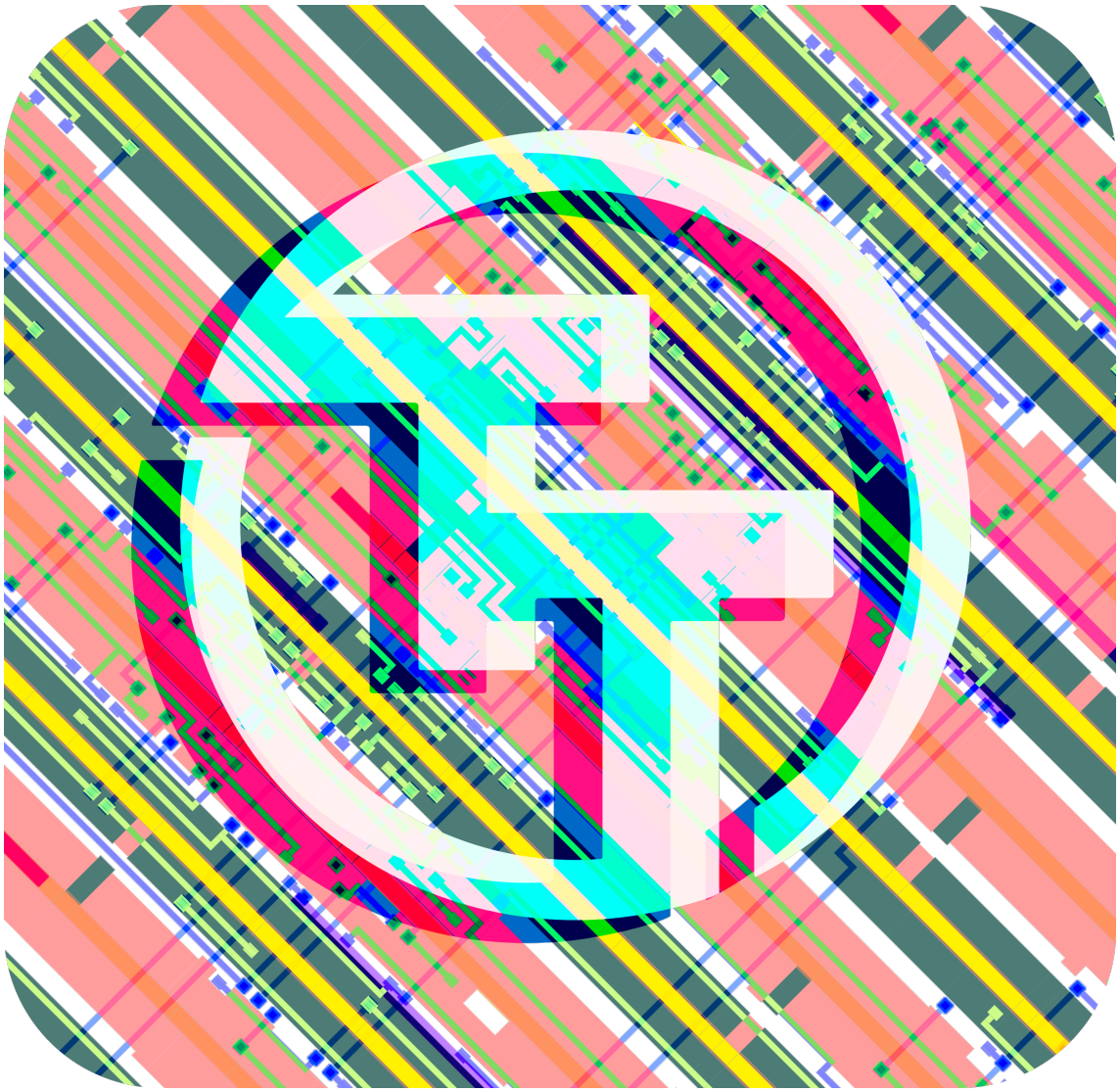


Tiny Tapeout IHP 0p3 Datasheet

Project Repository

https://github.com/TinyTapeout/tinytapeout-ihp-0p3

May 10, 2025

Contents

Chip map

Full chip map

GDS render

Logic density (local interconnect layer)

Projects

Chip ROM [0]

- Author: Uri Shaked
- Description: ROM with information about the chip
- GitHub repository
- HDL project
- Mux address: 0
- Extra docs
- Clock: 0 Hz

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: `&quot;TT\xFA\xBB&quot;`
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.

Pinout

#	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 [1]

- Author: Tiny Tapeout
- Description: Factory test module
- GitHub repository
- HDL project
- Mux address: 1
- Extra docs
- Clock: 0 Hz

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`).

Pinout

#	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]

simple-viii [2]

- Author: strau
- Description: A simple 8-bit CPU Architecture
- GitHub repository
- HDL project
- Mux address: 2
- Extra docs
- Clock: 50000000 Hz

How it works

How to test

External hardware

Pinout

#	Input	Output	Bidirectional
0			cs flash
1			SD0
2			SD1
3			SCK
4			SD2
5			SD3
6			cs ram
7			

R-2R DAC [3]

- Author: htfab
- Description: Basic 8 bit R-2R DAC for IHP
- GitHub repository
- Analog project
- Mux address: 3
- Extra docs
- Clock: 0 Hz

How it works

Combines 25 identical resistors (plus 2 dummies) into an 8 bit R-2R resistor ladder DAC.

