1 for a Digilent QSPI PMOD.

Ideally the `reg` input should be high to make the VGA outputs registered. Otherwise, they are just as they come out of internal combo logic, which may not always meet timing (and hence might be unstable). I’ve done it this way so I can test the difference (if any).

`debug` can be asserted to show current state of POV (point-of-view) registers, which might come in handy when trying to debug SPI writes.

`inc_px` and `inc_py` can be set high to continuously increment their respective player X/Y position register. Normally the registers should be updated via SPI, but this allows someone to at least see a demo in action without having to implement the SPI host controller. NOTE: Using either of these will suspend POV updates via SPI.

Unlike the TT07 version, this one combines the two separate SPI peripheral interfaces into one, allowing both the POV and other registers to be updated from the same interface.

External hardware

Tiny VGA PMOD on dedicated outputs (`uo`).

Optional SPI controllers to drive `ui_in[2:0]`.

Optional external SPI ROM for textures.

TBC. Please contact me if you want to know something in particular and I'll put it into the documentation!

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	spi_sck	red[1]	Out: digilent_tex_csb / Out: moser_tex_csb
1	spi_sdi	green[1]	I/O: digilent_tex_io0 / I/O: moser_tex_io0
2	spi_csb	blue[1]	In: digilent_tex_io1 / In: moser_tex_io1
3	debug	vsync_n	Out: digilent_tex_sclk / Out: moser_tex_sclk
4	inc_px	red[0]	In: SPARE / In: moser_tex_io2
5	inc_py	green[0]	In: gen_texb / In: moser_tex_io3
6	reg	blue[0]	In: digilent_tex_io2 / N/A: moser_CS1
7	tex_pmod_type	hsync_n	In: digilent_tex_io3 / N/A: moser_CS2

VGA Screensaver with Tiny Tapeout Logo

by **Uri Shaked**

256 | 25.175 MHz | HDL Project

`github.com/TinyTapeout/tt-logo-screensaver`

"Tiny Tapeout Logo bouncing around the screen (640x480, TinyVGA Pmod)"

How it works

Displays a bouncing Tiny Tapeout logo on the screen, with animated color gradient.

Figure 256.1: Tiny Tapeout screensaver

How to test

Connect to a VGA monitor. Set the following inputs to configure the design:

- `tile` (ui_in[0]) to repeat the logo and tile it across the screen,
- `solid_color` (ui_in[1]) to use a solid color instead of an animated gradient.

If you have a Gamepad Pmod connected, you can also use the following controls:

- Start button: start/pause bouncing

- Left/right/up/down: change the bouncing direction (if bouncing) or move the logo around the screen (if paused)

External hardware

- Tiny VGA Pmod
- Optional: Gamepad Pmod

Project Pinout

Digital Pins

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

Linear Timecode (LTC) generator with I2C control

by **Thomas Flummer**

258 | 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

Orion Iron Ion [TT08 demo competition]

by **Toivo Henningsson**

0260 50.4 MHz HDL Project

github.com/MichaelBell/ttihp-orion-iron-ion

"My contribution to the TT10 demo competition"

Intro

Curly / Medieval presents

my contribution to the Tiny Tapeout 10 demo competition (which unfortunately got cancelled due the Efabless shutdown). Code, graphics, and music by Curly (Toivo Henningsson) of Medieval.

The demo was originally written for the Sky130 process (see the sky130 branch), but was updated to the IHP process before submission.

The demo can be seen at https://youtu.be/VCQJCVPyYjU (captured from a Verilator simulation).

How it works

The demo code contains a few different parts, mainly:

- Logo
- Star field
- Floating point unit; used for the twister and spiralling balls
- Synthesizer and sequencer

The video output is produced in VGA mode 640x480 @60 Hz. The logic is clocked at 50.4 MHz, giving two clock cycles per pixel.

Logo

Just like in https://github.com/toivoh/tt08-demo, the logo is made of big pixels (32x32 this time vs 16x16 before), where each big pixel is split along the diagonal into two triangles. The title is chosen so that the logo should hopefully compress well (and also alludes to the space theme and the echo feeling in the music). The dimensions of the characters are chosen so that

patterns should repeat at a scale of 4 pixels, to try to make them more visible to the tool's logic minimizer.

Star field

The star field keeps track of the current x and y coordinates with two extra subpixel bits of precision, increasing x or y by up to 4 units in each step. This is to allow the antialiasing effect where stars can move a fraction of a pixel per frame; since the output is RGB222, 2 subpixel position bits are enough. The size of the step to take to the next x or y value is taken from a coarse table that is made so that the resulting x and y curve should be similar to turning in space. The table entries have higher precision, so that they can cause a dithering between e g increments of 2 and 3 or 3 and 4.

The distribution of stars is computed using a pseudorandom function that depends on the current x and y value. This is inspired by the pseudorandom function used in the synth (see below). The algorithm can be described as

```
{r0, dr} = bitshuffle({jx, jy}, pattern1)
r = r0
for i=1:2
    r = bitshuffle(r, pattern2) + dr
    r = bitshuffle(r, pattern3)

intensity = r[5:4]
if intensity == 0: intensity = 1
if r[3:0] != 0: intensity = 0
```

where `ix, iy` are input bits for the current position where we should determine if there is a star, and its brightness. `bitshuffle(x, pattern)` permutes the bits of `x` using the fixed `pattern`. This can be done with only wires, so should be cheap. The combination of addition and bit shuffling means that the effects of input bits propagate in a quite unpredictable way to affect the output bits, creating a pseudorandom behavior. To find good star patterns, I simulated random bit shuffles with a script until I found a pattern that I liked.

The input to the random algorithm is not the full pixel positions `ix` and `iy`, but `jx = ix>>2` and `jy = iy>>1`. Only every fourth x pixel position and every second y pixel position can hold a star, with black in between. This allows the random algorithm time to converge before the star's intensity is needed.

A typical way to do the antialiasing of the stars would be to calculate numbers `px` and `py` that described how well the star aligns with the current x and y position, and use `px*py` to modulate the intensity. To save on logic, `min(px, py)` is used instead. A small lookup table is then used to calculate the 2 bit pixel value from `min(px, py)` and the star's intensity.

Floating point unit - twister and spiralling balls effects

The demo contains a small floating point unit for approximate floating point calculations, with 5 exponent bits and 11 mantissa bits. The FPU implements addition and subtraction in a similar way to most FPUs. It also supports approximate operations for multiplication and square root (approximate division could be supported as well, but wasn't needed). The approximate operations assume that the concatenation of exponent and mantissa bits are a logarithmic representation of the floating point number: multiply adds `{exponent, mantissa}`, while square root shifts it right by one. The result is quite inaccurate, but good enough for the computations needed, and contributes some interesting jaggedness to the twister. These cheap approximations of multiply and square root were the motivation to use an FPU in the first place.

Conversion from fixed point to floating point is done by creating a floating point number with a fixed exponent and placing the fixed point number (with its sign bit inverted) in the mantissa, which produces a floating point number representation of `fixed_point_number+bias`. The bias is then subtracted. To convert in the other direction, the bias is added to a floating point number, and the fixed point result is read out from the mantissa.

The FPU code uses the approximation `cos(0.5*pi*t) = 1 - t^2, abs(t) <= 1`. There didn't seem to be a point in making a more accurate representation given the inaccuracy of the multiplication.

The ALU has a single accumulator register and 5 general purpose registers. One of the inputs to the ALU is always the accumulator, while the other can come from a register, constant, or time varying fixed point value from the outside. The result is always written to the accumulator, and the value in the accumulator can then be written to one of the other registers.

Twister and spiral of balls effects

These are implemented by running a short FPU program during horizontal blanking before each scan line starts (different programs for different effects). The program computes up to five x positions for the scan line, which are used to draw horizontal spans of light and dark blue. The x positions share space with the mantissas in the FPU registers. The programs are written in such a way that they use fewer and fewer FPU registers as they are being overwritten with x positions.

Breaking down FPU instructions into cycles

To save area, each FPU instruction is broken into up to 3 single-cycle micro-operations:

- Add/sub:
  - Determine which argument has the largest magnitude

    - Add/Subtract
    - Normalize, calculating the correct exponent and shifting the mantissa to the right position
- Multiply:
    - Calculate carry out from mantissa sum
    - Add `{exponent, mantissa}`
- Single cycle instructions:
    - Load
    - Square root

The FPU code is stored as instructions rather than micro-operations, which should save some area. One instruction is executed every 4 cycles, which lets the program code be indexed by a running timer.

The logic that represents the program ROM for the FPU has quite high latency. A multicycle timing constraint is used to allow data paths that go through the program ROM to take two cycles. This means that no micro-operation is executed until the second cycle of running each instruction. (The multicycle constraint turned out not to be needed for the IHP version, and was removed.)

Synthesizer

The synthesizer produces output samples at 63 kHz, 10 bit resolution. This gives it 800 cycles (half a scan line) per sample, and the usable ouput range of PWM values is 0 - 800. One voice sample is calculated in 64 cycles, which gives time to calculate 12 voices at the same time, plus a little time to update for the next sample. On average, the voices need to have a peak amplitude <= 64 steps to fit into the output range; one step per cycle.

The voices are used as follows:

- 4x2 melody/harmony voices: 4 channels with 2 voices per channel with detuning to get a fatter sound
    - The frequency is slightly higher for one of the voices in each pair than the other
- Prenoise: the pedal tone with rhythmically changing timbre that runs throughout most of the demo
- Bass drum
- Hihat
- Visualization voice - not heard, used to produce the visible waveforms on screen
    - Can calculate two waveforms per scan line

Aliasing considerations

The waveforms are designed to keep aliasing artifacts relatively low:

- The melody/harmony waveforms use piecewise linear sections without a too steep slope, and avoid slopes of less than one unit per sample, which keeps aliasing down.
- The bass drum has a similar approach, using a clipped triangle wave that gradually sinks in frequency.
- The prenoise waveform is kept at a power of 2 frequency, since it would be hard to antialias. The music has been written around this limitation, with the prenoise as a pedal tone/ostinato.
  - Initially, the pedal tone is the tonic note.
  - After the music modulates down by a fifth towards the end, the pedal tone is now the fifth instead.
- The hihat is pure noise and doesn't need any antialiasing considerations.

To gradually reduce the volume of each voice, it is clamped to a decreasing maximum amplitude. This simple method changes the waveform as the volume reduces, but keeps the slopes in their original range. If the volume had been reduced by multiplication, increasing aliasing artifacts would results as the effective range gets reduced.

ALU

The synthesizer is based around a small ALU, with a small set of registers

- 11 bit accumulator
- 10 bit output accumulator
- 10 bit output register
- 23 bit oscillator divided into low and high 11 bit parts plus top bit
- two flag bits (predicates)

The oscillator is used to calculate the phase of the waveforms, keep track of time in the demo, index the notes for the music and to know which frame to display.

Calculating voice phases from the shared oscillator

There are no registers to keep track of the phases of different synth voices. Instead, for each sample of a voice that needs to be computed, the first 30 cycles are used to compute `phase = (freq * osc) >> n` to produce an 11 bit phase in the accumulator. The bits above 11 are truncateed, since the waveform repeats after one phase. `freq` varies between 256 and 511 to choose a note, while `n` selects the octave.

The product `phase = (freq * osc) >> n` is calculated using shifts and adds (at most one of each per cycle), discarding low order bits when they are not needed anymore, and high order bits that will not be needed. To illustrate the method, say that we want to calculate bits `10:3` of the product of an 8 bit and an 11 bit number. The calculation can be visualized as follows:

```
             ********
            ********
           ********
          ********
         ?*******|
        ??******  |
       ???*****   |
      ????****    |
     ?????***     |
    ??????**      |
 + ???????*       |
--------------------
= ????????********???
```

We have 11 product terms to add up, each with 8 bits.

- We proceed from the smallest term, adding up terms.
- In the first phase, the bottom bit in the current sum is not needed in the final output, so it can be dropped since no later term will change it (they are all shifted further to the left).
- In the second phase, we are out of bottom bits to ignore, and we can instead start to ignore top bits in the terms, since they are above the range of bits needed in the result.
- By changing the number of steps in the first phase, we can change the shift amount `n`.

This way, we can use the same number of bits to store each intermediate result as is needed to store the final result. The implementation proceeds by shifting the intermediate result right by one for each step, switching to rotate right when coming to the second phase (to preserve the bits that are rotated out). When all relevant terms have been added, the result will have been rotated back to the correct position.

Evaluating waveforms using a single accumulator as intermediate storage

The synth ALU has only a single intermediate register to work with, the accumulator (to save area). To evaluate a piecewise linear function (which the melodic/harmonic waveforms are made of), it first evaluates one or several conditions on the current accumulator value to know which piece of the piecewise linear function that it should evaluate, storing the results in the predicates. Then, it can use the predicate values to choose how to transform the accumulator.

Melody/harmony voices

There are two waveforms used for the melody/harmony voices: saw like and pulse like. An ideal sawtooth wave includes a sudden jump every period, and an ideal pulse wave contains two. To simulate a gradually closing lowpass

filter, these jumps have been changed to ramps. The slope of the ramps is gradually decreased as a note ages.

The pulse like waveform is also uses pulse width modulation by a triangle wave, which is added to the intermediate phase after it itself has been made into a triangle wave as a step in the waveform computation.

Bass drum

The bass drum approximates a clamped triangle wave with exponentially decreasing frequency. This was a bit challenging to implement, since there is no register to store the bass drum's phase between samples, instead it has to be recalculated at each sample from the linearly increasing oscillator.

The bass drum uses a variation on the multiplication algorithm, takeing the lower bits of the oscillator and calculating an approximate square. Let `x` be the relevant oscillator bits in fixed point. As `x` goes from 0 to one, the function

```
y = (2-x)^2 mod 1
```

wraps around 3 times, with the slope at the end being half of the slope at the beginning. Several such quadratic sections are shifted into decreasing octaves to give an approximation of an exponentially decaying frequency. The square approximation is calculated using a variation of the multiplication algorithm described above. It uses only 8 slopes per octave, which seems to be barely enough to give the impression of a continuously descending pitch, but allows the algorithm to use the top 11 bits of the oscillator as one of its inputs.

The bass drum phase is used to evaluate a triangle wave, which is clamped to gradually reduce its volume.

Hihat

The hihat is created by noise clamped to a decreasing amplitude. This kind of noise is traditionally created with a Linear Feedback Shift Register (LFSR), but that would have required space for registers to hold the LFSR state. Instead, a new noise value is calculated for each sample based on the oscillator, using the algorithm

```
acc = osc_low
for i=0:3
    acc = bitshuffle(acc, pattern) + osc_high
    acc = acc + (acc >> 1)
```

The `pattern` used for shuffling is fixed, all it needs is an 11-bit wide multiplexer. After four iterations, the result sounds like noise.

Prenoise pedal tone

The prenoise waveform used as a pedal tone is computed using a simplification of the noise algorithm above, with only one iteration:

```
acc = <selected bits from osc>
acc = bitshuffle(acc, pattern)
acc = acc + rotate_right(acc, 1)  # could use acc *= 3 instead
```

The permutation used in the `bitshuffle` step is the same as in the noise case above, and has been chosen for the sonic results it produces in the prenoise case. The result is a waveform with a power of 2 period that changes timbre in a rhythmic manner.

Visualization voice

The visualization voice can evaluate the waveform from any of the other voices, using the current scanline's y position instead of the oscillator, to keep the waveform steady from frame to frame. Two waveforms can be evaluated per scan line, one to be shown on the left side of the screen and one on the right side. The synth is synchronized with the display output so that a new visualization waveform sample is computed in the middle of each scan line and one in hblank. There are 3 registers to store the evaluated waveforms, to keep track of two waveforms and have access to the value from the previous scanline when a waveform is displayed.

Sequencer

The notes to play are taken from logic that represents a note ROM, and it has quite high latency. Out of the 64 cycles used to compute each voice sample, the first cycle is used to wait for the output from the note ROM to stabilize, and the synthesizer doesn't do anything with the note data until the second cycle. This is accomplished with a multicycle timing constraint. (The multicycle constraint turned out not to be needed for the IHP version, and was removed.)

The note ROM contains data for 4 channels. For each channel and time position `pos`, it outputs an enable flag, a note value (note and octave), and an age value `t0`. The actual age is computed as `t = t0 + pos` (with wraparound). This allows `t0` to be piecewise constant in the note ROM. The age `t` is filled out with low order bits from the oscillator, and used to modulate the waveform and volume of notes as they age. Weaker notes can be achieved by starting them at a higher age.

How to test

Plug in a TinyVGA compatible Pmod on the TT08 demo board's out Pmod. Plug in a Pmod compatible with Mike's audio Pmod on the TT08 demo board's bidir Pmod. Set all inputs to zero to get the default behavior. The demo starts directly after reset.

External hardware

This project needs

- a TinyVGA VGA Pmod.
- Mike's audio Pmod.

Project Pinout

Digital Pins

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

Pattern-Guided Arithmetic Optimizations with MLIR kulisch bf16

by **L. Ledoux**

261 500 kHz HDL Project

`github.com/Bynaryman/ttihp04`

"Streaming byte-fed S3FDP seq+comb accumulator generated from MLIR for-loop."

How it works

`tt_um_lledoux_bf16_diminished_kulisch` wraps a generated seq+comb arithmetic core built from MLIR with Emeraude + CIRCT.

The flow detects a loop pattern in MLIR (`scf.for` + `memref.load/store` + `arith.mulf/addf`) and emits a specialized S3FDP accumulator instead of generic floating-point datapath logic.

Canonical loop shape:

```
scf.for %k = %c0 to %c2 step %c1 {
  %x = memref.load %a[%k] : memref<2xf32>
  %y = memref.load %b[%k] : memref<2xf32>
  %acc = memref.load %c[%c0] : memref<2xf32>
  %m = arith.mulf %x, %y : f32
  %s = arith.addf %acc, %m : f32
  memref.store %s, %c[%c0] : memref<2xf32>
}
```

Arithmetic model (S3FDP)

S3FDP is used as a truncated Kulisch-like fixed-point accumulation strategy:

- multiply input terms,
- accumulate in a constrained fixed-point-like internal format,
- convert back to `f32`.

Specialization used in this project:

- `ovf=2`
- `msb=4`
- `lsb=-6`
- `chunk_size=16`

Llama / PyTorch source example

The source pattern used in experiments:

```python
class LlamaFfnSublayer(nn.Module):
    """Llama FFN sublayer using SwiGLU (SiLU-gated linear unit)."""

    def __init__(self, dim: int = 512, hidden_dim: int | None =
None, multiple_of: int = 256):
        super().__init__()
        if hidden_dim is None:
            hidden_dim = 4 * dim
            hidden_dim = int(2 * hidden_dim / 3)
            hidden_dim = multiple_of * ((hidden_dim + multiple_of -
1) // multiple_of)
        self.w_gate = nn.Linear(dim, hidden_dim, bias=False)
        self.w_up = nn.Linear(dim, hidden_dim, bias=False)
        self.w_down = nn.Linear(hidden_dim, dim, bias=False)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        gate = F.silu(self.w_gate(x))
        up = self.w_up(x)
        return self.w_down(gate * up)
```

High-level linalg excerpt:

```
...
%8 = linalg.generic
  {ins(%7: tensor<1x2x16xf32>) outs(%5: tensor<1x2x16xf32>) {
    %19 = arith.negf %in : f32
    %20 = math.exp %19 : f32
    %21 = arith.addf %20, %cst_1 : f32
    %22 = arith.divf %cst_1, %21 : f32
    linalg.yield %22 : f32
  }} -> tensor<1x2x16xf32>

%9 = linalg.generic
  {ins(%8, %7: tensor<1x2x16xf32>, tensor<1x2x16xf32>)
   outs(%5: tensor<1x2x16xf32>)} {
    %19 = arith.mulf %in, %in_7 : f32
    linalg.yield %19 : f32
  } -> tensor<1x2x16xf32>
...
```

Internal IR snapshots

Comb-specialized stage (`generated/ir-stages/20-flopoco-comb.mlir`):

```
%s3fdp_accum.r = hw.instance "s3fdp_accum"
@s3fdp_accum_core_wE8_wF23_cs16(
  clk: %3: !seq.clock, reset: %arg1: i1, x: %1: i32, y: %2: i32
) -> (r: i32)
```

Seq/HW aggregate stage (`generated/ir-stages/60-hw-aggregate.mlir`):

```
%27 = seq.clock_gate %26, %3
%s3fdp_accum.r = hw.instance "s3fdp_accum"
@s3fdp_accum_core_wE8_wF23_cs16(
  clk: %27: !seq.clock, reset: %reset: i1, x: %18: i32, y: %25: i32
) -> (r: i32)
seq.write %c_mem[%false] %s3fdp_accum.r wren %3 {latency = 1 :
i64} : !seq.hlmem<2xi32>
```

SV-lowered stage (`generated/ir-stages/90-hw-to-sv.mlir`):

```
%c_mem = sv.reg : !hw.inout<uarray<2xi32>>
sv.alwaysff(posedge %clk_0) {
  sv.if %4 {
    %33 = sv.array_index_inout %c_mem[%false] : !
hw.inout<uarray<2xi32>>, i1
    sv.passign %33, %s3fdp_accum.r : i32
  }
}
```

Interface protocol

Input stream on `ui_in[7:0]`:

- 20-byte frame, little-endian
- `a[0..1]` IEEE754 `f32` (8 bytes)
- `b[0..1]` IEEE754 `f32` (8 bytes)
- `c0` IEEE754 `f32` seed (4 bytes)

Execution:

- hold core reset during load,
- release reset after byte 20,
- wait 3 cycles,
- output one 32-bit result on `uo_out[7:0]` as 4 little-endian bytes.

Frame slot: 27 cycles (`20 load + 3 run + 4 output`).

`uio` pins are unused.

Generation and test

Generate core + IR stages:

```
./scripts/generate_s3fdp_core.sh
```

Run simulation:

```
cd test
make clean
make -B
```

Waveform screenshot:

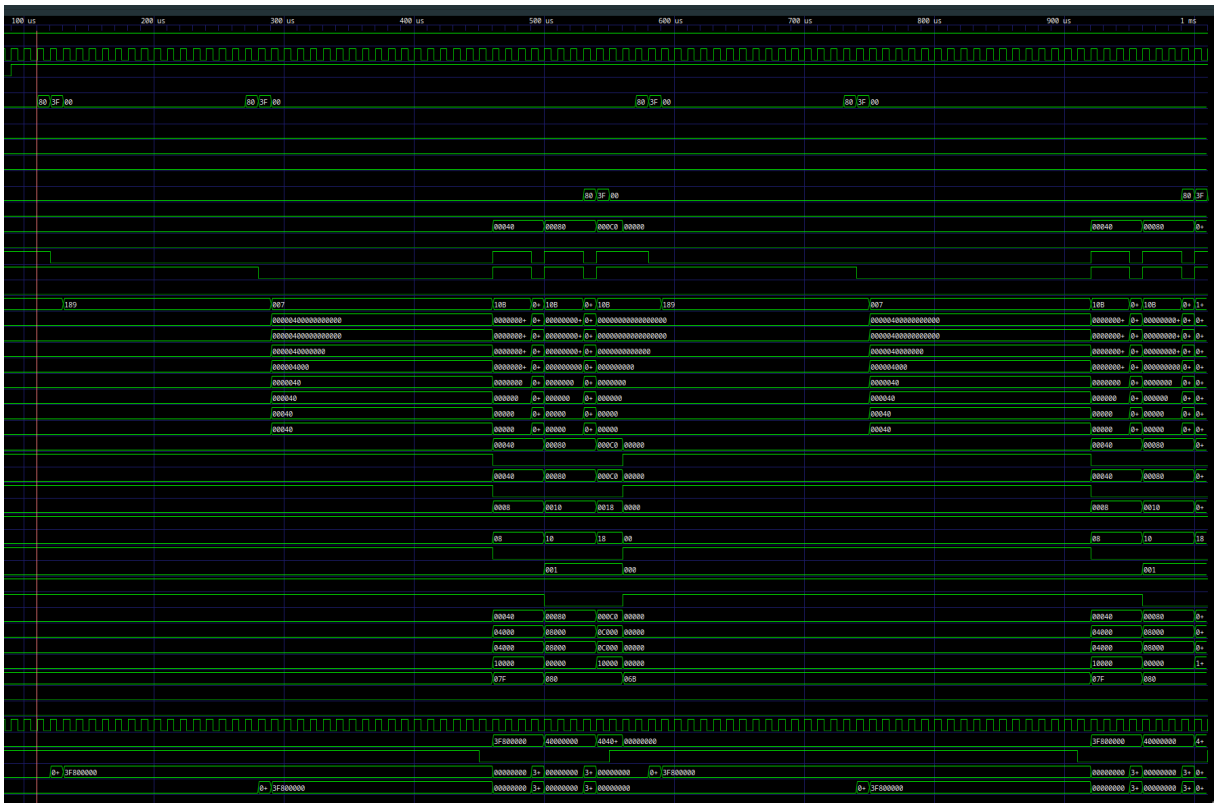


Figure 261.1: Waveform screenshot

External hardware

No external hardware is required.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	in_byte[0]	out_byte[0]	unused
1	in_byte[1]	out_byte[1]	unused
2	in_byte[2]	out_byte[2]	unused
3	in_byte[3]	out_byte[3]	unused
4	in_byte[4]	out_byte[4]	unused
5	in_byte[5]	out_byte[5]	unused
6	in_byte[6]	out_byte[6]	unused
7	in_byte[7]	out_byte[7]	unused

ROTFPGA v2

by **htfab**

0262 | 10 MHz | HDL Project

`github.com/htfab/ttihp0p4-rotfpga`

"A reconfigurable logic circuit made of identical rotatable tiles"

How it works

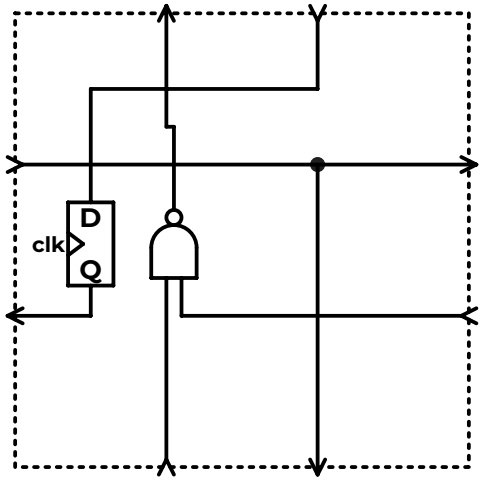


Figure 262.1: Logic tile

A reconfigurable logic circuit built from identical copies of the tile above containing a NAND gate, a D flip-flop and a buffer, with each tile individually rotated or reflected as described by the FPGA configuration. Port of the original ROTFPGA from Caravel to TinyTapeout.

