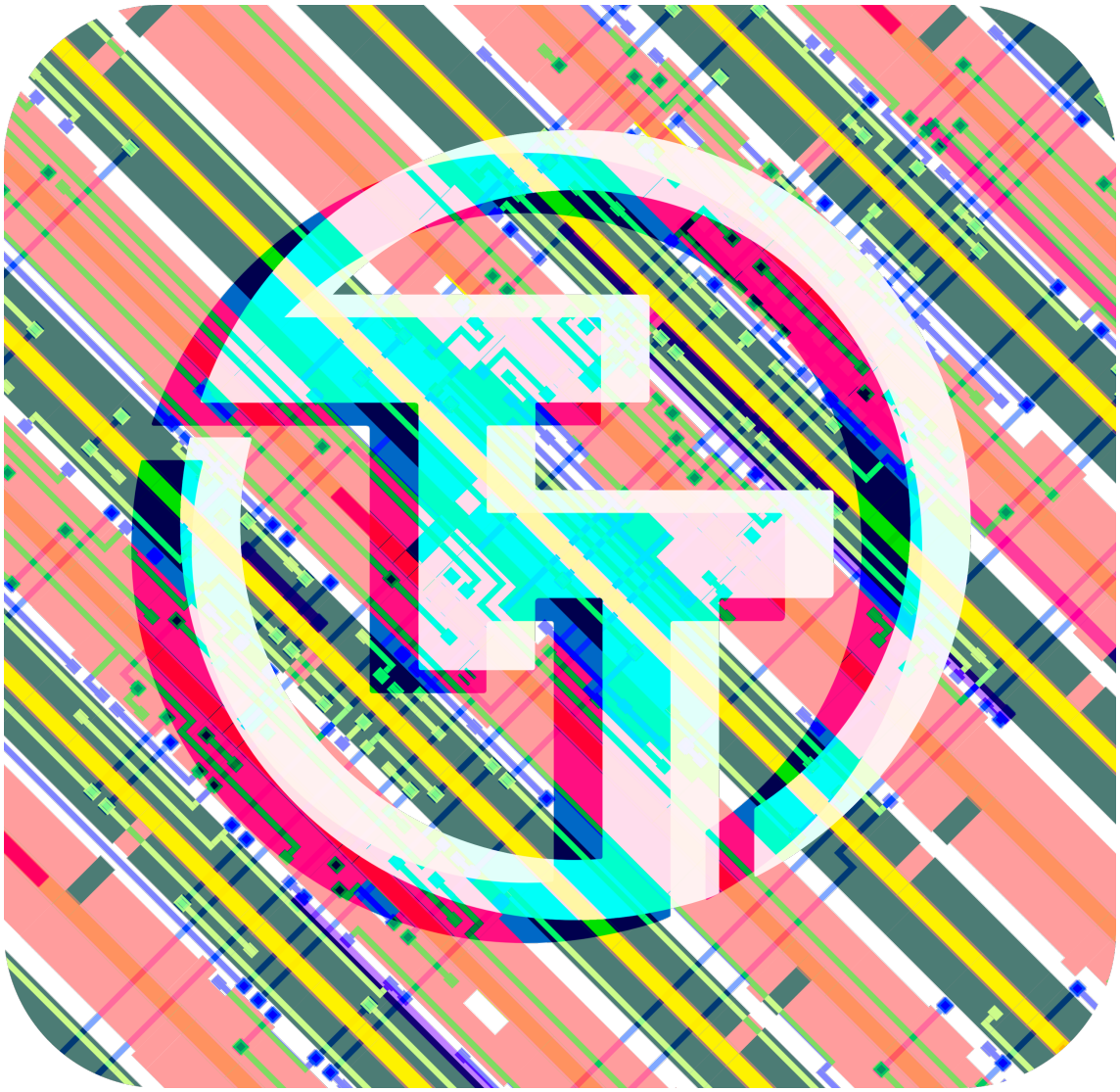


Tiny Tapeout 8 Datasheet

Project Repository

https://github.com/TinyTapeout/tinytapeout-08

September 16, 2024

Contents

Chip map

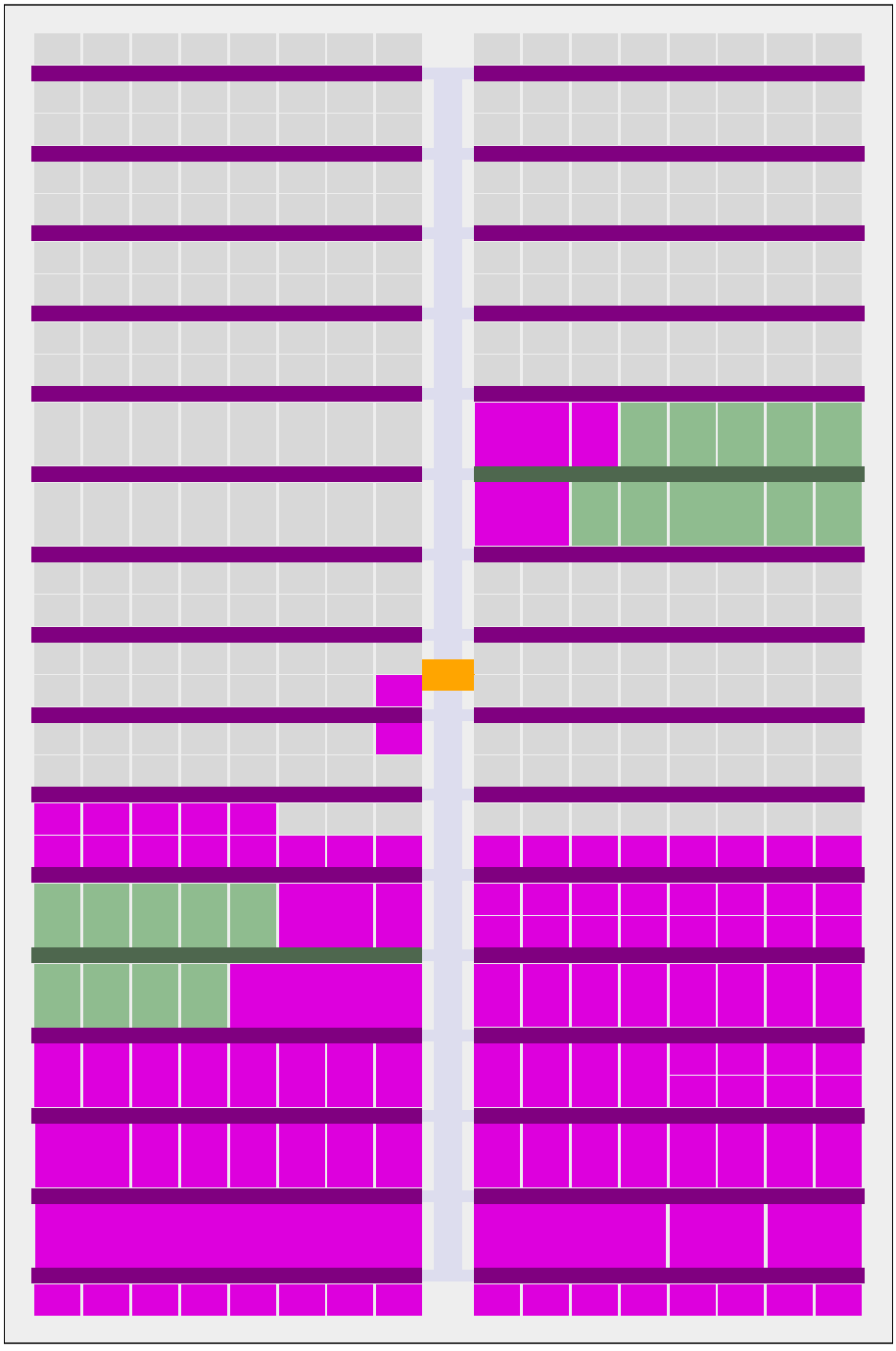


Figure 1: Full chip map

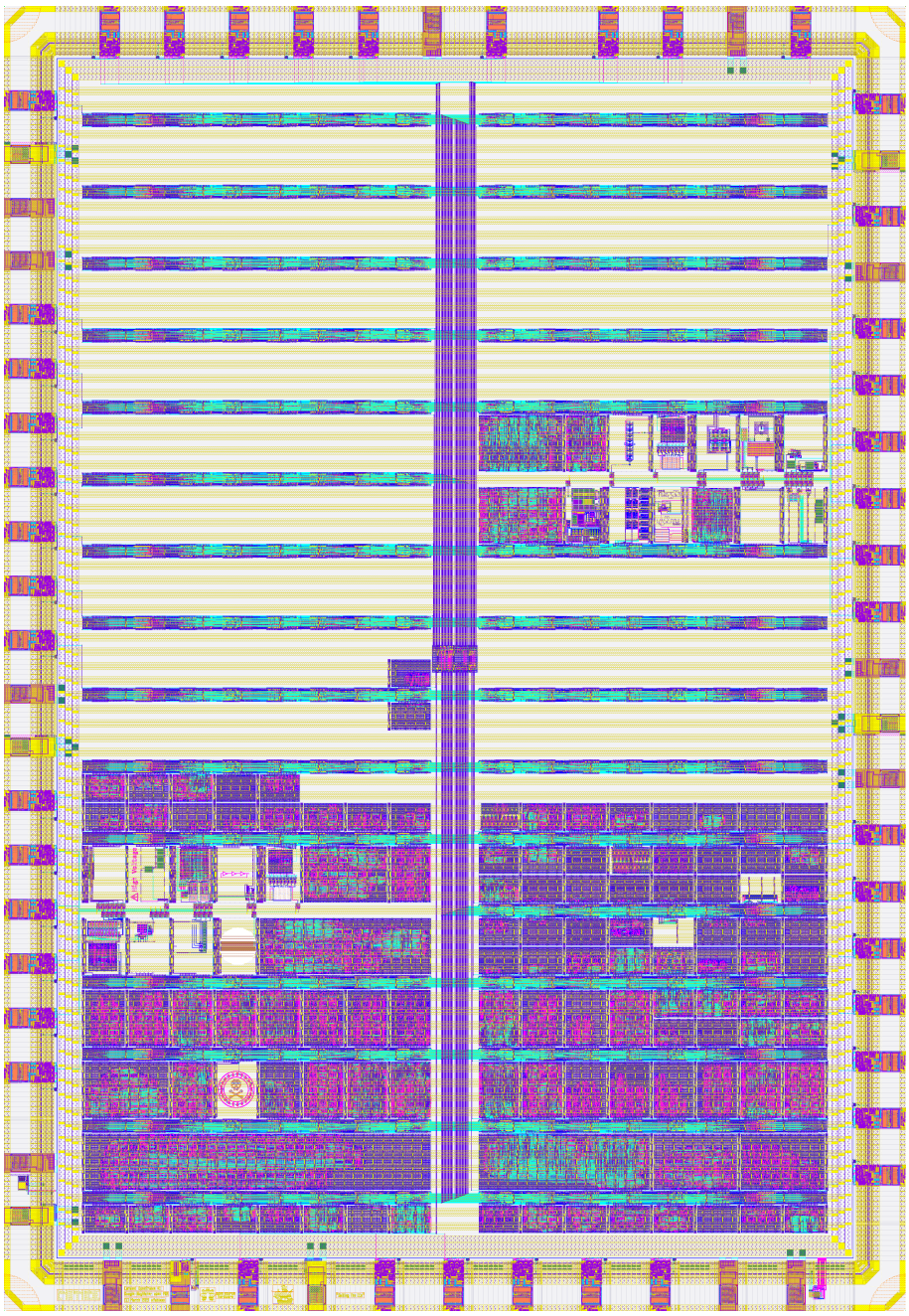


Figure 2: GDS render

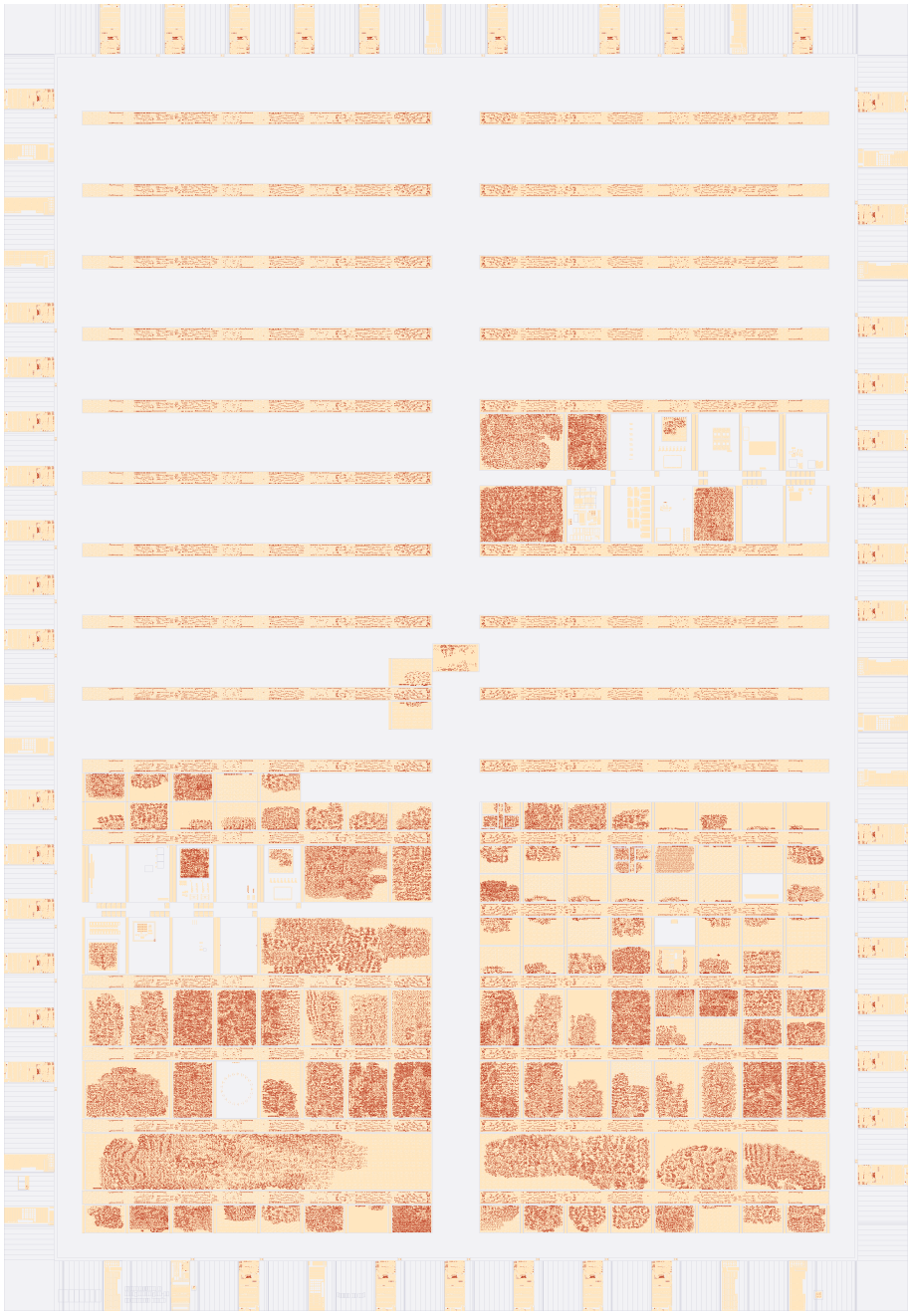


Figure 3: Logic density (local interconnect layer)

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.

How to test

Read the ROM contents by setting the address pins and reading the data pins. The first 16 bytes of the ROM are 7-segment encoded and contain the shuttle name and commit hash. You can read them by toggling the first four DIP switches and 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]	

TinyTapeout 8 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]

RGB Mixer [135]

- Author: Tianmin (Kevin) Kong
- Description: First ASIC Project!
- GitHub repository
- HDL project
- Mux address: 135
- Extra docs
- Clock: 10000000 Hz

How it works

Debounce the inputs, drive an encoder module, and output a PWM signal for each encoder.

How to test

Twist each encoder and the LEDs attached to the outputs should change in brightness.

By setting the debug port to 0, 1 or 2, the internal value of each encoder is output on the bidirectional outputs.

External hardware

Use 3 digital encoders attached to the first 6 inputs.

Pinout

#	Input	Output	Bidirectional
0	enc0 a	pwm0	encoder bit 0
1	enc0 b	pwm1	encoder bit 1
2	enc1 a	pwm2	encoder bit 2
3	enc1 b		encoder bit 3
4	enc2 a		encoder bit 4
5	enc2 b		encoder bit 5
6	debug bit 0		encoder bit 6
7	debug bit 1		encoder bit 7

Counter [137]

- Author: Jasmin Mittelman
- Description: Counts till 99 when button is pressed and shows light depending if counter is even or odd.
- GitHub repository
- Wokwi project
- Mux address: 137
- Extra docs
- Clock: 50000 Hz

How it works

There are three modules: MyCounter (tests if the button is pressed or released), score (recieves the increment signal from MyCounter and sends the 7 segment information back to Wokwi module), and Wokwi (Takes the inputs from the buttons and clock. Then it sends the information to MyCounter. Next, it sends the information recieved by MyCounter back out to the 7 segment display and LEDs).