How to test

Use `ui_in[7:0]` to set the input in binary. The output analog voltage is available on `ua[0]`.

External hardware

A multimeter or some other device to measure the output voltage (ADC, oscilloscope, etc.)

Pinout

#	Input	Output	Bidirectional
0	IN[0]		
1	IN[1]		
2	IN[2]		
3	IN[3]		
4	IN[4]		
5	IN[5]		
6	IN[6]		
7	IN[7]		

Analog pins

ua#	analog#	Description

VGA clock [32]

- Author: Matt Venn
- Description: Shows the time on a VGA screen
- GitHub repository
- HDL project
- Mux address: 32
- Extra docs
- Clock: 31500000 Hz

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

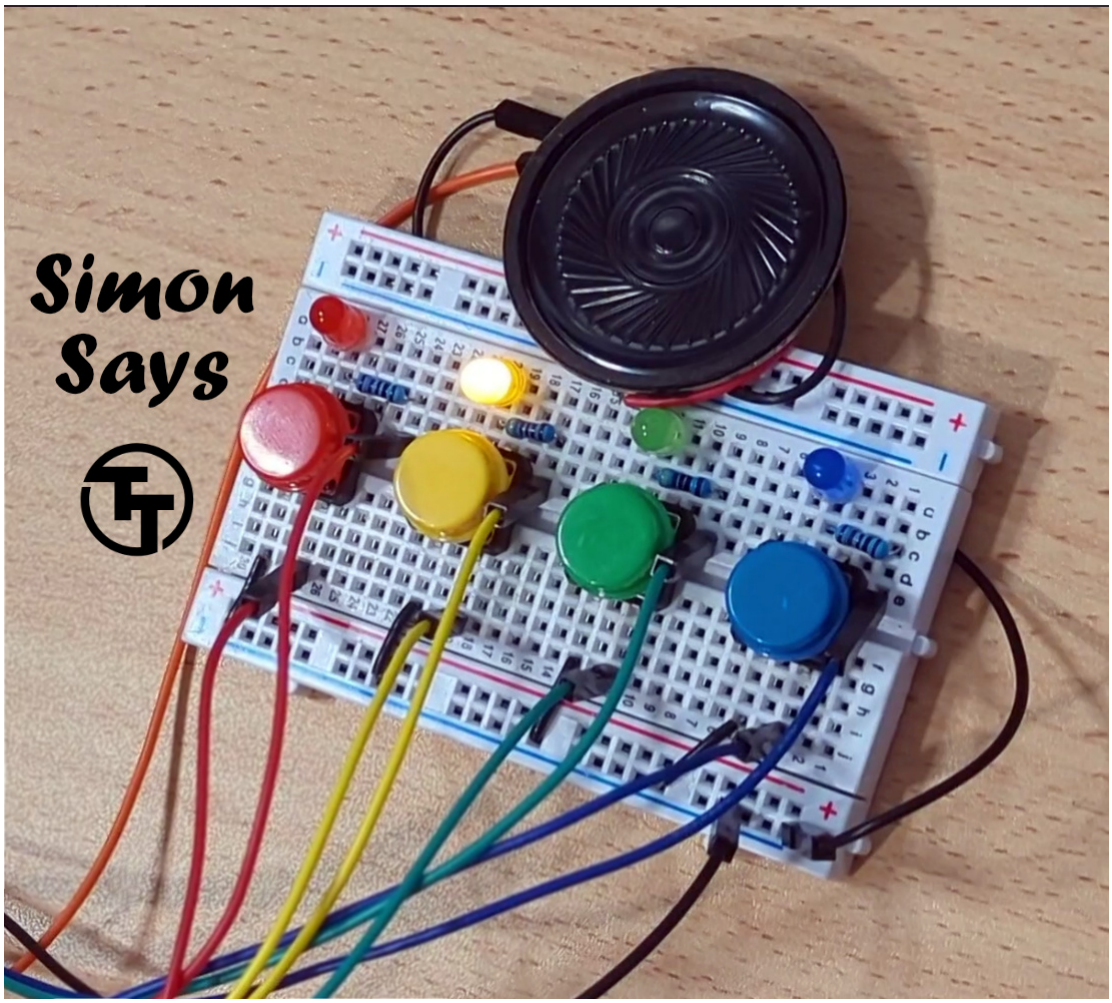
#	Input	Output	Bidirectional

Pinout

#	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	

Simon Says memory game [35]

- Author: Uri Shaked
- Description: Repeat the sequence of colors and sounds to win the game
- GitHub repository
- HDL project
- Mux address: 35
- Extra docs
- Clock: 50000 Hz



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.