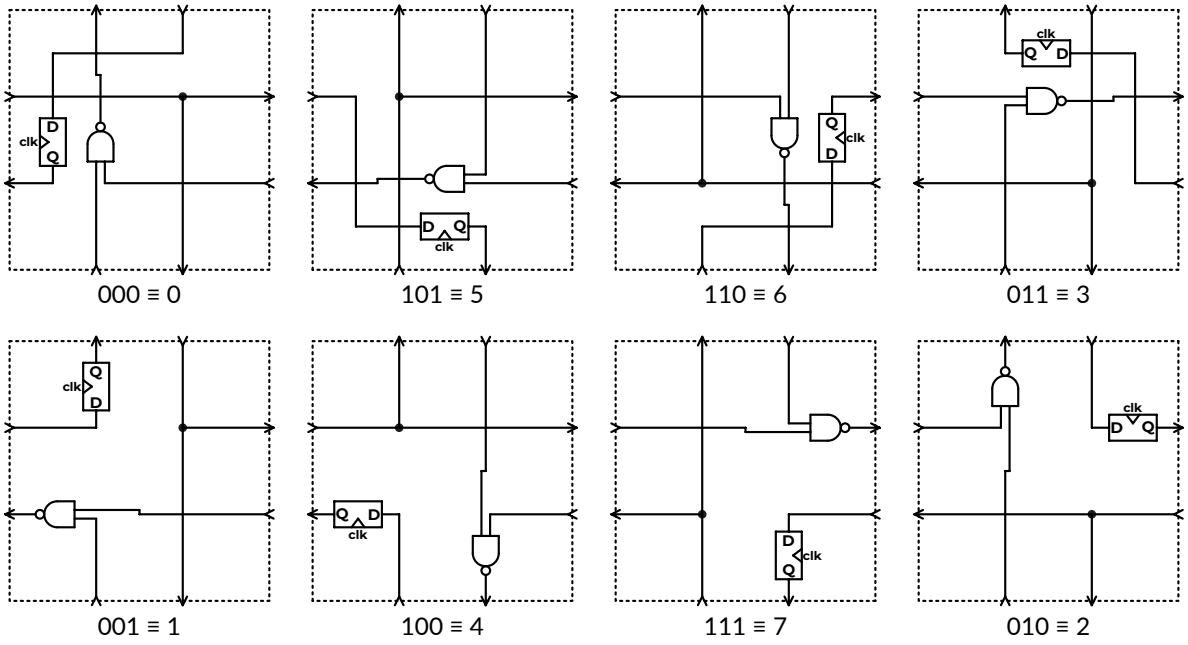
Porting the design required a 50-fold decrease in chip area which was achieved using a combination of cutting corners, heavy optimization and a few design changes. In particular:

- The FPGA was reduced from 24×24 to 8×8 tiles. There are 8 inputs and 8 outputs instead of 12 each.
- To compensate for smaller size, tiles can also be mirrored in addition to rotation.
- Tiles (being the most repeated part of the design) were rewritten as hand-optimized gate-level Verilog.
- Each tile only contains 1 flip-flop (the one exposed to the user). Configuration is now stored in latches.
- Configuration and reset are performed using a routing-efficient scan chain, so the design is no longer routing constrained. This allows standard cells to be placed with >80% density.

- Openlane and its components are 2 years more mature, hardening the same HDL more efficiently.

Configuration

Each tile can be configured in 8 possible orientations. Bits 0, 1 and 2 correspond to a diagonal, horizontal and vertical flip respectively. Any rotation or reflection can be described as a combination:



(The bottom row looks somewhat different, but we just rearranged the wires so that the inputs and outputs line up with the unmirrored tiles.)

Tiles are arranged in an 8×8 grid:

- Top, bottom, left and right inputs and outputs are connected to the tile in the respective direction.
- Tiles mostly wrap around, e.g. the bottom output of a cell in the last line connects to the top input of the cell in the first line.
- As an exception to the wrapping rules, left inputs in the first column correspond to chip inputs and right outputs in the last column correspond to chip outputs.
- There is a scan chain meandering through all the tiles, visiting lines from top to bottom and within each line going from left to right.

This is a 4×4 model of the tile grid, showing regular i/o as black and the scan chain as grey:

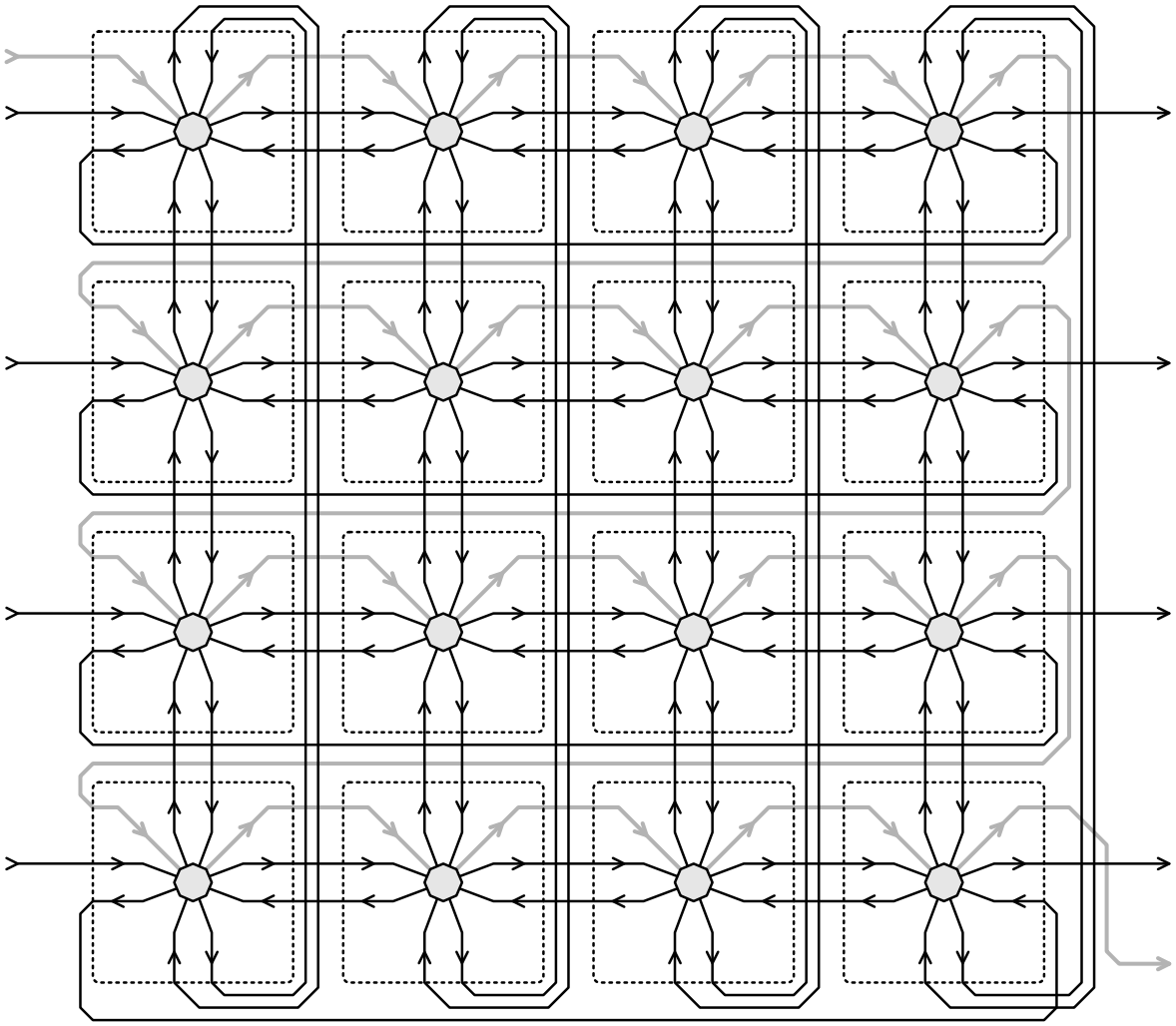


Figure 262.2: Grid model

When the *scan enable* input is 0, the FPGA operates normally and each tile sets its flip-flop to the input it receives from one of the neighboring tiles according to its current rotation/reflection. When *scan enable* is 1, it sets the flip-flop to the value received through the scan chain instead. This allows us to set the initial state of each flip-flop and also to query their state later for debugging. With some extra machinery it also allows us to change the rotations/reflections.

When the 2-bit *configuration* input is is 01, each cell updates its *vertical flip* bit to the current value of its flip-flop. Similarly, for 10 it sets the *horizontal flip* and for 11 it sets the *diagonal flip*. When *configuration* is 00, all three flip bits are latched and the orientation doesn't change.

One can thus configure the FPGA by sending the sequence of all *diagonal flip* bits through the scan chain, then setting *configuration* to 11 and back to 00, then sending all *horizontal flip* bits, setting *configuration* to 10 and back to 00, and finally sending the *vertical flip* bits and setting *configuration* to 01 and back to 00.

Note that in order to save space the flip bits are stored in latches, not registers. Changing the *configuration* input from 00 to 11 or vice versa can cause a race condition where it is temporarily 01 or 10, overwriting the horizontal or vertical flip bits. Therefore one should configure the diagonal flips first.

Loop breaker

The user design may intentionally or inadvertantly contain combinational loops such as ring oscillators. To help debug such designs, the chip has a loop breaker mechanism using a *loop breaker enable* input as well as a 2-bit *loop breaker class* input.

Tiles are assigned to loop breaker classes:

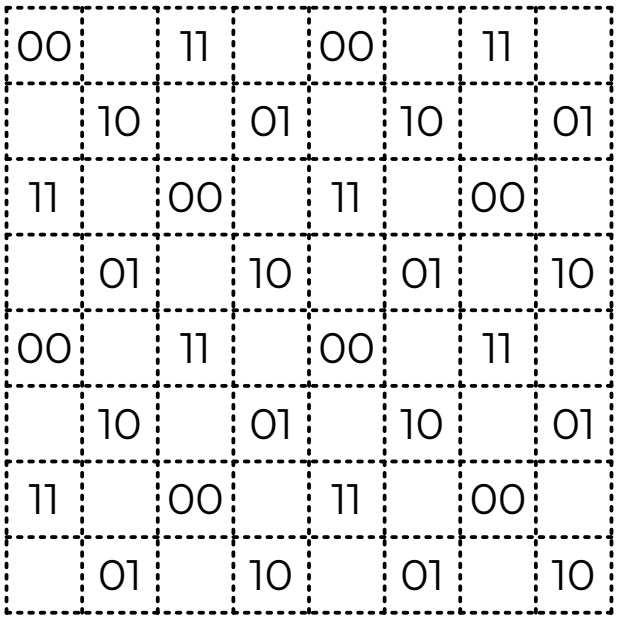


Figure 262.3: Loop breaker tile classes

The loop breaker latches a tile output if and only if the following conditions are all met:

- The *loop breaker enable* input is 1.
- The current tile has a non-empty class that is different from the *loop breaker class* input.
- The output doesn't come from the tile's flip-flop.

The loop breaker has the following properties:

- If *loop breaker enable* is 1 and *loop breaker class* is constant, there are no combinational loops running. If we also pause the clock, the circuit keeps a steady state.
- If *loop breaker enable* is 1 and we cycle *loop breaker class* through all possible values repeatedly while the clock is paused, everything will eventually propagate. If we also assume that the design has no race conditions, it will behave in the same way as if *loop breaker enable* was 0.

Reset

Setting the *active-low reset* input to 0 has the following effect:

- Override *scan enable* to 1, *scan chain* input to 0 and disengage the latches for vertical, horizontal and diagonal flips. When kept low for 64 clock cycles this will reset the state and configuration in every tile.
- Override *loop breaker enable* to 1 and *loop breaker class* to 00. This ensures that we play nice with other designs on TinyTapeout and keep a steady state while our design is not selected.

Pin mapping

Input pins:

- `clk` provides a clock signal for the flip-flops
- `rst_n` is the *active-low reset* described above
- `ui_in[7:0]` are passed to the leftmost column of tiles as inputs from the left

Output pins:

- `uo_out[7:0]` come from the rightwards output of the rightmost column of tiles

Bidirectional pins:

- `uio_in[0]` is the *scan enable* input
- `uio_in[1]` is the *scan chain* input
- `uio_in[3:2]` are the *configuration* input bits
- `uio_in[4]` is the *loop breaker enable* input
- `uio_in[6:5]` are the *loop breaker class* input bits
- `uio_out[7]` is the *scan chain* output

How to test

Follow the test suite the `test` directory. It sets up the FPGA with the following two configurations and runs a battery of tests on each.

Test configuration 1 used for upload, download, single-step and propagation tests:

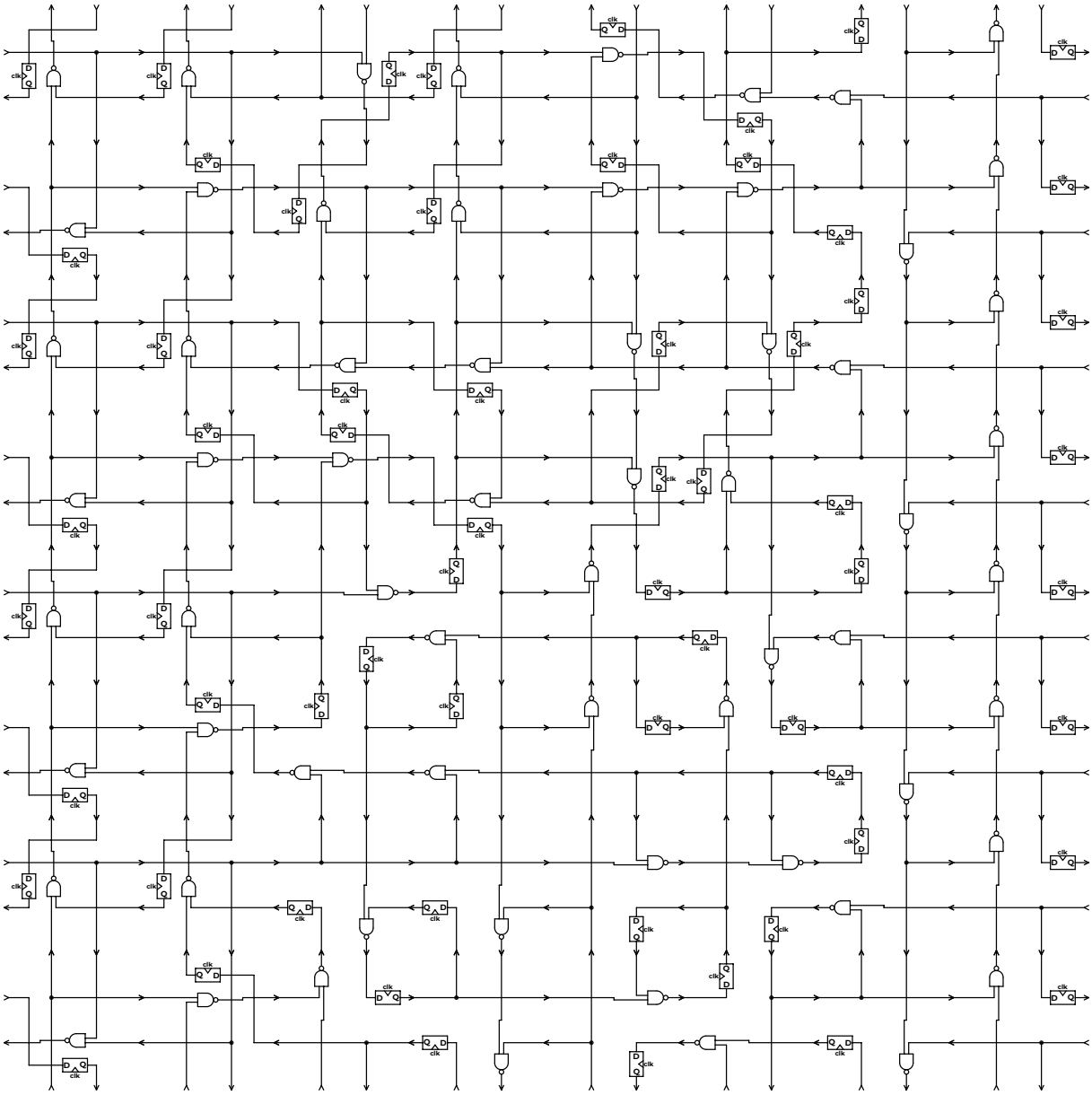


Figure 262.4: Diagram corresponding to `fpga_config` in `test.py`

Test configuration 2 used for testing the loop breaker with manual and automatic cycles:

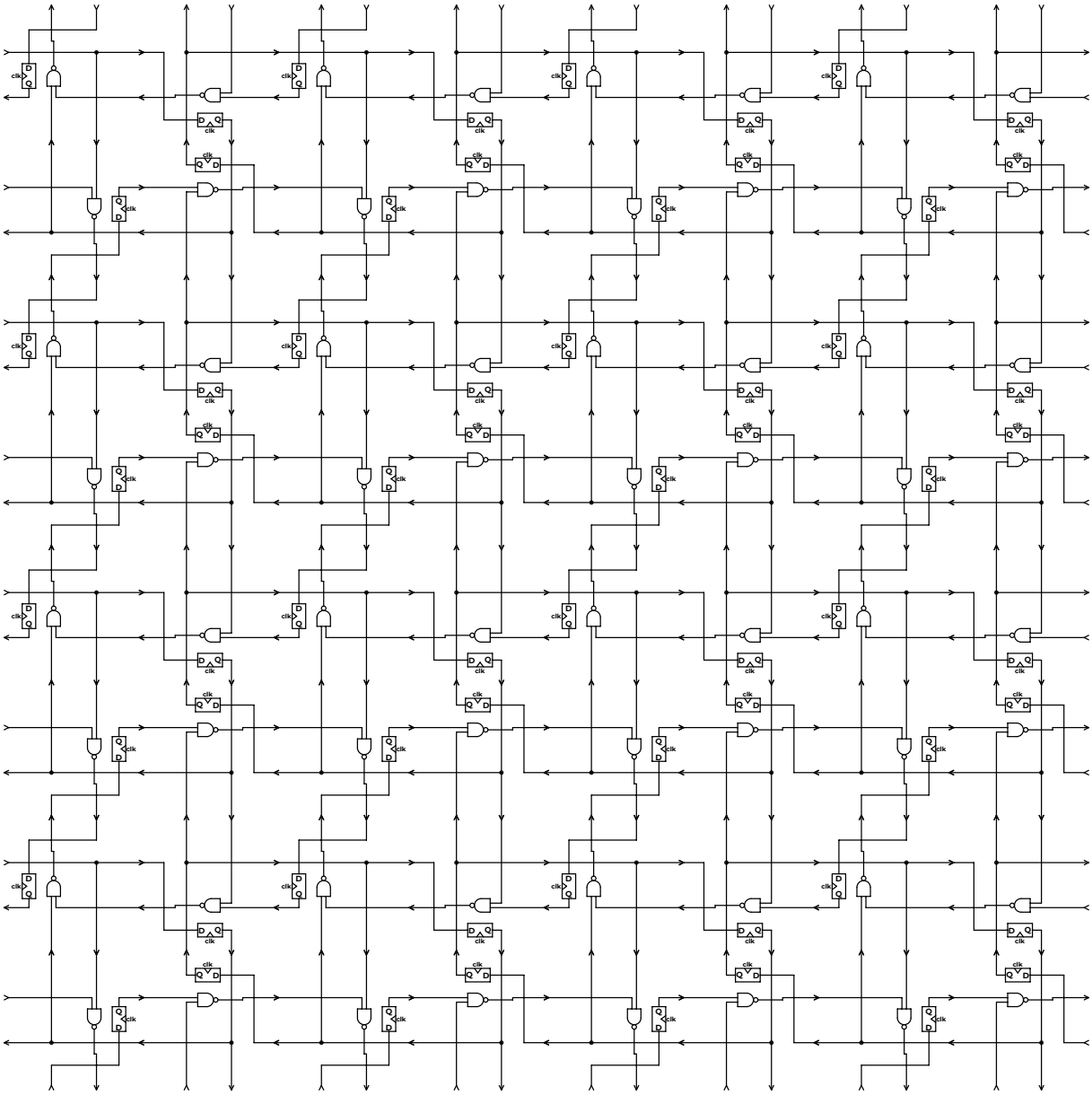


Figure 262.5: Diagram corresponding to `cfg` from the loop breaker test in `test.py`

External hardware

None

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	tile(0,0) left in	tile(7,0) right out	_scan enable_ input
1	tile(0,1) left in	tile(7,1) right out	_scan chain_ input
2	tile(0,2) left in	tile(7,2) right out	_configuration_ input bit 0

#	Input	Output	Bidirectional
3	tile(0,3) left in	tile(7,3) right out	_configuration_ input bit 1
4	tile(0,4) left in	tile(7,4) right out	_loop breaker enable_ input
5	tile(0,5) left in	tile(7,5) right out	_loop breaker class_ input bit 0
6	tile(0,6) left in	tile(7,6) right out	_loop breaker class_ input bit 1
7	tile(0,7) left in	tile(7,7) right out	_scan chain_ output

ihp_cmos51_prism

by **Ken Pettit**

264 64 MHz HDL Project

github.com/kdp1965/ihp-cmos51-prism-lite

"A programmable FSM"

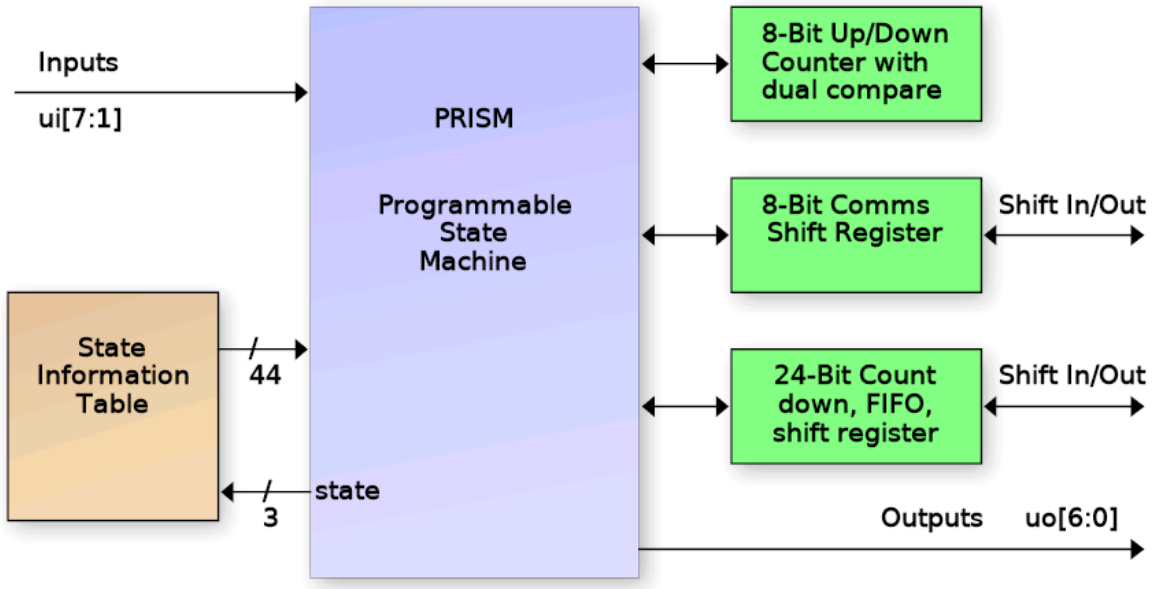
PRISM

Author: Ken Pettit

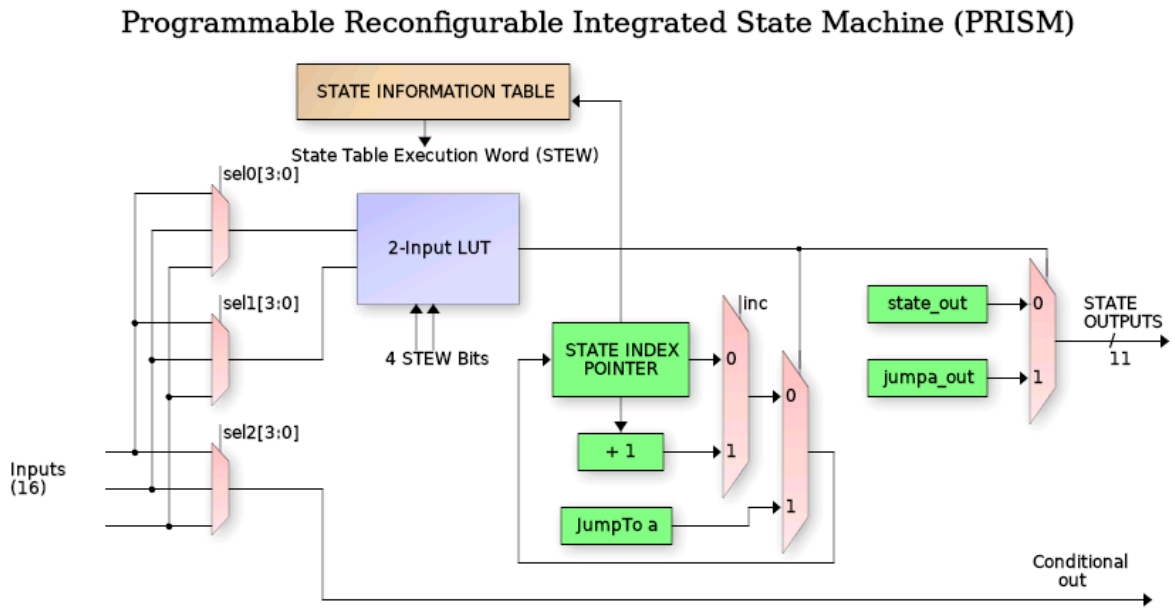
Peripheral index: 8

What it does

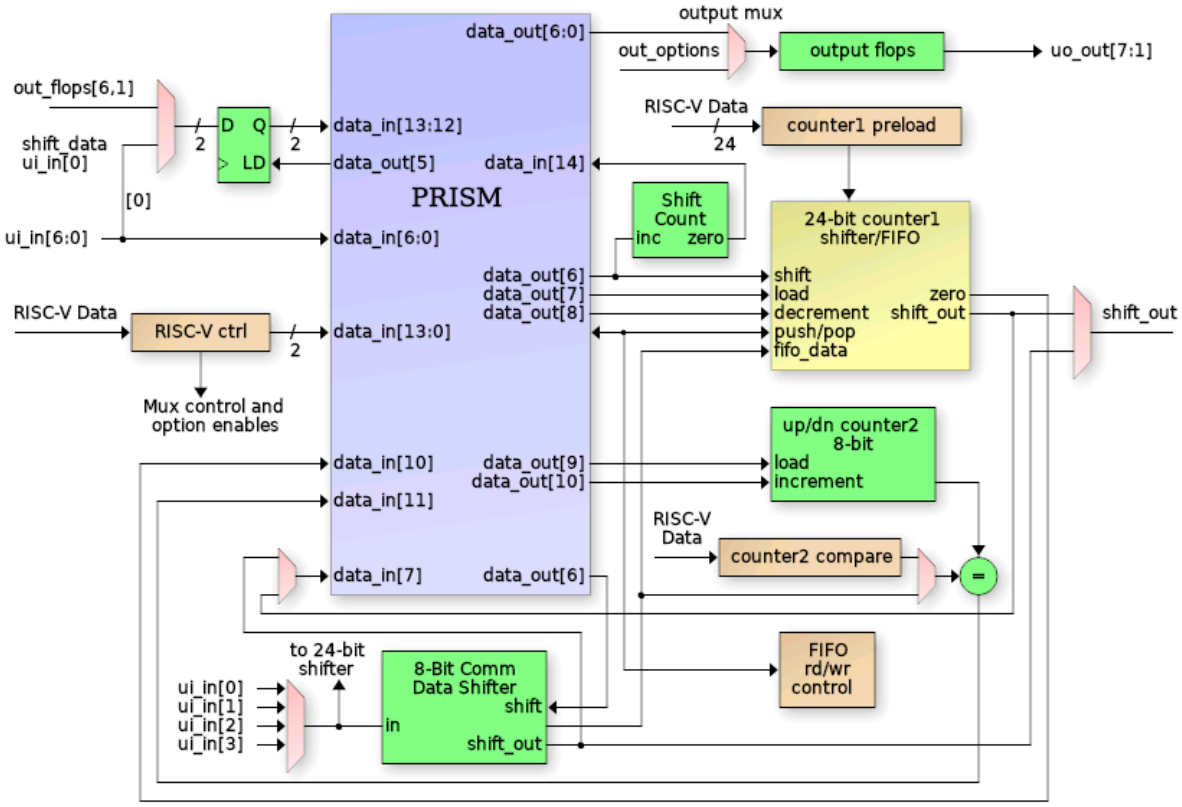
This is a Programmable Reconfigurable Indexed State Machine (PRISM) that executes a Verilog coded Mealy state machine loaded via a runtime loadable configuration bitstream generated by a custom branch of Yosys. PRISM includes it's own counter, shift register and compare sub-peripherals for controlling input/output data as well as timing operation.