How to test

1. Click the red button to increment the counter
2. Blue LED indicates the counter is an even number
3. Red LED indicates the counter is an odd number
4. Press the black reset button to reset the counter back to 0

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	CLK	LED0	SEG_G
1	RST	LED1	DIG1
2	BTN0	SEG_A	DIG2
3		SEG_B	
4		SEG_C	

#	Input	Output	Bidirectional
5		SEG_D	
6		SEG_E	
7		SEG_F	

Why not? [139]

- Author: sylefeb
- Description: One tile something
- GitHub repository
- HDL project
- Mux address: 139
- Extra docs
- Clock: 25000000 Hz

How it works

This is a single tile 'demo' hacked on the very last day, basically during coffee breaks. It's using an old rotozoom trick, and is otherwise pretty simple.

Music is ... well it is an attempt ;)

How to test

Plug VGA pmod, power up, enjoy.

External hardware

VGA PMOD, Audio PMOD

Pinout

#	Input	Output	Bidirectional
0		R1	
1		G1	
2		B1	
3		VS	
4		R0	
5		G0	
6		B0	
7		HS	Audio (output)

Dummy Counter [141]

- Author: Chinmay
- Description: A 16-bit counter
- GitHub repository
- HDL project
- Mux address: 141
- Extra docs
- Clock: 0 Hz

How it works

Like a 16-bit counter

How to test

Like a 16-bit counter

External hardware

NA

Pinout

#	Input	Output	Bidirectional
0	count_en	b0	b8
1	mult_en	b1	b9
2	m_a0	b2	b10
3	m_a1	b3	b11
4	m_a2	b4	b12
5	m_b0	b5	b13
6	m_b1	b6	b14
7	m_b2	b7	b15

TinyFPGA resubmit for TT08 [143]

- Author: Emilian Miron
- Description: TinyFPGA
- GitHub repository
- HDL project
- Mux address: 143
- Extra docs
- Clock: 0 Hz

How it works

Configure the FPGA then look at outputs.

How to test

Configure the FPGA then look at outputs. See the tests.

External hardware

No special hardware needed.

Pinout

#	Input	Output	Bidirectional
0	input0	output0	n/a
1	input1	output1	n/a
2	input2	output2	n/a
3	input3	output3	n/a
4		output4	n/a
5		output5	n/a
6	cmd0	output6	n/a
7	cmd1	output7	n/a

VGA donut [227]

- Author: Andy Sloane
- Description: Renders a 3D torus on a VGA display
- GitHub repository
- HDL project
- Mux address: 227
- Extra docs
- Clock: 48000000 Hz

VGA Donut

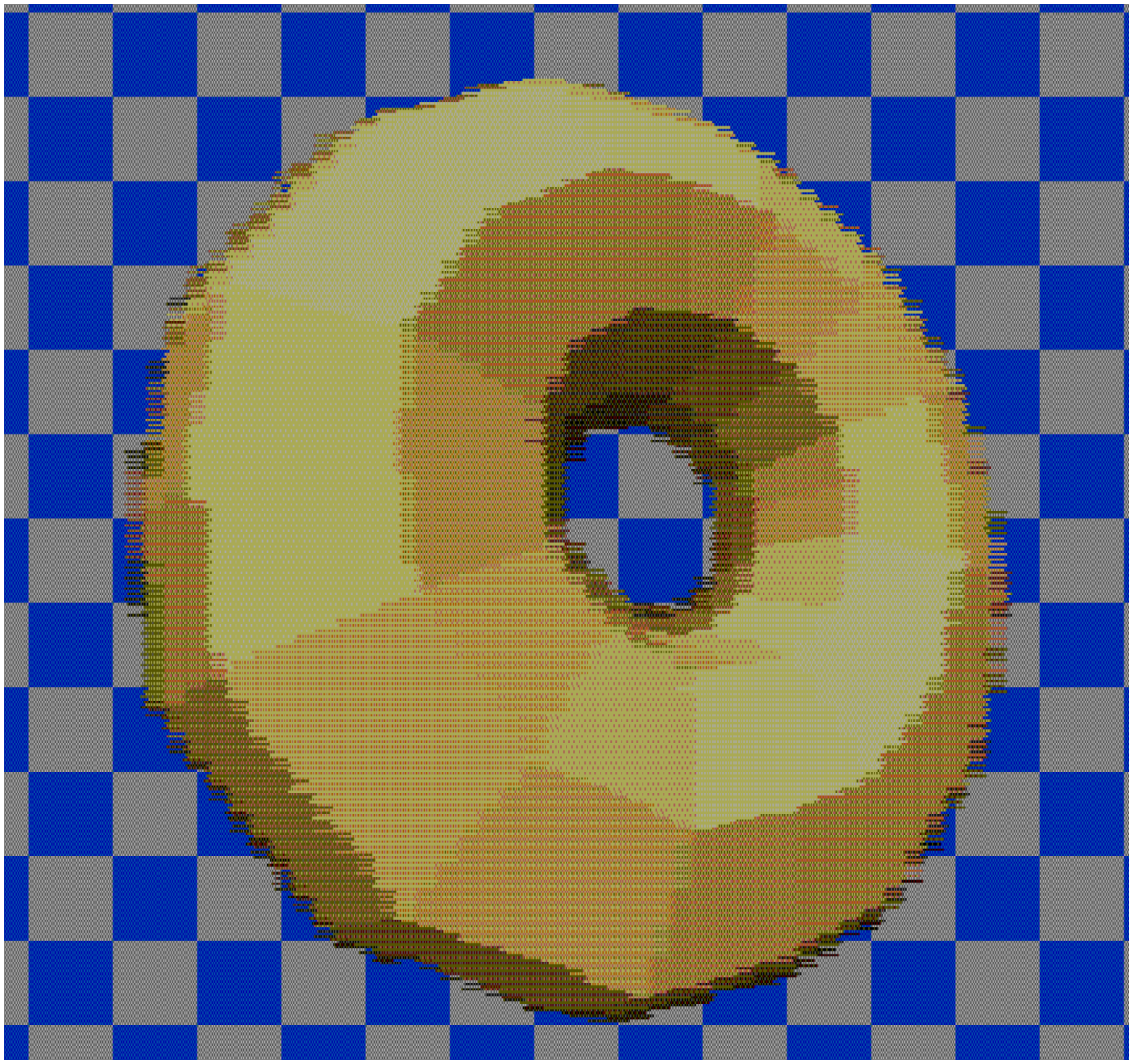


Figure 4: preview

How it works Renders a faceted donut to a VGA monitor.

Like my other demo on tt08, this runs in a weird VGA resolution: 1220x480, but still 4:3 aspect ratio like 640x480.

Interestingly, it is not actually rendering any polygons; this is sphere traced (AKA raymarched), using a CORDIC unit to calculate the distance between a point and the surface of the torus. But, because we don't have much time (we're racing the VGA beam!), we do just two or three CORDIC iterations, which causes the donut surface to actually become polyhedral. This trick was accidentally discovered by Bruno Levy while playing with a C version of my original donut code and I had to try it out in Verilog – so here we are.

The reason it has such low horizontal resolution is because it's doing 16 ray marching steps per "pixel", with five CORDIC iterations unrolled into one clock cycle (three iterations for the major axis, and two for the minor axis).

In order to fit this into 2x2 TinyTapeout tiles, a lot of sacrifices were made; for one, it doesn't have a multiplier so the ray steps are by approximate orders of magnitude. New donut "pixels" are rendered every 16 clock cycles, so the demo makes heavy use of dithering in both space and time – the video looks much better than the screenshot above.

How to test Connect VGA Pmod to output, set clock to 48MHz, and give it a reset pulse.

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

Pinout

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

LDO BG IREF OSC [229]

- Author: Lukas Baumann
- Description: Analog building-blocks for Tiny Tapeout 8
- GitHub repository
- Analog project
- Mux address: 229
- Extra docs
- Clock: 0 Hz

Analog Building Blocks for Tiny Tapeout 8

- Bandgap
  - OTA
- Ref. Current Source
- LDO
  - OPAMP
- Ring Oscillator

Pinout

#	Input	Output	Bidirectional
0	OSC_EN	OSC_OUT	
1	OSC_RESET		
2			
3			
4			
5			
6			
7			

Analog pins

ua#	analog#	Description
0	0	LDO_OUT

Bias Generator [231]

- Author: Rod Burt
- Description: A test chip for a simple CMOS beta multiplier current source.
- GitHub repository
- Analog project
- Mux address: 231
- Extra docs
- Clock: 0 Hz

How it works

A test chip to evaluate mismatch of a simple CMOS beta multiplier 1uA current source. Eight identical cells are oriented in one direction and another eight cells are rotated 90 degrees for evaluation.

How to test

All current outputs (i_out) are wired together and brought out on ua[0]. Disable individual cells 0 -> 7 with ui[0] -> ui[7] and cells 8 -> 15 with uio[0] -> uio[7].

External hardware

None

Pinout

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

Analog pins

ua#	analog#	Description
0	1	i out

AICD Playground [235]

- Author: Leo Moser
- Description: A mixed-signal test project for the analog IC design course at Graz University of Technology.
- GitHub repository
- Analog project
- Mux address: 235
- Extra docs
- Clock: 1000000 Hz

How it works

AICD Playground is a mixed-signal test project for the analog IC design course at Graz University of Technology.

It integrates a digital 8-bit controller with several analog IPs such as up/down-levelshifters, an R2R-DAC and a comparator. The IPs are connected in a way that a simple SAR-ADC is formed.

How to test

To initialize the memory, you need to set the `mode` pin to high. Now data can be send via the SPI interface, which is then written into the memory. After a rising edge of `mode`, the address points to zero and is incremented with each SPI transaction. Send 64 bytes to initialize the whole memory.

Next, set `mode` to low. The CPU starts executing the program. Depending on the program that is loaded, an analog voltage is applied to `ua[1]` via the R2R-DAC. This voltage is compared with the voltage at `ua[0]` and the result can be read by the CPU. In this way, a simple SAR-ADC can be programmed.

External hardware

No external hardware necessary.

Pinout

#	Input	Output	Bidirectional
0	port_i[0]	port_o[0]	CS
1	port_i[1]	port_o[1]	MOSI
2	port_i[2]	port_o[2]	MISO
3	port_i[3]	port_o[3]	SCK
4	port_i[4]	port_o[4]	mode
5	port_i[5]	port_o[5]	debug_i
6	port_i[6]	port_o[6]	debug_o[0]
7	port_i[7]	port_o[7]	debug_o[1]

Analog pins

ua#	analog#	Description
0	0	adc_in
1	5	dac_out

Raw_Transistors [237]

- Author: Isshu Wakita
- Description: Performance evaluation of Raw_Transistors
- GitHub repository
- Analog project
- Mux address: 237
- Extra docs
- Clock: 0 Hz

How it works

The circuit consists of raw NMOS and PMOS transistors.

How to test

ua[0] and ua[1] are connected to the source and drain of the MOSFETs (NMOS1, NMOS2, and PMOS1). The length/width are 0.25um/1.0um for all MOSFETs. ua[2] provides the gate bias for NMOS1, ua[3] provides the gate bias for NMOS2, and ua[4] provides the gate bias for PMOS1. Set the appropriate bias voltages, then measure the current.

External hardware

A source-measure unit is used to measure currents.

Pinout

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

Analog pins

ua#	analog#	Description
0	0	source
1	4	gate1
2	1	gate2
3	3	gate3
4	2	drain

TT08 Differential Receiver test [239]

- Author: Sylvain Munaut
- Description: Small test module to test functionality of a differential input receiver
- GitHub repository
- Analog project
- Mux address: 239
- Extra docs
- Clock: 0 Hz

How it works

FIXME

How to test

FIXME

External hardware

FIXME

Pinout

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

Analog pins

ua#	analog#	Description
0	5	clk_n
1	0	clk_p
2	4	data_n
3	1	data_p
4	3	ibias

Simple Stopwatch [256]

- Author: Fabio Ramirez Stern
- Description: A simple stopwatch counting in 100th seconds and outputing it via SPI to a MAX7219 chip controlling an 8 digit 7-segment display.
- GitHub repository
- HDL project
- Mux address: 256
- Extra docs
- Clock: 1000000 Hz

How it works

A clock divider turns 1 MHz into 100 Hz, which drives a stopwatch going from 00:00:00 to 59:59:99. To achieve this, a chain of two types of counting circuit, one per digit gives it's output to an SPI master that encodes the result to be displayed on a 7-segment display with at least 6 digits.

How to test

The start/stop button toggles the clock, the lap time button pauses the display, while the clock keeps running in the background. Pressing it again re-enables the display. The time can be reset with the reset button on input 2, or with the chip/PCB wide reset. The PCB wide reset affects everything, the input pin driven reset does only resets the counters.

External hardware

2-3 buttons, one for start/stop and one for lap times. For the reset, either a third button or the dev board's reset for the whole chip can be used. 1 MAX7219/MAX7221 driven 7-segment display, or something that can interpret the SPI signal according to the MAX's specifications.

Pinout

#	Input	Output	Bidirectional
0	start/stop	SPI MOSI	

#	Input	Output	Bidirectional
1	lap time	SPI CS (active low)	
2	reset (active high)	SPI CLK	
3	skip display setup (only output time, active high during reset)	stopwatch enabled (counting up)	
4		display enabled (goes low when showing lap time)	
5			
6			
7			

VGA Pong with NES Controllers [257]

- Author: Brandon S. Ramos
- Description: Pong using 2 NES Controllers with a VGA display
- GitHub repository
- HDL project
- Mux address: 257
- Extra docs
- Clock: 25175000 Hz

How it works

This project is designed to play Pong with two players using NES controllers which output to a VGA compatible monitor.

How to test

You will need two NES controllers which will take in 3 wires (not including power and ground). Hook up the connections as shown in the bidirectional I/O.

Bidirectional:

1. NES_Controller_Left[0] data
2. NES_Controller_Left[1] clock
3. NES_Controller_Left[2] latch
4. NES_Controller_Right[0] data
5. NES_Controller_Right[1] clock
6. NES_Controller_Right[2] latch
7. NC
8. NC You will also need the hook up the output to a VGA breakout board. I created my own using a perfboard and some resistors but you can use the TinyTapeout VGA PMOD, just ensure that you hook up r0,r1 on the VGA PMOD both to r from the output as my design only uses 1 bit for each signal.

Output:

1. h_sync
2. v_sync
3. r
4. g

5. b
6.
7.
8.

External hardware

- VGA PMOD or your own VGA breakout board
- 2 NES controllers
- VGA compatible monitor

Pinout

#	Input	Output	Bidirectional
0		h_sync	NES_Controller_Left[0]
1		v_sync	NES_Controller_Left[1]
2		r	NES_Controller_Left[2]
3		g	NES_Controller_Right[0]
4		b	NES_Controller_Right[1]
5			NES_Controller_Right[2]
6			
7			

RGB Mixer demo [258]

- Author: Matt Venn
- Description: Zero to ASIC demo project
- GitHub repository
- HDL project
- Mux address: 258
- Extra docs
- Clock: 10000000 Hz

How it works

Debounce the inputs, drive an encoder module, and output a PWM signal for each encoder.

How to test

Twist each encoder and the LEDs attached to the outputs should change in brightness.

By setting the debug port to 0, 1 or 2, the internal value of each encoder is output on the bidirectional outputs.

External hardware

Use 3 digital encoders attached to the first 6 inputs.

Pinout

#	Input	Output	Bidirectional
0	enc0 a	pwm0	encoder bit 0
1	enc0 b	pwm1	encoder bit 1
2	enc1 a	pwm2	encoder bit 2
3	enc1 b		encoder bit 3
4	enc2 a		encoder bit 4
5	enc2 b		encoder bit 5
6	debug bit 0		encoder bit 6
7	debug bit 1		encoder bit 7

VGA clock [260]

- Author: Matt Venn
- Description: Shows the time on a VGA screen
- GitHub repository
- HDL project
- Mux address: 260
- 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	

Find The Damn Issue [261]

- Author: Leonel Gouveia Ergin (Synogate), Michael Offel (Synogate)
- Description: USB to UART/SPI/I2C/JTAG/GPIO adapter
- GitHub repository
- HDL project
- Mux address: 261
- Extra docs
- Clock: 12000000 Hz

How it works

It is a bit bang device to interface that can be used to communicate to various devices over UART, SPI, 3wire, I2C, JTAG, GPIO or many custom protocols. To the host it registers as a USB Communication Device Class (CDC) device.

UART mode It is in UART mode by default. In UART mode it can be used as a standard CDC device with configurable baud rate and 8 data bits. You can make use of any tool that supports COM ports, like Visual Studio's Serial Monitor, to send and receive data or configure the baud rate.

Pins TX, RX, DTR and RTS are used in UART mode. DTR and RTS can be set by most tools and can be used as GPIO. There is no flow control implemented.