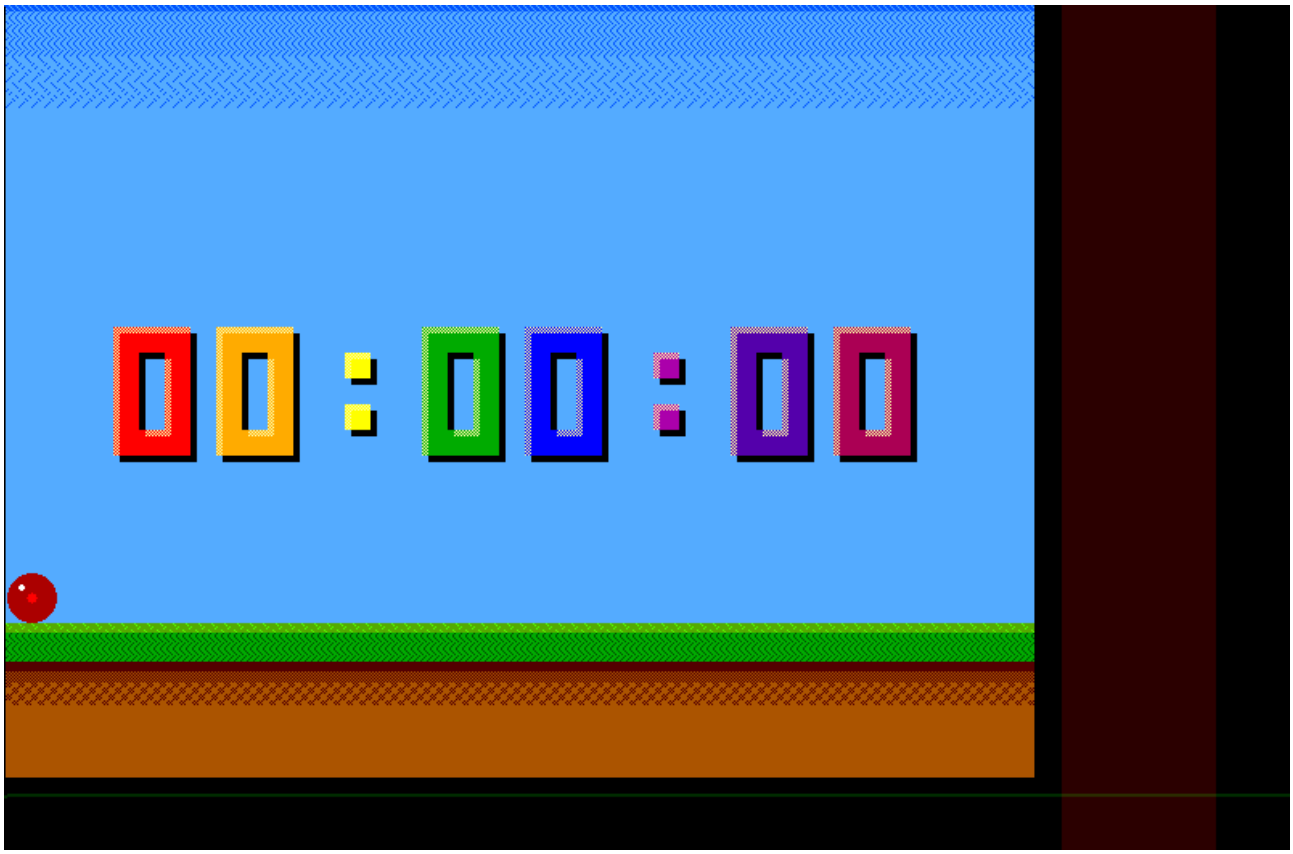
Pinout

#	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	

Fun VGA Clock [64]

- Author: algofoogle (Anton Maurovic)
- Description: Simple VGA demo for IHP tapeout (inc. Matt Venn's VGA clock)
- GitHub repository
- HDL project
- Mux address: 64
- Extra docs
- Clock: 25000000 Hz

How it works



Typical Verilog design that generates VGA timing and RGB222 colour outputs compatible with the Tiny VGA PMOD.

It produces a bouncing ball animation over the top of an adaptation of Matt Venn's VGA clock, from here: https://github.com/mattvenn/tt08-vga-clock

How to test

- Plug in a VGA monitor via Tiny VGA PMOD.
- Set `mode` input to 0, i.e. specifying 640x480 60Hz from a 25MHz clock.

- Set `show_clock` input to 1.
- Set `pmod_select` input to 0 for Tiny VGA PMOD. Otherwise, 1=Matt's VGA Clock PMOD.
- Supply a 25MHz clock (clock's actual seconds timer assumes exactly 25.000MHz).
- Assert reset.
- Pulse or hold the `adj_*` inputs to adjust hours, minutes, or seconds.

External hardware

Tiny VGA PMOD and VGA monitor is all you should need externally.

Pinout

#	Input	Output	Bidirectional
0	adj_hrs	r1	hmax
1	adj_min	g1	vmax
2	adj_sec	b1	hblank
3	pmod_select	vsync	vblank
4	show_clock	r0	visible
5		g0	
6		b0	
7	mode	hsync	

Ring Oscillator Worker [66]

- Author: algofoogle (Anton Maurovic)
- Description: Simple digital logic, doing work, driven by a ring oscillator
- GitHub repository
- HDL project
- Mux address: 66
- Extra docs
- Clock: 0 Hz

How it works

An internal simple digital counter block can be driven by an external clock or an internal ring oscillator, and then be fed data through external pins. It will run as fast as it can to try and produce a result, which can then be read back out.