The PRISM controller block is a programmable state machine that uses an N-bit (3 in this case) index register to track the current FSM state. That index is a pointer into the State Information Table (SIT) to request the State Execution Word (STEW).



What can it do?



Operating priciples of PRISM

PRISM supports FSM designs up to 8 states and includes controllable peripherals such as counters, communication shift register, FIFO and interrupt

support. It also features an integrated debugger with 2 breakpoints, single-stepping and readback of state information. Due to long combinatorial delays, PRISM operates from a divide-by-two clock (32Mhz max). The following is a block diagram of the PRISM controller:

Each state is encoded with a 44-bit (in the case of TinyQV PRISM) execution word that controls the FSM output values, input values and state transition decision tree for that state. The peripheral is operated by loading a "Chroma" (more on that below), or execution personality in the 352 bit configuration array and programming the 14-bit control word to set peripheral behaviors (like choosing 24-bit shift register vs. 24-bit counter, choosing pin locations for shift data in/out, etc.).

Once a chroma has been loaded, the control register programmed and the PRISM enabled, the FSM will start at state 0. Eight of the bits in the State Execution Word (STEW) specify which of 16 inputs get routed to the 2-input Look Up Table (LUT) that makes the decision for jumping to the specified state (stored in 3 bits of the STEW). While in any state, there is a set of 11 (from the STEW) output bits that drive the PRISM block outputs when the LUT output is zero (no jump) and 11 more that are output during a jump (transitional outputs).

Each state also has an independent 16-input mux (4-bits from STEW) driving a 1-input LUT to drive a "conditional output". This is an output who's value is not strictly depedent on the static values in the STEW for the current state, but rather depends on the selected input during that state.

State Looping (important)

In larger PRISM implementations, each state has "dual-compare" with two N-bit LUTs which allows jumping to one of two possible states. Due to size restrictions, this peripheral does not include dual-compare. Instead the PRISM implementation has (in each state's STEW), a single "increment state" bit.

In any state where the 'inc' bit is set and the LUT output is FALSE (i.e no jump), then the state will increment to the next state, and the "starting state" of the first occurance of this will be saved (i.e. start of loop). Then each successive state can test a different set of inputs to jump to different states. When a state is encountered with the 'inc' bit NOT set, PRISM will loop back to the "starting state" and loop through that set of states until the first TRUE from a LUT causing a jump, clearing the loop.

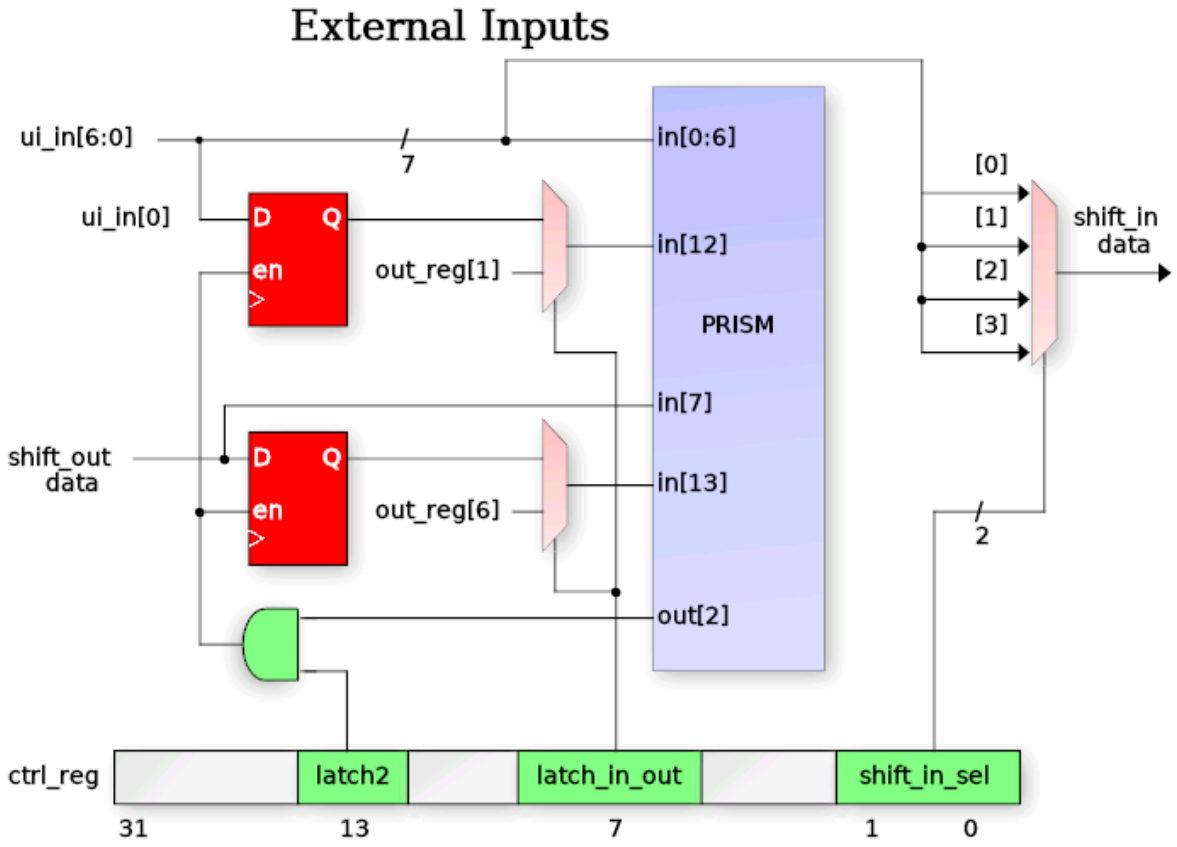
TL/DR

1. Load a Chroma defining the FSM and enable PRISM.
2. State starts at zero.
3. Each state chooses up to 3 of 16 inputs via config bits.
4. 2-input LUT decides if "jump to defined state" occurs.

5. Increment bit decides if “state looping” is in effect.
6. State looping ends when first LUT jump occurs.
7. Outputs bits from STEW for “non-jump” and “transitional jump”.
8. One conditional output based on single selected input per state.

External PRISM Inputs

Inputs to the PRISM engine come from the ui_in[6:0] pins of the TinyTapeout ASIC (pin 7 is not used as this is a UART pin for TinyQV). All 7 ui_in[6:0] pins are directly selectable by the running chroma for state transition or conditional output decisions.



In addition to direct input to the PRISM, a few inputs also have special functions (refer to the diagram below for visual). Input ui_in[0] can be programmatically latched by the chroma using PRISM out[2] when configured via the ctrl_reg latch2 and latch_in_out bits. The latched input version becomes available on PRISM input in[12]. This allows for detection of rising or falling edges.

Additionally any one of inputs in[3:0] can be used as shift register “shift_in_data” to feed the 8-bit and 24-bit shift registers. Selection of which pin of in[3:0] is made using ctrl_reg field “shift_in_sel”.

The diagram also shows that shift_out_data can be latched via the out[5] signal. It is to allow protocols like SPI to present serial output data on one

edge while shifting occurs on the opposite edge (by selecting in[13] as the controlling input for the cond_out[0] conditional output.

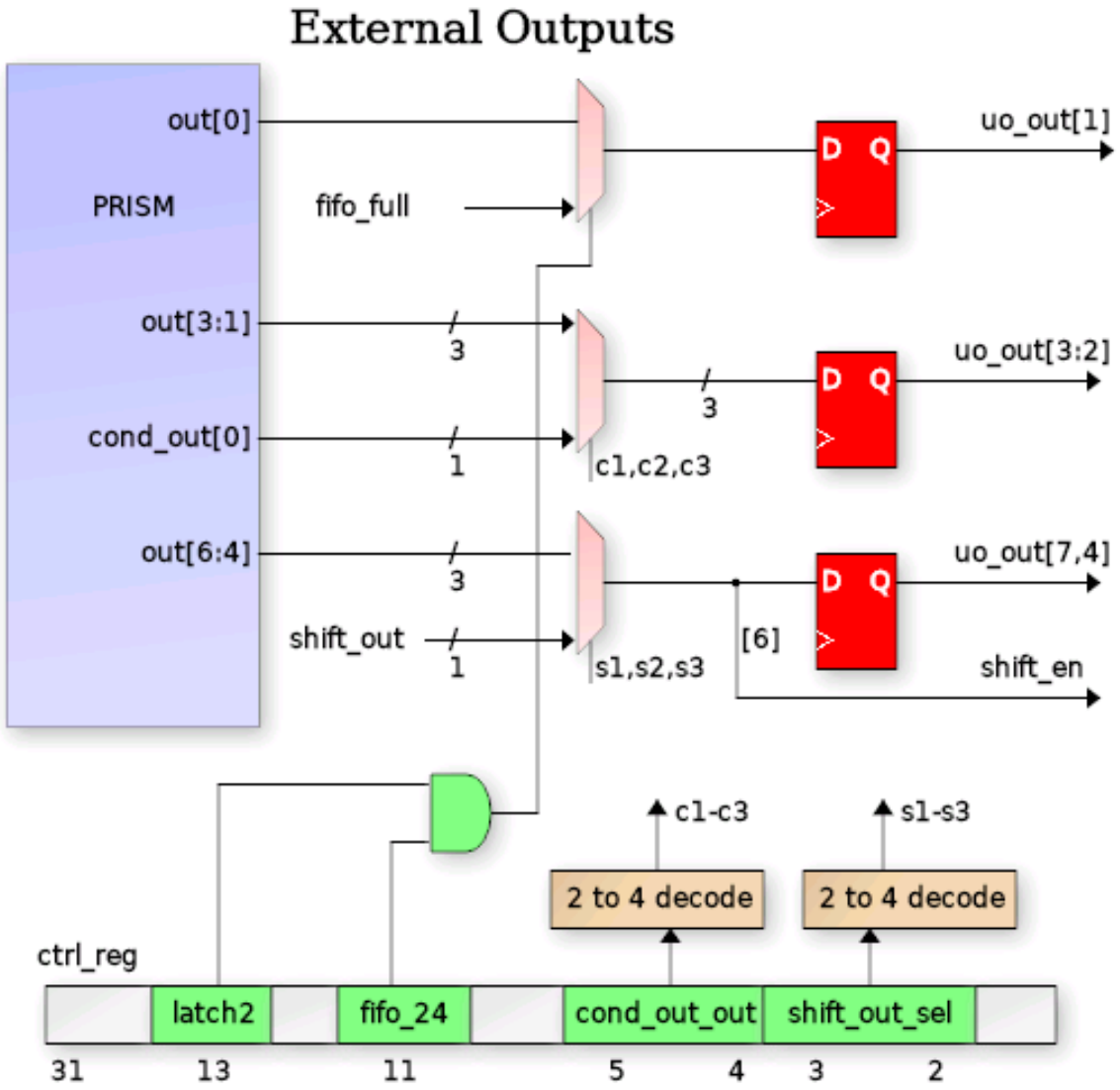
The in[13:12] inputs to PRISM can also be driven by the registered version of out[6] and out[1] bits. This allows a chroma to use a single FSM state to perform multiple output operations using the in-state vs. transitional outputs. For instance, the shift_en signal is output on out[6]. The following code will perform a rising plus falling shift edge in a single FSM state since a registered version of the shift_en is used to detect when to transition to the next state.

```
STATE_SHIFT_BITS:
   begin
      // Shift the next bit out
      shift_en = 1'b1;

      // Detect the rising shift_en bit to go to next state
      if (shift_en_in)
      begin
         next_state = STATE_DELAY;
         shift_en = 1'b0;
      end
   end
```

External PRISM Outputs

The PRISM has 11 outputs, 7 of which (out[6:0]) can directly drive uo_out[7:1] pins. Note that uo_out[0] is not used since it is the TinyQV UART TX pin. However some of the out[6:0] outputs from PRISM also have special functions, and the uo_out[7:1] pins also have muxes allowing other types of output (such as shift_out data, conditional out, etc.). Below is a diagram of the external outputs for the PRISM peripheral.



Outputs uo_out[1], out[5] and out[7] are controled directly by PRISM outputs 0,4 and 6 respectively. Though PRISM out[6] also serves as the “shift_enable” signal when performing either an 8-bit or 24-bit shift operation (via enable bits in the ctrl_reg).

Outputs uo_out[4:2] can be driven either directly from PRISM out[3:1], or can have muxed data:

- cond_out_sel (2-bits) when non-zero selects cond_out[0] conditional output.

Outputs uo_out[7:5] can be driven either directly from PRISM out[6:4], or can have muxed data:

- shift_out_sel (2-bits) when non-zero selects 8 or 24 bit shift out data.

Output uo_out[0] can be driven either directly from PRISM out[0] or can be the “fifo_full” signal when fifo_24 and latch2 config bits are set in ctrl_reg (more on those bits in the 3-Byte FIFO section).

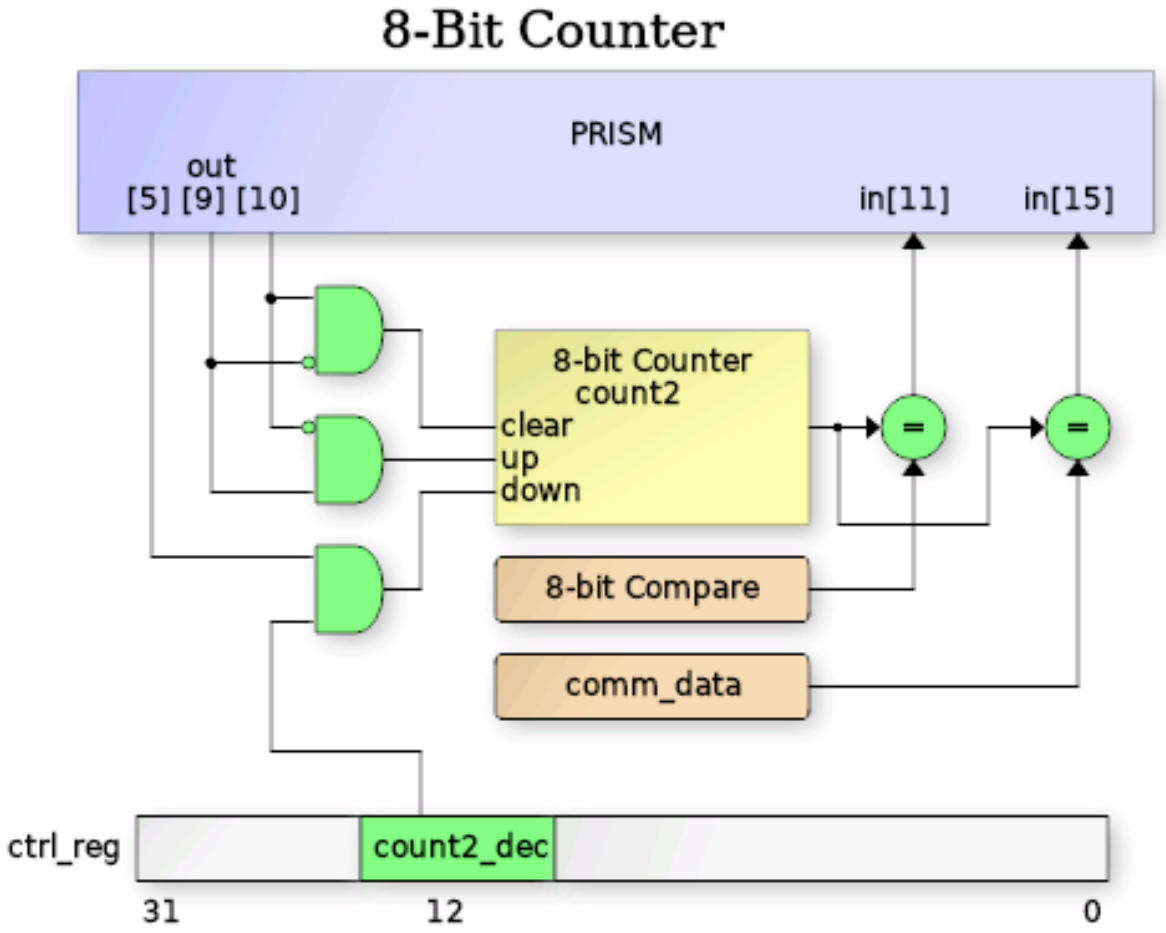
Peripherals

The PRISM Peripheral has, itself, peripherals. Those are:

1. 8-Bit communication shift register (left or right)
2. 8-Bit count up/down register with compare (dedicated compare reg or comm register).
3. 24-bit register: Count down with zero detect, shift left / right or 3-byte FIFO
4. 5-bit Shift bit counter (automatic)
5. Input shift data source selection
6. Output shift data destination selection
7. Conditional output destination selection
8. Controllable latched inputs (2-bits)

8-Bit Counter (count2)

The 8-bit counter is an up/down/clear counter controllable from the PRISM chroma. There are three PRISM outputs plus a ctrl_reg bit that contols the operation (see figure below). Upon reset (or disabling PRISM) the count will be zero. In any PRISM state, the counter can be cleared, incremented or decremented (assuming decrement has been enabled via the count2_dec ctrl_reg bit). The decrement is provided as a configurable feature in case it is not needed but uo_out[6] is needed (since they share the same out[5] signal).



The current 8-bit count2 value is compared against both the 8-bit count2_compare register (fixed register accessible by TinyQV with R/W access) *and* the comm_data register (also R/W accessible). The result of each of these compares are made available on PRISM inputs in[11] and in[15]. A running chroma can use these compare inputs for timing, terminal count checking, etc.

Chroma

Chroma are PRISM's version of "personalities". Each chroma is a unique hue of PRISM's spectrum of behavior. Chroma's are coded as Mealy state machines in Verilog to define FSM inputs, outputs and state transitions:

```
always @(posedge clk or negedge rst_n)
   if (~rst_n)
      curr_state <= 3'h0;
   else
      curr_state <= fsm_enable ? next_state : 'h0;

always_comb
begin
   pin_out[5:0]   = 6'h0;
```

```
   count1_dec     = 1'b0;
   etc.

   case (curr_state)
   STATE_IDLE:       // State 0
      begin
         // Detect I/O shift start
         if (host_in[HOST_START])
         begin
            // Load inputs
            pin_out[GPIO_LOAD] = 1'b0;

            // Load 24-bit shift register from preload (our
OUTPUTS)
            count1_load = 1'b1;
            next_state = STATE_LATCH_INPUTS;
         end
      end
   STATE_LATCH_INPUTS:  // State 1
      begin
         next_state = STATE_SHIFT_BITS;
      end
   etc.
end
```

Chroma are compiled into PRISM programmable bitstreams via a custom fork of Yosys (see link below) using a configuration file describing the architecture in the TinyQV PRISM peripheral. In addition to bitstream generation, the Yosys PRISM backend also calculates the ctrl_reg value for configuring the PRISM peripheral muxes, etc. There are several output formats including C, Python and columnar list:

ST	Mux0	Mux1	Mux2	Inc	JmpA	OutA	Out	CfgA	CfgB	STEW
0	8	0	0	0	1	100	001	a	0	288000120 10
1	0	0	0	0	2	001	000	f	0	3c0080040 00
2	d	0	0	0	3	001	041	a	0	28008416 01a
3	e	0	0	1	2	001	001	5	0	14008014 01d
4	0	0	9	0	5	001	000	f	2	bc00800b2 00
5	8	0	0	0	0	001	001	5	0	1400801001 0
6	0	0	0	0	0	001	000	f	0	3c0080000 00
7	0	0	0	0	0	001	000	f	0	3c0080000 00

The table has the following fields

- ST: the state (obvious)
- Mux0: Selects input for LUT2 input 0 (jump decision)

- Mux1: Selects input for LUT2 input 1
- Mux2: Selects input for LUT1 Conditional Output
- Inc: Set when next state looping is requested (i.e. 'else state <= ST_A')
- JmpA: The "Jump to" state if LUT2 output is TRUE
- OutA: Outputs during "jump to" JmpA state (LSB is PRISM out[0])
- Out: Output during no-jump, steady-state dwelling
- CfgA: The LUT2 4-bit lookup table values
- CfgB: The LUT1 2-bit lookup table values
- STEW: The complete word aggregrated in proper bit order

Register map

Document the registers that are used to interact with your peripheral

Address	Name	Access	Description
0x00	CTRL	R/W	Control register - see [CTRL Register] below
0x04	DBG_CTRL	W	Debug control
0x0C	DBG_STAT	W	Debug status
0x10	CFG_LSB	W	Config LSB (write second)
0x14	CFG_MSB	W	Config MSB (write first)
0x18	HOST_DATA	R/W	Host data - see [HOST_DATA Register]
0x19	FIFO_DATA	R	FIFO read data - see [HOST_DATA Register]
0x1A	FIFO_STAT	R	FIFO status - see [HOST_DATA Register]
0x1B	HOST_IN	R/W	Host input - see [HOST_DATA Register]
0x20	COUNT_CFG	R/W	Counter config - see [COUNT_CFG Reg]
0x24	COUNT_VAL	R	Counter values - see [COUNT_VAL Reg]
0x34	DECISION_TREE	R	Decision tree data
0x38	OUTPUT_DATA	R	Output data
0x3C	INPUT_DATA	R	Input data

Bit-field Details

CTRL Register (0x00)

Bit(s)	Name	Description
31	Interrupt clear	Write 1 to clear interrupt
30	PRISM enable	Enable/disable PRISM executino
23	ui_in[7]	Direct read of ui_in bit 7
22-16	latched_out	Latched output values
13	latch5	Use prism_out[5] to latch inputs
12	count2_dec	Enable count2 decrement mode
11	fifo_24	Enable 24-bit FIFO mode
10	shift_24_en	Enable 24-bit shift mode
9	shift_dir	Shift direction (0=left, 1=right)
8	shift_en	Enable shift operations
7	latch_in_out	Latch input/output mode
6-4	cond_out_sel	Conditional output selection
3-2	shift_out_sel	Shift output selection
1-0	comm_in_sel	Communication input selection

DBG_CTRL Register (0x04)

Bit(s)	Name	Description
13-11	new_si_val	New state index to set
10	new_si	Set to 1 to set new state index
9-7	bp1	Breakpoint 1 state index
6-4	bp0	Breakpoint 0 state index
3	bp1_en	Enable breakpoint 1
2	bp0_en	Enable breakpoint 0
1	single_step	Toggle 1->0 to single step PRISM
0	halt_req	Set to 1 to halt the PRISM

DBG_STAT Register (0x0C)

Bit(s)	Name	Description
8-7	break_active	Which breakpoint is active
6	debug_halt	Indicates if PRISM is halted
5-3	next_si	Next state index to jump to

2-0	curr_si	Current state index

HOST_DATA Register (0x18)

Bit(s)	Name	Description
31-26	Reserved	Reserved bits
25-24	host_in	Host input data
23-21	Reserved	Reserved bits
20-19	fifo_rd_ptr	Reserved bits
19-18	fifo_wr_ptr	Reserved bits
17	fifo_full	FIFO full status
16	fifo_empty	FIFO empty status
15-8	fifo_rd_data	FIFO read data
7-0	comm_data	Communication data

FIFO_READ Register (0x19)

Byte-mode access of the 3-byte RX FIFO. A byte read from this address will "pop" the byte from the fifo, incrementing the read point, updating the FULL/EMPTY flags, etc.

FIFO_STATUS Register (0x1a)

Byte-mode access of the RX FIFO status.

Bit(s)	Name	Description
31-26	Reserved	Reserved bits

COUNT_CONFIG Register (0x20)

Bit(s)	Name	Description
31-24	count2_compare	8-bit Compare value
23-0	count1_preload	24-bit Preload value

COUNT_VALUES Register (0x24)

Bit(s)	Name	Description
31-24	count2	Current value of 8-bit counter
23-0	count1	Current value of 24-bit counter

DEBUG_OUT Register (0x30)

Bit(s)	Name	Description
31-0	debug_output	Debug output bits

DECISION_TREE Debug Register (0x34)

Bit(s)	Name	Description
31-0	decision_tree	Compare matches and LUT inputs

OUTPUT_DATA Debug Register (0x38)

Bit(s)	Name	Description
31-11	Reserved	Reserved bits
10-0	out_data	Current state output data

INPUT_DATA Debug Register (0x3C)

Bit(s)	Name	Description
31-16	Reserved	Reserved bits
15-0	in_data	Current input data

How to test

1. First define a Chroma FSM transitions, inputs and outputs.
2. Write Verilog to describe your FSM in Mealy format.
3. Specify the Control Word Mux and options bit settings in the Verilog: shift_en: Enable shifter mode in either comm_data or 24-bit count1 shift_24_en: Specify if shift is comm_data or count1 shift_dir: Set left or right shift direction fifo_24: Use 24-bit count1 as RX FIFO etc.
4. Clone and build the custom Yosys fork for generating PRISM bitstreams. (2)
5. Generate your chroma bitstream using the PRISM Yosys version. (3)
6. Copy the generated chroma and ctrl_reg value from either the .py or .c file.
7. Pass the chroma and ctrl_reg value to the “load_chroma” function.
8. The chroma should start running immediately.

(1) See examples in chromas directory.

(2) The Makefile in the chromas directory has a ‘make yosys’ target for cloning and building this from github sources.

(3) If you put your chroma in the ‘chromas’ directory and follow the naming convention, simply typing ‘make’ within that directory will build your chroma. Results appear in the ‘output’ directory.

- https://github.com/kdp1965/tinyqv-prism-lite/tree/main/chromas
- https://github.com/kdp1965/yosys-prism

External hardware

No external HW required other than anything custom you might want to control from the programmable FSM.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	UART RX	fsm_out0	—
1	fsm_in0	fsm_out1	—
2	fsm_in1	fsm_out2	—
3	fsm_in2	fsm_out3	spi_miso
4	fsm_in3	fsm_out4	spi_cs_n
5	fsm_in4	fsm_out5	spi_clk
6	fsm_in5	fsm_out6	spi_mosi
7	fsm_in6	UART TX	—

miniMAC_5L

by **Yann Guidon**

`0266` `100 MHz` `HDL Project`

`github.com/ygdes/miniMAC_5L`

> *"16-bit Scrambler/Framer/Error detector for IHP SG2CMOS5L PDK"*

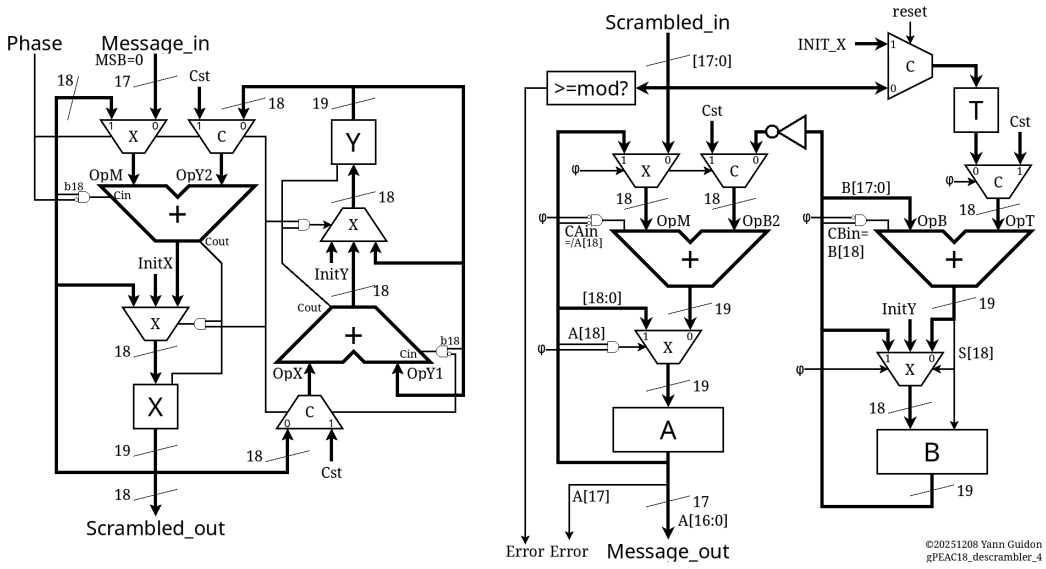
Note: after the messy tapeout on iHP26a, porting to CMOS5L PDK.