BitBang mode In BitBang mode, the device can be used similar to an FTDI MPSSE. **To enter BitBang mode set the baud rate to 57600 and parity to even.** A description of the protocol and its commands can be found in /libs/gatery/doc/BitBangEngine/BitBangEngine.md. There is also a collection of examples and a c++ header-only API in /example/. In contrast to the FTDI chips this is not a clone. It does not pretend to be from FTDI, nor does it support the FTDI driver or API. It acts as a standard CDC device in BitBang mode and can be used by and program that supports writing and reading to serial ports.

Note that on tiny tapeout we choose to follow the pinout templates of the tiny tapeout wiki. The documentation is written for the default pinout. Instead, refer to the pinout table below for each pins function.

How to test

1. Connect a device to communicate to. A good one is a LIS2DH12, for its wide range of protocols and available example code in this repo.
2. Connect USB_DP and USB_DN to your computer's USB. And attach the external pull up of 1.5k ohm to 3.3V.
3. Compile and run /example/LIS2DH12.cpp using CMake on Linux or Windows.
4. You should see sensor readings from the device on your screen.

External hardware

An external pull up of 1.5k to 3.3V on USB_DP is required.

Pinout

#	Input	Output	Bidirectional
0	GPIOh0	GPIOh0/DTR	GPIOl0-CS
1	GPIOh1	GPIOh1/RTS	GPIOl1-MOSI/TX
2	GPIOh2	GPIOh2	GPIOl2-MISO/RX
3	GPIOh3	GPIOh3	GPIOl3-CLK
4	GPIOh4	GPIOh4	GPIOl4-TMS
5	GPIOh5	GPIOh5	GPIOl5-WAIT
6	GPIOh6	GPIOh6	USB_DP
7	GPIOh7	GPIOh7	USB_DN

PWM generator [262]

- Author: Matea Samuel
- Description: Generate pwm signal with configurable period - 12-bit - and duty cycle - 1%-99%.
- GitHub repository
- HDL project
- Mux address: 262
- Extra docs
- Clock: 50000000 Hz

How it works

This design is intended to function as a PWM generator. It contains two 12-bit registers: one for the duty cycle (duty_reg) and one for the period (period_reg). When the sel signal is set to "0," the duty_reg is selected, and when sel is "1," the period_reg is selected. If values for the duty cycle or period are provided at the input, they are written to the registers only when wr_en is set to "1." For the duty cycle, only 7 bits (from 0 to 6) are used, with the remaining bits hardcoded to 0. The value for period_reg can range from 2 to 4095 (using 12 bits). The pwm_out signal will be available only when out_en is set to "1."

How to test

Connect the output to an oscilloscope and verify whether the frequency and duty cycle correspond to your expectations.

External hardware

The only external hardware required is the wire for the pwm_out signal and 15 inputs (such as a microcontroller, digital pattern generator, etc.) to set the period, duty cycle and controls signals: sel, wr_en, and out_en.

Pinout

#	Input	Output	Bidirectional
0	in[0]	pwm_out	in[8]
1	in[1]		in[9]

#	Input	Output	Bidirectional
2	in[2]		in[10]
3	in[3]		in[11]
4	in[4]		
5	in[5]		out_en
6	in[6]		sel
7	in[7]		wr_en

Bucket Brigade [263]

- Author: Wallie Everest
- Description: Switched capacitor delay line
- GitHub repository
- Analog project
- Mux address: 263
- Extra docs
- Clock: 100000 Hz

How it works

A cascade of NMOS and capacitors perfomr a delay line.

How to test

A two-phase non-overlapping clock is applied to DIN_0 and DIN_1.

External hardware

A 1Vpp audio signal is applied to UA_0. A summing amplifier is connected to analog outputs UA_1 and UA_2.

Pinout

#	Input	Output	Bidirectional
0		PHASE_1	
1		PHASE_2	
2			
3			
4			
5			
6			
7			

Analog pins

ua#	analog#	Description
0	11	AIN

DMTD [264]

- Author: Armaan Gomes
- Description: A Dual Mixer Timer DIfferential
- GitHub repository
- HDL project
- Mux address: 264
- Extra docs
- Clock: 0 Hz

How it works

Cool stuff makes cool stuff happen Explain how your project works

How to test

Plug cool stuff into the chip and it will output cool stuff Explain how to use your project

External hardware

You need some cool microphones and a cool clock generator and a cool i2s reciever List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Pinout

#	Input	Output	Bidirectional
0	Bit Clock (3.072 Mhz)	PCM Out Mic 0	Delay Select 0
1	LR Clock (48Khz)	PCM Out Mic 1	Delay Select 1
2	PDM Input Mics 0,1	PCM Out Mic 2	Delay Select 2
3	PDM Input Mics 2,3	PCM Out Mic 3	Delay Select 3
4	PDM Input Mics 4,5	PCM Out Mic 4	Delay Select 4
5	PDM Input Mics 6,7	PCM Out Mic 5	Beamformed PCM Output
6		PCM Out Mic 6	Mic Clock
7		PCM Out Mic 7	

Ring Oscillators [265]

- Author: Matt Venn
- Description: Ring Oscillators using analog output pins
- GitHub repository
- Analog project
- Mux address: 265
- Extra docs
- Clock: 0 Hz

How it works

Aiming to create 2 ring oscillators at around 600MHz and 300MHz. The output will be quite attenuated due to the pad.

- Ring oscillator 1 is made of 18 inverters and a NAND gate for enable.
- Ring oscillator 2 is made of 36 inverters and a NAND gate for enable.

To get a good output current, a 2 stage inverter is used with large drive transistors.

- Ring oscillator 1
- Ring oscillator 2
- Driver

The output waveform of the 600MHz is expected to be as shown in the cyan trace (out_parax). The ring_out_parax and pre_drive_parax are internal signals. See the xschem test bench for more details.

How to test

- Enable 600 MHz oscillator 1 by setting user input pin 0 high and measure the signal at analog output 0.
- Enable 300 MHz oscillator 2 by setting user input pin 1 high and measure the signal at analog output 1.

External hardware

Oscilloscope.

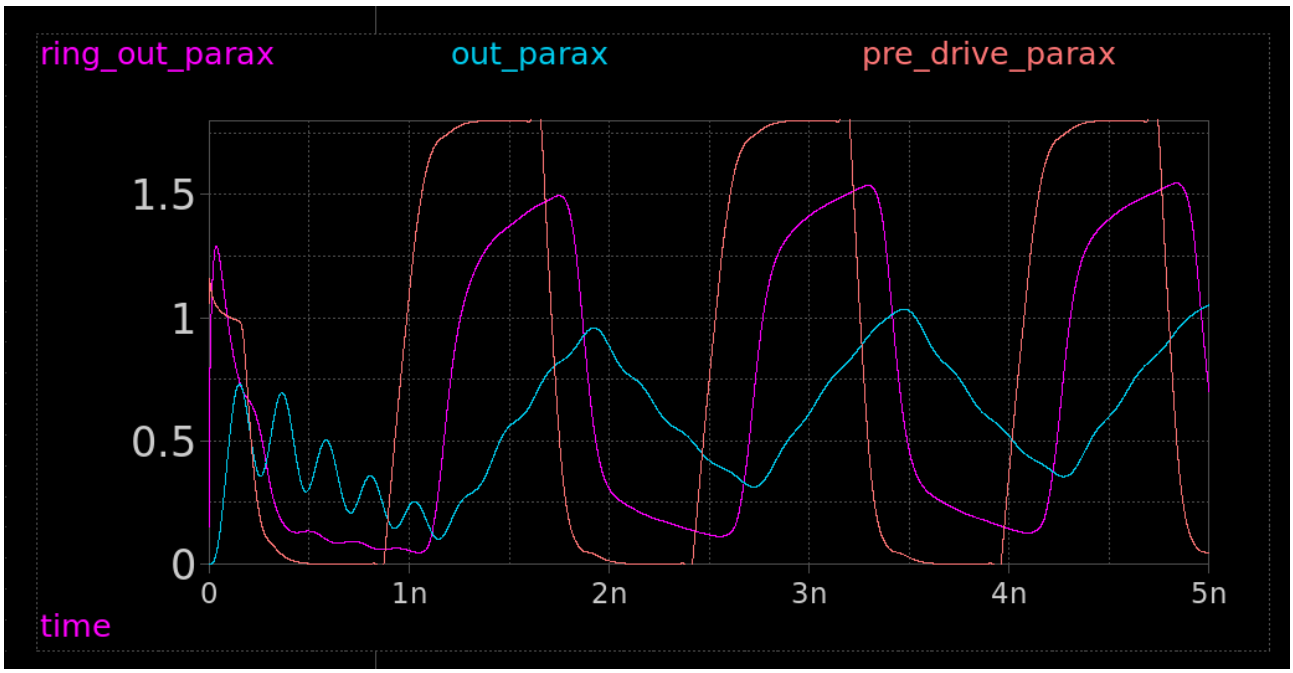


Figure 5: output waveform

Pinout

#	Input	Output	Bidirectional
0	Enable ring 1	ring_oscillator1	
1	Enable ring 2	ring_oscillator2	
2			
3			
4			
5			
6			
7			

Analog pins

ua#	analog#	Description
0	11	
1	6	

I2S to PWM [266]

- Author: Armaan Gomes
- Description: An 8-bit I2S to PWM convertor
- GitHub repository
- HDL project
- Mux address: 266
- Extra docs
- Clock: 0 Hz

How it works

Cool stuff makes cool stuff happen Explain how your project works

How to test

Plug cool stuff into the chip and it will output cool stuff Explain how to use your project

External hardware

You need some cool microphones and a cool clock generator and a cool i2s reciever List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Pinout

#	Input	Output	Bidirectional
0	Bit Clock (3.072 Mhz)	PCM Out Mic 0	Delay Select 0
1	LR Clock (48Khz)	PCM Out Mic 1	Delay Select 1
2	PDM Input Mics 0,1	PCM Out Mic 2	Delay Select 2
3	PDM Input Mics 2,3	PCM Out Mic 3	Delay Select 3
4	PDM Input Mics 4,5	PCM Out Mic 4	Delay Select 4
5	PDM Input Mics 6,7	PCM Out Mic 5	Beamformed PCM Output
6		PCM Out Mic 6	Mic Clock
7		PCM Out Mic 7	

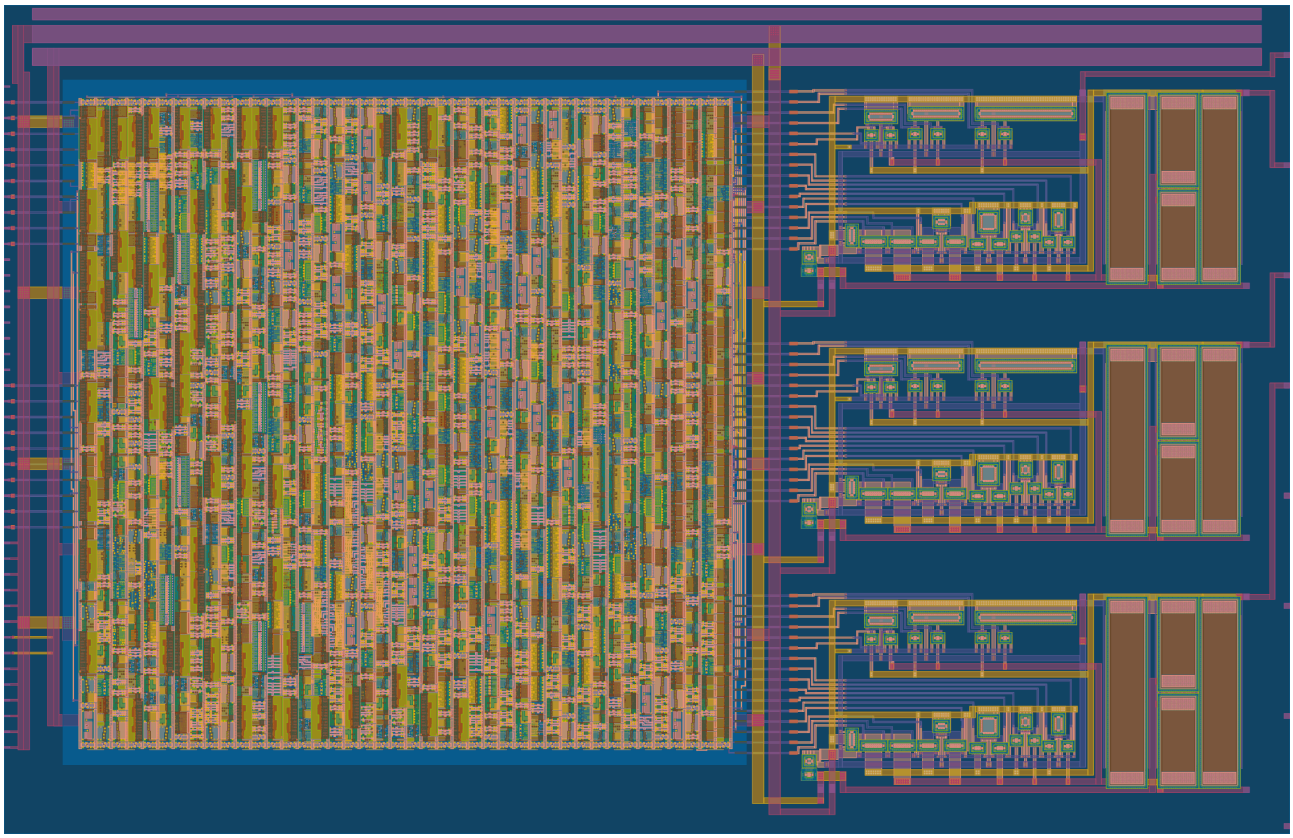
TT08 VGA FUN! [267]

- Author: algofoogle (Anton Maurovic)
- Description: Rough 24-bit VGA DAC tests with digital control block
- GitHub repository
- Analog project
- Mux address: 267
- Extra docs
- Clock: 25000000 Hz

Overview

This project (tt08-vga-fun) uses (roughly-designed) current steering DACs to hopefully produce analog outputs that can produce an adequate RGB888 (24-bit) VGA image, based on patterns that can be generated from a simple digital controller. This improves on my previous tt06-grab-bag – my 1st analog ASIC project, included on TT06, using 3 RDAC instances instead.

With these current steering DACs, I'm hoping for an improved slew rate (estimated to be about 60-80nS; still below the target of 40nS, but better than the TT06 version which was estimated to be about 240nS).

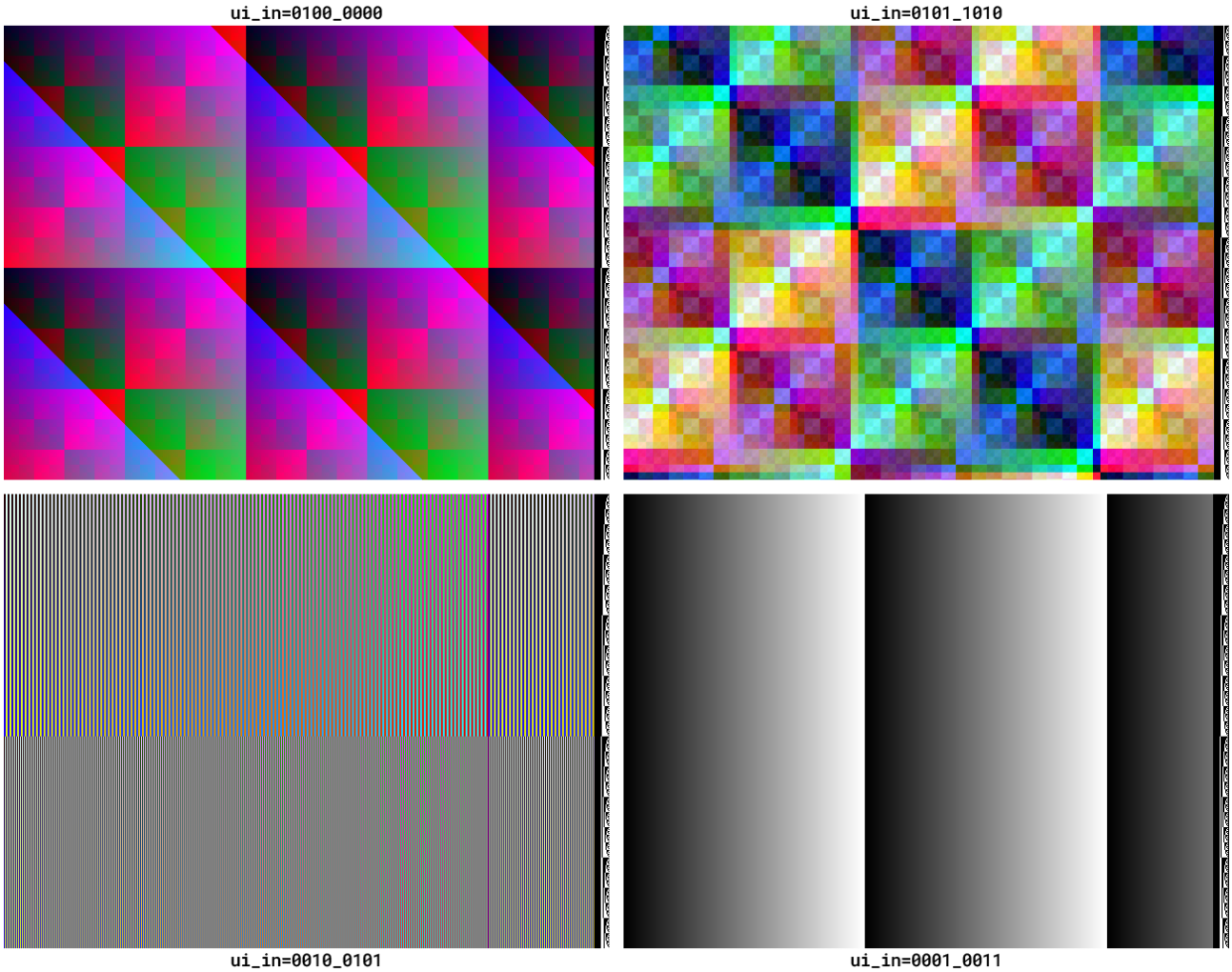


Note that the analog R/G/B outputs (`ua[1:3]`) are expected to be in the range 0.9-1.8V, and high impedance, while VGA requires a 0.0-0.7V range and 75Ω impedance. Thus, external opamps will be required.