How to test

1. Set `clock_sel`=1 (internal ring oscillator is used as the clock source). Ring-osc clock, divided by 16, should be present on `cdebug` – expected to be on the order of 12.5MHz to 25MHz.
2. Set `mode`=0 (we're going to load the number of cycles for which we want the worker to run).
3. Assert reset. No need to supply a clock on `clk`. Expect `done==0`.
4. Load a sequence of 4 bytes: a rising edge on `shift` loads each byte, in turn, via `din[7:0]`. First 2 bytes are a starting value (MSB first). The next 2 bytes are a cycle count. In `mode==0` this cycle count is used, while in `mode==1` it is repurposed as the addend for an adder experiment.
5. After the 4th byte has been loaded, the worker should start, and set `done==1` when it finishes.
6. When done, `dout[7:0]` should be presenting the first byte (MSB) of the output data; shift out 4 bytes in total via `dout[7:0]` by raising `shift` each time again (which in turn loads 4 more bytes, so it will start again). The first 2 bytes out are the 'starting value' incremented by the counter value (i.e. it should be the starting value, plus the internal counter value), and the last 2 bytes are the internal counter value (which started at 0).

External hardware

Nothing special. Probably just an oscilloscope to see how fast it actually yields a result.

Pinout

#	Input	Output	Bidirectional
0	din[0]	dout[0]	shift
1	din[1]	dout[1]	clock_sel
2	din[2]	dout[2]	mode
3	din[3]	dout[3]	stop
4	din[4]	dout[4]	
5	din[5]	dout[5]	running
6	din[6]	dout[6]	done
7	din[7]	dout[7]	cdebug

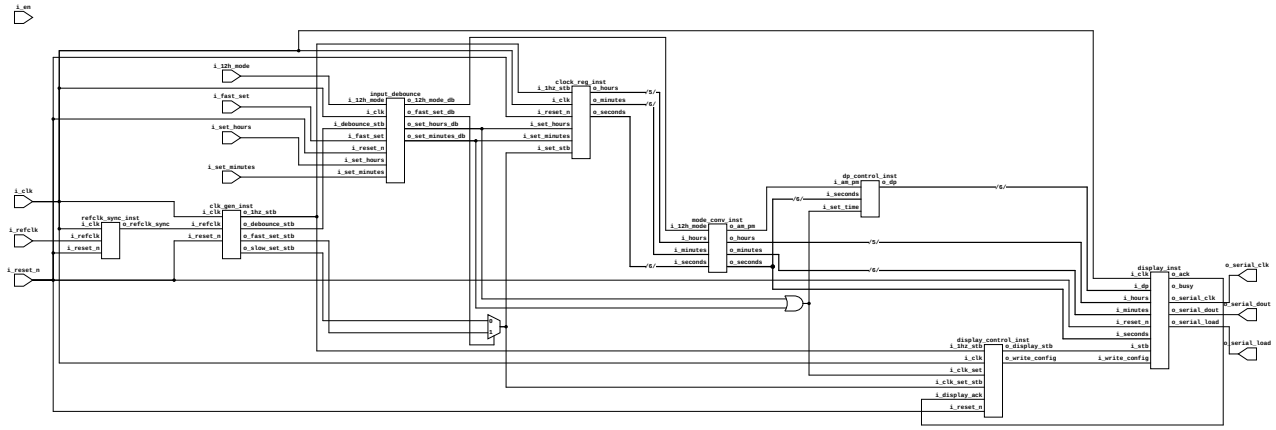
7-Segment Digital Desk Clock [98]

- Author: Samuel Ellicott
- Description: 7-Segment Desk Clock
- GitHub repository
- HDL project
- Mux address: 98
- Extra docs
- Clock: 50000000 Hz

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 Tape-out SPI

Pinout

#	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			

Linear Feedback Shift Register [128]

- Author: Steve Jenson <stevej@gmail.com>
- Description: An implementation of a Linear Feedback Shift Register for ttihp0p3
- GitHub repository
- HDL project
- Mux address: 128
- Extra docs
- Clock: 0 Hz

How it works

Read the `ui_out` pins, each read should be different than the last. To reset the shift register, reset the chip, or set the 'write_enable' pin high after offering a value on `ui_in` as a seed.

How to test

Read several bytes from `ui_in`, they should each be different.

External hardware

No external hardware needed other than to read the pins.

Pinout

#	Input	Output	Bidirectional
0	Seed Bit 1	LFSR Bit 1	Write Enable
1	Seed Bit 2	LFSR Bit 2	
2	Seed Bit 3	LFSR Bit 3	
3	Seed Bit 4	LFSR Bit 4	
4	Seed Bit 5	LFSR Bit 5	
5	Seed Bit 6	LFSR Bit 6	
6	Seed Bit 7	LFSR Bit 7	
7	Seed Bit 8	LFSR Bit 8	

Prism [129]

- Author: bleeptrack
- Description: a hypnotic prism floating in a starry night
- GitHub repository
- HDL project
- Mux address: 129
- Extra docs
- Clock: 25175000 Hz

How it works

It's a visual. not much to do currently :)

How to test

Hook it up into the VGA dongle and turn it on!