What is this miniMAC

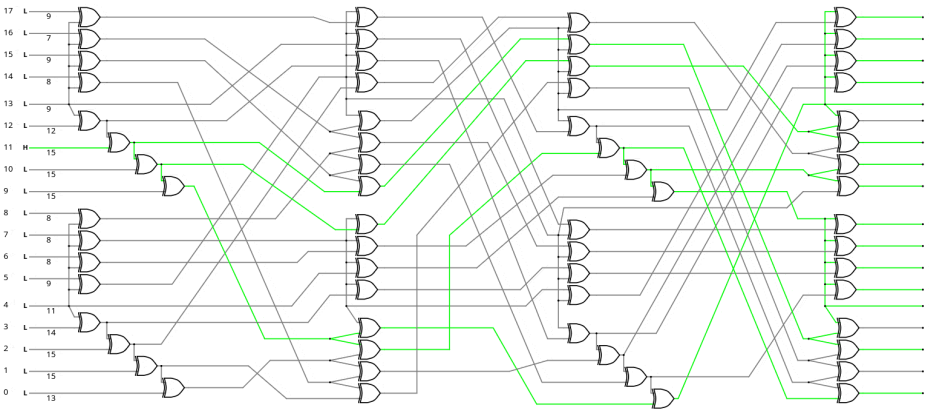
IMPORTANT: This custom circuit and protocol is not at all compliant or even compatible, even remotely linked to any 802.3 standard. It's all explained and detailed on Hackaday at https://hackaday.io/project/198914

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:

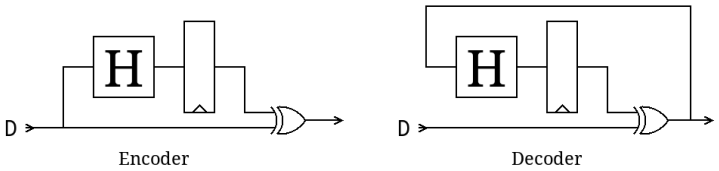
- The term "gPEAC" means "generalised Pisano with End-Around Carry" (see https://hackaday.io/project/178998), a class of PRNG/scrambler/checksum that uses different mathematics than Galois-based LFSR. The gPEAC18 unit is a non-power-of-two additive-based scrambler-checksum, with modulus=258114, orbit=66623095108 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:

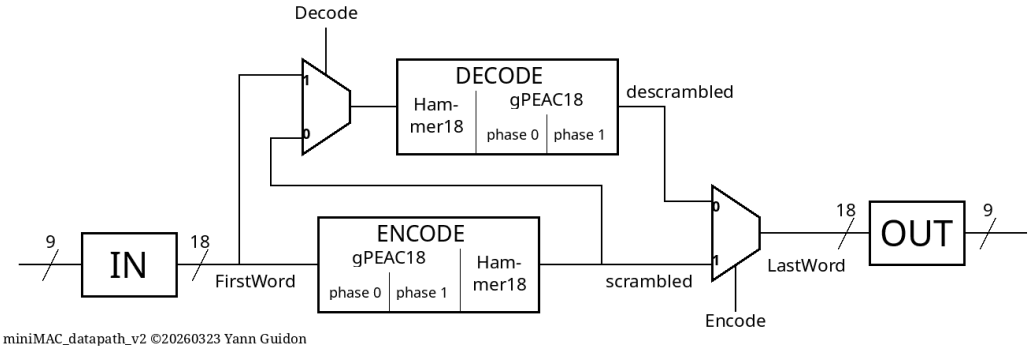
- For encoding, the input data goes through gPEAC then Hammer is inserted at the end of the last cycle.
- For decoding, the scrambled data goes through Hammer at the start of the first cycle of gPEAC descrambling.

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

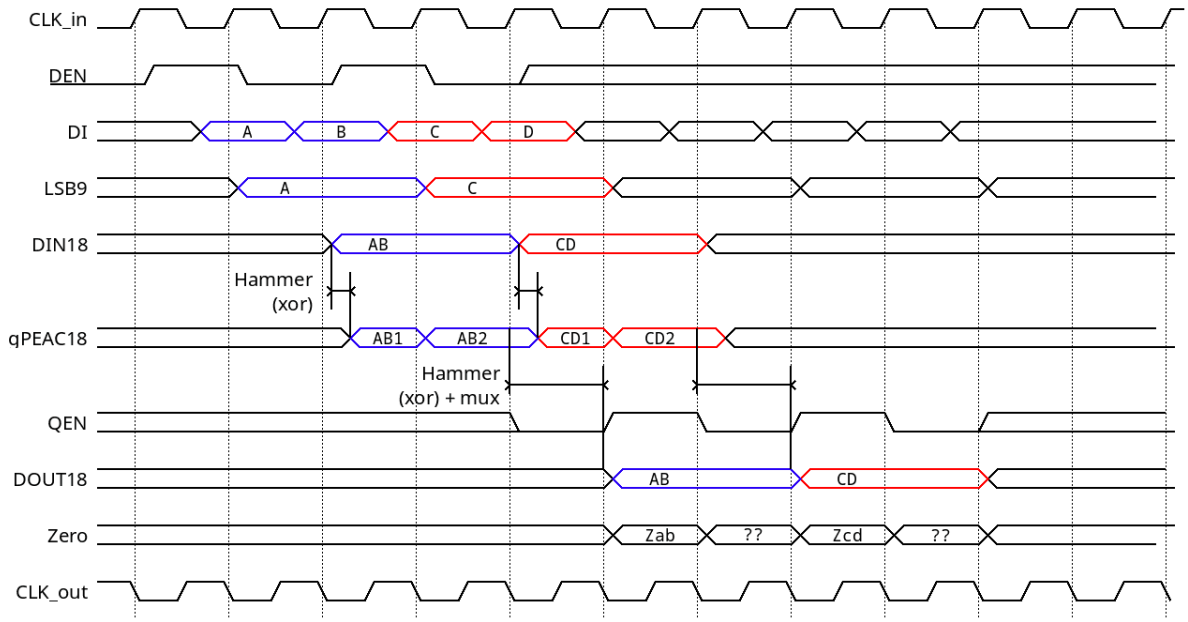
First let's examine the pinout. The inputs:

- 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 : loopback mode => encodes then decodes, the delayed output (8 cycles) must be identical to the input.,
  - Enc=0, Dec=1 : decode mode => the cleartext input is scrambled and output after 5 cycles,

    - Enc=1 : encode mode => the scrambled input is restored to cleartext after 5 cycles.
- 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.

Next developments

This started as a VHDL to Verilog+IHP PDK port but it will likely grow to a more-featured project.

- I'll try to get 2 boards to test both coder and decoder in a chain, to simulate noisy communications.
- In parallel I try to design a decent PHY, a much more difficult endeavour.
- Logically, the MAC must be completed with a FSM, a buffer and a host interface, likely in the next tapeout so I can play with memory blocks.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	DI0	DO0	D08
1	DI1	DO1	QEN
2	DI2	DO2	CLK_out
3	DI3	DO3	Zero
4	DI4	DO4	Enc
5	DI5	DO5	Dec
6	DI6	DO6	DEN
7	DI7	DO7	DI8

TWI Monitor

by **Nicklaus Thompson**

0291 | 50 MHz | HDL Project

github.com/FangameEmpire/ttihp0p4-twi-monitor

"A Two Wire Interface (I2C) bus monitor"

How it works

This project is a Two-Wire Interface (I2C) monitor. The TWI side is essentially a shift register and does not respond like a slave or have an address. The system runs at 50 MHz and uses a UART baud rate of 115200. The system cannot currently capture repeated TWI frames, but captured single frames during testing on an FPGA.

How to test

You can use an Arduino and any TWI-compatible module to generate TWI frames to view. The frames will be converted to three bytes, those being {addr, R/W}, {data}, and {{4{Addr Ack}}, {4{Data Ack}}}. I use Coolterm to view the hex output, you can download it at https://freeware.the-meiers.org/.

External hardware

This project needs an external UART to USB adapter if you want to connect it to your PC.

Project Pinout

Digital Pins

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

TinyMOA-IHP0P4-8x8

by **Ezra Wolf**

0293 | 50 MHz | HDL Project

github.com/EzraWolf/TinyMOA-IHP0P4-8x8

"8x8 digital compute-in-memory (DCIM) core on an experimental IHP SG13G2 CMOS5L shuttle."

How it works

Digital compute-in-memory (DCIM) 8x8 array core.

How to test

Documentation in progress.

External hardware

Documentation in progress.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	data_in[0]	result[0]	wen (in)
1	data_in[1]	result[1]	execute (in)
2	data_in[2]	result[2]	read_next (in)
3	data_in[3]	result[3]	acc_clear (in)
4	data_in[4]	result[4]	col_sel[0] (out)
5	data_in[5]	result[5]	col_sel[1] (out)
6	data_in[6]	result[6] (zero)	col_sel[2] (out)
7	data_in[7]	result[7] (zero)	done (out)

Ring osc on VGA

by **algofoogle (Anton Maurovic)**

`0295` `25.175 MHz` `HDL Project`

`github.com/algofoogle/ttihp0p4-vga-ring-osc`

"VGA display visualisation of a ring oscillator doing work"

How it works

This is a TTIHP0p4 experimental resubmission of the TTGF0p2 version: ttgf0p2-vga-ring-osc

Manually-instantiated ihp-sg13cmos5l inverter cells form a chain out of chain segments of varying lengths, allowing the user to select given points in the overall chain to loop back to produce a ring oscillator. This makes a configurable ring oscillator that is expected to be able to oscillate from about 20MHz up to 850MHz (theoretically 3.7GHz but this probably won't work).

This (or an external clock) then can be selected to drive a "worker" module: a counter which counts up to 3000.

Alongside this is a VGA sync generator which takes its pixel colour from whatever is in the upper 6 bits of the worker's counter at the time. The worker is reset during HBLANK of each VGA line.

It's expected that at the faster ring oscillator speeds, the counter will reach its target of 3000 sooner than the width of the VGA line but with some jitter... or the counter/compare logic will break down because it's too fast.

How to test

Set `clksel2[1:0]` to 0.

Set `clksel[3:0]` to (say) 10, or anything greater than 1.

Set `mode[1:0]` to 0 (though these are unused at the time of writing; TBA).

Set `vga_mode` to 0.

Attach a Tiny VGA PMOD to `uo_out`.

Supply a 25MHz clock to the system `clk`, and assert reset for at least 2 clocks.

Expect to see vertical coloured bars on screen, but expect some jitter. Their width should increase as you increase `clksel`.

Measure the ring oscillator (or rather, the selected clock source) on `uio_out[7:4]`: `uio_out[4]` is the raw oscillator output, and the higher bits are the oscillator divided by powers of 2.

More testing notes:

- When `vga_mode==1`, `clk` should be 26.6175MHz (106.47 MHz ÷ 4) to drive a 1440x900 60Hz VGA display.
- When `clksel2` is:
  - 0: Just rely on `clksel`.
  - 1: Use fixed 25-deep inv_2 ring oscillator.
  - 2: Use fixed 25-deep inv_4 ring oscillator.
  - 3: Use inverted `clk`.
  - NOTE: options 1 and 2 require `clksel > 1` (any value will do) to enable the rings.
- When `clksel2==0` and `clksel` is:
  - 0: Use `clk`.
  - 1: Use `altclk`.
  - For values 2 and above, use an inv_1-based ring oscillator tapped at...
  - 2: => 3 => 3.70 GHz
  - 3: => 5 => 2.22 GHz
  - 4: => 9 => 1.23 GHz
  - 5: => 13 => 855 MHz
  - 6: => 19 => 585 MHz
  - 7: => 25 => 444 MHz
  - 8: => 33 => 337 MHz
  - 9: => 41 => 271 MHz
  - 10: => 57 => 195 MHz
  - 11: => 65 => 171 MHz
  - 12: => 97 => 115 MHz
  - 13: => 161 => 69.0 MHz
  - 14: => 289 => 38.4 MHz
  - 15: => 545 => 20.4 MHz

(NOTE: Frequencies are ROUGHLY estimated, and it's expected that going above 855MHz internally probably won't work).

External hardware

Tiny VGA PMOD and a VGA monitor.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	clksel[0]	r7	IN: clksel2[0]
1	clksel[1]	g7	IN: clksel2[1]

#	Input	Output	Bidirectional
2	clksel[2]	b7	—
3	clksel[3]	vsync	—
4	altclk	r6	OUT: osc
5	mode[0]	g6	OUT: div2
6	mode[1]	b6	OUT: div4
7	vga_mode	hsync	OUT: div8

VGA clock

by **Matt Venn**

0297 | 31.5 MHz | HDL Project

`github.com/mattvenn/ttihp0p4-vga-clock`

"Shows the time on a VGA screen"

How it works

Races the beam! Font is pre generated and loaded into registers. 6 bit colour keeps register count low.

Every minute the colours cycle.

How to test

Hook up a VGA monitor to the outputs and provide a clock at 31.5 MHz.

Adjust time with the inputs[2:0], and choose the type of VGA PMOD with the input[3].

External hardware

VGA PMOD - you can use one of these VGA PMODs:

- https://github.com/mole99/tiny-vga
- https://github.com/TinyTapeout/tt-vga-clock-pmod

Set input[3] low to use tiny-vga and high to use vga-clock

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	adjust hours	hsync / R1	—
1	adjust minutes	vsync / G1	—
2	adjust seconds	B0 / B1	—
3	PMOD type select	B1 / VS	—
4	—	G0 / R0	—
5	—	G1 / G0	—
6	—	R0 / B0	—
7	—	R1 / HS	—

Silicon Strummer

by **Pranav M**

#299 HDL Project

`github.com/pranav0x0112/Silicon-Strummer`

"VGA fretboard guitar"

How it works

Silicon Strummer turns a VGA monitor into a playable guitar fretboard. The screen is split into a grid of 8 columns (frets) and 6 rows (strings), giving 48 unique notes total. Each cell in the grid is about 80x80 pixels.

A yellow cursor shows which note is currently selected. You move the cursor around using buttons, and when you hold the sound button, the chip outputs a square wave tone through the audio pin at the frequency of the selected note.

The pitch increases as you move right (higher fret) and down (lower string number to higher string number), going from about 100 Hz in the top-left corner all the way to about 2 kHz in the bottom-right corner, just like a real guitar neck.

The display uses the TinyVGA PMOD pin mapping. Grid lines are drawn in dim blue, string lines run horizontally in dim green, and the selected cell lights up in bright amber yellow. Everything else is black.

Audio is generated as a simple square wave oscillator. A counter counts down from a precomputed divider value based on the selected note, and toggles the output pin each time it hits zero. The divider is calculated as:

```
divider = 125000 - (note_index x 2510)
```

where note_index goes from 0 (top-left) to 47 (bottom-right).

The cursor position updates once per frame on the rising edge of vsync, so movement feels smooth and consistent at 60 fps.

How to test

Connect a TinyVGA PMOD to the output pins and a speaker or audio circuit to uio_out[7].

Use the buttons like this:

- ui_in[0] moves the cursor right to the next fret
- ui_in[1] moves the cursor left to the previous fret
- ui_in[2] moves the cursor up to the previous string

- ui_in[3] moves the cursor down to the next string
- ui_in[4] hold this to enable audio output

Move the cursor to any cell on the fretboard and hold the sound button to hear the note at that position. Try moving across frets to hear pitch go up, and across strings to hear bigger jumps in pitch.

You can also test it in the Tiny Tapeout VGA Playground at https://vga-playground.com by clicking the ui_in buttons on screen to move the cursor and pressing button 4 to hear audio.

External hardware

- TinyVGA PMOD connected to uo_out for VGA display output
- Speaker or audio output circuit connected to uio_out[7] for square wave audio output

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Move right (fret +)	R1 (TinyVGA)	—
1	Move left (fret -)	G1 (TinyVGA)	—
2	Move up (string -)	B1 (TinyVGA)	—
3	Move down (string +)	vsync (TinyVGA)	—
4	Enable audio	R0 (TinyVGA)	—
5	—	G0 (TinyVGA)	—
6	—	B0 (TinyVGA)	—
7	—	hsync (TinyVGA)	Audio out (square wave)

LISA 8-Bit Microcontroller

by **Ken Pettit**

326 | 18 MHz | HDL Project

github.com/kdp1965/ihp-cmos5l-um-lisa

"8-Bit Microcontroller SOC with 128 bytes Flop based RAM"

What is LISA?

LISA is a Microcontroller built around a custom 8-Bit Little ISA (LISA) micro-processor core. It includes several standard peripherals that would be found on commercial microcontrollers including timers, GPIO, UARTs and I2C. The following is a block diagram of the LISA Microcontroller:

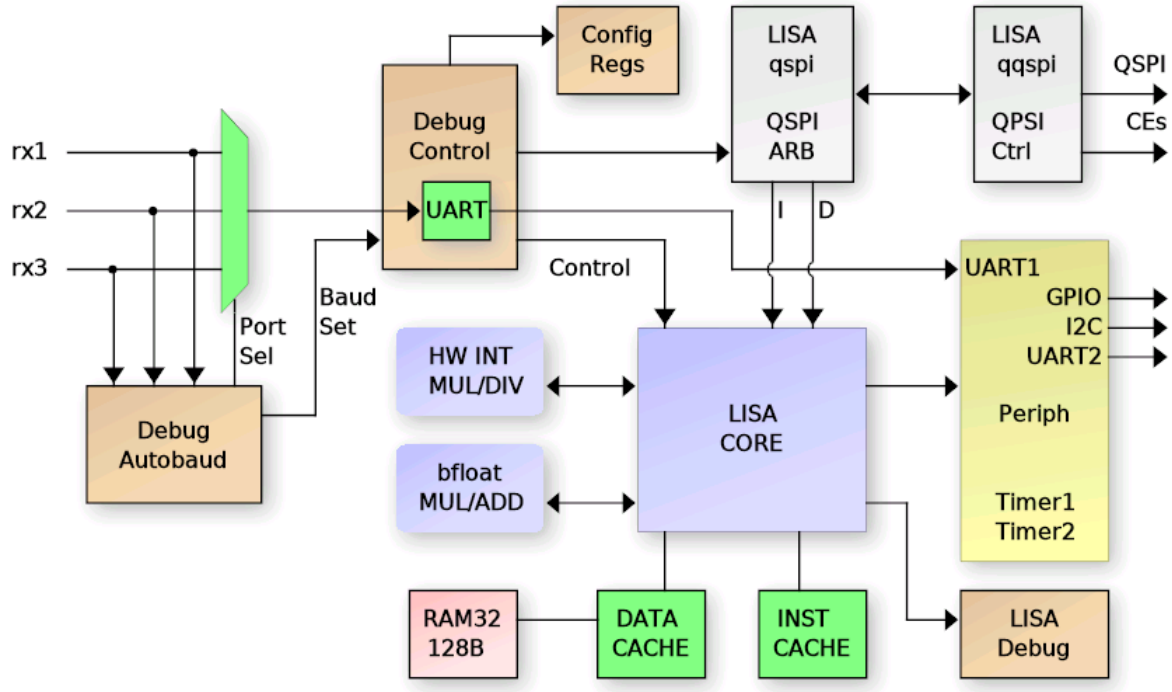


Figure 974.1: LISA Microcontroller Block Diagram

- The LISA Core has a minimal set of register that allow it to run C programs:
  - Program Counter + Return Address Resister
  - Stack Pointer and Index Register (Indexed DATA RAM access)
  - 8-bit Accumulator + 16-bit BF16 Accumulator and 4 BF16 registers

Deailed list of the features

- Harvard architecture LISA Core (16-bit instruction, 15-bit address space)
- Debug interface
  - UART controlled
  - Auto detects port from one of 3 interfaces

    - Auto detects the baud rate
    - Interfaces with SPI / QSPI SRAM or FLASH
    - Can erase / program the (Q)SPI FLASH
    - Read/write LISA core registers and peripherals
    - Set LISA breakpoints, halt, resume, single step, etc.
    - SPI/QSPI programmability (single/quad, port location, CE selects)
- (Q)SPI Arbiter with 3 access channels
    - Debug interface for direct memory access
    - Instruction fetch
    - Data fetch
    - Quad or Single SPI. Hereafter called QSPI, but supports either.
- Onboard 128 Byte RAM for DATA / DATA CACHE
- Data bus CACHE controller with 8 16-byte CACHE lines
- Instruction CACHE with a single 4-instruction CACHE line
- Two 16-bit programmable timers (with pre-divide)
- Debug UART available to LISA core also
- Dedicated UART2 that is not shared with the debug interface
- 8-bit Input port (PORTA)
- 8-bit Output port (PORTB)
- 4-bit BIDIR port (PORTC)
- I2C Master controller
- Hardware 8x8 integer multiplier
- Hardware 16/8 or 16/16 integer divider
- Hardware Brain Float 16 (BF16) Multiply/Add/Negate/Int16-to-BF16
- Programmable I/O mux for maximum flexibility of I/O usage.

It uses a 32x32 1RW DFFRAM macro to implement a 128 bytes (1 kilobit) RAM module. The 128 Byte ram can be used either as a DATA cache for the processor data bus, giving a 32K Byte address range, or the CACHE controller can be disabled, connecting the Lisa processor core to the RAM directly, limiting the data space to 128 bytes. Inclusion of the DFFRAM is thanks to Uri Shaked (Discord urish) and his DFFRAM example.

Reseting the project **does not** reset the RAM contents.

Connectivity

All communication with the microcontroller is done through a UART connected to the Debug Controller. The UART I/O pins are auto-detected by the debug_autobaud module from the following choices (RX/TX):

```
ui_in[3]  / ui_out[4]     RP2040 UART interface
uio_in[4] / uio_out[5]    LISA PMOD board (I am developing)
uio_in[6] / uio_out[5]    Standard UART PMOD
```

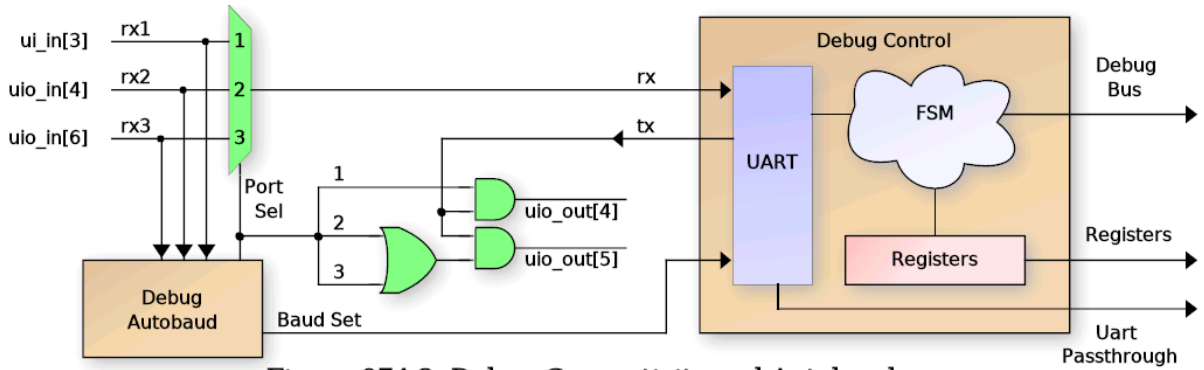


Figure 974.2: Debug Connectivity and Autobaud

The RX/TX pair port is auto-detected after reset by the autobaud circuit, and the UART baud rate can either be configured manually or auto detected by the autobaud module. After reset, the ui_in[7] pin is sampled to determine the baud rate selection mode. If this input pin is HIGH, then autobaud is disabled and ui_in[6:0] is sampled as the UART baud divider and written to the Baud Rate Generator (BRG). The value of this divider should be: clk_freq / baud_rate / 8 - 1. Due to last minute additions of complex floating point operations, and only 2 hours left on the count-down clock, the timing was relaxed to 20MHz input clock max. So for a 20MHz clock and 115200 baud, the b_div[6:0] value would be 42 (for instance).

If the ui_in[7] pin is sampled LOW, then the autobaud module will monitor all three potential RX input pins for LINEFEED (ASCII 0x0A) code to detect baud rate and set the b_div value automatially. It monitors bit transistions and searches for three successive bits with the same bit period. Since ASCII code 0x0A contains a “0 1 0 1 0” bit sequence, the baud rate can be detected easily.

Regardless if the baud rate is set manually or using autobaud, the input port selection will be detect automatically by the autobaud. In the case of manual buad rate selection, it simply looks for the first transition on any of the three RX pins. For autobaud, it selects the RX line with three successive equivalent bit periods.

Debug Interface Details

The Debug interface uses a fixed, Verilog coded Finite State Machine (FSM) that supports a set of commands over the UART to interface with the microcontroller. These commands are simple ASCII format such that low-level testing can be performed using any standard terminal software (such as minicom, tio. Putty, etc.). The ‘r’ and ‘w’ commands must be terminated using a NEWLINE (0x0A) with an optional CR (0x0D). Responses from the debug interface are always terminated with a LINFEED plus CR sequence (0x0A, 0x0D). The commands are as follows (responsce LF/CR ommited):

Command	Description

v	Report Debugger version. Should return: lisav1.2
wAAVVVV	Write 16-bit HEX value ‘VVVV’ to register at 8-bit HEX address ‘AA’.
rAA	Read 16-bit register value from 8-bit HEX address ‘AA’.
t	Reset the LISA core.
l	Grant LISA the UART. Further data will be ignored by the debugger.
+++	Revoke LISA UART. NOTE: a 0.5s guard time before/after is required.

NOTE: All HEX values must be a-f and not A-F. Uppercase is not supported.

Debug Configuration and Control Registers

The following table describes the configuration and LISA debug register addresses available via the debug ‘r’ and ‘w’ commands. The individual register details will be described in the sections to follow.