How it works

There is a digital control block which can be controlled by the state of the `ui_in` pins at reset. It has various test modes, and a pass-through mode.

Here are some of the test patterns it can produce, but note that the image probably won't be this clear because of: (a) poor matching; and (b) slew simulated to be worse than 40nS will lead to a little bit of horizontal smearing:



The digital control block internally drives 3 (RGB) colour channels, each of which has 8 positive and 8 negative polarity bits. This complementary polarity is required for switching the binary-weighted current steering transistors either one way or the other, maintaining an equal (estimated) current of 500µA per channel. Each channel's internal current sum is then converted to a voltage with a pull-up resistor that is about 2.3kΩ.

Additionally the first analog output pin (`ua[0]`) is the internal `VbiasR` of the red channel DAC (gate voltage for current mirroring); this is for testing, but could possibly also be pulled up or down a little to see what effect it has on the red channel's output.

How to test

TBC.

External hardware

Probably an op-amp on each analog output, plus a VGA connector.

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	bias1_in
6	mode[6] / dac_in[6]	b6	bias2_in
7	mode[7] / dac_in[7]	hsync	bias3_in

Analog pins

`ua#`	`analog#`	Description
0	10	VbiasR
1	7	r
2	9	g
3	8	b

CEJMU Beers and Adders [268]

- Author: Prof. Dr.-Ing. Matthias Jung, Philipp Wetzstein, Derek Christ, Jonathan Hager
- Description: Several projects to show in lectures. Includes a simple state-machine, a decoder and two 24 bit adders. Refer to documentation for details
- GitHub repository
- HDL project
- Mux address: 268
- Extra docs
- Clock: 12000000 Hz

How it works

The goal of our design is to be able to show different RTL designs on a real chip in our lectures. Therefore, an internal multiplexer selects different projects. The multiplexer is controlled by uio_in[1:0]. The following designs can be selected:

- state machine that models a vending machine
- decoder to attach the vending machine to a coin acceptor
- 24 bit Ripple Carry Adder
- 24 bit Carry Lookahead Adder

How to test

- 00: A state machine, which models a vending machine. This state machine outputs 1, if 1.50€ have been fed into it. Inputs are taken from ui_in[1:0] with the following meaning: 00 = 0€ (nothing changes), 01 = 0.50€, 10 = 1€, 11 = undefined
- 01: A module that decodes pulses coming from a coin acceptor into coin ids. The number of pulses is equivalent with the decoded id. With a second instance of the vending machine automaton, this module makes it possible to physically insert coins into the machine.
- 10: Ripple Carry Adder with 24 bit input and 25 bit output
- 11: Carry Lookahead Adder with 24 bit input and 25 bit output

Since we only have 8 bit input and output, an internal logic is responsible for taking the inputs in 8 bit chunks and outputting the results in 8 bit chunks. This logic can be used as follows:

1. Select the adder you want to use: uio_in[1:0] == 10 (RCA) or 11 (CLA)

2. Reset the chip for at least one cycle
3. ui_in[7:0] should now be assigned a[23:16]
4. Wait for one cycle, repeat with a[15:8], a[7:0]
5. Repeat with b[23:16], b[15:8], b[7:0]
6. The inputs are now read into the design and will be send to the adders by asserting uio_in[2] to 1 (this is done to have a reference signal when measuring)
7. If you are ready to read the outputs, set uio_in[3] to 1 and wait one cycle
8. z[23:16] can now be read from uo_out
9. Wait one cycle, z[15:8] can now be read
10. Repeat for z[7:0]

Note that the overflows of both adders are always brought out to uio_out[7:6] to allow measurements. A reset upon changing the design is required to ensure valid results

External hardware

No external hardware is strictly required. Since the goal of both adders is to measure the difference in execution speed, an oscilloscope is helpful. The decoder for the coin acceptor was designed for the HX-916

Pinout

#	Input	Output	Bidirectional
0	Multiplexed to all designs (refer to documentation for details)	Multiplexed from all designs (refer to documentation for details)	Select design (input)
1	...	...	Select design (input)
2	...	...	start_calc
3	...	...	output_result
4	...	...	unused
5	...	...	unused
6	...	...	overflow bit of RCA (output)
7	...	...	overflow bit of CLA (output)

Dickson Charge Pump [269]

- Author: Uri Shaked
- Description: Pumps the input voltage up to ~3.65V
- GitHub repository
- Analog project
- Mux address: 269
- Extra docs
- Clock: 2000000 Hz

How it works

A 3-stage dickson charge pump. The output voltage is `Vout = 4*(VPWR - Vd) = ~3.6 V` where `VPWR` is the digital input voltage (1.8 V), and Vd is the diode drop (~0.9 V). The output voltage is divided by two and available at the `ua[0]` pin.

How to test

Apply a clock signal of 2 MHz to the `clk` input. In TT07, the analog pin voltage is limited to VDDIO/VDDA (usually 3.3 V), so the output voltage will be divided by two. You can measure the divided output voltage at the `ua[0]` (vout_div) pin.

Simulation results

Post layout simulation showing the output voltage `x1.vout` and the divided output voltage on ta `ua[0]` pin. The output voltage stabilizes at ~3.65 V, and the divided output voltage at ~1.82 V. The current draw is about 623.5 nA (measured by adding a 1k resistor between `ua[0]` and VGND in simulation).

The following graph shows the input clock, the intermediate voltages at the output of each stage, the output voltage, and the divided voltage as they rise during the first 10 us of operation.

Project layout

Pinout

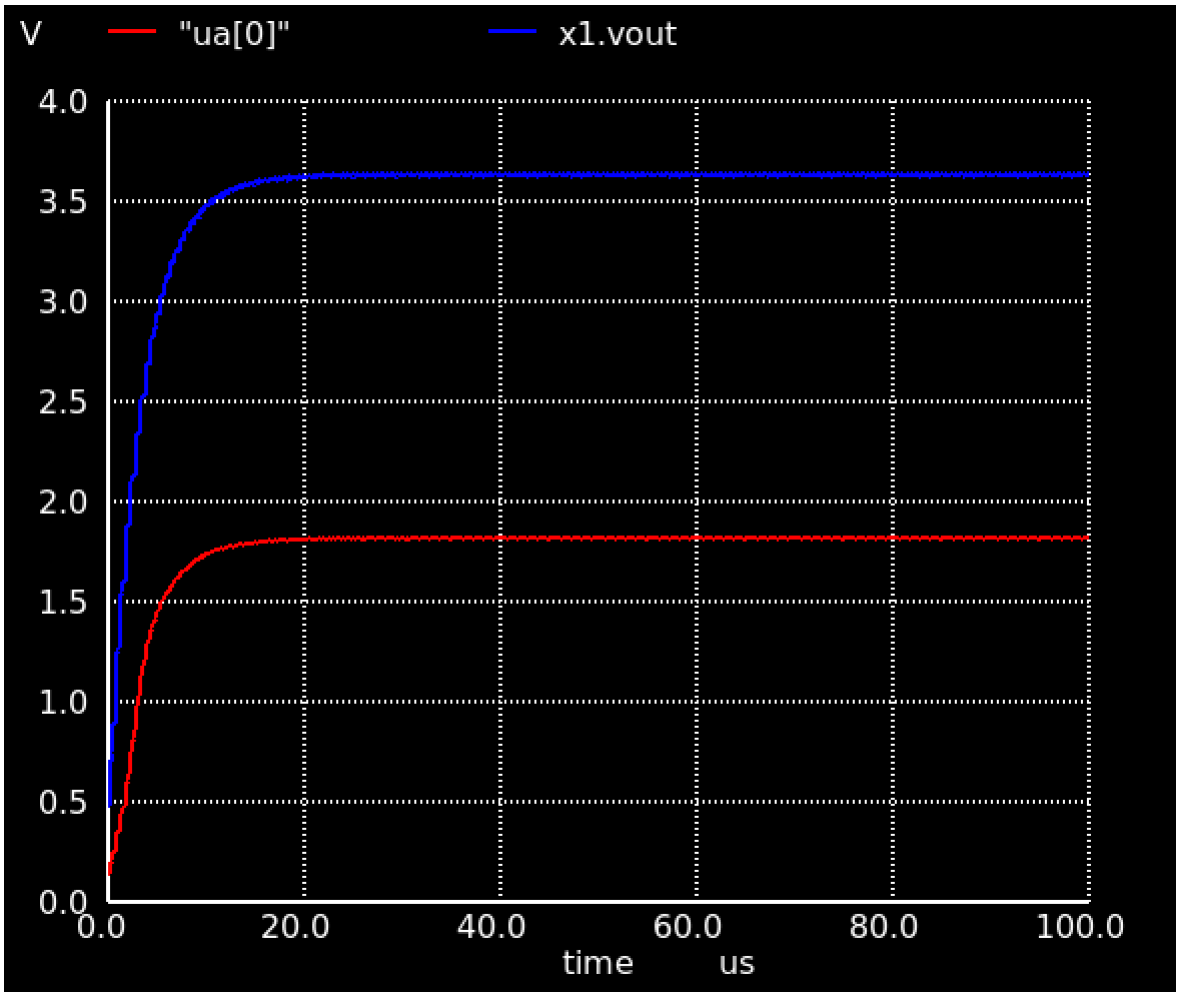


Figure 6: output voltage and divided voltage

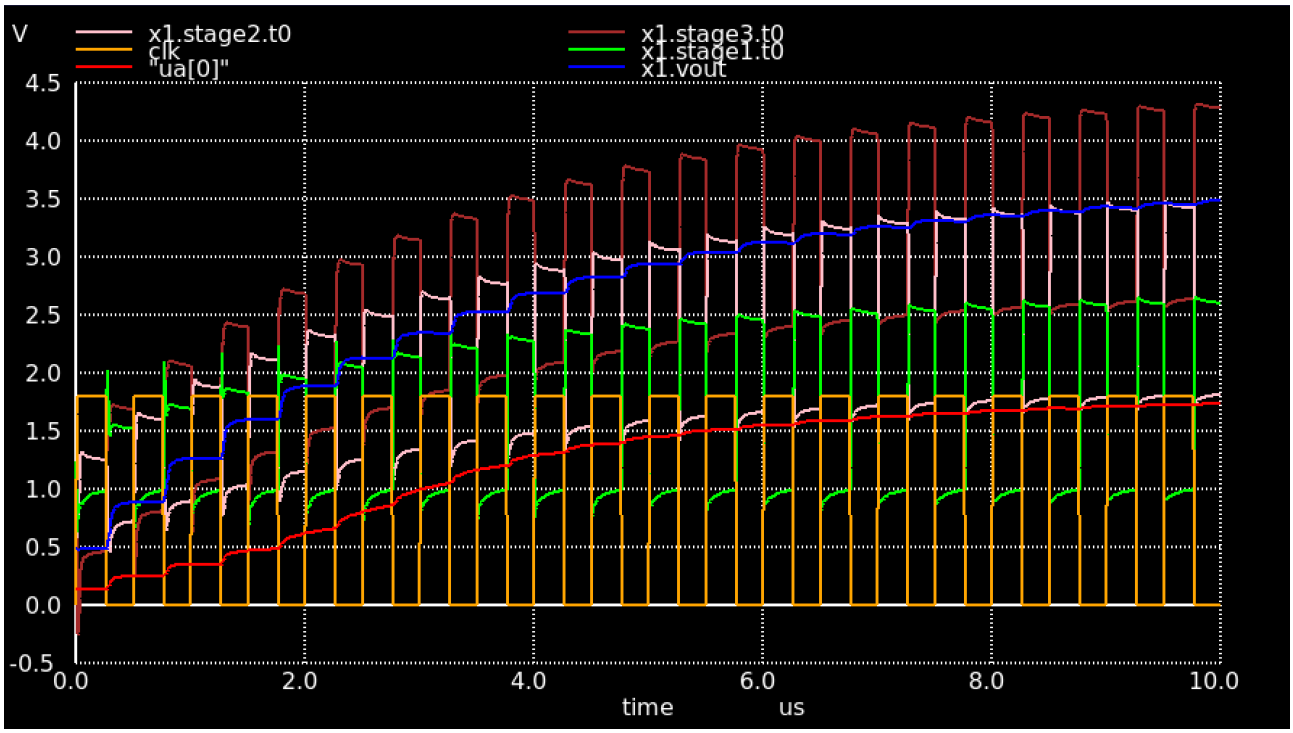


Figure 7: output voltage and intermediate voltages

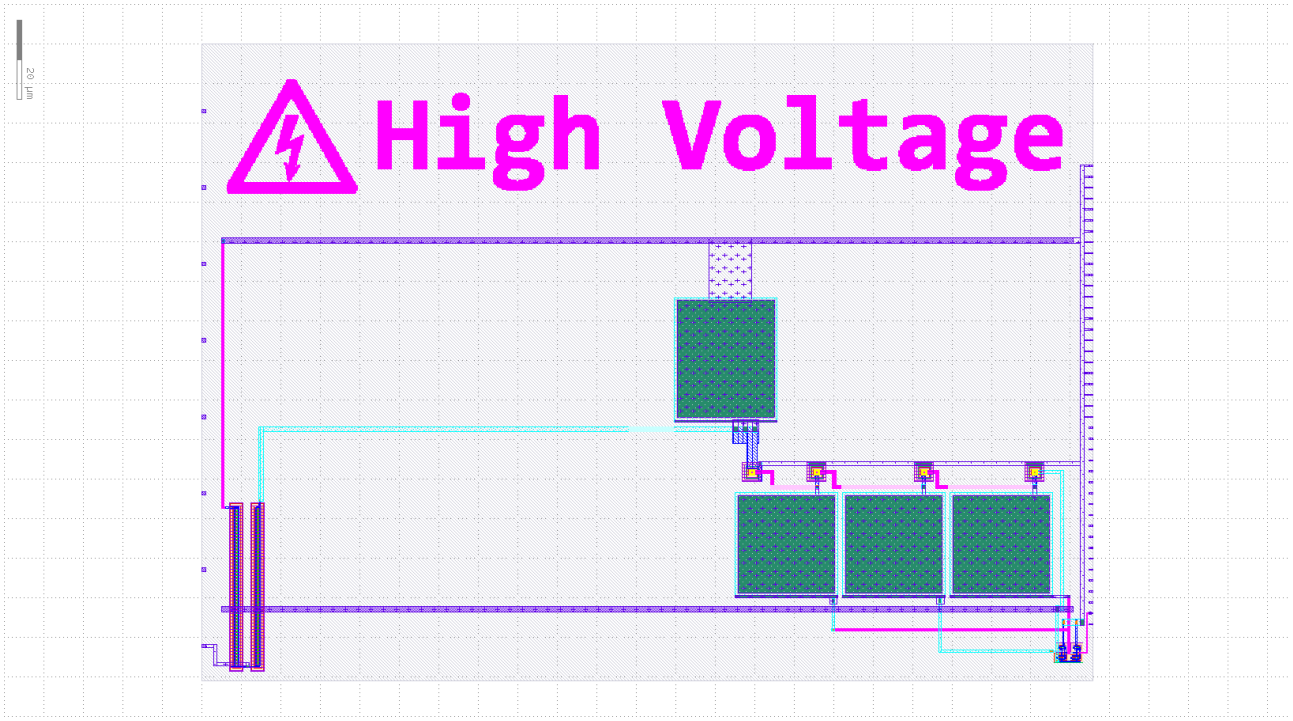


Figure 8: Project layout

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

Analog pins

ua#	analog#	Description
0	8	vout_div

resfuzzy [270]

- Author: roshan
- Description: calculation
- GitHub repository
- HDL project
- Mux address: 270
- Extra docs
- Clock: 0 Hz

How it works

The project implements a fuzzy logic system that estimates "risk" based on rainfall and soil moisture. It uses triangular membership functions to evaluate these inputs as low, medium, or high. Three fuzzy rules fire depending on the overlap between rainfall and soil moisture conditions. The system calculates the weighted average of the rule strengths to produce the risk value. If no rules fire (i.e., denominator is zero), the output risk is zero. The module updates the risk value on the clock edge when ef is enabled.