External hardware

Mole99

Pinout

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

Antonalog analog VGA [130]

- Author: algofoogle (Anton Maurovic)
- Description: Rough 24-bit VGA DAC tests with digital control block
- GitHub repository
- Analog project
- Mux address: 130
- Extra docs
- Clock: 25000000 Hz

Overview

My first attempt at an IHP analog design.

How it works

A digital block based on my tt08-vga-fun project drives 3x 8-bit DACs to produce analog VGA outputs.

In this case, the DACs are simple R2R DACs (where R=8660).

How to test

TBC.

External hardware

TBC.

Pinout

#	Input	Output	Bidirectional
0	mode[0] / dac_in[0]	r7	vblank_out
1	mode[1] / dac_in[1]	g7	hblank_out
2	mode[2] / dac_in[2]	b7	
3	mode[3] / dac_in[3]	vsync	
4	mode[4] / dac_in[4]	r6	
5	mode[5] / dac_in[5]	g6	

#	Input	Output	Bidirectional
6	mode[6] / dac_in[6]	b6	
7	mode[7] / dac_in[7]	hsync	

Analog pins

ua#	analog#	Description

E-ink display driver [131]

- Author: Tim Edwards
- Description: Test driver for Adafruit 2.13 inch e-ink display
- GitHub repository
- HDL project
- Mux address: 131
- Extra docs
- Clock: 50 Hz

How it works

This is an example hardware driver for an e-ink display. Adafruit makes a nice series of small e-ink displays, but they are designed for an Arduino and driven by software. This project shows how to build a display driver in verilog. To keep memory overhead to a minimum, it operates like a VGA screen saver, displaying simple patterns that can be computed in real time as the pixel positions are counted and transmitted to the driver.

The driver instantiates an SPI master which communicates with the SSD1680 chipset on the e-ink display. Whenever a bit from the input PMOD is set to "1", and initialization sequence is send to the display, followed by a transmission of the display image, followed by a deep sleep power-down. Once in deep sleep mode, the displayed image will remain indefinitely, even if the display is disconnected from the development board.

How to test

The input/output PMOD is used to connect to the e-ink display pins. Since the e-ink display is not PMOD-compatible, it is necessary to install a header onto the e-ink display and create a bundle of jumper wires to connect to the PMOD as follows:

pin signal direction PMOD pin

ECS: uio[0] output 1

MOSI: uio[1] output 2 MISO: uio[2] input 3 SCK: uio[3] output 4 SRCS: uio[4] output 7 RST: uio[5] output 8 BUSY: uio[6] input 9 D/C: uio[7] output 10 GND: 11 or 5 VIN: 12 or 6

To test the eight example patterns, raise one of the input pins to value "1". This can be done with a set of external buttons on the input PMOD, or the input PMOD value can be set from software.

ui[5] is a special case in which the contents of the display board's SRAM are copied directly to the e-ink display. This uses an unusual method in which the SRAM is set to a sequential read mode and then is left enabled while the e-ink display is initialized. Commands being sent to the display are ignored by the SRAM, which outputs one bit on every clock cycle. The SRAM contents are then copied into the display starting at offset address 30 (which is the number of SPI bytes clocked while initializing the display). The SRAM is volatile and so unprogrammed at power-up. It can be programmed using the "pass-through" mode, in which the SRAM's SPI can be bit-banged from the ui[] port using software. Enable "pass-through" mode by setting ui[7:4] to 0xf, then bit-bang using ui[0] for clock and ui[1] for data (if the SRAM is given a READ command, then output from the SRAM can be read from uo[0]). First put the SRAM into sequential mode with command 0x01 0x40. End pass-through mode with ui = 0x00, then re-enter pass-through mode with ui = 0xf0. Continue with the command 0x02 0x00 0x1e and then write 3904 bytes of image data (32 bytes x 122 lines). End pass-through mode again with ui = 0x00, then display the image data with ui = 0x20.

External hardware

Every e-ink display has a very specific driver, and making a general-purpose driver is prohibitive for Tiny Tapeout. The project is designed to drive the Adafruit 2.13" e-ink display, Product ID: 4197, URL https://www.adafruit.com/product/4197 (as of this writing, cost is $22.50).

Pinout

#	Input	Output	Bidirectional
0	All white	Bitbang SCK	SRAM MISO (out)
1	All black	Bitbang MOSI	
2	Vertical stripes		MISO (in, unused)
3	Horizontal stripes		SCK (out)
4	Small checkerboard		SRCS (out, =1)
5	User SRAM contents		RSTB (out)
6	Large checkerboard		BUSY (in)
7	Low-res smiley face		D/C (out)

SPI test [193]

- Author: Caio Alonso da Costa
- Description: SPI test
- GitHub repository
- HDL project
- Mux address: 193
- Extra docs
- Clock: 50000000 Hz

How it works

SPI test design based from https://github.com/calonso88/tt07_alu_74181

See that design's docs for information about the SPI peripheral.