ADDR	Description	ADDR	Description
0x00	LISA Core Run Control	0x12	LISA1 QSPI base address
0x01	LISA Accumulator / FLAGS	0x13	LISA2 QSPI base address
0x02	LISA Program Counter (PC)	0x14	LISA1 QSPI CE select
0x03	LISA Stack Pointer (SP)	0x15	LISA2 QSPI CE select
0x04	LISA Return Address (RA)	0x16	Debug QSPI CE select
0x05	LISA Index Register (IX)	0x17	QSPI Mode (QUAD, flash, 16b)
0x06	LISA Data bus	0x18	QSPI Dummy read cycles
0x07	LISA Data bus address	0x19	QSPI Write CMD value
0x08	LISA Breakpoint 1	0x1a	The ‘+++’ guard time count
0x09	LISA Breakpoint 2	0x1b	Mux bits for uo_out
0x0a	LISA Breakpoint 3	0x1c	Mux bits for uio
0x0b	LISA Breakpoint 4	0x1d	CACHE control
0x0c	LISA Breakpoint 5	0x1e	QSPI edge / SCLK speed
0x0d	LISA Breakpoint 6	0x20	Debug QSPI Read / Write
0x0f	LISA Current Opcode Value	0x21	Debug QSPI custom command
0x10	Debug QSPI Address (LSB16)	0x22	Debug read SPI status reg

0x11	Debug QSPI Address (MSB8)	—	—

LISA Processor Interface Details

The LISA Core requires external memory for all Instructions and Data (well, sort of for data, the data CACHE can be disabled then it just uses internal DF-FRAM). To accomodate external memory, the design uses a QSPI controller that is configurable as either single SPI or QUAD SPI, Flash or SRAM access, 16-Bit or 24-Bit addressing, and selectable Chip Enable for each type of access. To achieve this, a QSPI arbiter is used to allow multiple accessors as shown in the following diagram:

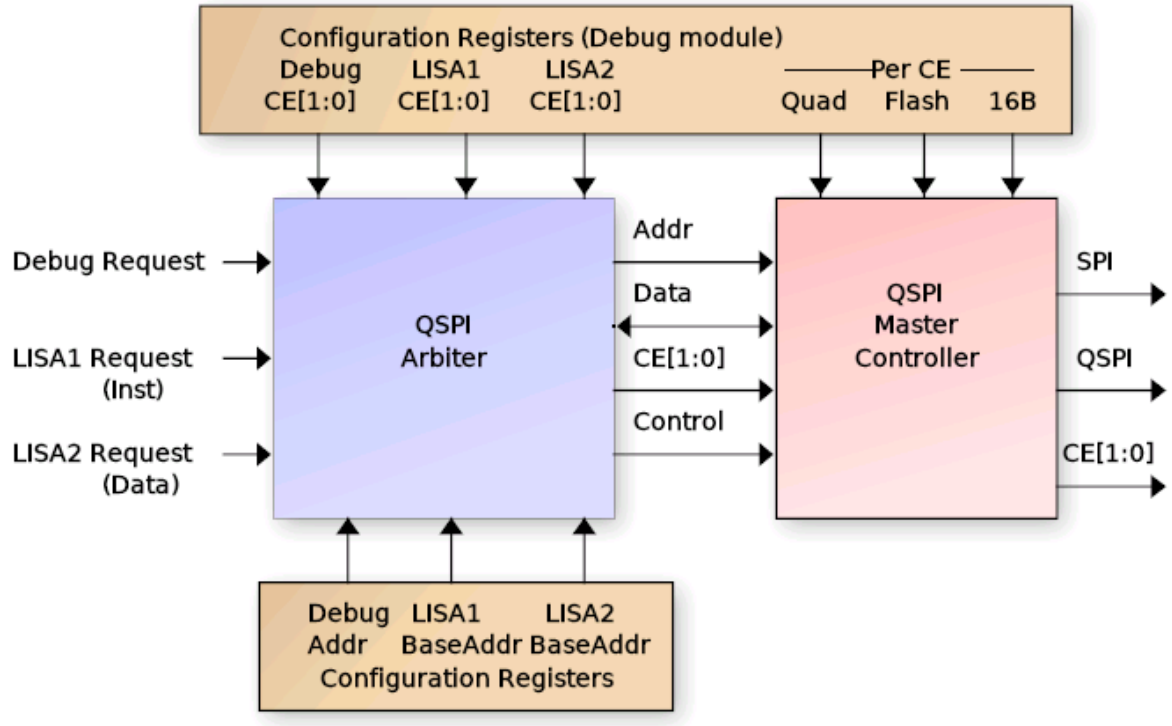


Figure 974.3: (Q)SPI Controller Interface Diagram

The arbiter is controlled via configuration registers (accessible by the Debug controller) that specify the operating mode per CE, and CE selection bits for each of the three interfaces:

- Debug Interface
- LISA1 (Instruction fetch)
- LISA2 (Data read/write)

The arbiter gives priority to the Debug accesses and processes LISA1 and LISA2 requests using a round-robbin approach. Each requestor provides a 24-bit address along with 16-bit data read/write. For the Debug interface, the address comes from the configuration registers directly. For LISA1, the address is the Program Counter (PC) + LISA1 Base and for LISA2, it is the

Data Bus address + LISA2 Base. The LISA1 and LISA2 base addresses are programmed by the Debug controller and set the upper 16-bits in the 24-bit address range. The PC and Data address provide the lower 16 bis (8-bits overlapped that are 'OR'ed together). The BASE addresses allow use of a single external QSPI SRAM for both instruction and data without needing to worry about data collisions.

When the arbiter chooses a requestor, it passes its programmed CE selection to the QSPI controller. The QSPI controller then uses the programmed QUAD, MODE, FLASH and 16B settings for the chosen CE to process the request. This allows LISA1 (Instruction) to either execute from the same SRAM as LISA2 (Data) or to execute from a separate CE (such as FLASH with permanent data storage).

Additionally the Debug interface has special access registers in the 0x20 - 0x22 range that allow special QSPI accesses such as FLASH erase and program, SRAM programming, FLASH status read, etc. In fact the Debug controller can send any arbitrary command to a target device, using access that either provide an associated address (such as erase sector) or no address. The proceedure for this is:

1. Program Debug register 0x19 with the special 8-bit command to be sent
2. Set the 9-th bit (reg19[8]) to 1 if a 16/24 bit address needs to be sent)
3. Perform a read / write operation to debug address 0x21 to perform the action.

Simple QSPI data reads/write are accomplished via the Debug interface by setting the desired address in Debug config register 0x10 and 0x11, then performing read or write to address 0x20 to perform the request. Reading from Debug config register 0x22 will perform a special mode read of QSPI register 0x05 (the FLASH status register).

Data access to the QSPI arbiter come from the Data CACHE interface (described later), enabling a 32K address space for data. However the design has a CACHE disable mode that directs all Data accesses directly to the internal 128 Byte RAM, thus eliminating the need for external SRAM (and limiting the data bus to 128 bytes).

Programming the QSPI Controller

Before the LISA microcontroller can be used in any meaningful manner, a SPI / QSPI SRAM (and optionally a NOR FLASH) must be connected to the Tiny Tapeout PCB. Alternately, the RP2040 controller on the board can be configured to emulate a single SPI (the details for configuring this are outside the scope of this documentation ... search the Tiny Tapeout website for details.). For the CE signals, there are two operating modes, fixed CE output and Mux Mode 3 "latched" CE mode. Both will be described here. The other standard SPI signals are routed to dedicated pins as follows:

Pin	SPI	QSPI	Notes
uio[0]	CE0	CE0	—
uio[1]	MOSI	DQ0	Also MOSI prior to QUAD mode DQ0
uio[2]	MISO	DQ1	Also MISO prior to QUAD mode DQ1
uio[3]	SCLK	SCLK	—
uio[4]	CE1	CE1	Must be enabled via uio MUX bits
uio[6]	-	DQ2	Must be enabled via uio MUX bits
uio[7]	-	DQ3	Must be enabled via uio MUX bits

For Special Mux Mode 3 (Debug register 0x1C uio_mux[7:6] = 2'h3), the pinout is mostly the same except the CE signals are not constant. Instead they are "latched" into an external 7475 type latch. This mode is to support a PMOD board connected to the uio PMOD which supports a QSPI Flash chip, a QSPI SRAM chip, and either Debug UART or I2C. For all of that functionality, nine pins would be required for continuous CE0/CE1, however only eight are available. So the external PMOD uses uio[0] as a CE "latch" signal and the CE0/CE1 signals are provided on uio[1]/uio[2] during the latch event. This requires a series resistor as indicated to allow CE updates if the FLASH/SRAM is driving DQ0/DQ1. The pinout then becomes:

Pin	SPI/QSPI	Notes
uio[0]	ce_latch	ce_latch HIGH at beginning of cycle
uio[1]	ce0_latch/MOSI/DQ0	Connection to FLASH/SRAM via series resistor
uio[2]	ce1_latch/MISO/DQ1	Connection to FLASH/SRAM via series resistor
uio[3]	SCLK	—
uio[6]	-/DQ2	Must be enabled via uio MUX bits
uio[7]	-/DQ3	Must be enabled via uio MUX bits

This leaves uio[4]/uio[5] available for use as either UART or I2C.

Once the SPI/QSPI SRAM and optional FLASH have been chosen and connected, the Debug configuration registers must be programmed to indicate the nature of the external device(s). This is accompilished using Debug registers 0x12 - 0x19 and 0x1C. To programming the proper mode, follow these steps:

1. Program the LISA1, LISA2 and Debug CE Select registers (0x14, 0x15, 0x16) indicating which CE to use.
   - 0x14, 0x15, 0x16: {6'h0, ce1_en, ce0_en} Active HIGH

2. Program the LISA1 and LISA2 base addresses if they use the same SRAM:
    - 0x12: {LISA1_BASE, 8'h0} | {8'h0, PC}
    - 0x13: {LISA2_BASE, 8'h0} | {8'h0, DATA_ADDR}
3. Program the mode for each Chip Enable (bits active HIGH)
    - 0x17: {10'h0, is_16b[1:0], is_flash[1:0], is_quad[1:0]}
4. For Quad SPI, Special Mux Mode 3, or CE1, program the uio_mux mode:
    - 0x1C:
        - [7:6] = 2'h2: Normal QSPI DQ2 select
        - [7:6] = 2'h3: Special Mux Mode 3 (Latched CE)
        - [5:4] = 2'h2: Normal QSPI DQ3 select
        - [5:4] = 2'h3: Special Mux Mode 3
        - [1:0] = 2'h2: CE1 select on uio[4]
5. For RP2040, you might need to slow down the SPI clock / delay between successive CE activations:
    - 0x1E:
        - [3:0] spi_clk_div: Number of clocks SCLK HIGH and LOW
        - [10:4] ce_delay: Number clocks between CE activations
        - [12:11] spi_mode: Per-CE FALLING SCLK edge data update
6. Set the number of DUMMY ready required for each CE:
    - 0x18: {8'h0, dummy1[3:0], dummy0[3:0]
7. For QSPI FLASH, set the QSPI Write opcode (it is different for various Flashes):
    - 0x19: {8'h0, quad_write_cmd}

NOTE: For register 0x1E (SPI Clock Div and CE Delay), there is only a single register, meaning this register value applies to both CE outputs. Delaying the clock of one CE will delay both, and adding delay between CE activations does not keep track of which CE was activated. So if two CE outputs are used and a CE delay is programmed, it will enforce that delay even if a different CE is used. This setting is really in place for use when the RP2040 emulation is being used in a single CE SRAM mode only (i.e. you have no external PMOD with a real SRAM / FLASH chip. In the case of real chips on a PMOD, SCLK and CE delays (most likely) are not needed. The Tech Page on the Tiny Tapeout regarding RP2040 SPI SRAM emulation indicates a delay between CE activations is likely needed, so this setting is provided in case it is needed.

Architecture Details

Below is a simplified block diagram of the LISA processor core. It uses an 8-bit accumulator for most of its operations with the 2nd argument predominately coming from either immediate data in the instruction word or from a memory location addressed by either the Stack Pointer (SP) or Index Register (IX).

There are also instructions that work on the 15-bit registers PC, SP, IX and RA (Return Address). As well as floating point operations. These will be covered in the sections to follow.

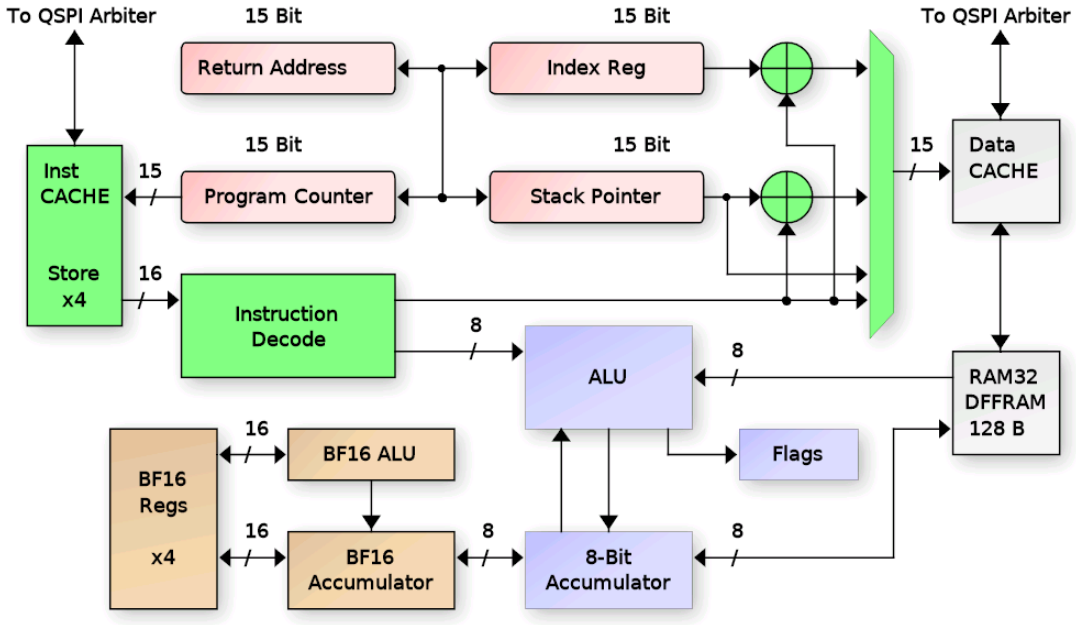


Figure 974.4: Simplified LISA Processor Block Diagram

Addressing Modes

Like most processors, LISA has a few different addressing modes to get data in and out of the core. These include the following:

Mode	Data	Description
Register	Rx[n -: 8]	Transfers between registers (ix, ra, facc, etc.).
Direct	inst[n:0]	N-bit data stored in the instruction word.
NextOp	(inst+1)[14:0]	Data stored in the NEXT instruction word.
Indirect	mem[inst[n:0]]	Address of the data is in the instruction word.
Periph	periph[inst[n:0]]	Accesses to the peripheral bus.
Indexed	mem[sp/ix+inst[n:0]]	The SP or IX register is added to a fixed offset.
Stack	mem[sp]	Stack pointer points to the data (push/pop).

The Control Registers

To run meaninful programs, the Program Counter (PC) and Stack Pointer (SP) must be set to useful values for accessing program instructions and data. The

PC is automatically reset to zero by rst_n, so that one is pretty much automatic. All programs start at address zero (plus any base address programmed by the Debug Controller). But as far as the LISA core is concerned, it knows nothing of base addresses and believes it is starting at address zero.

Next is to program the SP to point to a useful location in memory. The Stack is a place where C programs store their local variable values and also where we store the Return Address (RA) if we need to call nested routines, etc. The stack grows down, meaning it starts at a high RAM address and decrements as things are added to the stack. Therefore the SP should be programmed with an address in upper RAM. LISA supports different Data bus modes through it's CACHE controller, including CACHE disable where it can only access 128 bytes. But for this example, let's assume we have a full range of 32K SRAM available. The LISA ISA doesn't have an opcode for loading the SP directly. Instead it can load the IX register directly with a 15-bit value using NextOp addressing, and it supports "xchg" opcodes to exchange the IX register with either the SP or RA. So to load the SP, we would write:

```
Example:
  ldx     0x7FFF      // Load IX with value in next opcode
  xchg_sp             // Exchange IX with SP
```

The IX register can be programmed as needed to access other data within the Data Bus address range. This register is useful especially for accessing structures via a C pointer. The IX then becomes the value of the pointer to RAM, and Indexed addressing mode allows fixed offsets from that pointer (i.e. structure elements) to be accessed for read/write.

Loading the PC indirectly can be done using the "jmp ix" opcode which does the operation pc <= ix. Loading ix from the pc directly is not supported, though this can be accomplished using a function call and opcodes to save RA (sra) and pop ix:

```
 Example:
   get_pc:
     sra          // Push RA to the stack (Save RA)
     pop_ix       // Pop IX from the stack
     ret          // Return. Upon return, IX is the same as PC
```

Conditional Flow Processing

Program flow is controlled using flags (zero, carry, sign), arithemetic mode (amode) and condition flags (cond) to determine when program branches should occur. Specific opcode update the flags and condition registers based on results of the operation (AND, OR, IF, etc.). Then conditional branches are made using bz, bnz and if (and variants ifte "if-then-else" and iftt "if-then-then"). Also available are rc "Return if Carry" and rz "Return if Zero", though these are less useful in C programs as typically a routine uses local variables

and the stack must be restored prior to return, mandating a branch to the function epilog to restore the stack and often the return address. Below is a list of the opcodes used for conditional program processing:

Legend for operations below:

- acc_val = inst[7:0]
- pc_jmp = inst[14:0]
- pc_rel = pc + sign_extend(inst[10:0])

Opcode	Operation	Encoding	Description
jal	pc <= pc_jmp	0aaa_aaaa_aaaa_aaaa	Jump And Link (call).
—	ra <= pc	—	—
ret	pc <= ra	1000_1010_0xxx_xxxx	Return
reti	pc <= ra	1000_11xx_iiii_iiii	Return Immediate.
—	acc <= acc_val	—	—
br	pc <= pc_rel	1011_0rrr_rrrr_rrrr	Branch Always
bz	pc <= pc_rel	1011_1rrr_rrrr_rrrr	Branch if Zero.
—	if zero=1	—	—
bnz	pc <= pc_rel	1010_1rrr_rrrr_rrrr	Branch if Not Zero.
—	if zero=0	—	—
rc	pc <= ra	1000_1011_0xxx_xxxx	Return if Carry
—	if carry=1	—	—
rz	pc <= ra	1000_1011_1xxx_xxxx	Return if Zero
—	if zero=1	—	—
call_ix	pc <= ix	1000_1010_100x_xxxx	Call indirect via IX
—	ra <= pc	—	—
jump_ix	pc <= ix	1000_1010_101x_xxxx	Jump indirect via IX
if	cond <= ??	1010_0010_0000_0ccc	If. See below.
iftt	cond <= ??	1010_0010_0000_1ccc	If then-then. See below.
ifte	cond <= ??	1010_0010_0001_0ccc	If then-else. See below.

The IF Opcode

The “if” opcode and it’s variants “if-then-then” and “if-then-else” control program flow in a slightly different manner than the others. Instead of affecting the value of the PC directly, they set the two condition bits “cond[1:0]” to indicate which (if any) of the two following opcodes should be executed. the cond[0] bit represents the next instruction and cond[1] represents the instruction following that. All three “if” forms take an argument that checks

the current value of the FLAGS to set the condition bits. The argument is encoded as the lower three bits of the instruction word ard operate as shown in the following table:

Condition	Test	Encoding	Description
EQ	zflag=1	3'h0	Execute if Equal
NE	zflag=0	3'h1	Execute if Not Equal
NC	cflag=0	3'h2	Execute if Not Carry
C	cflag=1	3'h3	Execute if Carry
GT	cSigned & zflag	3'h4	Execute if Greater Than
LT	cSigned & zflag	3'h5	Execute if Less Than
GTE	cSigned	zflag	3'h6
LTE	cSigned	zflag	3'h7

The "if" opcode will set cond[0] based on the condition above and the cond[1] bit to HIGH. It only affects the single instruction following the "if" opcode. The "iftt" opcode will set both cond[0] and cond[1] to the same value based on the condition above. It means "if true, execute the next two opcodes". And the "ifte" opcode will set cond[0] based on the condition above and cond[1] to the OPPOSITE value, meaning it will execute either the following instruction OR the one after that (then-else).

```
Example:
  ldi      0x41        // Load A immediate with ASCII 'A'
  cpi      0x42        // Compare A immediate with ASCII 'B'
  ifte     eq          // Test if the compare was "Equal"
  jal      L_equal     // Jump if equal
  jal      L_different // Jump if different
```

The above code will load the "jal L_equal" opcode but will not execute it since the compare was Not Equal. Then it will execute the "jal L_different" opcode. Note that if the compare were "ifte ne", it would call the L_equal function and then upon return would not execute the "L_different" opcode. This is because the cond[1] code is saved with the Return Address (RA) during the call and restored upon return. This means the FALSE cond[1] code would prevent the 2nd opcode from executing. As an opcode gets executed, the cond[1] value is shifted into the cond[0] location, and the cond[1] is loaded with 1'b1.

Direct Operations

To do any useful work, the LISA core must be able to load and operate on data. This is done through the accumulator using the various addressing modes. The diagram below details the Direct addressing mode where data is stored directly in the opcode / instruction word:

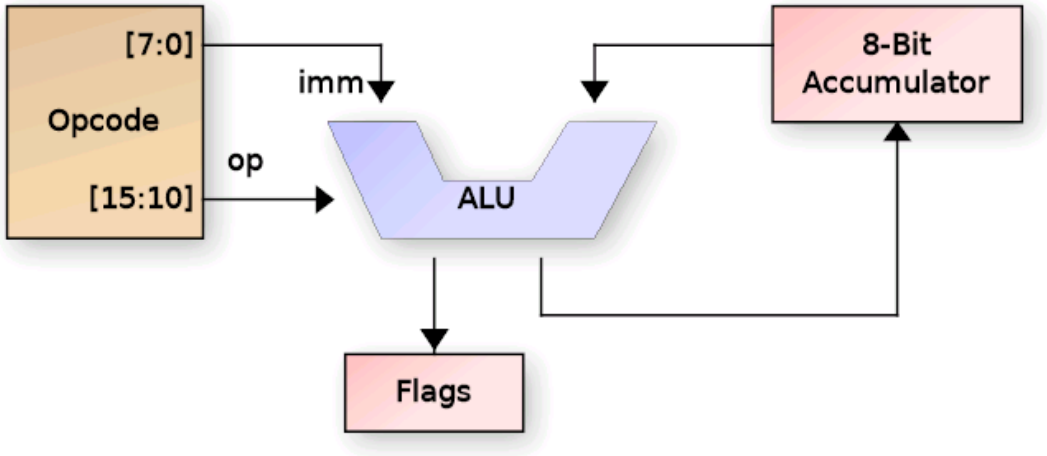


Figure 974.5: Accumulator Direct Operations Diagram

The instructions that use direct addressing are:

Opcode	Operation	Encoding	Description
adc	A <= A + imm + C	1001_00xx_iiii_iiii	ADD immediate with Carry
ads	SP <= SP + imm	1001_01ii_iiii_iiii	ADD SP + signed immediate
adx	IX <= IX + imm	1001_10ii_iiii_iiii	ADD IX + signed immediate
andi	A <= A & imm	1000_01xx_iiii_iiii	AND immediate with A
cpi	Z,C <= A >= imm	1010_01xx_iiii_iiii	Compare A >= immediate
cpi	Z,C <= A >= imm	1010_01xx_iiii_iiii	Compare A >= immediate

Accumulator Indirect Operations

The Accumulator Indirect operations use immediate data in the instruction word to index indirectly into Data memory. That memory address is then used to load, store or both load and store (swap) data with the accumulator.

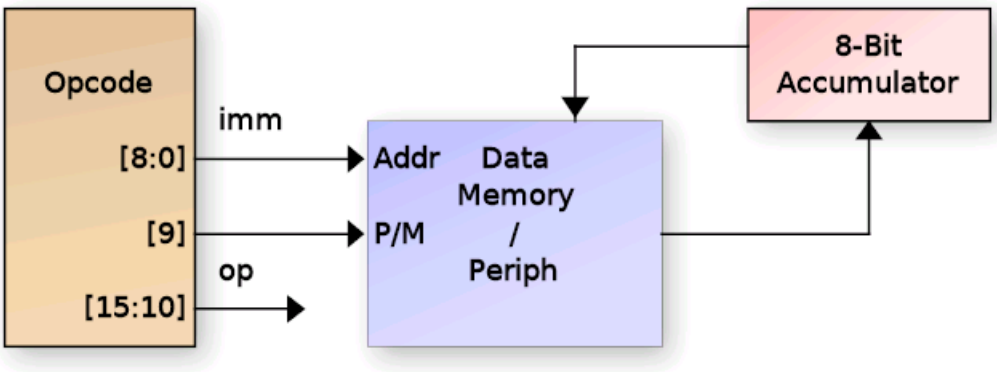


Figure 974.6: Accumulator Indirect Operations Diagram

Opcode	Operation	Encoding	Description
lda	A <= M[imm]	1111_01pi_iiii_iiii	Load A from Memory/Peripheral
sta	M[imm] <= A	1111_11pi_iiii_iiii	Store A to Memory/Peripheral
swapi	A <= M[imm]	1101_11pi_iiii_iiii	Swap Memory/Peripheral with A
—	M[imm] <= A	—	—

- p = Select Peripheral (1'b1) or RAM (1'b0)
- iiii = Immediate data

Indexed Operations

Indexed operations use either the IX or SP register plus a fixed offset from the immediate field of the opcode. The selection to use IX vs SP is also from the opcode[9] bit. The immediate field is not sign extended, so only positive direction indexing is supported. This was selected because this mode is typically used to access either local variables (when using SP) or C struct members (when using IX), and in both cases, negative index offsets aren't very useful. The following is a diagram of indexed addressing:

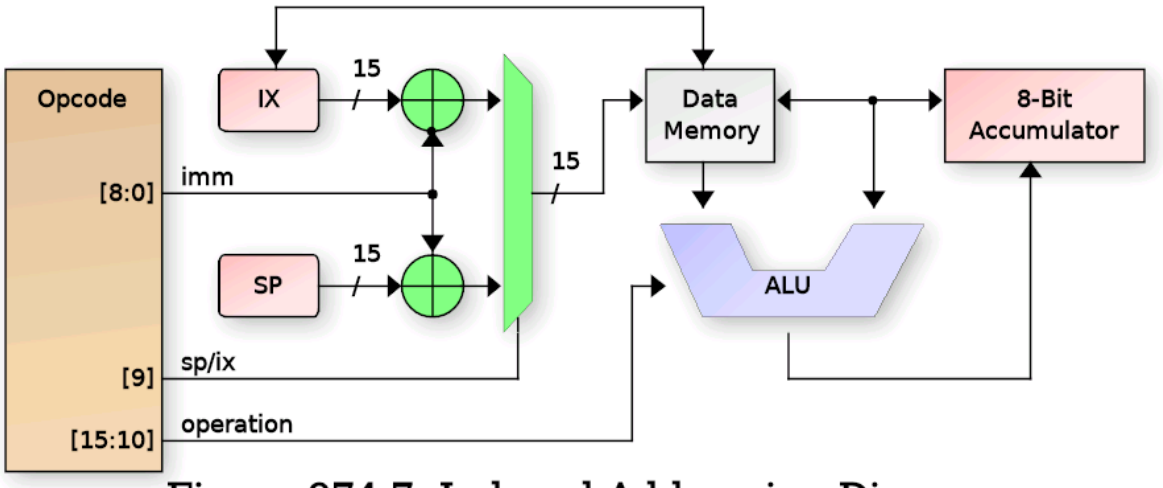


Figure 974.7: Indexed Addressing Diagram

Opcode	Operation	Encoding	Description
add	A <= A+ M[ind]	1100_00si_iiii_iiii	ADD index memory to A
and	A <= A & M[ind]	1101_00si_iiii_iiii	AND A with index memory
cmp	A >= M[ind]?	1110_10si_iiii_iiii	Compare A with index memory
dcx	M[ind] -= 1	1001_11si_iiii_iiii	Decrement the value at index memory
inx	M[ind] += 1	1110_01si_iiii_iiii	Increment the value at index memory

ldax	A <= M[ind]	1111_00si_iiii_iiii	Load A from index memory
ldxx	IX <= M[SP+imm]	1100_110i_iiii_iiii	Load IX from memory at SP+imm
mul	A <= A*M[ind]L	1100_10si_iiii_iiii	Multiply index memory * A, keep LSB
mulu	A <= A*M[ind]H	1000_01si_iiii_iiii	Multiply index memory * A, keep MSB
or	A <= A	M[ind]	1101_10si_iiii_iiii
stax	M[ind] <= A	1111_10si_iiii_iiii	Store A to index memory
stxx	M[SP+imm] <= IX	1100_111i_iiii_iiii	Save IX to memory at SP+imm
sub	A <= A-M[ind]	1100_10si_iiii_iiii	SUBtract index memory from A
swap	A <= M[ind]	1110_11si_iiii_iiii	Swap A with index memory
—	M[ind] <= A	—	—
xor	A <= A ^ M[ind]	1110_00si_iiii_iiii	XOR A with index memory

Legend for table above:

- ind = IX or SP + immediate
- s = Select IX (zero) or SP (one)
- iiii = Immediate data

The Zero and Carry flags are updated for most of the above operations. The Carry flag is only updated for math operations where a Carry / Borrow could occur.