How to test

To test the fuzzy logic system,simulate different conditions by changing the input data_bus (rainfall/soil moisture data).Test the values of 80, 10, and 50 with ss (sensor select) toggling between 0 and 1 to activate the fuzzy logic. The expected output is a different risk value based on these input scenarios (FF,55,AA) (High,Low,Medium) respectively.

External hardware

8 switches connected to the input ui_in[7:0] pins 1 switch to the uio_in[0] pin 8 bit LED is needed to show the output values uo_out[7:0]

Pinout

#	Input	Output	Bidirectional
0	Input data from the sensors	risk value	sensor select
1	Input data from the sensors	risk value	

#	Input	Output	Bidirectional
2	Input data from the sensors	risk value	
3	Input data from the sensors	risk value	
4	Input data from the sensors	risk value	
5	Input data from the sensors	risk value	
6	Input data from the sensors	risk value	
7	Input data from the sensors	risk value	

TT08 Analog Factory Test [271]

- Author: Sylvain Munaut
- Description: Test structures for TT08 analog support
- GitHub repository
- Analog project
- Mux address: 271
- Extra docs
- Clock: 0 Hz

How it works

FIXME

How to test

FIXME

External hardware

FIXME

Pinout

#	Input	Output	Bidirectional
0	ena_1v8_n	digital_out	digital_out
1	ena_3v3_n	digital_out	digital_out
2		digital_out	digital_out
3		digital_out	digital_out
4		digital_out	digital_out
5		digital_out	digital_out
6	uio_oe	digital_out	digital_out
7	digital_in	digital_out	digital_out

Analog pins

ua#	analog#	Description
0	11	ibias
1	6	vapwr_sense
2	10	vgnd_sense
3	7	vdpwr_sense
4	9	loopback_a
5	8	loopback_b

Micro tile container (group 2) [320]

- Author: Uri Shaked
- Description: Experimental microtile container
- GitHub repository
- HDL project
- Mux address: 320
- Extra docs
- Clock: 0 Hz

Selecting the active project

Use uio[1:0] to select the active micro-tile project.

Project 0 - Test

- Repo: https://github.com/TinyTapeout/tt-micro-tiles-experiment
- Author: Uri Shaked
- Description: Micro tiles test module

How it works

The micro tiles test module is a simple module that demonstrates the use of the micro tile interface.

It has two modes of operation:

1. Mirroring the input pins to the output pins (when `rst_n` is low).
2. Outputing a counter on the output pins and the bidirectional pins (when `rst_n` is high).

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 observe that the counter is output on the output pins (`uo_out`).

Project 1 - Wokwi Doodle

- Repo: https://github.com/dlmiles/tt08-micro-wokwi-doodle (micro)
- Repo: https://github.com/dlmiles/tt08-wokwi-doodle (1x1)
- Wokwi: https://wokwi.com/projects/408272151035187201
- Module: tt_um_wokwi_408272151035187201
- Author: Darryl Miles
- Description: TT08 Wokwi Doodle

How to build instructions in README.md of tt08-micro-wokwi-doodle

Picture of circuit also in README.md

How it works

I don't know how it works, it is a doodle.

The aim is to see if a random doodle can be made to count on the 7SEG.

Maybe it can, maybe it can't, let the truth tables work it out.

How to test

Send all possible input combinations to the project and see what happens.

It has never been tested to find out if it is possible to observe a full set of 7SEG font states at the output.

External hardware

None, just the standard Tiny Tapeout PCB.

Project 2 - NCO

- Repo: https://github.com/gfg-development/tt-micro-tiles-nco
- Author: Gerrit Grutzeck
- Description: Micro tiles nummerical controlled oscillator, which generates a PDM stream of a sawtooth

How it works

On `ui_in` the desired frequency is set. The hightest output frequency is the clock divided by $2^{21}/(2^8 - 1)$. The lowest possible frequency is the clock divided by 2^{21}. First a phase accumulator is used to generate the sawtooth. The resulting waveform is then converted into a PDM stream in a second stage. This generated PDM datastream is output via `uo_out[7]`.

How to test

Connect the audio Pmod to the output ports or any other low pass filter with a speaker to `uo_out[7]`. Then configure the clock to a meaningfull frequency (e.g. 50 MHz for frequencies between 6 kHz and 24 Hz). Finally set the `ui_in` pins to the desired frequency (= clk / 2^{21} * `ui_in`). After applying the reset, the sawtooth will be generated.

Project 3 - Micro Maze

- Repo: https://github.com/htfab/micro-maze
- Author: htfab
- Description: A simple fixed maze game with 7-segment output

How it works

The player can walk around the maze, showing the adjacent walls on the 7-segment display.

How to test

Use the first four inputs to move up, down, left or right.

Pinout

#	Input	Output	Bidirectional
0	in[0]	out[0]	sel[0]
1	in[1]	out[1]	sel[1]
2	in[2]	out[2]	

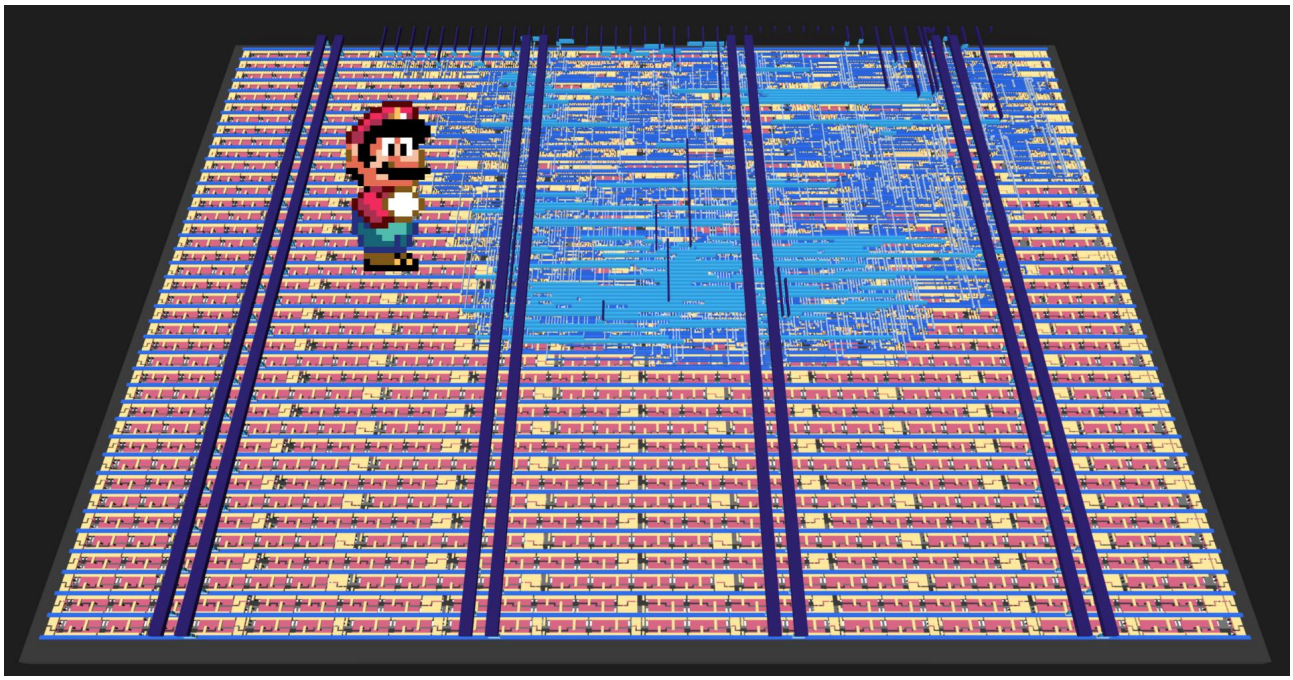
#	Input	Output	Bidirectional
3	in[3]	out[3]	
4	in[4]	out[4]	
5	in[5]	out[5]	
6	in[6]	out[6]	
7	in[7]	out[7]	

Super Mario Tune on A Piezo Speaker [321]

- Author: Milosch Meriac
- Description: Plays Super Mario Tune over a Piezo Speaker connected across uio_out[1:0]
- GitHub repository
- HDL project
- Mux address: 321
- Extra docs
- Clock: 100000 Hz

How it works

This design will play Super Mario Tune over a Piezo Speaker connected across bidir[0:1] and bidir[7]. The speaker is driven in differential PWM mode to increase its output power. The changed pinout accomodates for the Tiny Tapeout Audio Pmod.



(see also the interactive version of this design)

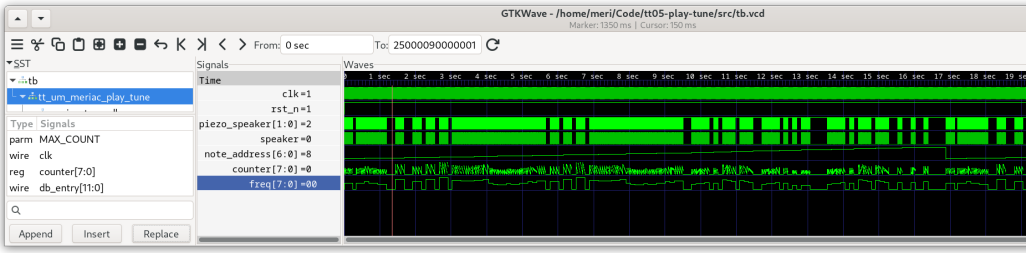
Additionally - for testing purposes, the inputs ui[7:0] are copied to the hex segment display 1:1 (uo[7:0]).

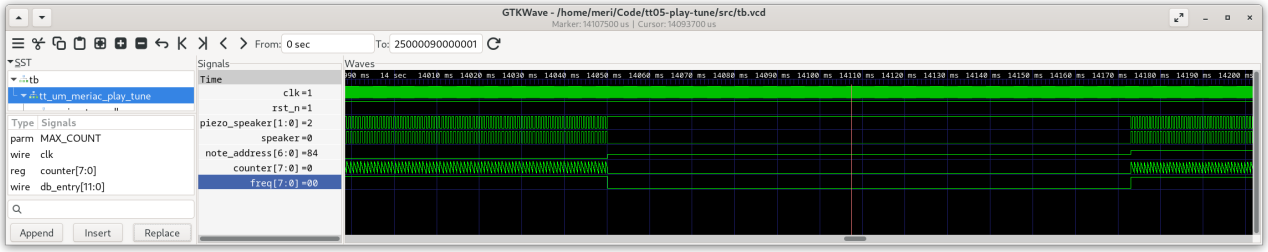
Verilog Design Files

- Playback Logic

- Autogenerated Super Mario Tune Storage. This project contains a Python-based script for converting a RTTL ringtone into optimized Verilog. An additional script converts TIM-file waveforms from the Verilog simulator back to a WAV-sound file to verify the correctness of the hardware-based player's sound.

PWM Waveform in Verilog Simulation Output Using GTKWave for visualization of Simulation Results:





How to test

Provide 100kHz clock on clk, briefly lower reset (rst_n) and bidir[1:0]/bidir[7] will play a differential sound wave over piezo speaker (Super Mario Tune).

External hardware

Piezo speaker connected across bidir[1:0] (loud) or between bidir[7] and GND (less loud). Alternatively you can connect the Tiny Tapeout Audio Pmod to the bidir port to listen to the music.

Pinout

#	Input	Output	Bidirectional
0	input pin 0	ui[0]	piezo_speaker_p (uio_out[0])
1	input pin 1	ui[1]	piezo_speaker_n (uio_out[1])
2	input pin 2	ui[2]	GND
3	input pin 3	ui[3]	GND
4	input pin 4	ui[4]	GND

#	Input	Output	Bidirectional
5	input pin 5	ui[5]	GND
6	input pin 6	ui[6]	GND
7	input pin 7	ui[7]	piezo_speaker_n (uio_out[7])

Metaballs [322]

- Author: Johannes Hoff
- Description: You can't prove it's not metaballs
- GitHub repository
- HDL project
- Mux address: 322
- Extra docs
- Clock: 50000000 Hz

How it works

An attempt at metaballs on a very rushed timeline. Keep your hopes down. Including for this documentation.

How to test

Should work like other VGA projects. No sound.

External hardware

VGA PMOD

Pinout

#	Input	Output	Bidirectional
0		R[1]	
1		G[1]	
2		B[1]	
3		vsync	
4		R[0]	
5		G[0]	
6		B[0]	
7		hsync	

AES Inverse S-box [323]

- Author: Dag Arne Osvik
- Description: Advanced Encryption Standard (AES) Inverse S-box
- GitHub repository
- HDL project
- Mux address: 323
- Extra docs
- Clock: 125000000 Hz

How it works

This circuit computes the inverse S-box of the Advanced Encryption Standard (AES).

How to test

Set the input byte, then read back the result from uo_out (unregistered) or uio_out (registered).

External hardware

None.

Pinout

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

Flame demo [324]

- Author: Konrad Beckmann & Linus Mårtensson
- Description: Flame demo
- GitHub repository
- HDL project
- Mux address: 324
- Extra docs
- Clock: 25000000 Hz

Flame - Konrad & Linus tinytapeout08 demo compo entry

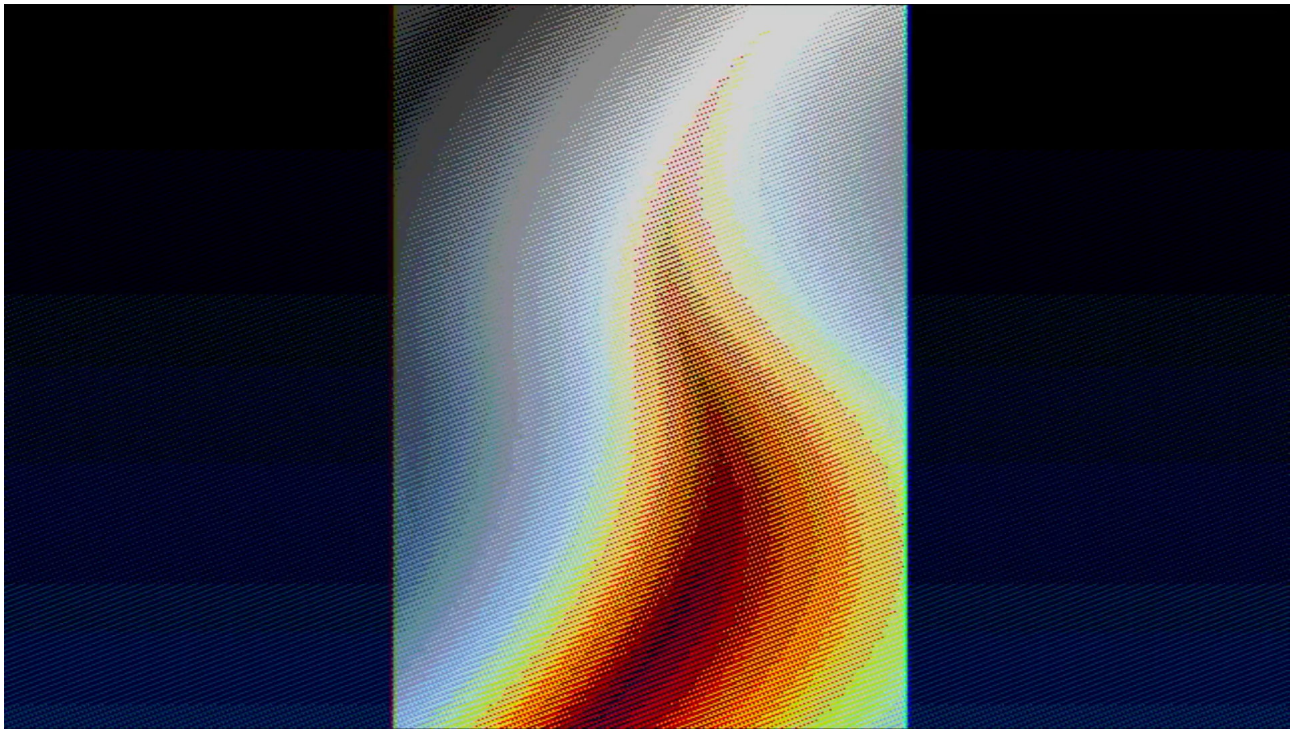


Figure 9: preview

How it works It shows a flame and plays audio. The VGA output is standard 640x480@60Hz, audio is simple 1 bit PWM.

How to test Run clock at 25MHz, connect VGA and sound Pmods, and give it a reset pulse.