Small improvement done on the spi_reg 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.

Design been configured with 8 read/write 8 bit registers and 8 read only 8 bit status registers.

The first read/write register also drives the 7 segment display.

How to test

Use SPI1 Master peripheral in RP2040 to start communication on SPI interface towards this design. Remember to configure the SPI mode using the switches in DIP switch (if you'd like to have CPOL=1 and CPHA=1). Alternatively, don't use the DIP switches and use the RP2040 GPIOs to configure the SPI mode in the desired mode.

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, firstb
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].

External hardware

Not required. Write to the first register to set the LEDs on the demoboard.

External hardware

None.

Pinout

#	Input	Output	Bidirectional
0	cpol	spare[0]	
1	cpha	spare[1]	
2		spare[2]	
3		spare[3]	spi_miso

#	Input	Output	Bidirectional
4		spare[4]	spi_cs_n
5		spare[5]	spi_clk
6		spare[6]	spi_mosi
7		spare[7]	

Demo by a1k0n [194]

- Author: Andy Sloane
- Description: Tiny Tapeout demo competition entry
- GitHub repository
- HDL project
- Mux address: 194
- Extra docs
- Clock: 48000000 Hz

a1k0n's tinytapeout08 demo compo entry

How it works It's a standalone VGA+sound demo that fits in two tiles; you'll just have to see. The demo is short, looping after about 25 seconds.

This was developed with a 48MHz clock, so it's in a funky VGA video mode – it's standard 640x480@60Hz VGA timing and 4:3 aspect ratio, but with 1220 horizontal

pixels instead of 640. All graphics are dithered down to RGB222 with a Bayer matrix which alternates each frame. Because of the dithering and the weird resolution, it looks best on a real CRT, but any VGA monitor ought to work.

Sound is generated using a 16-bit sigma-delta DAC on io7 from an internal 3-channel synth (triangle, noise, and square waves).

Sines and cosines are generated by an old HAKMEM trick which generates a slightly off-center circle but that doesn't matter in this application:

```
cos_new = cos - (sin>>k)
sin_new = sin + (cos_new(!)>>k)
```

The plane is rendered by doing a bit-by-bit non-restoring division of the y coordinate during the horizontal blanking interval to find a fixed point reciprocal, which is then used as an x increment for the plane u coordinate. As a drastic simplification, the plane v coordinate is *also* the x increment value (when you do the math, it turns out they are proportional).

Starfield is generated by an LFSR that increments every line which provides an x-offset and speed for each star by picking out individual bits of the LFSR state.

The "TT08" logo uses the outline of an old demo font, but the actual coloring is procedural as it would take too much combinational logic to reproduce exactly.

Soundtrack is a riff on "Crooner" by Drax/Vibrants, composed as a bunch of text in a Python script with limitations on song structure and octave range. Kick drum and bass share the triangle channel, lead arpeggios on square, and hihat noise.

I'm not super happy about the "programmer colors" everywhere, but I ran out of room trying to add palettes.

How to test Run clock at 48MHz, connect VGA and sound Pmods, and give it a reset pulse (falling edge).

External hardware Follows the democompo hardware rules:

TinyVGA Pmod for video on o[7:0].

1-bit sound on io[7], compatible with Tiny Tapeout Audio Pmod, or any basic ~20kHz RC filter on io7 to an amplifier will work.

Pinout

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

SPI-connected PWM generator [195]

- Author: Damir G
- Description: SPI-connected PWM generator featuring 8 outputs with 2 independent generators and 4 total PWM channels
- GitHub repository
- HDL project
- Mux address: 195
- Extra docs
- Clock: 10000000 Hz

How it works

Will add later

How to test

Will add later

External hardware

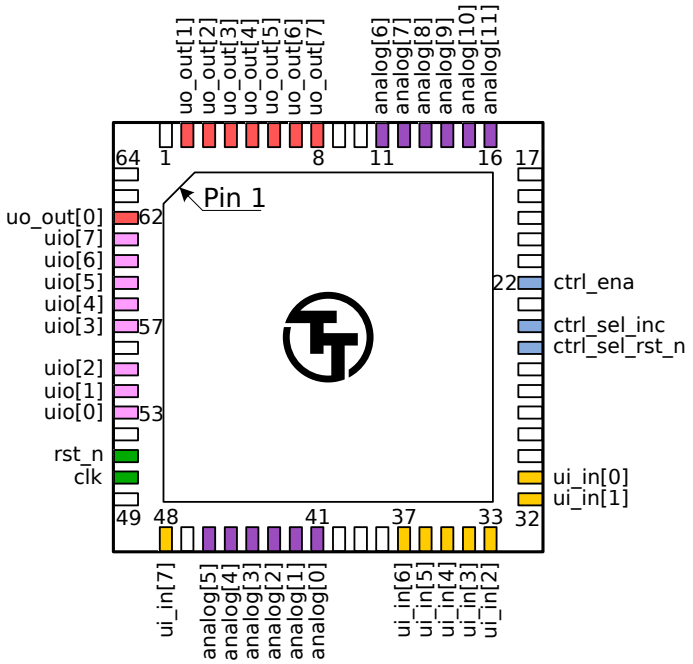
Will add later

Pinout

#	Input	Output	Bidirectional
0	SCLK	OUT_0	
1	COPI	OUT_1	
2	nCS	OUT_2	
3		OUT_3	
4		OUT_4	
5		OUT_5	
6		OUT_6	
7		OUT_7	CIPO

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. 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.