Carry	Zero
adc	add and
add	or xor
sub	cmp sub
cmp	dcx inx
dcx	swap ldax
inx	mul mulu

Stack Operations

Stack operations use the current value of the SP register to PUSH and POP items to the stack in opcode. As items are PUSHed to the stack, the SP is

decremented after each byte, and as they are POPed, the SP is incremented prior to reading from RAM.

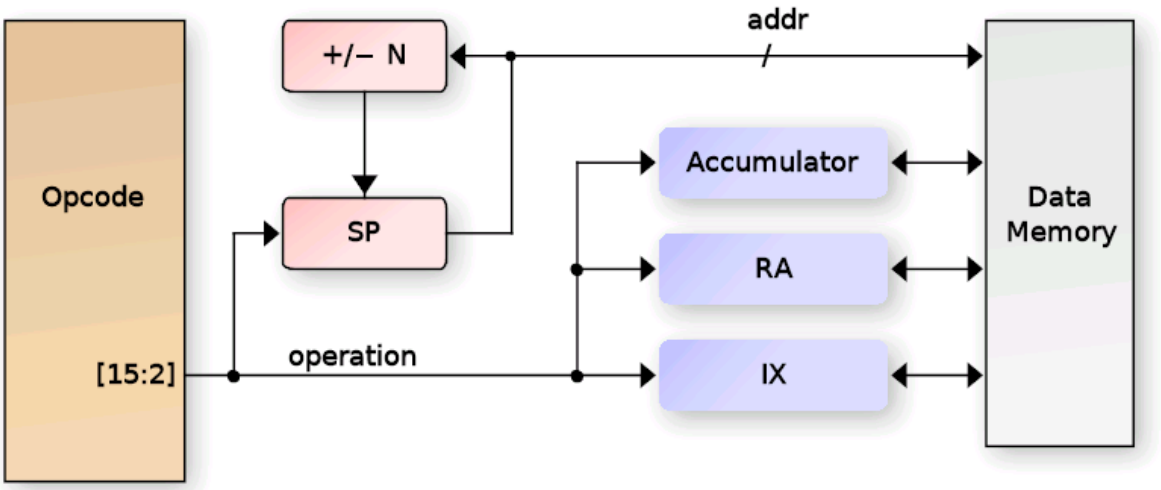


Figure 974.8: Stack Addressing Diagram

Opcode	Operation	Encoding	Description
lra	RA <= M[SP+1]	1010_0001_0110_01xx	Load {cond,RA} from stack
—	SP += 2	—	—
sra	M[SP] <= RA	1010_0001_0110_00xx	Save {cond,RA} to stack
—	SP -= 2	—	—
push_ix	M[SP] <= IX	1010_0001_0110_10xx	Save IX to stack
—	SP -= 2	—	—
pop_ix	IX <= M[SP+1]	1010_0001_0110_11xx	Load IX from stack
—	SP += 2	—	—
push_a	M[SP] <= A	1010_0000_100x_xxxx	Save A to stack
—	SP -= 1	—	—
pop_a	A <= M[SP+1]	1010_0000_110x_xxxx	Load A from stack
—	SP += 1	—	—

How to test

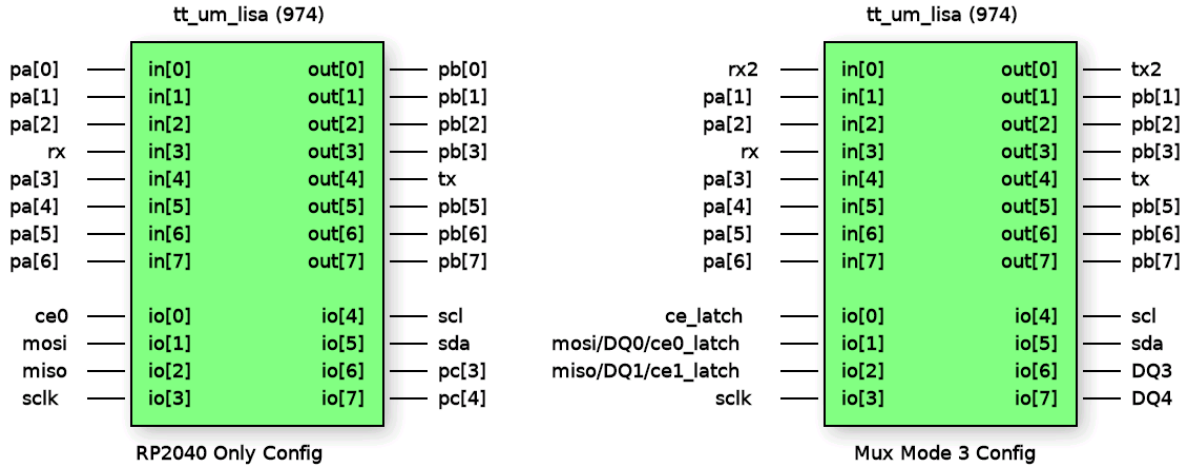
You will need to download and compile the C-based assembler, linker and C compiler I wrote (will make available) Also need to download the Python based debugger.

- Assembler is fully functional
  - Includes limited libraries for crt0, signed int compare, math, etc.

    - Libraries are still a work in progress
- Linker is fully functional
- C compiler is somewhat functional (no float support at the moment) but has *many* bugs in the generated code and is still a work in progress.
- Python debugger can erase/program the FLASH, program SPI SRAM, start/stop the LISA core, read SRAM and registers.

Legend for Pinout

- pa: LISA GPIO PortA Input
- pb: LISA GPIO PortB Output
- b_div: Debug UART baud divisor sampled at reset
- b_set: Debug UART baud divisor enable (HIGH) sampled at reset
- baud_clk: 16x Baud Rate clock used for Debug UART baud rate generator
- ce_latch: Latch enable for Special Mux Mode 3 as describe above
- ce0_latch: CE0 output during Special Mux Mode 3
- ce1_latch: CE1 output during Special Mux Mode 3
- DQ1/2/3/4: QUAD SPI bidirection data I/O
- pc_io: LISA GPIO Port C I/O (direction controllable by LISA)



Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	pa[0]/b_div[0]/rx2	pb[0]/tx2	ce0/ce_latch
1	pa[1]/b_div[1]/rx2	pb[1]/tx2	mosi/dq1/ce0_latch
2	pa[2]/b_div[2]/rx2	pb[2]/tx2	miso/dq2/ce1_latch
3	pa[3]/b_div[3]/rx	pb[3]	sclk
4	pa[4]/b_div[4]	pb[4]/tx	rx /pc_io[0]/scl/ce1
5	pa[5]/b_div[5]	pb[5]	tx /pc_io[1]/sda
6	pa[6]/b_div[6]	pb[6]	scl /pc_io[2]/dq2/rx

#	Input	Output	Bidirectional
7	pa[7]/b_set(autobaud_disable)	pb[7]/baud_clk	sda/pc_io[3]/dq3

Asicle v2

by **htfab**

0330 | 25.175 MHz | HDL Project

`github.com/htfab/ttihp0p4-asicle2`

"Wordle clone in raw silicon"

How it works

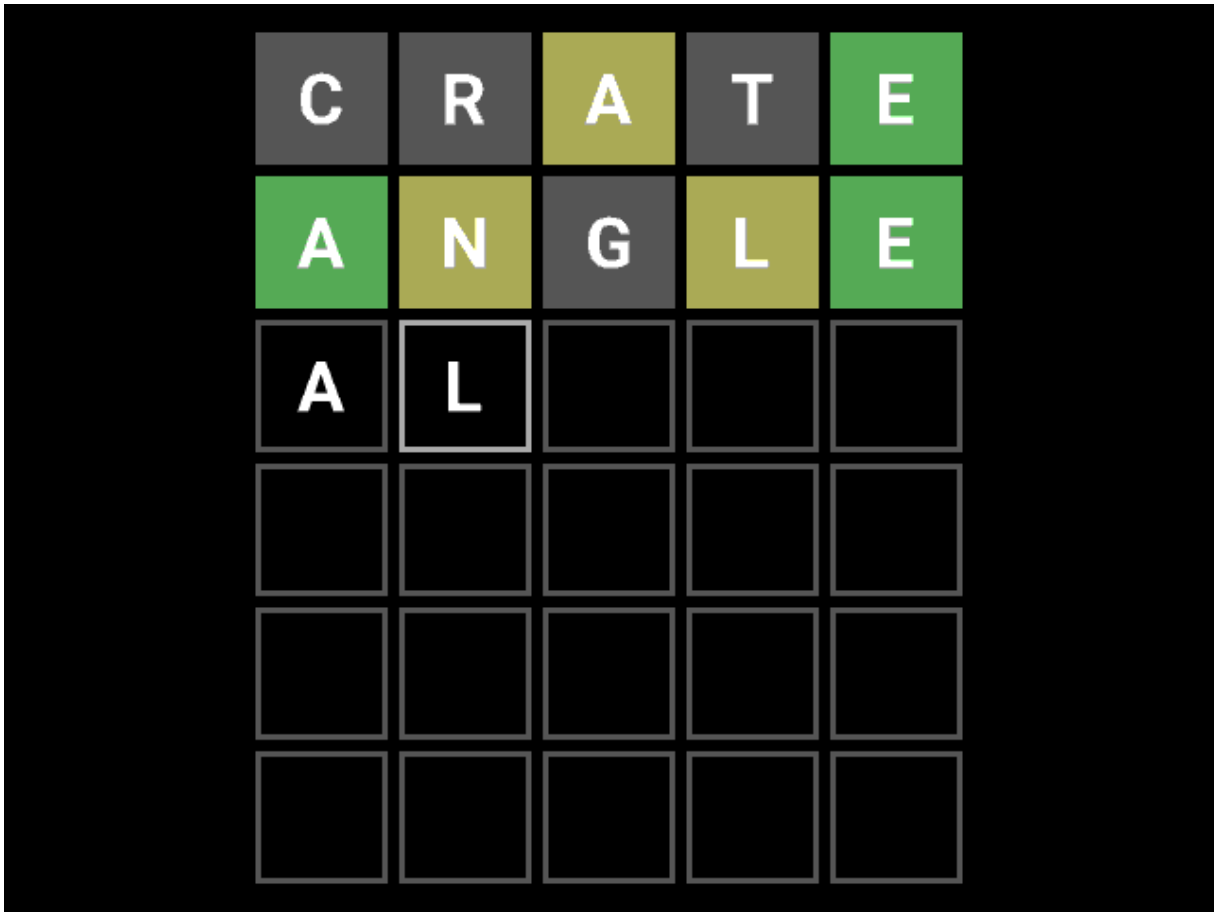


Figure 330.1: VGA screenshot with questions CRATE and ANGLE

Asicle is a Wordle clone implemented directly in hardware. A first version of it was taped out on the Google-Skywater MPW6 shuttle. This second version is a rewrite for Tiny Tapeout.

The 25-fold decrease in area mostly comes from moving the word list and font bitmaps to external flash on the QSPI Pmod, with some architectural changes to compensate for slower memory access. The design was also adapted to the Tiny Tapeout ecosystem by using the Gamepad Pmod for input and the Tiny VGA Pmod for output.

How to test

- Connect the Pmods:
  - Gamepad to input port (optional, you can also drive the input pins using the commander app or momentary push buttons)
  - QSPI to bidirectional port
  - Tiny VGA to output port
- Flash the data file to the QSPI Pmod using the Tiny Tapeout flasher
- Select the design
- Set the clock to 25.175 MHz and reset the design
- Play the game

If you haven't played Wordle before, the aim is to guess a five-letter English word in six attempts. Each time you get feedback: a green square indicates that the letter is correct, a yellow square indicates that it appears in the hidden word but at a different position, and a grey square means that the letter doesn't appear in the solution at all.

Gamepad controls:

- ↑ ↓: change the letter in the selected position
- ← →: move the selection
- A: make a guess
- START: start a new game (if the current one is finished)
- SELECT+START: start a new game (any time)
- SELECT+X: show the solution*
- SELECT+Y: re-roll the solution*

(* only in debug mode)

Direct input using `ui_in`:

- 0 1: change the letter in the selected position
- 2 3: move the selection
- 4: make a guess
- 5: start a new game
- 6: show the solution
- 7: re-roll the solution

External hardware

- Tiny VGA Pmod
- QSPI Pmod
- Gamepad Pmod (optional)

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	up	tinyvga: r1	qspi: cs0 (flash)
1	down	tinyvga: g1	qspi: sd0/mosi
2	left	tinyvga: b1	qspi: sd1/miso
3	right	tinyvga: vsync	qspi: sck
4	guess / gamepad: latch	tinyvga: r0	qspi: sd2
5	new game / gamepad: clock	tinyvga: g0	qspi: sd3
6	peek (debug) / gamepad: data	tinyvga: b0	qspi: cs1 (unused)
7	roll (debug)	tinyvga: hsync	qspi: cs2 (unused)

SotaSoC

by **SotaTek**

`490` `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.

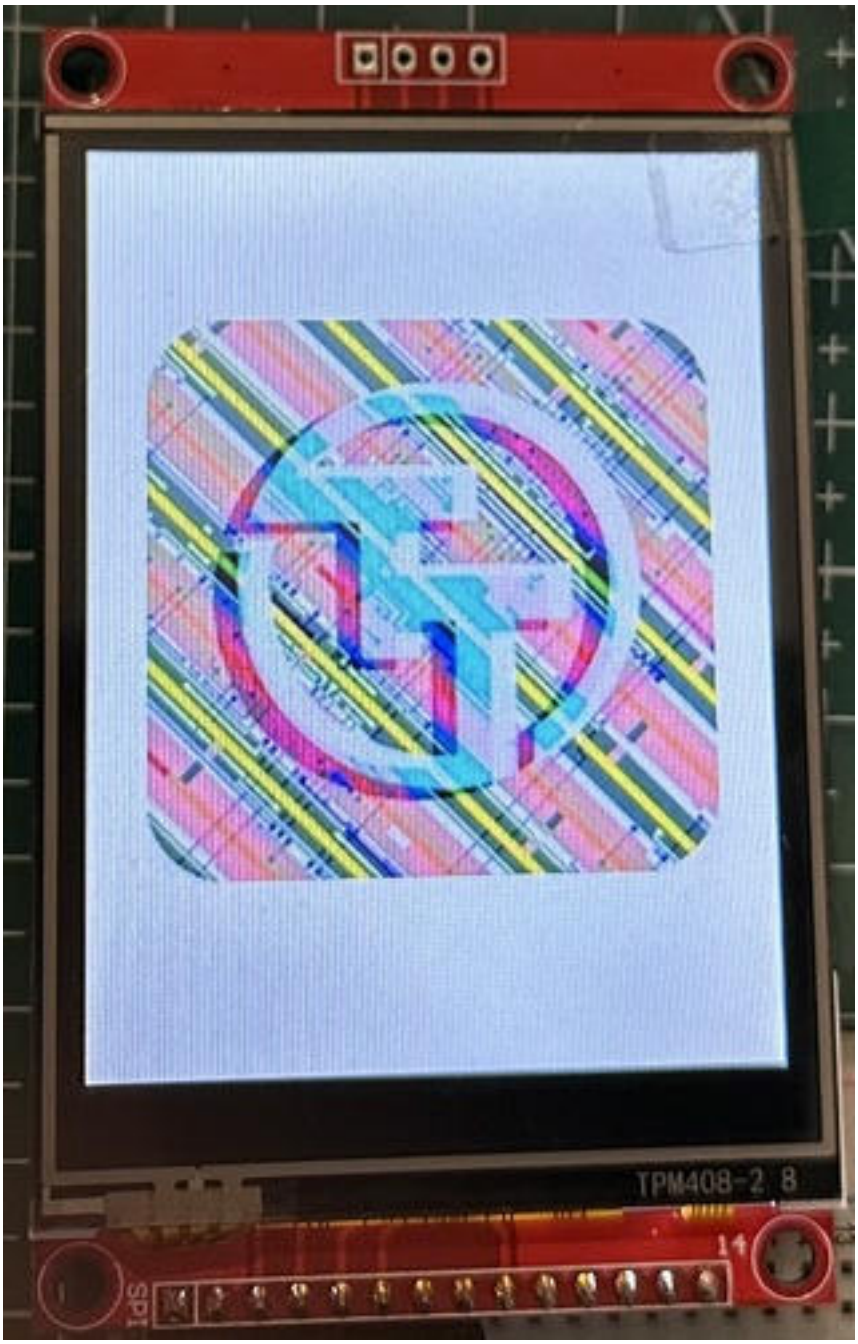


Figure 490.1: LCD demo (320×240 ST7789)

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:

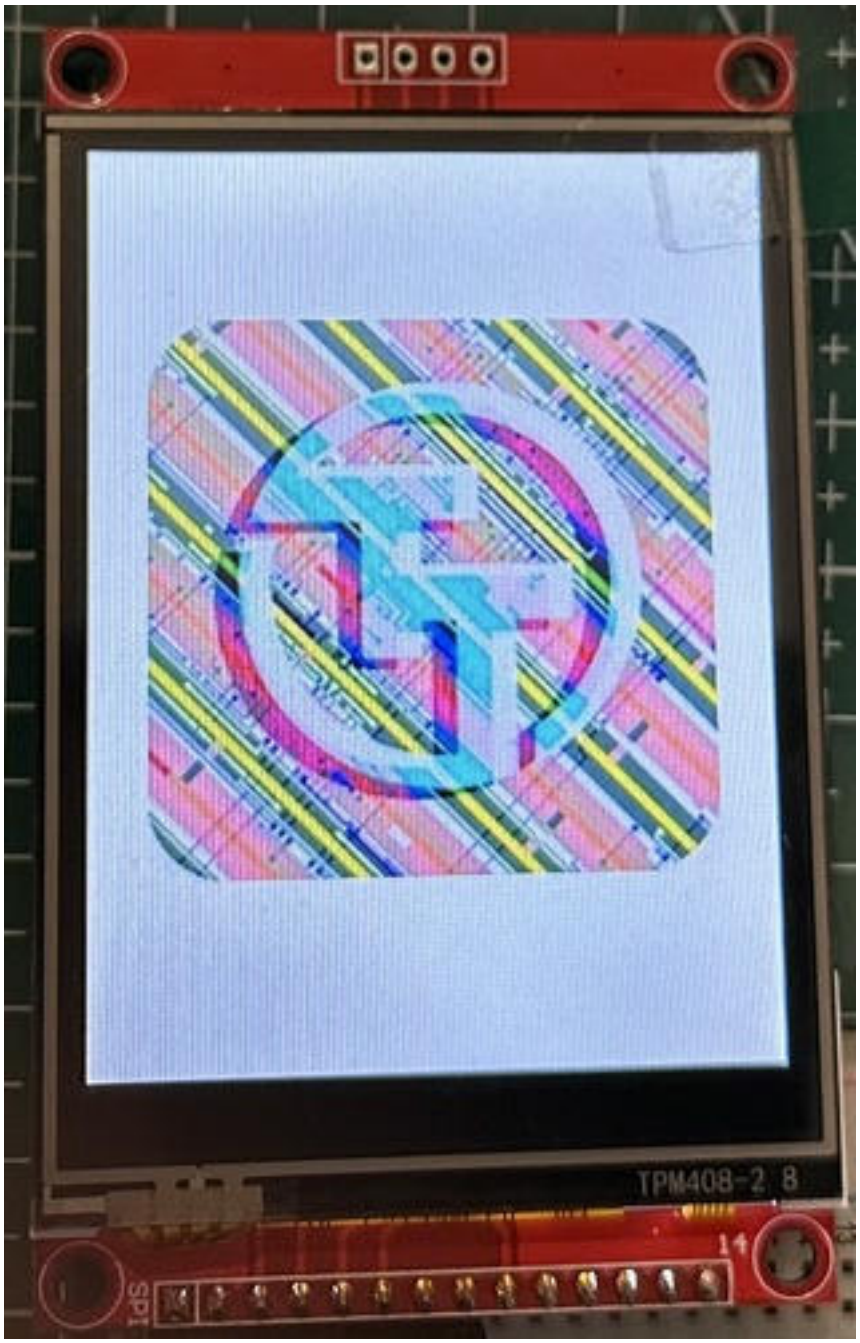


Figure 490.2: LCD demo (320×240 ST7789)

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

Photo Frame

by **Mike Bell**

513 | 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

TinyScanChain5L

by **Yann Guidon**

515 | 50 MHz | HDL Project

`github.com/ygdes/TinyScanChain5L`

"Low footprint scan chain for iHP PDK / SG13CMOS5L"

This is a port of the project https://github.com/YannGuidon/TinyScanChain to IHP SG13CMOS5L PDK, before I restart the DTAP interface project (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)

.

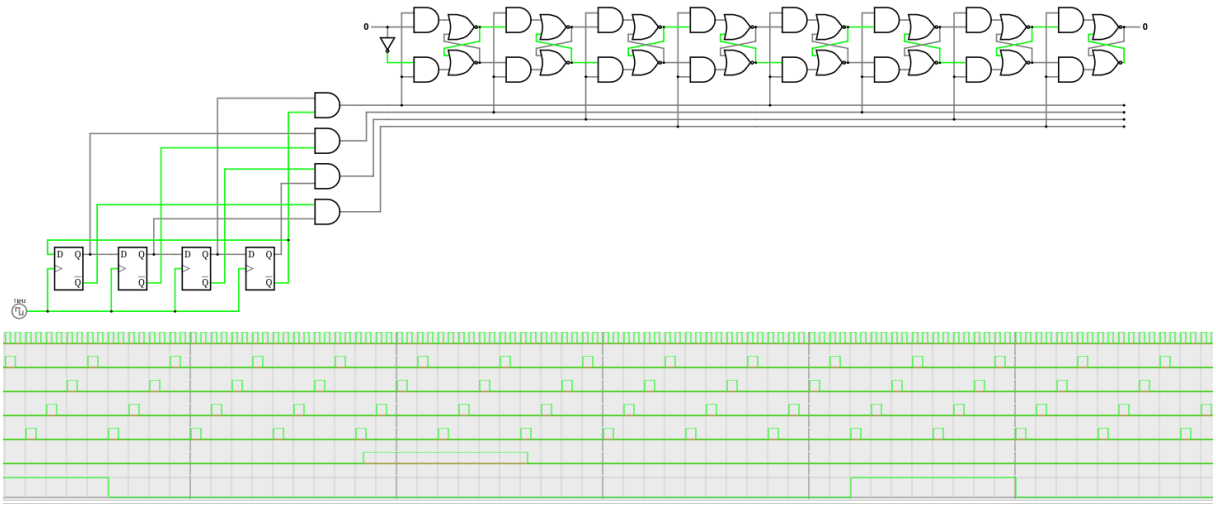
Resources

- 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.

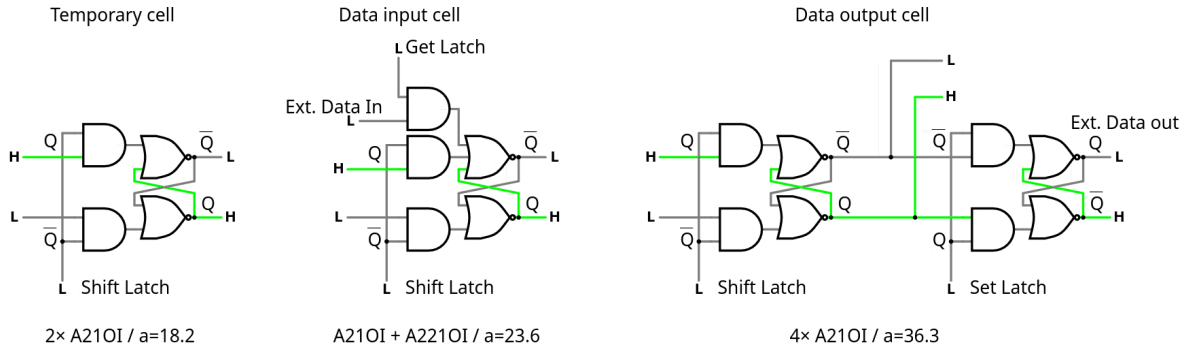
How it works

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

2048 sliding tile puzzle game (VGA)

by **Uri Shaked**

`516` `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

ttihp-HDSISO8RS

by **Yann Guidon**

`0517` `50 MHz` `HDL Project`

`github.com/YannGuidon/ttihp-HDSISO8RS_5L`

"Higher density Shift register - RS version"

What is is

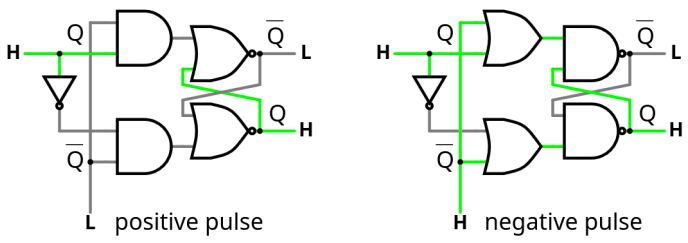
This tile delays a bit's value by 768+23=791 cycles at speeds above 20MHz (according to the synthesiser, but that's conservative). It is an attempt to beat storage packing density, as well as a test architecture for asynchronous shift registers, not made out of the larger DFF cells. This version packs 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! This version is a port to the experimental IHP SG13CMOS5L PDK.

How it works

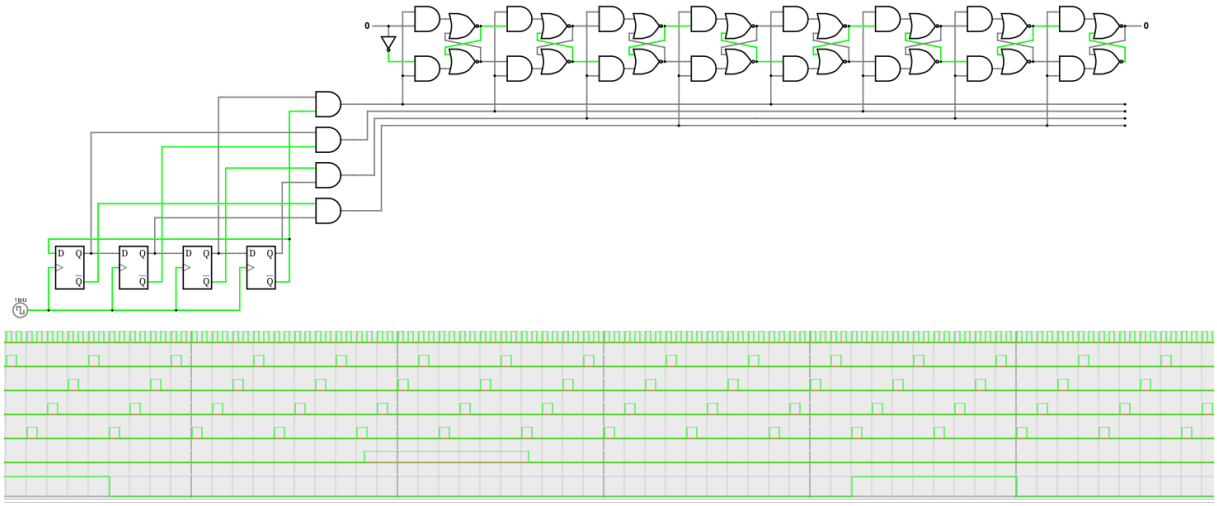
As the name implies, it's a high density shift register for deep digital delays. According to the PDK for CMOS IHP at https://github.com/IHP-GmbH/IHP-Open-PDK/blob/main/ihp-sg13g2/libs.ref/sg13g2_stdcell/doc/sg13g2_stdcell_typ_1p20V_25C.pdf

- 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.