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	ui_in[0]	ui_out[0]	uio_out[0]
1	ui_in[1]	ui_out[1]	uio_out[1]
2	ui_in[2]	ui_out[2]	uio_out[2]
3	ui_in[3]	ui_out[3]	uio_out[3]
4	ui_in[4]	ui_out[4]	uio_out[4]
5	ui_in[5]	ui_out[5]	uio_out[5]
6	ui_in[6]	ui_out[6]	uio_out[6]
7	ui_in[7]	ui_out[7]	uio_out[7]

Logic Test [325]

- Author: Eric Ulteig
- Description: Verify
- GitHub repository
- Wokwi project
- Mux address: 325
- Extra docs
- Clock: 0 Hz

How it works

Logic

How to test

Test

External hardware

None required

Pinout

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

SkyKing Demo [326]

- Author: Nicklaus Thompson
- Description: Types some text over an image of a plane flying into the sunset
- GitHub repository
- HDL project
- Mux address: 326
- Extra docs
- Clock: 25200000 Hz

How it works

The demo conststs of a static image of a passenger jet flying off into the sunset with a text overlay at the bottom that fills in character-by-character. The text begins typing immediately after reset, so it is likely that the entire text animation will complete before the VGA monitor recognizes the signal. It is best to view this demo in the VGA playground due to the timing issue. The project also includes demos to test some oscilloscope XY display PMODs I'm working on. The demos for these PMODs are both circles and they can be accessed by setting ui_in[0] high and using ui_in[1] to select the demo.

How to test

The demo runs automatically if all inputs are low. If ui_in[1:0] = 2'b01, an unrelated demo for a 1-PMOD XY display driver will play. If ui_in[1:0] = 2'b11, a demo for a 2-PMOD XY display driver will play.

External hardware

The demo requires the Tiny VGA PMOD on UO. The XY demos require either a 1-PMOD driver on UO, or a 2-PMOD driver on UO and UIO. The demo does not include audio.

Pinout

#	Input	Output	Bidirectional
0	0: VGA, 1: XY	ui[1:0] = 0 -> HS, 1 -> Trig, 3 -> Y0	ui[1] = 0 -> 1'b0, 1 -> X0
1	0: XY 1, 1: XY 2	ui[1:0] = 0 -> R0, 1 -> Y5, 3 -> Y2	ui[1] = 0 -> 1'b0, 1 -> X2
2		ui[1:0] = 0 -> G0, 1 -> X7, 3 -> Y4	ui[1] = 0 -> 1'b0, 1 -> X4
3		ui[1:0] = 0 -> B0, 1 -> X5, 3 -> Y6	ui[1] = 0 -> 1'b0, 1 -> X6
4		ui[1:0] = 0 -> VS, 1 -> Y6, 3 -> Y1	ui[1] = 0 -> 1'b0, 1 -> X1
5		ui[1:0] = 0 -> R1, 1 -> Y4, 3 -> Y3	ui[1] = 0 -> 1'b0, 1 -> X3
6		ui[1:0] = 0 -> G1, 1 -> X6, 3 -> Y5	ui[1] = 0 -> 1'b0, 1 -> X5
7		ui[1:0] = 0 -> B1, 1 -> X4, 3 -> Trig	ui[1] = 0 -> 1'b0, 1 -> X7

Micro tile container [327]

- Author: Uri Shaked
- Description: Experimental microtile container
- GitHub repository
- HDL project
- Mux address: 327
- Extra docs
- Clock: 0 Hz

How it works

Combined 4 micro tile sized projects into a single Tiny Tapeout tile.

Selecting the active project Use uio[1:0] to select the active micro-tile project.

Project 0 - Test

- Repo: https://github.com/TinyTapeout/tt-micro-tiles-experiment
- Author: Uri Shaked
- Description: Micro tiles test module

How it works The micro tiles test module is a simple module that demonstrates the use of the micro tile interface.

It has two modes of operation:

1. Mirroring the input pins to the output pins (when `rst_n` is low).
2. Outputing a counter on the output pins and the bidirectional pins (when `rst_n` is high).

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 observe that the counter is output on the output pins (`uo_out`).

Project 1 - Micro Shift Reg ALU

- Repo: https://github.com/MichaelBell/tt-micro-tiles-serial-compute
- Author: Michael Bell
- Description: Micro tiles shift register and 8-bit compute

How it works A 16-bit shift register that is clocked in from ui_in[0]. The low and high byte can be output on uo_out.

Additionally the result of certain computations of the low and high byte of the shift register can be latched and displayed:

- When ui_in[4] is high the result of ADDing the low and high bytes of the shift regsiter is latched
- When ui_in[5] is high the result of ANDing the low and high bytes of the shift regsiter is latched

How to test Clock data in on ui_in[0].

ui_in[2:1] select the output, as follows

ui_in[2:1]	Output
0	Low byte of shift register
1	High byte of shift register
2	Latched ADD result
3	Latched AND result

Finally, if rst_n is high the outputs mirror the inputs. Reset is otherwise unused.

Project 2 - PDM CIC Filter

- Repo: https://github.com/gfg-development/tt-micro-tiles-cic
- Author: Gerrit Grutzeck
- Description: Micro tiles CIC filter for PDM signals with a 2 stage filter and a downsampling of 4

How it works On ui_in[0] the input PDM datastream is received. Then it is processed in a CIC filter with 2 stages and a downsampling factor of 4. The resulting filtered samples are outputted on `uo_out[7:2]` and the downsampled clock on `uo_out[0]`. Furthermore on `uo_out[1]` the normal clock is available.

How to test Connect a PDM microphone as follows:

- Clock: `uo_out[1]`
- Data: `ui_in[0]`

Then configure the clock generator of the RP2040 to generate a clock, as needed by the microphone (typically around 2 MHz) and reset the design via `rst_n`. After the reset is removed again, the design is up and running, filtering the incoming datastream. With a connected logic analyzer or the RP2040 the filtered data can now be received on `uo_out[7:2]`, as well as the downsampled clock on `uo_out[0]`. The downsampled clock can be used to latch the filtered data.

Project 3 - Synchronous FIFO

How it works A generic synchronous First-In-First-Out (FIFO) buffer. It operates on a single clock domain and allows for buffering of data from an input interface (`ui_in`) to an output interface (`uo_out`), while ensuring that data is neither overwritten when full nor read when empty.

Parameters

- `DATA_WIDTH`: The width of the data in bits (default: 6).
- `DEPTH`: The depth of the FIFO in terms of the number of entries (default: 3).

Details: FIFO Depth:

- The default depth is set to 3 for practical reasons for a micro-tile, but this can be modified via the `DEPTH` parameter.

FIFO Width:

- The FIFO supports a 6-bit data width, which can also be adjusted via the `DATA_WIDTH` parameter.

Reset:

- The `rst_n` signal resets the FIFO, clearing its contents by resetting the write and read pointers (`wr_ptr` and `rd_ptr`).

Data Write:

- Data from the `ui_in` bus is written to the FIFO when the write enable signal (`wr_en`) is asserted (bit 6 of `ui_in`).

- The FIFO prevents writing if the buffer is full, indicated by the `o_full` signal.

Data Read:

- Data is read from the FIFO when the read enable signal (`rd_en`) is asserted (bit 7 of `ui_in`).
- The FIFO prevents reading if the buffer is empty, indicated by the `o_empty` signal.

Control Signals:

- `o_full`: Indicates when the FIFO has reached its maximum capacity.
- `o_empty`: Indicates when the FIFO has no data to read.

How to test

1. **Write Operation**:
    - Set the `wr_en` signal (`ui_in[6]`) high and ensure `o_full` is low (`uo_out[6]`).
    - Apply a 6-bit data value on the lower 6 bits of `ui_in[5:0]` until `o_full` is high.
2. **Read Operation**:
    - Set the `rd_en` signal (`ui_in[7]`) high and ensure `o_empty` is low (`uo_out[7]`).
    - Observe that data is read from the FIFO and appears on the lower 6 bits of `uo_out[5:0]`.
    - Once all the data has been read the `o_empty` signal will go high (`uo_out[7]`).

Inputs and Outputs Table

Signal	Description
`ui[0]`	FIFO Read Enable
`ui[1]`	FIFO Write Enable
`ui[2]`	FIFO Data Input 1
`ui[3]`	FIFO Data Input 2
`ui[4]`	FIFO Data Input 3
`ui[5]`	FIFO Data Input 4
`ui[6]`	FIFO Data Input 5
`ui[7]`	FIFO Data Input 6

Signal	Description
uo[0]	FIFO Empty Signal
uo[1]	FIFO Full Signal
uo[2]	FIFO Data Output 1
uo[3]	FIFO Data Output 2
uo[4]	FIFO Data Output 3
uo[5]	FIFO Data Output 4
uo[6]	FIFO Data Output 5
uo[7]	FIFO Data Output 6

Pinout

#	Input	Output	Bidirectional
0	in[0]	out[0]	sel[0]
1	in[1]	out[1]	sel[1]
2	in[2]	out[2]	
3	in[3]	out[3]	
4	in[4]	out[4]	
5	in[5]	out[5]	
6	in[6]	out[6]	
7	in[7]	out[7]	

4-bit CLA [328]

- Author: Wei Zhang
- Description: A 4 bit carry look-ahead adder
- GitHub repository
- HDL project
- Mux address: 328
- Extra docs
- Clock: 0 Hz

How it works

This is a 4-bit CLA. It can be used to construct the adder with higher bit.

How to test

The design has 3 input ports: a, b and ci It has 2 output ports: s and co a: an addend. b: the other addend. ci: the carry signal for the input. s: the output sum. co: the carry signal for the output.

External hardware

This project was tested by an U250 FPGA.

Pinout

#	Input	Output	Bidirectional
0	a[0]	s[0]	ci
1	a[1]	s[1]	
2	a[2]	s[2]	
3	a[3]	s[3]	
4	b[0]	co	
5	b[1]		
6	b[2]		
7	b[3]		

TT08 - experiments with latch-based shift registers [329]

- Author: Ciro Cattuto
- Description: A 512-bit latch-based shift register in 1 tile
- GitHub repository
- HDL project
- Mux address: 329
- Extra docs
- Clock: 0 Hz

How it works

This is an experiment. A 512-bit shift register (SR) implemented using D latches rather than D flip flops. The shift logic relies on a single pulse rippling along the shift register, from the output latch towards the input latch. The SR has one input, one output, and an edge-triggered control signal that controls the shift update. The SR shifts on either a rising or a falling edge of the control signal.

How to test

Shift zeros into the SR until it contains all zeros. Then shift in any sequence of 1s and 0s and observe it appear on the output of the SR after 512 transitions of the control signal.

External hardware

No external hardware required.

Pinout

#	Input	Output	Bidirectional
0	shift register input	shift register output	
1	shift control (edge-triggered)		
2			
3			
4			
5			
6			

#	Input	Output	Bidirectional
7			

Generate VGA output for Color Blindness Test [330]

- Author: Krushnasis Pradhan, Aniruddha Ranade
- Description: Generate VGA output which shall display the pattern similar to Ishihara Plates
- GitHub repository
- HDL project
- Mux address: 330
- Extra docs
- Clock: 0 Hz

How it works

Generates VGA output for displaying pattern similar to Ishihara plates used for conducting a color blindness test. (Disclaimer: Note that this is not an approved medical test and test setup. The pattern generated is for purely experimental purpose.)

How to test

This project will work out of the box. Just connect a VGA display via TinyVGA PMOD.

External hardware

TinyVGA PMOD to connect to a VGA display

Pinout

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

Abacus Lock [331]

- Author: Raunak Singh
- Description: digital lock that turns the servo on a specific sequence of switches and displays current combination entered on a LED bar
- GitHub repository
- Wokwi project
- Mux address: 331
- Extra docs
- Clock: 0 Hz

How it works

Abacus Lock is a digital combination lock that can be used in various projects. It moves the servo only when a specific sequence of switches is set. An LED Bar displays the status of the current lock combination in binary.

How to test

Follow Wokwi design and run the .ino file using Arduino: https://wokwi.com/projects/408154128142306305

External hardware

LED, LED Bar, Servo, Arduino Nano

This project was sponsored by The MITRE Corporation and MIT/LL Beaverworks Summer Institute

Pinout

#	Input	Output	Bidirectional
0	IN1	OUT1	
1	IN2	OUT2	
2	IN3	OUT3	
3	IN4	OUT4	
4	IN5	OUT5	
5	IN6	OUT6	
6	IN7	OUT7	

#	Input	Output	Bidirectional
7	IN8	OUT8	

DPM_Unit [332]

- Author: Sanjay Kumar M, Shylashree N, Ravish Aradhya H V, RV College Of Engineering, Neha R V, PES University
- Description: Design and Implementation of Dynamic Power management unit
- GitHub repository
- HDL project
- Mux address: 332
- Extra docs
- Clock: 100 Hz

Credits : We gratefully acknowledge the COE in Integrated Circuits and Systems (ICAS) and Department of ECE. Our special thanks to Dr K S Geetha (Vice Principal) and, Dr. K N Subramanya (principal) for their constant support and encouragement to do TAPEOUT in Tiny Tapeout 8 .

The code provided is a SystemVerilog module that implements a Dynamic Power Management Unit (DPMU) for an SoC (System on Chip). The DPMU dynamically adjusts voltage and frequency levels based on inputs such as performance requirements, temperature, battery level, and workload. The module uses a finite state machine (FSM) to manage transitions between different power states.

Key Components

Inputs and Outputs: Inputs (ui_in): The primary input signals include performance requirements, temperature sensor data, battery level, and workload. Outputs (uo_out, uio_out): These include the power-saving indicator, voltage levels, and frequency levels for different cores and memory. I/O (uio_in, uio_out, uio_oe): Handles bidirectional signals; however, in this design, uio_in is not used, and uio_out is used for output. Internal Signals:

State Variables: state and next_state manage the FSM that controls the DPMU's behavior. Power and Frequency Controls: Registers like vcore1, vcore2, vmem, fcore1, fcore2, and fmem store the voltage and frequency settings. Finite State Machine (FSM):

States: NORMAL: Default operating mode with standard voltage and frequency levels. PERFORMANCE: High-performance mode with maximum voltage and frequency levels. POWERSAVE: Low-power mode with reduced voltage and frequency levels. THERMAL_MANAGEMENT: Mode to handle high temperature by adjusting power levels moderately. BATTERY_SAVING: Mode to conserve battery by minimizing voltage and frequency levels.

State Transitions: Transitions between states occur based on the input conditions, such as high performance request, low battery level, high temperature, or low workload. Detailed Walkthrough Input and Output Mapping:

perf_req: Mapped to the least significant bit (LSB) of ui_in, indicating whether high performance is needed. temp_sensor: 2-bit signal derived from ui_in[3:2], providing temperature data. battery_level: 2-bit signal derived from ui_in[5:4], indicating the battery's charge status. workload_core: 3-bit signal derived from ui_in[7:6], representing the workload of a core. State Logic:

On each clock cycle (clk), the FSM checks the state and evaluates transitions based on inputs. In NORMAL state, if perf_req is high, the system transitions to PERFORMANCE state. If the battery level is low, it transitions to BATTERY_SAVING state. If the temperature is high, it transitions to THERMAL_MANAGEMENT state. If the workload is low, it transitions to POWERSAVE state. PERFORMANCE state sets all voltages and frequencies to maximum. If perf_req drops, it returns to NORMAL. POWERSAVE state reduces voltages and frequencies to conserve power. If the workload increases, it returns to NORMAL. THERMAL_MANAGEMENT state adjusts power levels to moderate values to manage high temperatures. If the temperature normalizes, it returns to NORMAL. BATTERY_SAVING state minimizes voltages and frequencies to conserve battery. If the battery level increases, it returns to NORMAL.

Output Assignment: The combined voltage (vcore1, vcore2, vmem) and frequency (fcore1, fcore2, fmem) values are assigned to the uio_out and uo_out outputs. The power_save signal is also part of the output, indicating whether the system is in power-saving mode. Behavior under Reset:

When the reset (rst_n) is low (active), the system resets to the NORMAL state.

Here's a table summarizing the expected output (uio_out, uo_out) based on the input (ui_in) and time using the provided testbench for the tt_um_dpmu module. The table provides the values for different states as the ui_in input changes over time.

Table 56: Testbench Expected Output

Time (ns)	**ui_in (Input)**	**State**	**uio_out (Expected Output)**	**uo_out (Expected Output)**
0	11110010	NORMAL	01010110	010_010010
10	00010010	PERFORMANCE	11111111	111_111111
30	11110010	NORMAL	01010110	010_010010
50	11110011	THERMAL_MANAGEMENT	01101111	011_011011
70	11110010	NORMAL	01010110	010_010010
90	11101010	THERMAL_MANAGEMENT	01101111	011_011011
110	11111010	BATTERY_SAVING	00000000	000_000000

Time (ns)	ui_in (Input)	State	uio_out (Expected Output)	uo_out (Expected Output)
130	11111110	BATTERY_SAVING	00000000	000_000000
150	11111010	BATTERY_SAVING	00000000	000_000000

Explanation of Table Columns:

Time (ns): The simulation time when the ui_in input is applied. ui_in (Input): The 8-bit input value applied to the design. State: The state of the FSM based on the ui_in input. The states are NORMAL, PERFORMANCE, THERMAL_MANAGEMENT, and BATTERY_SAVING. uio_out (Expected Output): The expected 8-bit output values for the uio_out signals. uio_out[0]: Power save mode indicator. uio_out[2:1], uio_out[4:3], uio_out[6:5]: Voltage controls. uio_out[7]: Part of fcore1[0]. uo_out (Expected Output): The expected 8-bit output values for the uo_out signals. uo_out[0:1]: Part of fcore1[2:1]. uo_out[4:2]: fcore2[2:0]. uo_out[7:5]: fmem[2:0].