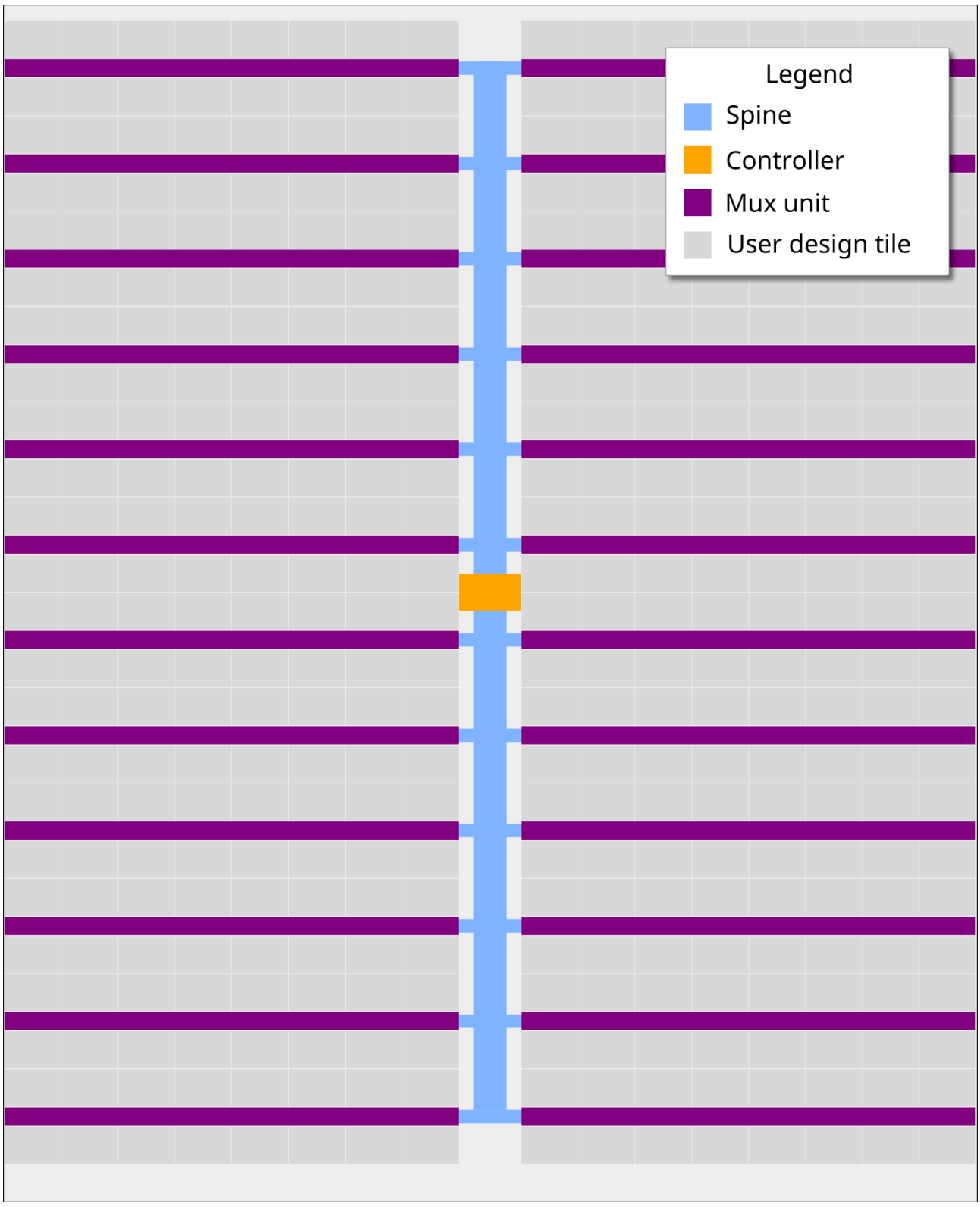
It has the following features:

- 10 dedicated inputs
- 8 dedicated outputs
- 8 bidirectional IOs
- Supports up to 512 user designs (32 mux units, each with up to 16 designs)
- Designs can have different sizes. The basic unit is a called a tile, and each design can occupy up to 16 tiles.