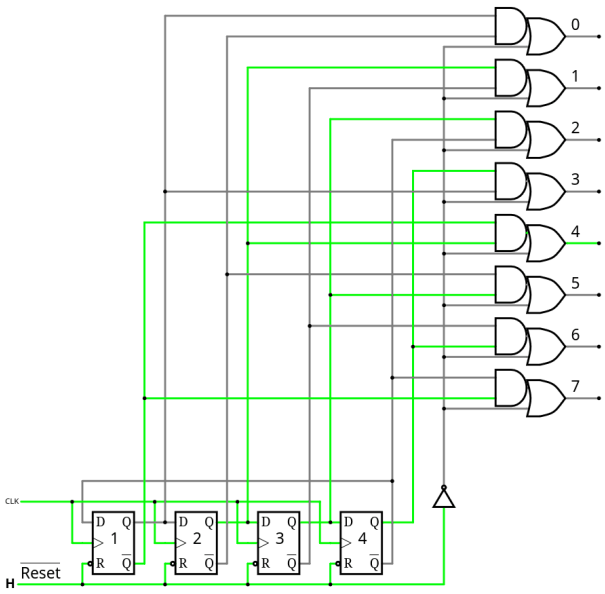
- External reset pin EXT_RST (asserted at 0 like the internal one) overrides the internal reset, don't let it float. A weak pull-up to 1 is advised.
- External clock (pin EXT_CLK) can be selected when pin CLK_SEL=1 (don't let them float).
- Always assert EXT_RST (to 0) while changing the state of CLK_SEL.

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

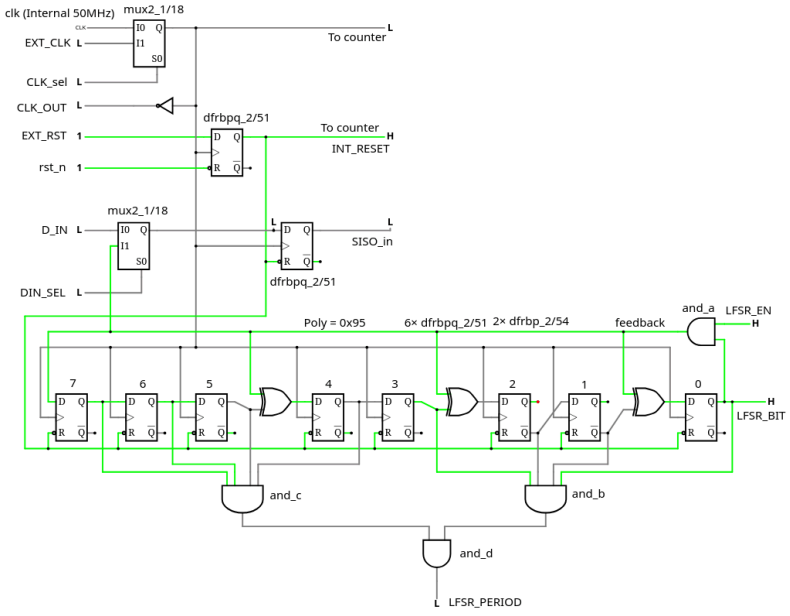
An 8-bit LFSR is integrated to ease testing. Thus an oscilloscope and a variable frequency oscillator are enough to characterise the achieveable speed. To use it,

- 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

ttihp-HDSISO8

by **Yann Guidon**

0519 | 50 MHz | HDL Project

`github.com/YannGuidon/ttihp-HDSISO8_5L`

"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. This version is a port to the experimental IHP SG13CMOS5L PDK.

How it works

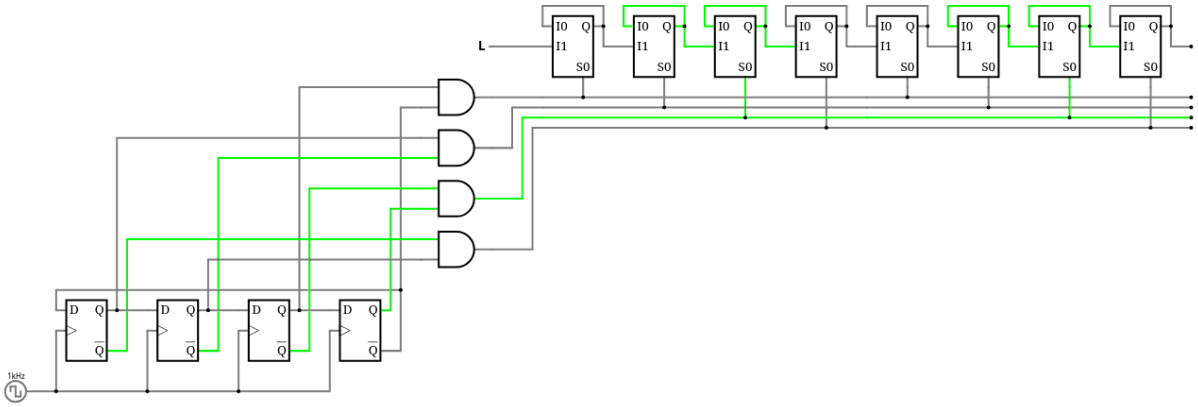
As the name implies, it's a high density shift register for deep digital delays. According to the PDK for CMOS IHP at https://github.com/IHP-GmbH/IHP-Open-PDK/blob/main/ihp-sg13g2/libs.ref/sg13g2_stdcell/doc/sg13g2_stdcell_typ_1p20V_25C.pdf

- 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:

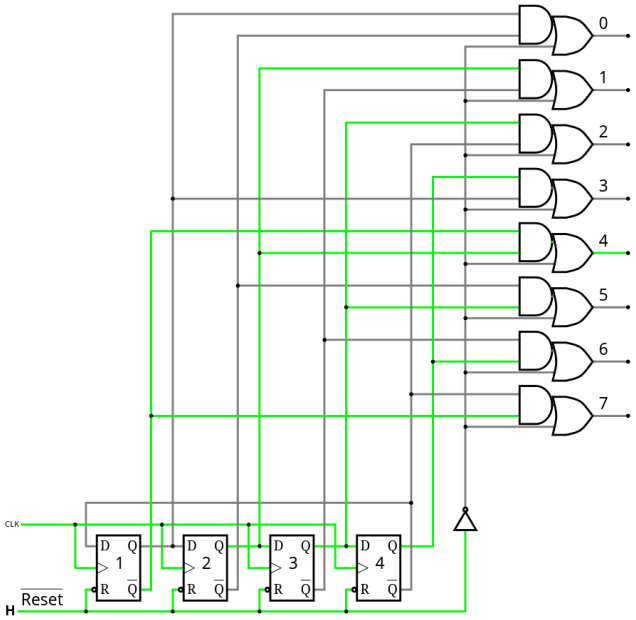
- Clock and Reset can be asserted by external pins and internal signals.
- The pin CLK_OUT copies the currently selected clock (negated), for external triggering and troubleshooting. If it oscillates, you’re good.
- External reset pin EXT_RST (asserted at 0 like the internal one) overrides the internal 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

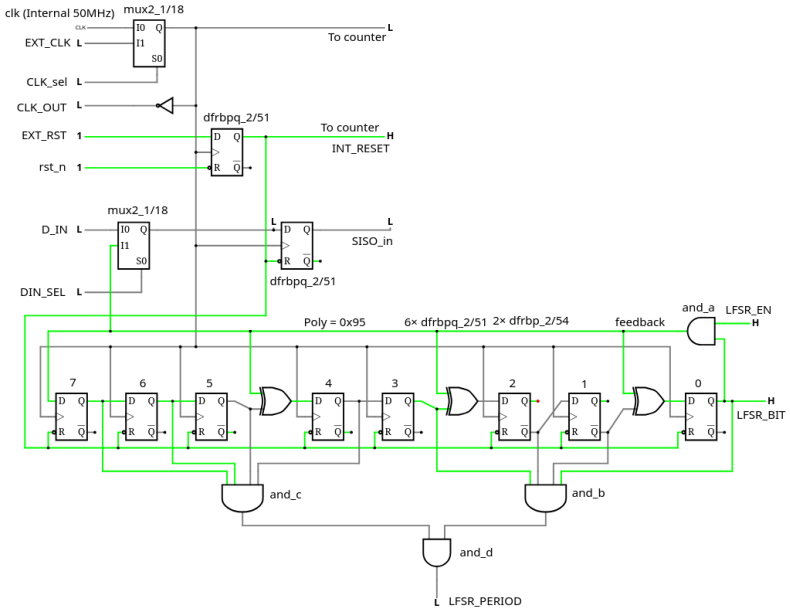
An 8-bit LFSR is integrated to ease testing. Thus an oscilloscope and a variable frequency oscillator are enough to characterise the achieveable speed. To use it,

- 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

microlane demo project

by **htfab**

`0521` `5 Hz` `HDL Project`

`github.com/htfab/ttihp0p4-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	—	—	—

SIC-1 8-bit SUBLEQ Single Instruction Computer

by **Uri Shaked**

522 HDL Project

`github.com/urish/tt-subleq-sic1`

"*Hardware implementation of the 8-bit Single Instruction Computer*"

How it works

SIC-1 is an 8-bit Single Instruction computer. The only instruction it supports is SUBLEQ: Subtract and Branch if Less than or Equal to Zero. The instruction has three operands: A, B, and C. The instruction subtracts the value at address B from the value at address A and stores the result at address A. If the result is less than or equal to zero, the instruction jumps to address C. Otherwise, it proceeds to the next instruction.

Memory map

The SIC-1 computer has an address space of 256 bytes, and and 8-bit program counter. The first 253 bytes are used for the program memory, and the last 3 bytes are used for input, output, and for halting the computer:

Address	Label	Read	Write
253	@IN	`ui` pins	Ignored
254	@OUT	Returns 0	`uo` pins
255	@HALT	Returns 0	Ignored

Setting the program counter to 253, 254, or 255 will halt the computer.

Each instruction is 3 bytes long, and the program counter is incremented by 3 after each instruction, except when a branch is taken.

For more information, check out the SIC-1 Assembly Language Reference.

Execution cycle

Each instruction takes two cycles to execute, regardless of whether a branch is taken or not. The execution of an instruction is divided into the following stages:

1. a. Write result: writes the result of the previous instruction back to memory. b. Read instruction: reads A, B, and C from the memory at the address pointed to by the program counter (PC).

2. Read data: Reads the values at addresses A and B and calculate the result of valA - valB. If the result is less than or equal to zero, set the PC to C. Otherwise, increment the PC by 3 to point to the next instruction.

The pseudocode for the execution cycle is as follows:

```
(1) result = valA - valB
    next_PC = result <= 0 ? C : PC + 3
    memory[A] <= result
    A <= memory[next_PC]
    B <= memory[next_PC+1]
    C <= memory[next_PC+2]
    PC <= next_PC
(2) valA <= memory[A]
    valB <= memory[B]
```

Where `valA` and `valB` are internal registers that hold the values read from memory at addresses A and B, respectively.

The <= symbol indicates a memory or register write operation, while the = symbol indicates a combinational assignment.

Control signals

The `uio` pins are used to load a program into the computer, and to control the computer:

uio pin	Name	Type	Description
0	run	input	Start the computer
1	halted	output	Computer has halted
2	set_pc	input	Set the program counter to the value on ui pins
3	load_data	input	Load the value from the ui pins into the memory addressed by PC
4	out_strobe	output	Pulsed for one clock cycle when the computer writes to @OUT (`uo` pins)
5	dbg[0]	input	Debug select bit 0
6	dbg[1]	input	Debug select bit 1
7	dbg[2]	input	Debug select bit 2

Debug interface

The `dbg` pins are used to expose internal signals for debugging on the `uo` pins. When the `dbg` pins are set to 0, the `uo` pins will output the @OUT value. For other values of `dbg`, the `uo` pins will output the following signals:

dbg[2:0]	Signal

0	None
1	PC
2	A
3	B
4	C
5	valA
6	result (valA - valB)
7	state (2 bits)

The `state` signal is a 2-bit value that represents the current state of the computer, corresponding to the execution cycle stages described above (1 and 2), and a 2-bit value of 0 when the computer is halted.

Programming the SIC-1

You can use the online SIC-1 app to compile and simulate your SIC-1 programs. Click on "Run game" and then "Apply for the job", close the "Electronic mail" popup. Paste the code and click on "Compile" (on the bottom left). You'll see the compiled code in the "Memory" window on the right, and will be able to step through the code.

To load a program and run a program, follow this sequence:

1. Set the `ui` pins to 0 (target address)
2. Pulse the the `load_pc` pin for a single clock cycle
3. Set the `ui` pins to the value you want to load
4. Pulse the `load_data` pin for a single clock cycle
5. Repeat steps 3-4 until you have loaded the entire program
6. Set the `ui` pins to the address you want to start at (usually 0)
7. Pulse the `set_pc` pin for a single clock cycle
8. Set the `run` pin to 1. The computer will start running the program, and the `halted` pin will go high when the program is done.

If you want to step through the program, you can pulse the `run` pin to advance one instruction at a time.

Reading the internal memory

You can read the internal memory of the SIC-1 while the program is halted. To do this, follow these steps:

1. Set the `run` pin to 0 to halt the computer, and wait for the `halted` pin to go high.
2. Set the `ui` pins to the address you want to read.
3. Pulse the the `load_pc` pin for a single clock cycle.
4. Set the `dbg` pins to `010` (binary for 2) to select the "A" debug signal.

5. Read the value on the `uo` pins.

The value on the `uo` pins will be the value stored in the internal memory at the address you specified.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	in[0]	out[0]	run
1	in[1]	out[1]	halted
2	in[2]	out[2]	set_pc
3	in[3]	out[3]	load_data
4	in[4]	out[4]	out_strobe
5	in[5]	out[5]	dbg[0]
6	in[6]	out[6]	dbg[1]
7	in[7]	out[7]	dbg[2]

Simon Says memory game

by **Uri Shaked**

`523` `50 kHz` `HDL Project`

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

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

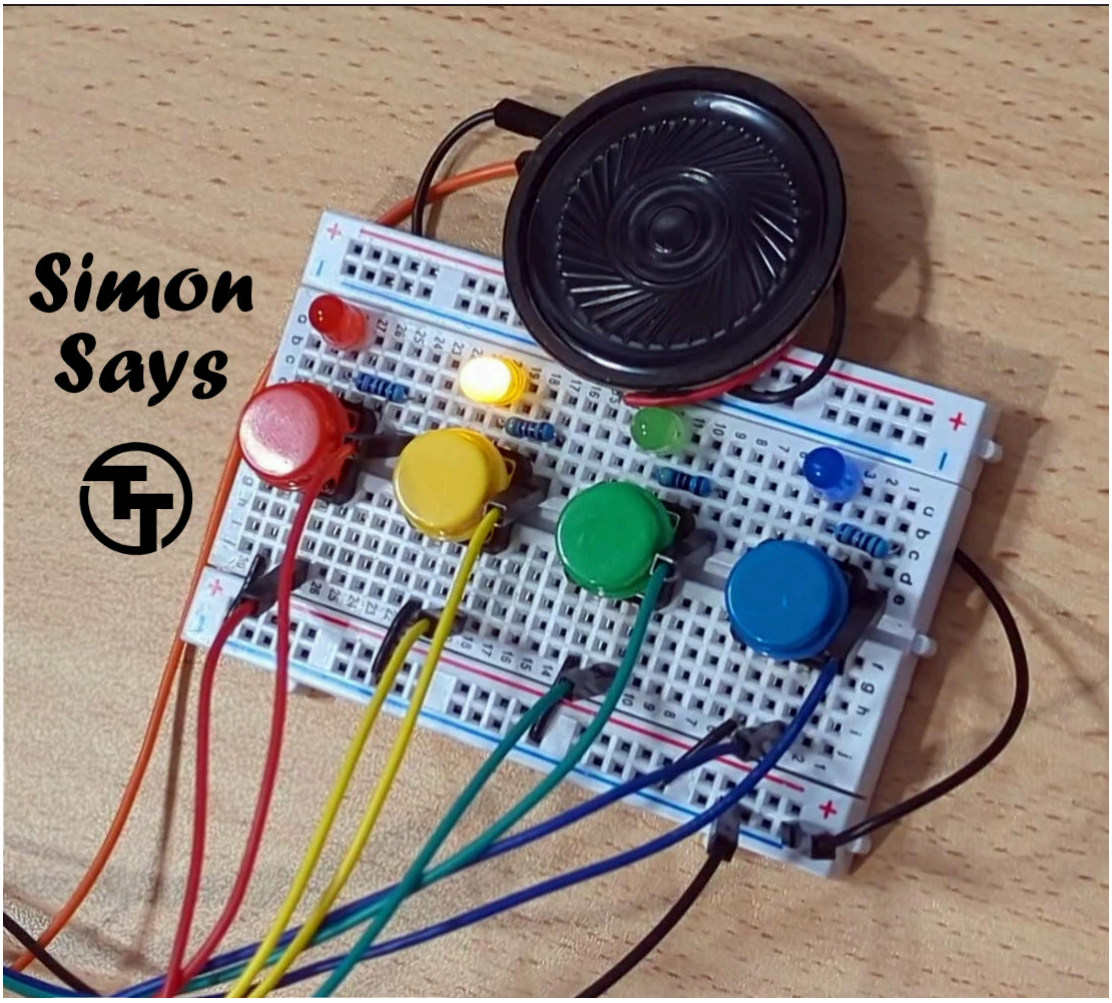


Figure 523.1: Simon Says Game

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	—

7-Segment Digital Desk Clock

by **Samuel Ellicott**

0577 | 50 MHz | HDL Project

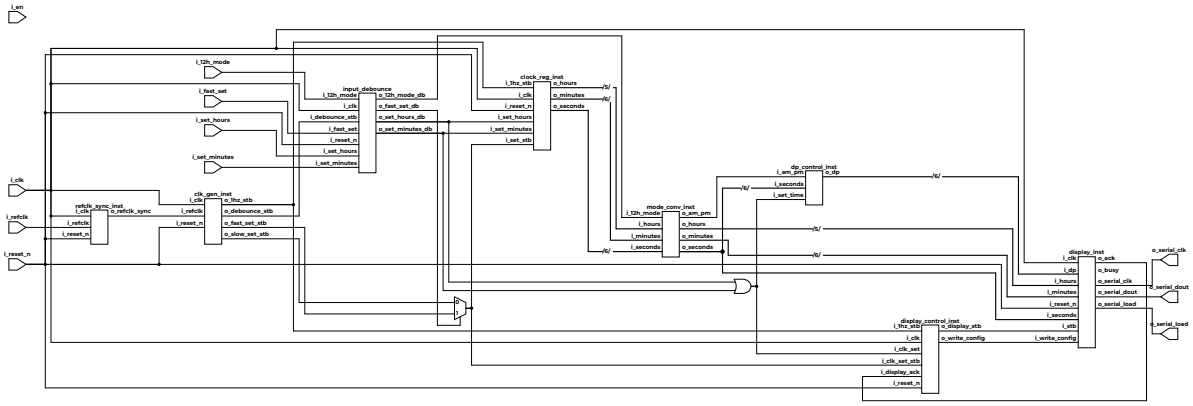
`github.com/sellicott/digital_clock_ihp03`

"7-Segment Desk Clock"

How it works

Simple digital clock, displays hours, minutes, and seconds in either a 24h format. Since there are not enough output pins to directly drive a 6x 7-segment displays, the data is shifted out over SPI to a MAX7219 in 7-segment mode. The time can be set using the `hours_set` and `minutes_set` inputs. If `set_fast` is high, then the the hours or minutes will be incremented at a rate of 5Hz, otherwise it will be set at a rate of 2Hz. Note that when setting either the minutes, rolling-over will not affect the hours setting. If both `hours_set` and `minutes_set` are presssed at the same time the seconds will be cleared to zero.

A block diagram of the system is shown below.



How to test

Apply a 5MHz clock to the clock pin and 32.786Khz signal to the refclk pin. Use the `hours_set` and `minutes_set` pins to set the time.

External hardware

Connect the BIDIR PMOD to a MAX7219 7-segment display, For reference Tiny Tapeout SPI

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	refclk	—	Display CS
1	—	—	Display MOSI
2	Fast/Slow Set	—	—
3	Set Hours	—	Display SCK
4	Set Minutes	—	—
5	12-Hour Mode	—	—
6	—	—	—
7	—	—	—

Glyph Mode HD

by **James Ross**

0579 | 25.175 MHz | HDL Project

`github.com/jar/ttihp0p4_glyph_mode_hd`

"*Improved Matrix digital rain animation with 4 VGA modes*"

How it Works

This is a standalone VGA demo that runs with or without input, replicating *The Matrix* Digital Rain effect.

Figure 579.1: example VGA output

Upon circuit reset, the glyphs will appear to fall from the top of the screen.

You can change the palette with the two pins, `ui_io[0]` and `ui_io[1]`.

NOTE The default VGA timing requires a pixel clock of 25.175 MHz. If you want to drive higher resolutions, the base clock rate must be adjusted accordingly with the Clocks table below. You must also set the two pins, `ui_io[6]` and `ui_io[7]`, to select your preferred mode.

How to Test

Plug into a VGA monitor and select this circuit to test. By default, the circuit must be clocked at (or very near) to **25.175 MHz**. There are four VGA timing modes, representing four different display resolutions, which must be both specifically clocked *and* have the pins `ui_io[7:6]` set according to the following table.

Clocks

Pins 6 and 7 paired with pixel clock

`ui_io[7:6]`	Clock (MHz)	VGA Timing Mode
(default) 0	25.175	640 x 480 @ 60 fps (VGA)
1	40.0	800 x 600 @ 60 fps (SVGA)

2	34.694	960 x 540 @ 60 fps (qHD)
3	64.0	1280 x 720 @ 60 fps (HD)

Additional Palette Input

The circuit accepts two pins `ui_io[0]` and `ui_io[1]` for palette selection:

`ui_io[1:0]`	Palette
(default) 0	Green
1	Red
2	Blue
3	Pride

External hardware

Requires the TinyVGA PMOD

Project Pinout

Digital Pins

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

Fast bfloat multiplication

by **Julia Desmazes**

0581 454.545 kHz HDL Project

`github.com/Essenceia/uselessly_fast_bfloat16_multiplier`

"*Uselessly fast bfloat16 multiplication*"

How it works

Trust me bro (but do sent data 4 bytes of the 2 values to multiply over 4 cycles with valid active)

How to test

Also trust me bro

External hardware

None

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	data_i[0]	res_o[0]	data_v_i
1	data_i[1]	res_o[1]	—
2	data_i[2]	res_o[2]	—
3	data_i[3]	res_o[3]	—
4	data_i[4]	res_o[4]	—
5	data_i[5]	res_o[5]	—
6	data_i[6]	res_o[6]	res_v_o
7	data_i[7]	res_o[7]	res_v_early_o

Register bank accessible through SPI and I2C

by **Caio Alonso da Costa**

583 | 50 MHz | HDL Project

github.com/calonso88/dtu_tt_workshop

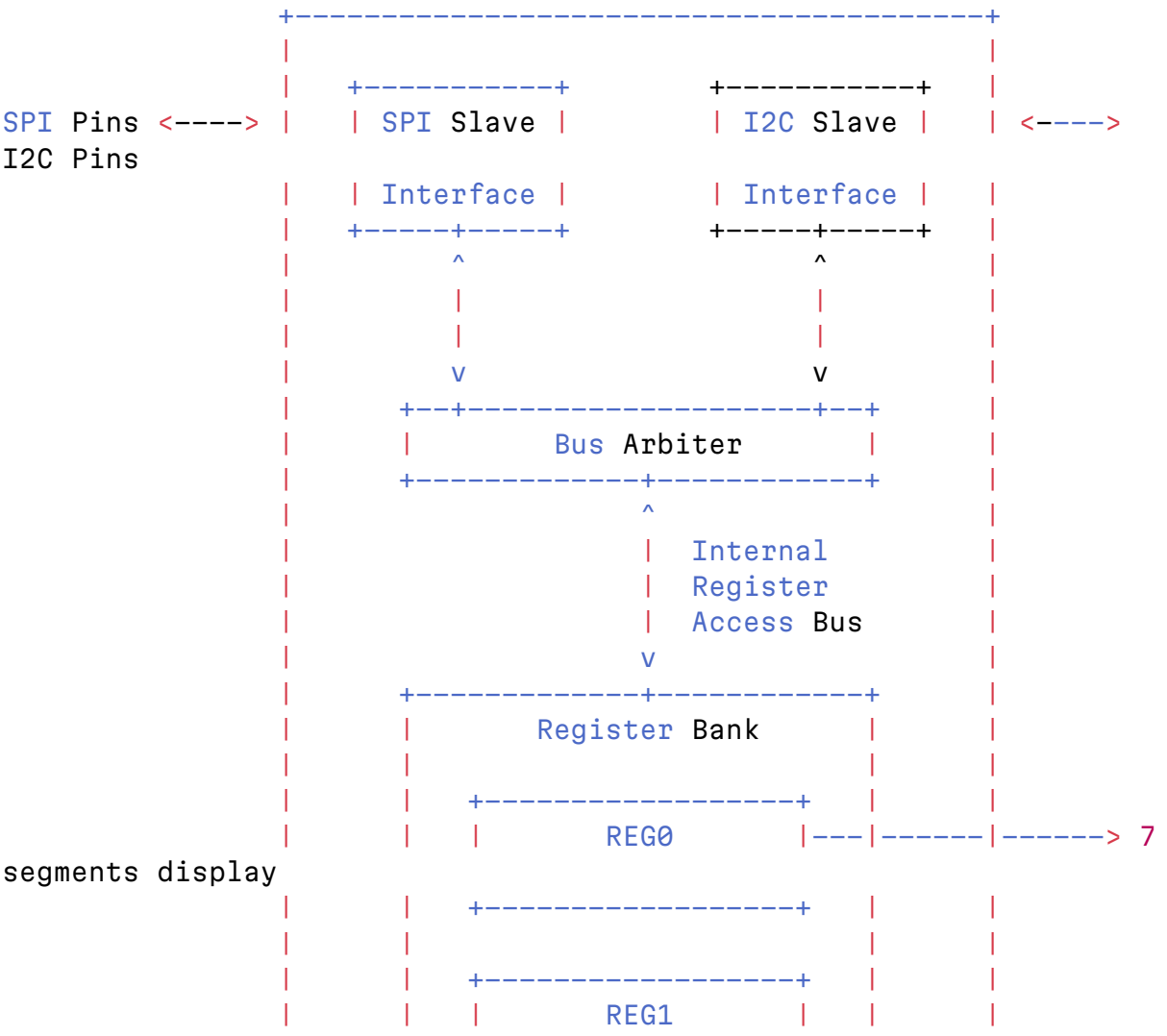
"Register bank accessible through SPI and I2C"

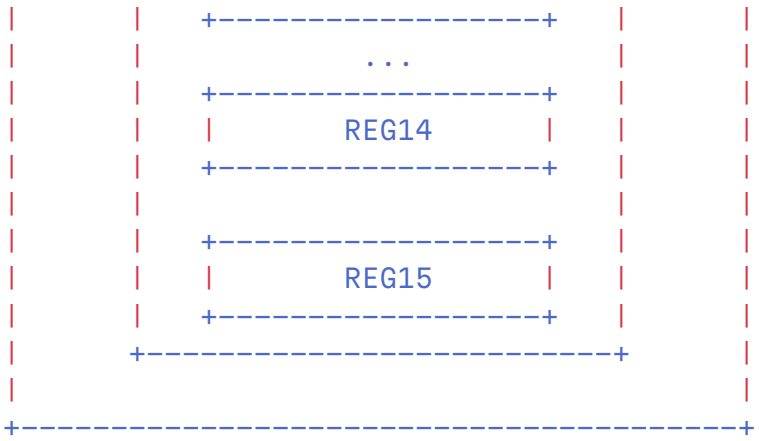
How it works

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)

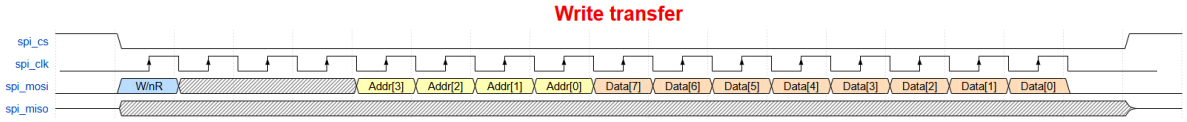


Figure 583.1: SPI Write

SPI Read (CPOL = 0, CPHA = 0)

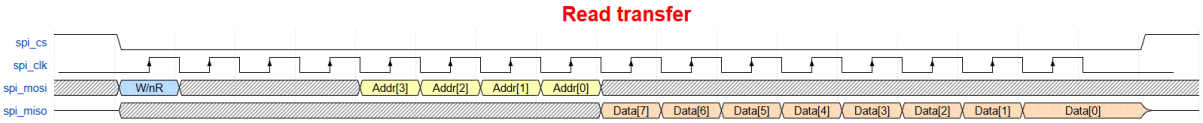


Figure 583.2: SPI Write

I2C Frame

Figure 583.3: 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]	—

VGA Rings

by **Uri Shaked**

`0585` `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	—

Glitcher

by **Philip Åkesson**

0587 | 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.