Explanation of Key Points: NORMAL State: When the inputs suggest a typical operating environment (e.g., ui_in = 11110010), the design operates with default voltage and frequency levels. PERFORMANCE State: Triggered by a performance request (perf_req = 1), leading to maximum voltage and frequency levels. THERMAL_MANAGEMENT State: Triggered by high temperature (temp_sensor = 10 or 11), moderates the voltage and frequency to prevent overheating. BATTERY_SAVING State: Triggered by low battery level (battery_level = 00 or 01), minimizing power consumption by reducing voltage and frequency to the lowest levels.

Testbench Operation: The testbench applies different ui_in values at specific simulation times. At each time step, it captures the output values (uio_out and uo_out) and compares them with the expected values as per the design's FSM logic. The $monitor statement continuously logs the input and output values, helping to verify the design's behavior at each time point.

Pinout

#	Input	Output	Bidirectional
0	ui_in[[0]	uo_out[0]	uio_out[0]
1	ui_in[[1]	uo_out[1]	uio_out[1]
2	ui_in[[2]	uo_out[2]	uio_out[2]
3	ui_in[[3]	uo_out[3]	uio_out[3]
4	ui_in[[4]	uo_out[4]	uio_out[4]
5	ui_in[[5]	uo_out[5]	uio_out[5]

#	Input	Output	Bidirectional
6	ui_in[[6]	uo_out[6]	uio_out[6]
7	ui_in[[7]	uo_out[7]	uio_out[7]

Obstacle Detection [333]

- Author: Emmy Xu
- Description: Does the logic of when to send certain signals when objects are close.
- GitHub repository
- HDL project
- Mux address: 333
- Extra docs
- Clock: 0 Hz

How it works

It takes in two different numbers one for a left sensor and one for a right sensor. There is a threshold value of 1 or 0, if the threshold has been passed, it will have a value of 1. If only one side has a value of one, it will send a 2'b10 meant for a motor to the opposite side. If both sides have a value of one, it will send 2'b01 to both sides meant for motors.

How to test

Set it up so that a value of 1 or 0 is going into the sensor pins and connect the output pins to something that can read what the chip is sending out. The reset pin resets when power is sent to it, which just makes it output 0s to all the outputs.

External hardware

Ultrasonic Sensors, Microcontroller, and Haptic Motors

Pinout

#	Input	Output	Bidirectional
0	sensor_left	left_buzz	
1	sensor_right	right_buzz	
2	reset		
3			
4			
5			

#	Input	Output	Bidirectional
6			
7			

Clock Divider [334]

- Author: Armaan Gomes
- Description: A clock divider with lag correction
- GitHub repository
- HDL project
- Mux address: 334
- Extra docs
- Clock: 0 Hz

How it works

Cool stuff makes cool stuff happen Explain how your project works

How to test

Plug cool stuff into the chip and it will output cool stuff Explain how to use your project

External hardware

You need some cool microphones and a cool clock generator and a cool i2s reciever List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Pinout

#	Input	Output	Bidirectional
0	Bit Clock (3.072 Mhz)	PCM Out Mic 0	Delay Select 0
1	LR Clock (48Khz)	PCM Out Mic 1	Delay Select 1
2	PDM Input Mics 0,1	PCM Out Mic 2	Delay Select 2
3	PDM Input Mics 2,3	PCM Out Mic 3	Delay Select 3
4	PDM Input Mics 4,5	PCM Out Mic 4	Delay Select 4
5	PDM Input Mics 6,7	PCM Out Mic 5	Beamformed PCM Output
6		PCM Out Mic 6	Mic Clock
7		PCM Out Mic 7	

Styler [335]

- Author: Rebecca G. Bettencourt
- Description: 16x16 bitmap manipulation based on text mode attributes.
- GitHub repository
- HDL project
- Mux address: 335
- Extra docs
- Clock: 0 Hz

How it works

The styler chip is used to transform a 16x16 character glyph bitmap based on a set of text mode attributes. It consists of a 4-bit scanline register, a 6-bit control register, a 16-bit bitmap register, and a 25-bit attribute register. Additionally, three independent input lines are used to control polarity of dim text (even or odd pixels), text blink rate, and cursor blink rate.

Typical use of the styler chip follows these steps:

1. Set output enable (input 6) HIGH and write enable (input 7) LOW.
2. Set the address (inputs 0-2) to 0.
3. Set the bidirectional pins to the physical scanline number.
4. Pulse `clk`.
5. Set output enable (input 6) LOW and write enable (input 7) HIGH.
6. Read the logical scanline number from the bidirectional pins.
7. Set output enable (input 6) HIGH and write enable (input 7) LOW.
8. Set the address (inputs 0-2) to 2.
9. Set the bidirectional pins to the right half of the row of the character bitmap corresponding to the logical scanline number.
10. Pulse `clk`.
11. Set the address (inputs 0-2) to 3.
12. Set the bidirectional pins to the left half of the row of the character bitmap corresponding to the logical scanline number.
13. Pulse `clk`.
14. Set output enable (input 6) LOW and write enable (input 7) HIGH.
15. Set the address (inputs 0-2) to 2.
16. Read the right half of the final character bitmap from the bidirectional pins.
17. Set the address (inputs 0-2) to 3.
18. Read the left half of the final character bitmap from the bidirectional pins.

You can also read from the dedicated output pins without changing output enable or write enable.

The register layout is as follows:

Address	Bits	Description
0	0-3	Input: physical scanline number; output: logical scanline number.
0	4-7	Input: ignored; output: 0.
1	0	Show cursor at bottom of character cell.
1	1	Show cursor at top of character cell.
1	2	Enable cursor blink.
1	3	Enable cursor.
1	4	Enable character underline, strikethrough, overline attributes.
1	5	Enable character blink, alternate attributes.
1	6-7	Input: ignored; output: 0.
2	0-7	Right half of character glyph bitmap.
3	0-7	Left half of character glyph bitmap.
4	0	X offset. (Determines which half of a double-width character.)
4	1	Double width.
4	2	Y offset. (Determines which half of a double-height character.)
4	3	Double height.
4	4	X premirror (flip input bitmap horizontally).
4	5	X postmirror (flip output bitmap horizontally).
4	6	Y premirror (invert physical scanline).
4	7	Y postmirror (invert logical scanline).
5	0	Bold.
5	1	Faint.
5	2	Italic.
5	3	Reverse italic.
5	4	Blink (text only, VT100-style).
5	5	Alternate (text and background, Apple II-style).
5	6	Inverse.
5	7	Hidden.
6	0	Underline.
6	1	Double underline.
6	2	Dotted underline.
6	3	Strikethrough.
6	4	Double strikethrough.
6	5	Dotted strikethrough.
6	6	Overline.
6	7	Double overline.
7	0	Dotted overline.

Address	Bits	Description
7	1-7	Input: ignored; output: 0.

The input pin assignments are as follows:

Pin	Description
0	A0 (address line 0).
1	A1 (address line 1).
2	A2 (address line 2).
3	Faint text polarity (even or odd pixels).
4	Text blink phase.
5	Cursor blink phase.
6	/OE (output enable).
7	/WE (write enable).

How to test

Test cases are to be determined.

External hardware

The styler chip is intended to be used as part of a larger text mode video display hardware project.

Pinout

#	Input	Output	Bidirectional
0	A0 (address)	D0	D0
1	A1 (address)	D1	D1
2	A2 (address)	D2	D2
3	dim text phase	D3	D3
4	text blink phase	D4	D4
5	cursor blink phase	D5	D5
6	/OE (output enable)	D6	D6
7	/WE (write enable)	D7	D7

nyan [448]

- Author: Peter Nørlund
- Description: Nyan Cat
- GitHub repository
- HDL project
- Mux address: 448
- Extra docs
- Clock: 25000000 Hz

How it works

The Nyan Cat animation and music on infinite repeat

How to test

Connect the TinyVGA PMOD to the `Out` PMOD and Mike' Audio PMOD to `Bidir` PMOD.

External hardware

- TinyVGA PMOD
- Mike's audio PMOD

Pinout

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

TSAL_TT [449]

- Author: Ephren Manning
- Description: FSAE EV Tractive System Active Light
- GitHub repository
- HDL project
- Mux address: 449
- Extra docs
- Clock: 8000000 Hz

How it works

This design is meant to fulfill FSAE Rule EV.5.9 on a student built electric vehicle. Rules are shown below. The digital design held on the TinyTapeout chip will take the digital value of an analog signal from an Analog Devices AD7476A 12-bit ADC chip. The value of the signal will be compared against a decided value representing that the tractive system is at 60V. Should the converted value be less than the decided value, a digital line driving a green LED will be driven high. Should the value be greater, a seperate digital line driving a red LED will flash at a rate of 4 hertz.

As of the 2024 Rules Ver. 1, operation is described as follows:

EV.5.9 Tractive System Active Light - TSAL

EV.5.9.1 The vehicle must include a Tractive Systems Active Light (TSAL) that must:

a. Illuminate when the GLV System is energized to indicate the status of the Tractive System

b. Be directly controlled by the voltage present in the Tractive System using hard wired electronics. Software control is not permitted.

c. Not perform any other functions.

EV.5.9.2 The TSAL may be composed of multiple lights inside a single housing

EV.5.9.3 When the voltage outside the Accumulator Container(s) exceeds T.9.1.1, the TSAL must:

a. Be Color: Red

b. Flash with a frequency between 2 Hz and 5 Hz

EV.5.9.4 When the voltage outside the Accumulator Container(s) is below T.9.1.1, the TSAL must:

a. Be Color: Green

b. Stay continuously illuminated

How to test

When testing, the digital line driving the green LED should be driven high only in the case that converted analog value is less than the comparison value. When the converted value is greater than or equal to the comparison value, the red LED should blink at a rate of 4 hertz. This requires that simulations be ran for upwards of a second to confirm LED blink speed.

External hardware

A PMOD AD1 from Digilent was used to test this project. The input/outputs on the TinyTapeout Demo board were configured so that this PMOD could be used on the top *(confirm) bidirectional port. Should a custom board be made to support functionality, the Analog Devices AD7476A or compatible 12-bit ADC converter will need to be used.

Pinout

#	Input	Output	Bidirectional
0	Comparison Value Bit 0	Green Led	Chip Select
1	Comparison Value Bit 1	Red Led	Serial Data
2	Comparison Value Bit 2		
3	Comparison Value Bit 3		Serial Clock
4	Comparison Value Bit 4		
5	Comparison Value Bit 5		
6	Comparison Value Bit 6		
7	Comparison Value Bit 7		

pulse_add [450]

- Author: Jonny Edwards
- Description: a temporal add in digital
- GitHub repository
- HDL project
- Mux address: 450
- Extra docs
- Clock: 0 Hz

How it works

This is a simple circuit to calculate:

- a simple add but temporal ...
- it's one out due to enable

How to test

All tested through cocotb

External hardware

I intend for this to be driven by the RP2040 and to work as a "coprocessor" for vector calculations Other.

Pinout

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

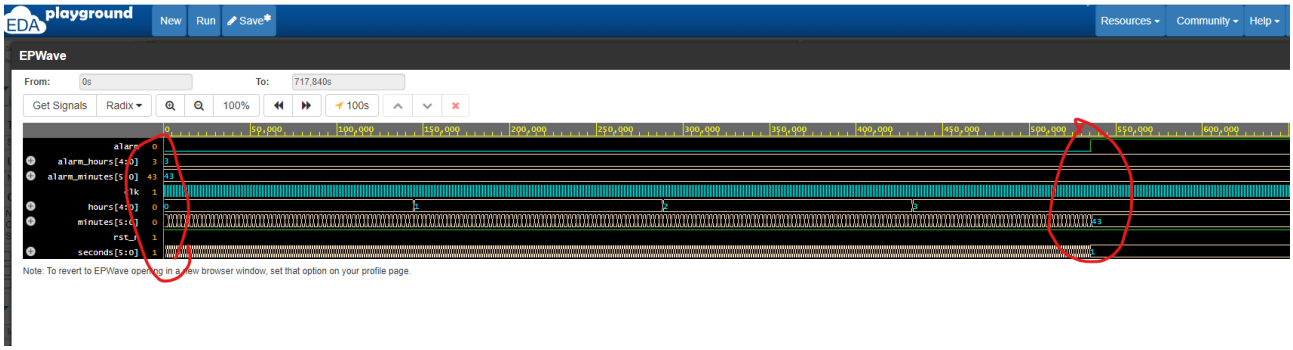
Alarm Clock [451]

- Author: Kapilan Karunakaran
- Description: A simple alarm clock
- GitHub repository
- HDL project
- Mux address: 451
- Extra docs
- Clock: 0 Hz

How it works

This project was sponsored by The MITRE Corporation and MIT/LL Beaverworks Summer Institute https://beaverworks.ll.mit.edu/CMS/bw/bwsi . This is a simple alarm clock. There are two inputs alarm_hours and alarm_minutes. These two inputs are to be set to values less than 23 and 59 respectively. The output pin "alarm" is expected to be asserted when the internal counters hours and minutes hit the expected alarm_hours and alarm_minutes respectively. Alarm output will be stuck at 1 until reset. Time at which Alarm is triggered are also sent out as output for comparison. Due to limited number of Inputs and outputs available, some of the inout pins are used as well. output enable pins are used to mask the inputs and outputs appropriately.

How to test



Use below testbench to input specific values to inputs and observe alarm asserted and output hours and minutes match with inputs. module tb();

reg clk, rst_n; reg [5:0] alarm_minutes; reg [4:0] alarm_hours;

tt_um_kapilan_alarm dut(.ui_in({alarm_minutes[2:0],alarm_hours[4:0]}), .uo_out(), .uio_in({5'b0,alarm_minutes[5:3]}), .uio_out(), .uio_oe(), .ena(1'b1), .clk(clk), .rst_n(rst_n));

initial begin rst_n = 1'b0; #100; rst_n = 1'b1; alarm_hours = 5'd3; alarm_minutes = 6'd43; end initial begin clk = 1'b0; forever begin #20; clk = ~clk; end end initial begin $dumpfile("alarm_dump.vcd"); $dumpvars(0,tb); #1000000; $finish; end

endmodule

External hardware

Pinout

#	Input	Output	Bidirectional
0	alarm_hours[0]	hours[0]	alarm_minutes[3]
1	alarm_hours[1]	hours[1]	alarm_minutes[4]
2	alarm_hours[2]	hours[2]	alarm_minutes[5]
3	alarm_hours[3]	hours[3]	
4	alarm_hours[4]	hours[4]	minutes[3]
5	alarm_minutes[0]	minutes[0]	minutes[4]
6	alarm_minutes[1]	minutes[1]	minutes[5]
7	alarm_minutes[2]	minutes[2]	

MAC [452]

- Author: Mahaa Santeep G, Shylashree N, Ravish Aradhya H V, RV College Of Engineering, Sneha R V, PES University
- Description: Design and Implementation of MAC Unit Using Dadda Multiplier and Kogge-Stone Adder
- GitHub repository
- HDL project
- Mux address: 452
- Extra docs
- Clock: 100 Hz

Credits : We gratefully acknowledge the COE in Integrated Circuits and Systems (ICAS) and Department of ECE. Our special thanks to Dr K S Geetha (Vice Principal) and, Dr. K N Subramanya (principal) for their constant support and encouragement to do TAPEOUT in Tiny Tapeout 8 .

How it works

The tt_um_mac module is a Multiply-Accumulate (MAC) unit designed for high-performance digital signal processing and embedded system applications. This module integrates a Dadda multiplier and a Kogge-Stone adder to achieve efficient and fast computations. The MAC unit performs a sequence of multiplication and accumulation operations, which are essential in various digital signal processing tasks, such as filtering and convolution. Functional Description Input and Output Ports ▪ Inputs: o ui_in (8-bit): Dedicated input for the first operand. o uio_in (8-bit): Input/Output interface for the second operand. o clk (1-bit): Clock signal to synchronize all operations. o rst_n (1-bit): Active-low reset signal to initialize the internal state of the MAC unit. ▪ Outputs: o uo_out (8-bit): Output that holds the final accumulated result. o uio_oe (8-bit): Output enable signal, set to 0 indicating the uio is used as input. o uio_out (8-bit): Unused output path in the current context. Internal Architecture

1. Dadda Multiplier The Dadda multiplier is a high-speed multiplier designed for efficient computation. It reduces the partial products in a sequence of reduction stages until the final product is obtained. In this design, a 4x4 Dadda multiplier is used to compute the 8-bit product of the two 4-bit operands, A and B.
2. Pipeline Registers Pipeline registers are implemented to enhance the performance of the MAC unit by storing intermediate results at each stage of the operation. This design uses two pipeline registers: ▪ Prod_stage: Holds the product of the multiplication. ▪ Sum_stage: Holds the result of the accumulation.