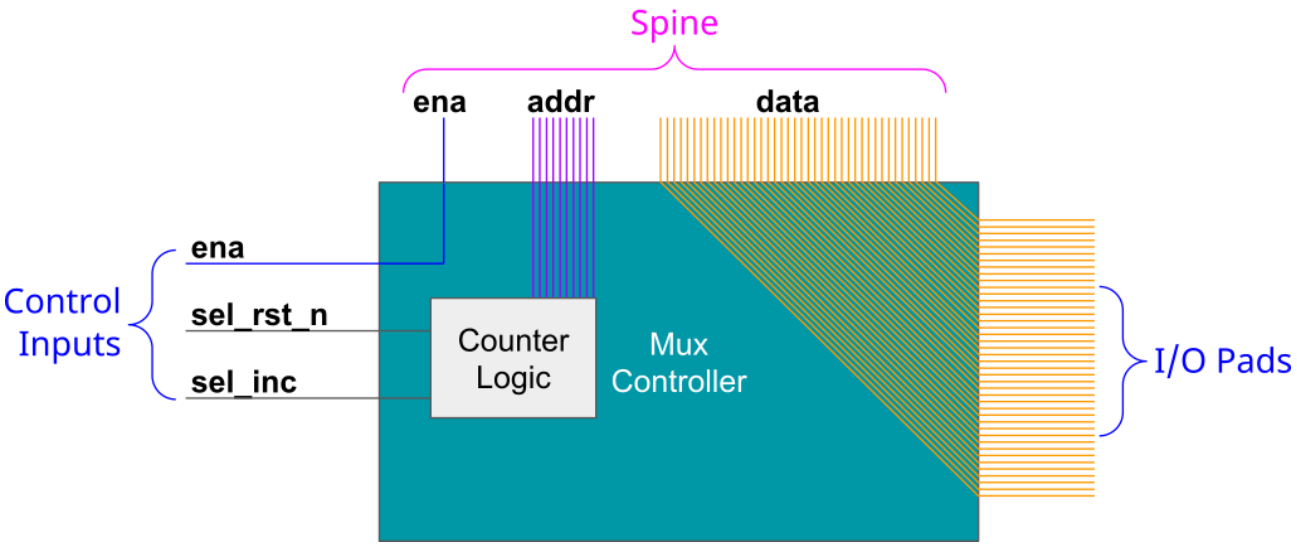
Operation

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 - connect the spine to individual user designs


Legend
Spine
Controller
Mux unit
User design tile

The Controller

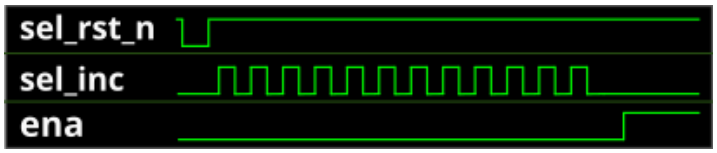


The mux controller has 3 inputs lines:

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:



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 projects demonstrates this setup: https://wokwi.com/projects/364347807664031745. 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 the muxes are connected together through the spine. The spine has the following signals going on 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 the chip IO pads.

The Multiplexer (The Mux)

Each mux branch is connected to up to 16 designs. It also has 5 bits of 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

QFN64 pin	Function	Signal
1	Mux Control	ctrl_ena
2	Mux Control	ctrl_sel_inc
3	Mux Control	ctrl_sel_rst_n
4	Reserved	(none)
5	Reserved	(none)
6	Reserved	(none)
7	Reserved	(none)
8	Reserved	(none)
9	Output	uo_out[0]
10	Output	uo_out[1]
11	Output	uo_out[2]
12	Output	uo_out[3]
13	Output	uo_out[4]
14	Output	uo_out[5]
15	Output	uo_out[6]
16	Output	uo_out[7]
17	Power	VDD IO
18	Ground	GND IO
19	Analog	analog[0]
20	Analog	analog[1]
21	Analog	analog[2]
22	Analog	analog[3]
23	Power	VAA Analog
24	Ground	GND Analog
25	Analog	analog[4]
26	Analog	analog[5]
27	Analog	analog[6]
28	Analog	analog[7]
29	Ground	GND Core
30	Power	VDD Core
31	Ground	GND IO
32	Power	VDD IO
33	Bidirectional	uio[0]
34	Bidirectional	uio[1]
35	Bidirectional	uio[2]
36	Bidirectional	uio[3]
37	Bidirectional	uio[4]
38	Bidirectional	uio[5]

QFN64 pin	Function	Signal
39	Bidirectional	uio[6]
40	Bidirectional	uio[7]
41	Input	ui_in[0]
42	Input	ui_in[1]
43	Input	ui_in[2]
44	Input	ui_in[3]
45	Input	ui_in[4]
46	Input	ui_in[5]
47	Input	ui_in[6]
48	Input	ui_in[7]
49	Input	rst_n †
50	Input	clk †
51	Ground	GND IO
52	Power	VDD IO
53	Analog	analog[8]
54	Analog	analog[9]
55	Analog	analog[10]
56	Analog	analog[11]
57	Ground	GND Analog
58	Power	VDD Analog
59	Analog	analog[12]
60	Analog	analog[13]
61	Analog	analog[14]
62	Analog	analog[15]
63	Ground	GND Core
64	Power	VDD Core

† Internally, there's no difference between `clk`, `rst_n`, and `ui_in` 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.

Funding

IHP PDK support for Tiny Tapeout was funded by The SwissChips Initiative.

The manufacturing of Tiny Tapeout IHP 0p2 silicon was funded by the German BMBF project FMD-QNC (16ME0831).

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 Tiny Tapeout 01 and volunteered time to improve docs and test the flow
- The team at YosysHQ and all the other open source EDA tool makers
- 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