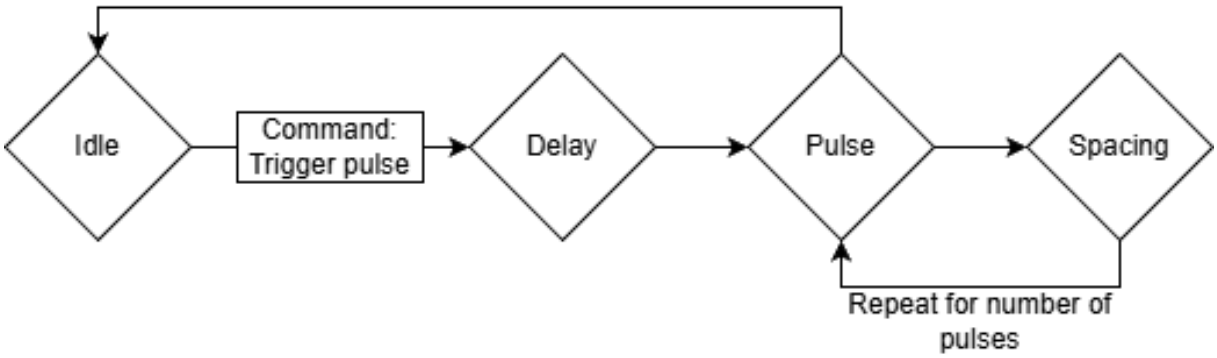


Figure 587.1: Diagram showing manual triggering over UART

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.

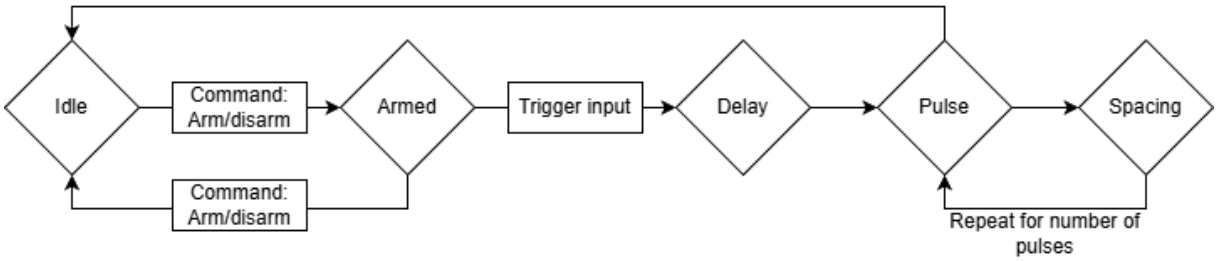


Figure 587.2: Diagram showing arming over UART and then using an external trigger

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.

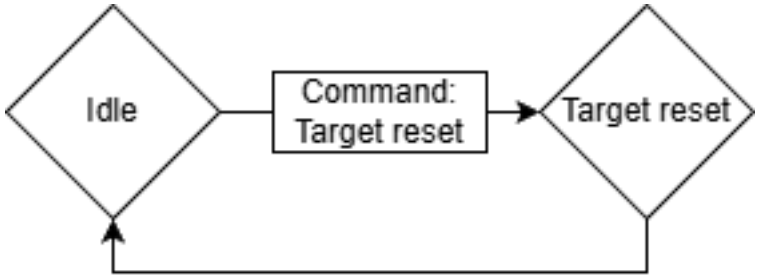


Figure 587.3: Diagram showing the “None” reset mode

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.

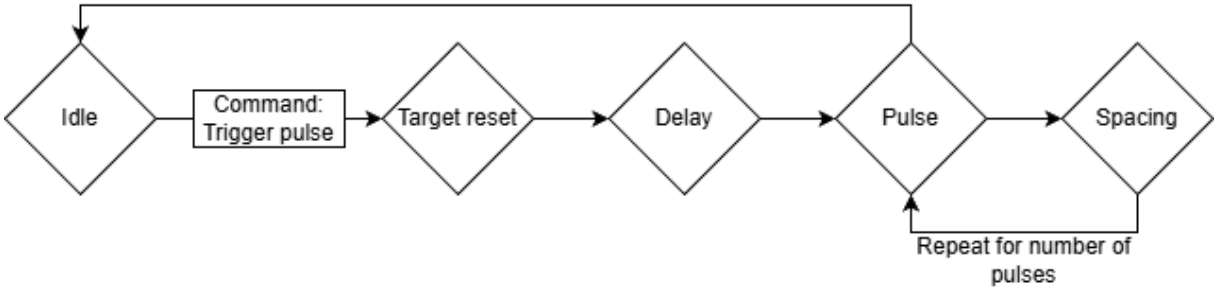


Figure 587.4: Diagram showing the “Pulse” reset mode

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]`).

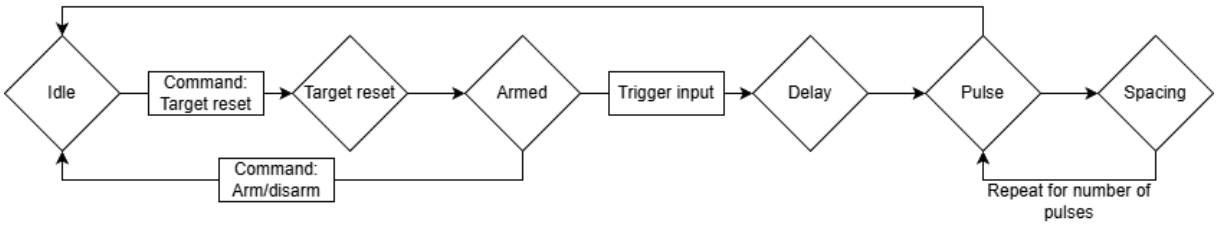


Figure 587.5: Diagram showing the “Arm” reset mode

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
```

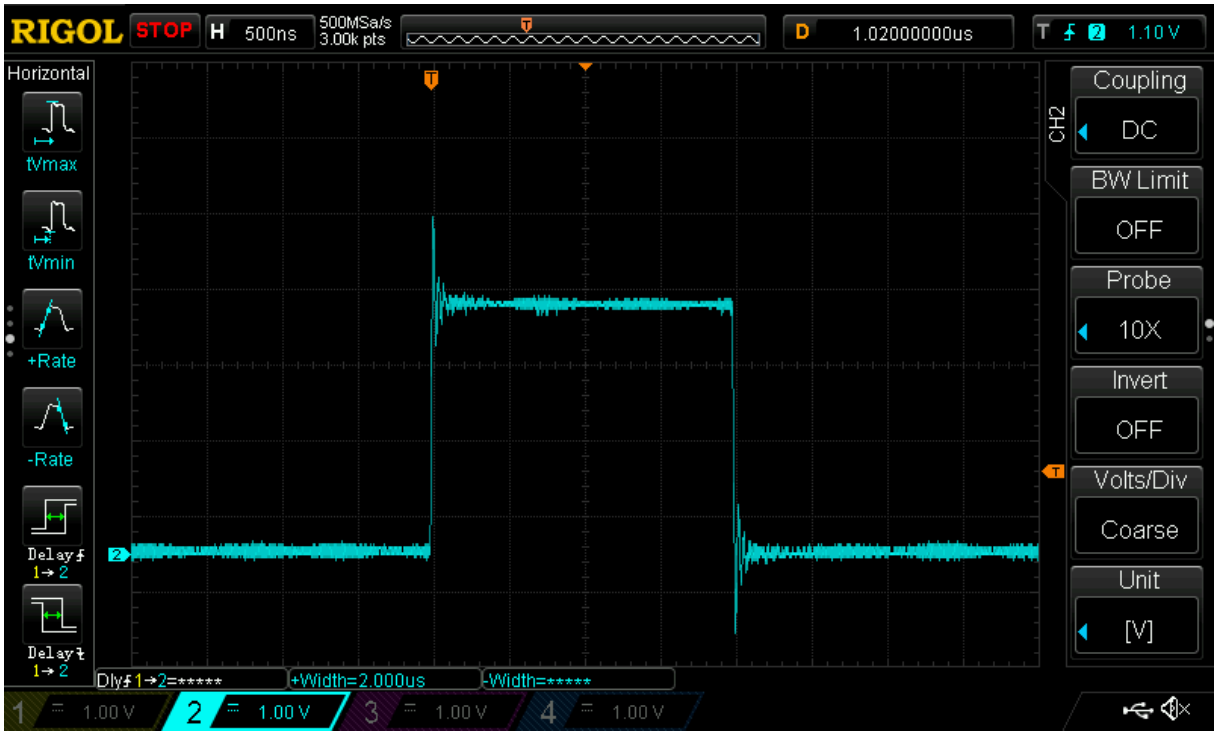


Figure 587.6: Oscilloscope capture of a single pulse, with width set to 100 clock cycles

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
```

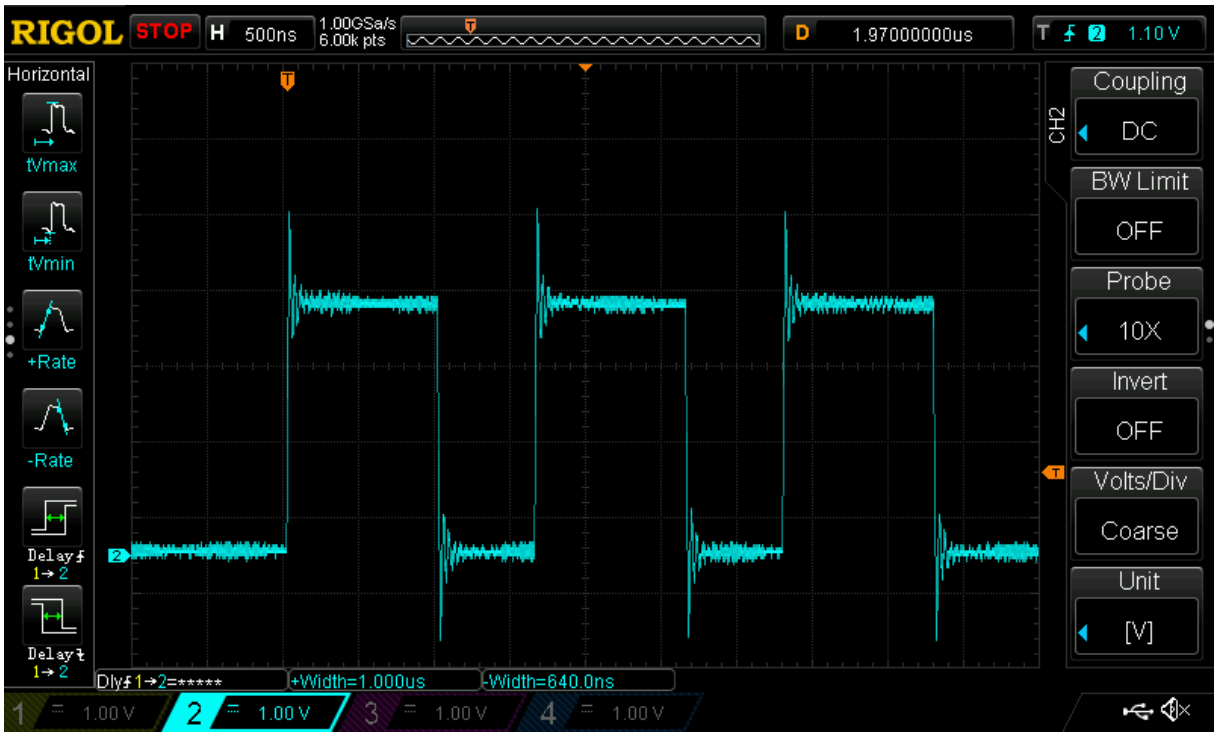


Figure 587.7: Oscilloscope capture of multiple pulses

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
```

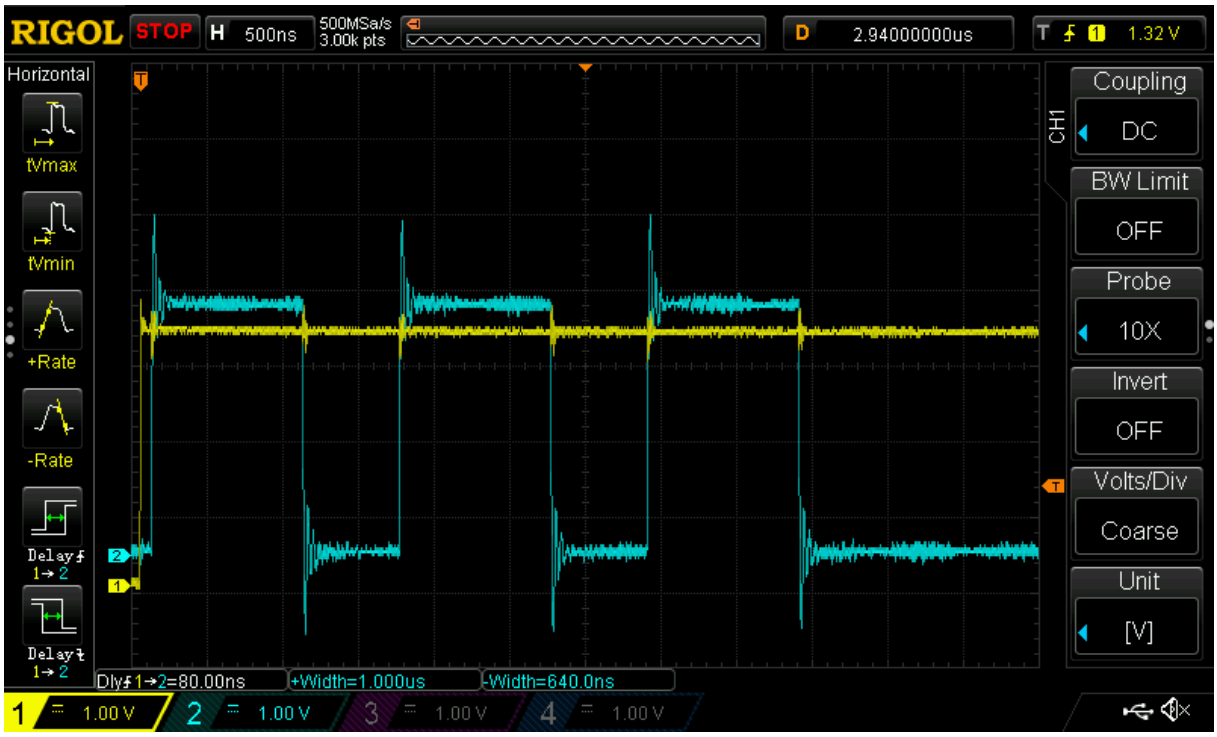


Figure 587.8: Oscilloscope capture of both trigger signal and pulse output with three pulses, pulse width 50 clock cycles and spacing 32 clock cycles.

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
```

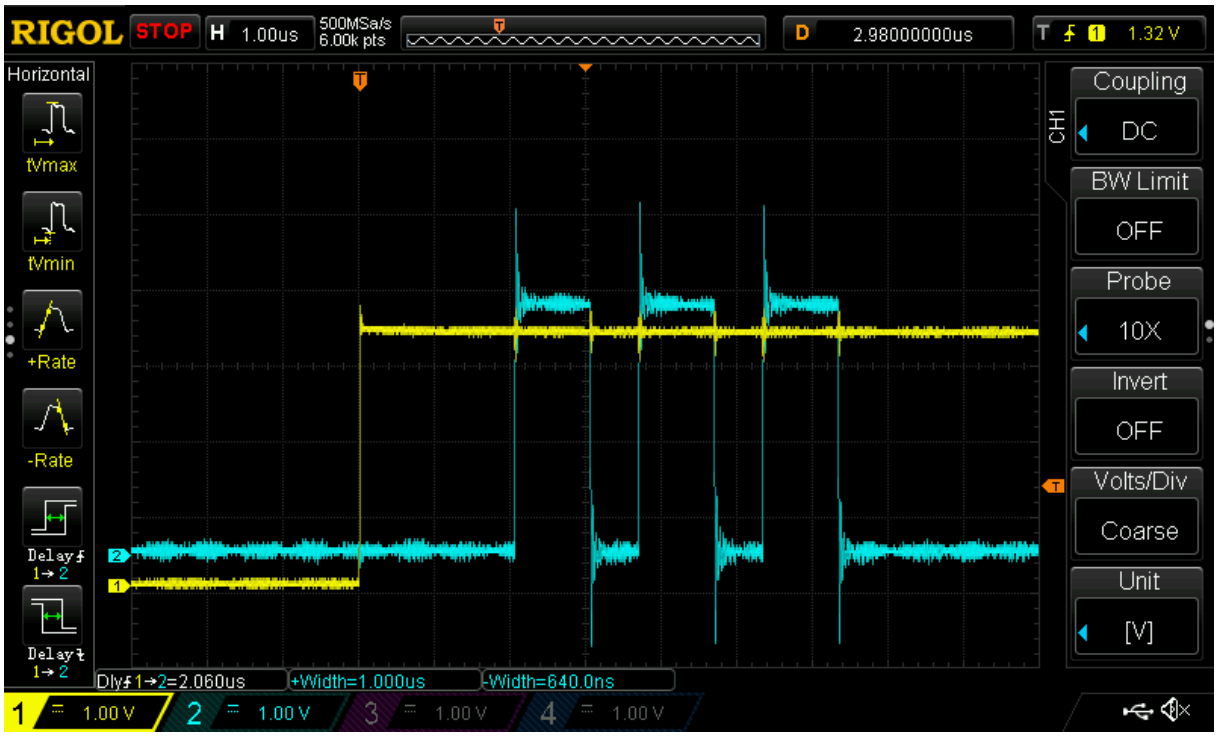


Figure 587.9: Oscilloscope capture of both trigger signal and pulse output with delay 100 cycles, three pulses, pulse width 50 clock cycles and spacing 32 clock cycles.

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

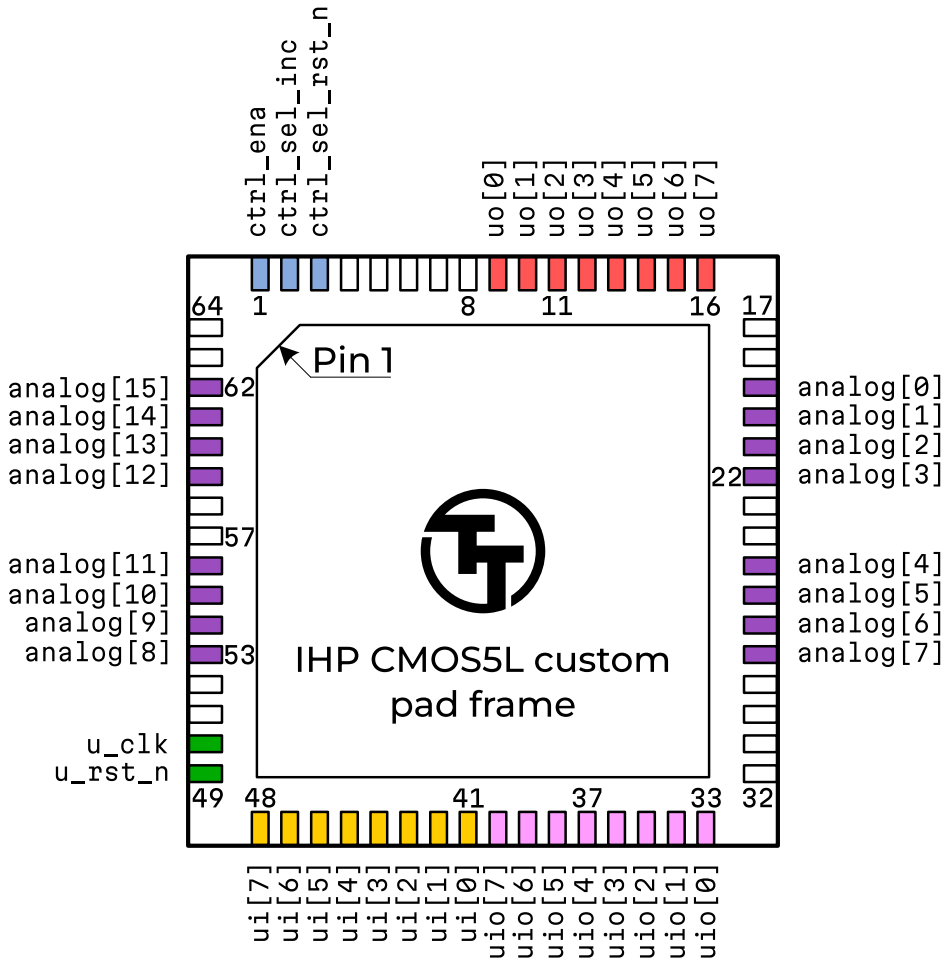
Digital Pins

#	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	—	—	—

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:

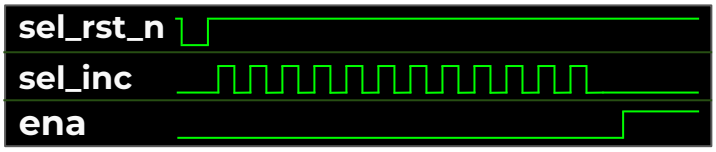


Figure 1: Mux signals for activating the design at address 12

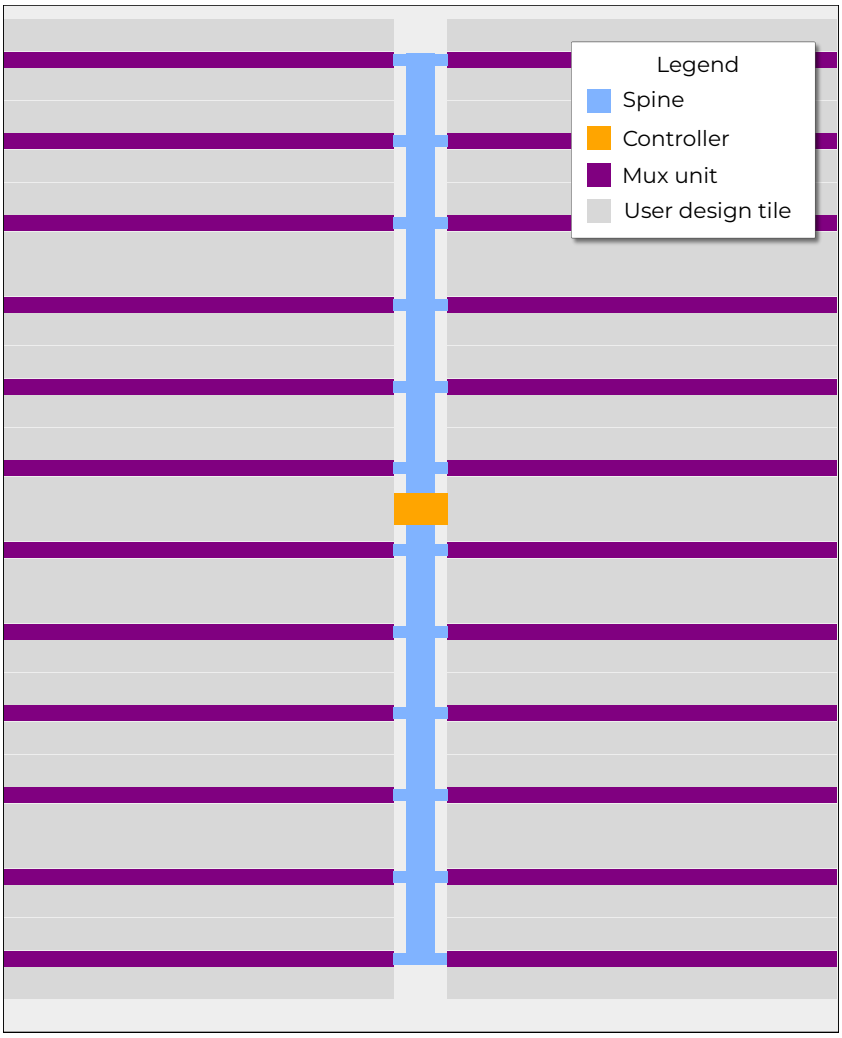


Figure 2: Mux Diagram

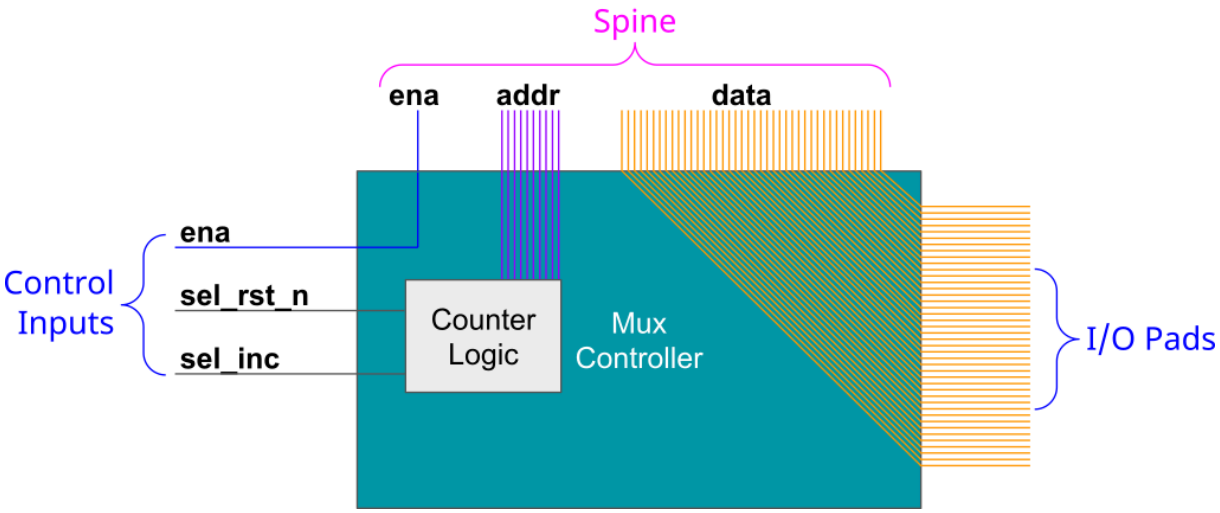


Figure 3: Mux Controller Diagram

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.
#123 #423/2	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