3. Kogge-Stone Adder The Kogge-Stone adder is a parallel-prefix form of a carry-lookahead adder, known for its high speed and efficiency in handling large bit-width additions. It computes the sum of the product and the current accumulator value (Acc), which is stored in the Sum_stage register.
4. Accumulator The accumulator (Acc) is a key component that stores the ongoing sum of the products. It is updated with the result from the Kogge-Stone adder on each clock cycle, allowing the MAC unit to perform repeated accumulation operations. Reset Behavior When the reset signal (rst_n) is asserted low, the pipeline registers (Prod_stage, Sum_stage) and the accumulator (Acc) are cleared, resetting the MAC unit to its initial state.

How to test

How to Test

To verify the functionality of the `tt_um_mac` module, a testbench (`tt_um_mac_tb`) has been provided. The testbench simulates different input scenarios and observes the output behavior of the `tt_um_mac` module to ensure that it works correctly.

- The testbench will output the results of the simulation, including the values of the inputs and the resulting output for each test case.
- Monitor the output in the console or waveform viewer to ensure the `tt_um_mac` module behaves as expected.

Example Test Scenarios Below is a summary of the test cases used in the `tt_um_mac_tb` testbench, along with their expected results.

Time (ns)	**`ui_in` (Input A)**	**`uio_in` (Input B)**	**Operation**	**Expected `uo_out` (Output)**
0-10	00000000 (0)	00000000 (0)	Reset	00000000 (0)
10-30	00000011 (3)	00000010 (2)	Multiply, Accumulate	00000110 (6)
30-50	00000001 (1)	00000100 (4)	Multiply, Accumulate	00001010 (10)
50-70	00000101 (5)	00000011 (3)	Multiply, Accumulate	00011001 (25)
70-90	00000111 (7)	00000010 (2)	Multiply, Accumulate	00100111 (39)
90-110	00000000 (0)	00000000 (0)	No Operation (Idle)	00100111 (39)

Time (ns)	ui_in (Input A)	uio_in (Input B)	Operation	Expected uo_out (Output)
110-130	00000001 (1)	00000001 (1)	Multiply, Accumulate	00101000 (40)

Monitoring Output During the simulation, you can monitor the console or waveform outputs for detailed step-by-step results. The testbench uses `$monitor` to display real-time updates of the inputs and the resulting output.

```
initial begin
    $monitor("Time=%0d | ui_in=%b, uio_in=%b | uo_out=%b", $time, ui_in,
end
```

This will provide you with a detailed trace of how the `tt_um_mac` module processes the inputs to generate the expected outputs.

Pinout

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

tt08-octal-alu [453]

- Author: Theo Kachelski
- Description: Executes binary operations on two octal numbers encoded in an 8-bit instruction and outputs the result to the 7-segment display.
- GitHub repository
- Wokwi project
- Mux address: 453
- Extra docs
- Clock: 0 Hz

How it works

This design executes binary operations on two octal numbers encoded in an 8-bit instruction and outputs the result to the 7-segment display. The instruction is made up of 8 total bits.

Bits 0 and 1 make up the operation code. Operations are according to the following table.

Bit 0	Bit 1	Operation
0	0	ADD
0	1	OR
1	0	AND
1	1	XOR

Figure 10: OP Bit Table

Bits 2, 3, and 4 make up operand A. Bit 2 is MSB and bit 4 is LSB.

Bits 5, 6, and 7 make up operand B. Bit 5 is MSB and bit 7 is LSB.

A full operation would be decoded like this

How to test

1. Craft a instruction following the above decoding table.
2. Enter the instruction on the TT08 PCB's input pin dip switches.

Bit	0	1	2	3	4	5	6	7
Decoded	OP1	OP2	A(MSB)	A	A(LSB)	B(MSB)	B	B(LSB)

Figure 11: OP Decoding

3. View the result on the 7 segment display

   Note: An overflow condition during an add operation will illumniate the 7 segment display's DP.

Examples:

1. 0 0 1 0 0 0 0 1 -> ADD 4, 1 -> Displays 5 on 7-Segment Display
2. 1 1 1 0 0 1 0 1 -> XOR 4, 5 -> Displays 1 on 7-Segment Display
3. 1 0 1 0 0 1 0 1 -> AND 4, 5 -> Displays 4 on 7-Segment Display
4. 0 1 1 1 0 1 0 1 -> OR 6, 5 -> Displays 7 on 7-Segment Display
5. 0 0 1 0 1 1 1 0 -> ADD 5, 6 -> 7-Segment Display DP illuminated indicating overflow condition

External hardware

The only external hardware required is the 7 segment display provided by the tt08 PCB.

Pinout

#	Input	Output	Bidirectional
0	Operation bit 1	Segment A	
1	Operation bit 2	Segment B	
2	Operand A bit 1 (MSB)	Segment C	
3	Operand A bit 2	Segment D	
4	Operand A bit 3 (LSB)	Segment E	
5	Operand B bit 1 (MSB)	Segment F	
6	Operand B bit 2	Segment G	
7	Operand B bit 3 (LSB)	Segment DP	

DPMU [454]

- Author: Sanjay Kumar M, Shylashree N, Ravish Aradhya H V, RV College Of Engineering, Neha R V, PES Unoversity
- Description: Design and Implementation of Dynamic Power management unit
- GitHub repository
- HDL project
- Mux address: 454
- Extra docs
- Clock: 100 Hz

Credits : We gratefully acknowledge the COE in Integrated Circuits and Systems (ICAS) and Department of ECE. Our special thanks to Dr K S Geetha (Vice Principal) and, Dr. K N Subramanya (principal) for their constant support and encouragement to do TAPEOUT in Tiny Tapeout 8 .

The code provided is a SystemVerilog module that implements a Dynamic Power Management Unit (DPMU) for an SoC (System on Chip). The DPMU dynamically adjusts voltage and frequency levels based on inputs such as performance requirements, temperature, battery level, and workload. The module uses a finite state machine (FSM) to manage transitions between different power states.

Key Components

Inputs and Outputs: Inputs (ui_in): The primary input signals include performance requirements, temperature sensor data, battery level, and workload. Outputs (uo_out, uio_out): These include the power-saving indicator, voltage levels, and frequency levels for different cores and memory. I/O (uio_in, uio_out, uio_oe): Handles bidirectional signals; however, in this design, uio_in is not used, and uio_out is used for output. Internal Signals:

State Variables: state and next_state manage the FSM that controls the DPMU's behavior. Power and Frequency Controls: Registers like vcore1, vcore2, vmem, fcore1, fcore2, and fmem store the voltage and frequency settings. Finite State Machine (FSM):

States: NORMAL: Default operating mode with standard voltage and frequency levels. PERFORMANCE: High-performance mode with maximum voltage and frequency levels. POWERSAVE: Low-power mode with reduced voltage and frequency levels. THERMAL_MANAGEMENT: Mode to handle high temperature by adjusting power levels moderately. BATTERY_SAVING: Mode to conserve battery by minimizing voltage and frequency levels.

State Transitions: Transitions between states occur based on the input conditions, such as high performance request, low battery level, high temperature, or low workload. Detailed Walkthrough Input and Output Mapping:

perf_req: Mapped to the least significant bit (LSB) of ui_in, indicating whether high performance is needed. temp_sensor: 2-bit signal derived from ui_in[3:2], providing temperature data. battery_level: 2-bit signal derived from ui_in[5:4], indicating the battery's charge status. workload_core: 3-bit signal derived from ui_in[7:6], representing the workload of a core. State Logic:

On each clock cycle (clk), the FSM checks the state and evaluates transitions based on inputs. In NORMAL state, if perf_req is high, the system transitions to PERFORMANCE state. If the battery level is low, it transitions to BATTERY_SAVING state. If the temperature is high, it transitions to THERMAL_MANAGEMENT state. If the workload is low, it transitions to POWERSAVE state. PERFORMANCE state sets all voltages and frequencies to maximum. If perf_req drops, it returns to NORMAL. POWERSAVE state reduces voltages and frequencies to conserve power. If the workload increases, it returns to NORMAL. THERMAL_MANAGEMENT state adjusts power levels to moderate values to manage high temperatures. If the temperature normalizes, it returns to NORMAL. BATTERY_SAVING state minimizes voltages and frequencies to conserve battery. If the battery level increases, it returns to NORMAL.

Output Assignment: The combined voltage (vcore1, vcore2, vmem) and frequency (fcore1, fcore2, fmem) values are assigned to the uio_out and uo_out outputs. The power_save signal is also part of the output, indicating whether the system is in power-saving mode. Behavior under Reset:

When the reset (rst_n) is low (active), the system resets to the NORMAL state.

Here's a table summarizing the expected output (uio_out, uo_out) based on the input (ui_in) and time using the provided testbench for the tt_um_dpmu module. The table provides the values for different states as the ui_in input changes over time.

Table: Testbench Expected Output

Time (ns) ui_in (Input) State uio_out (Expected Output) uo_out (Expected Output) 0 11110010 NORMAL 01010110 010_010010 10 00010010 PERFORMANCE 11111111 111_111111 30 11110010 NORMAL 01010110 010_010010 50 11110011 THERMAL_MANAGEMENT 10101011 011_011011 70 11110010 NORMAL 01010110 010_010010 90 11101010 THERMAL_MANAGEMENT 10101011 011_011011 110 11111010 BATTERY_SAVING 00000000 000_000000 130 11111110 BATTERY_SAVING 00000000 000_000000 150 11111010 BATTERY_SAVING 00000000 000_000000

Explanation of Table Columns:

Time (ns): The simulation time when the ui_in input is applied. ui_in (Input): The 8-bit input value applied to the design. State: The state of the FSM based on the ui_in input. The states are NORMAL, PERFORMANCE, THERMAL_MANAGEMENT, and BATTERY_SAVING. uio_out (Expected Output): The expected 8-bit output values for the uio_out signals. uio_out[0]: Power save mode indicator. uio_out[2:1], uio_out[4:3], uio_out[6:5]: Voltage controls. uio_out[7]: Part of fcore1[0]. uo_out (Expected Output): The expected 8-bit output values for the uo_out signals. uo_out[0:1]: Part of fcore1[2:1]. uo_out[4:2]: fcore2[2:0]. uo_out[7:5]: fmem[2:0].

Explanation of Key Points: NORMAL State: When the inputs suggest a typical operating environment (e.g., ui_in = 11110010), the design operates with default voltage and frequency levels. PERFORMANCE State: Triggered by a performance request (perf_req = 1), leading to maximum voltage and frequency levels. THERMAL_MANAGEMENT State: Triggered by high temperature (temp_sensor = 10 or 11), moderates the voltage and frequency to prevent overheating. BATTERY_SAVING State: Triggered by low battery level (battery_level = 00 or 01), minimizing power consumption by reducing voltage and frequency to the lowest levels.

Testbench Operation: The testbench applies different ui_in values at specific simulation times. At each time step, it captures the output values (uio_out and uo_out) and compares them with the expected values as per the design's FSM logic. The $monitor statement continuously logs the input and output values, helping to verify the design's behavior at each time point.

Pinout

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

Rotary Encoder WS2812B Control [455]

- Author: Fabio Ramirez Stern
- Description: A rotary encoder controls 12 WS2812B LEDs on a ring PCB.
- GitHub repository
- HDL project
- Mux address: 455
- Extra docs
- Clock: 40000000 Hz

How it works

The rotary encoder adds/subtracts from a variable that determines which LED to turn on. Periodically, the chip sends out a signal for 12 LEDs out via `uo0`, according to the WS2812B protocol. The button connected to `in2` inverts the LEDs, whether that happens gets also indicated through `out1`. Further, the register value of the variable will be put out via `out2` to `uo5`. The colour can be activated as follows: `in3` for green, `in4` for red and `in5` for blue. Intensity is set with the remaining two bits, `in6` and `in7`.

How to test

Connect the rotary encoder outputs to `in0` and `in1`. If your rotary encoder also has a built in push button, conntect that to `in2`, or use another switch with a pull down resistor. The LEDs should be wired in series. The first LED's `DIN` input needs to be connected to the `out0` of the chip.

Give the project a reset after power up and then rotate the encoder back and forth to see the light moving.

External hardware

1. A rotary encoder.
2. Any arrangement of 12 WS2812B like controlled LEDs. More or less will also work, you will just not get the full range, or some LEDs will stay off.

The seller called what I bought this: LED Ring 5V RGB WS2812B 12-Bit 37mm

Pinout

#	Input	Output	Bidirectional
0	rotary encoder: CLK	DOUT	
1	rotary encoder: DT	inverted	
2	rotary encoder: SW	count0	
3	green	count1	
4	red	count2	
5	blue	count3	
6	intensity1		
7	intensity2		

7 Segment Decode [456]

- Author: Jack Clayton
- Description: ASIC implementation of a university CPLD project which drives 4 multiplexed 7 segment displays, and scans a multiplexed keypad.
- GitHub repository
- HDL project
- Mux address: 456
- Extra docs
- Clock: 5000 Hz

How it works

The serial protocol implemented in this design consists of a simple single byte packet which instructs the CPLD which column of the keypad to multiplex into MISO, which screen should be displaying data, and what number should be displayed on the screen. The screen select signal also doubles up as instructing which row of the keypad to be scanned. Data is sent Most Significant Bit (MSB) first.

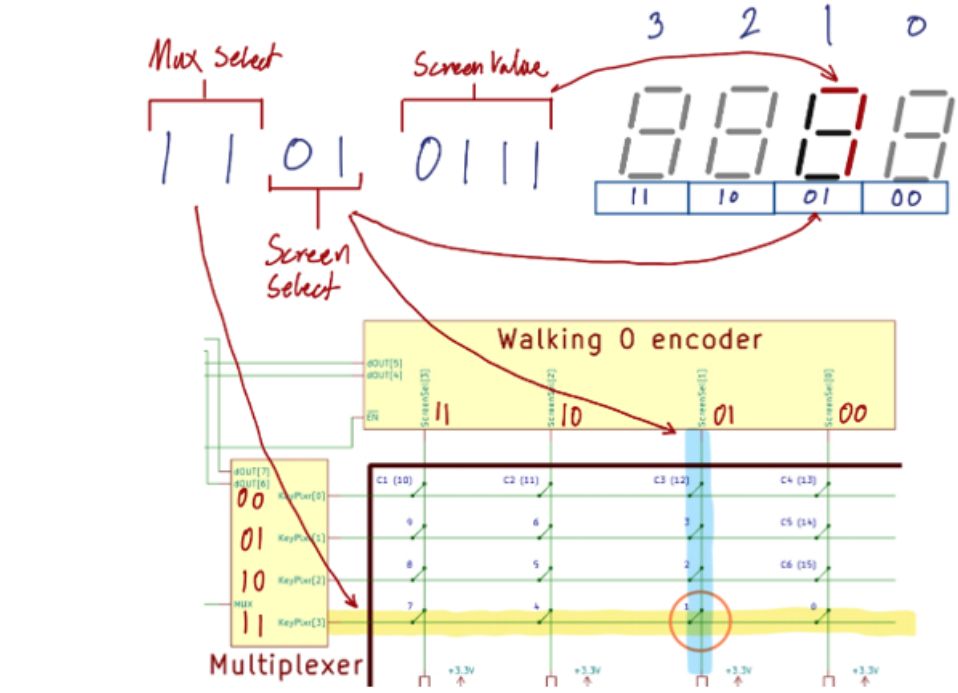


Figure 2.2.0.1 – SPI Protocol visual bit representations

Figure 12: SPI Protocol visual bit representations

The high impedance programming state is not implemented in this ASIC. It is represented as a bit out instead.

Bit 7	Multiplexer select MSB
Bit 6	Multiplexer select LSB
Bit 5	Screen select MSB
Bit 4	Screen select LSB
Bit 3	Screen data MSB
Bit 2	Screen data
Bit 1	Screen data
Bit 0	Screen data LSB

Figure 2.2.0.2 – SPI Protocol bit definitions

Figure 13: SPI Protocol bit definitions

2.2.2 Signal structure

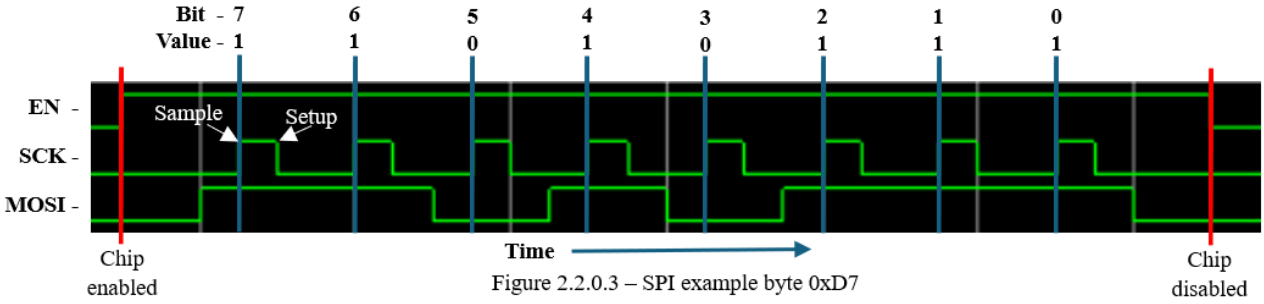


Figure 2.2.0.3 – SPI example byte 0xD7

Figure 14: SPI example byte 0xD7

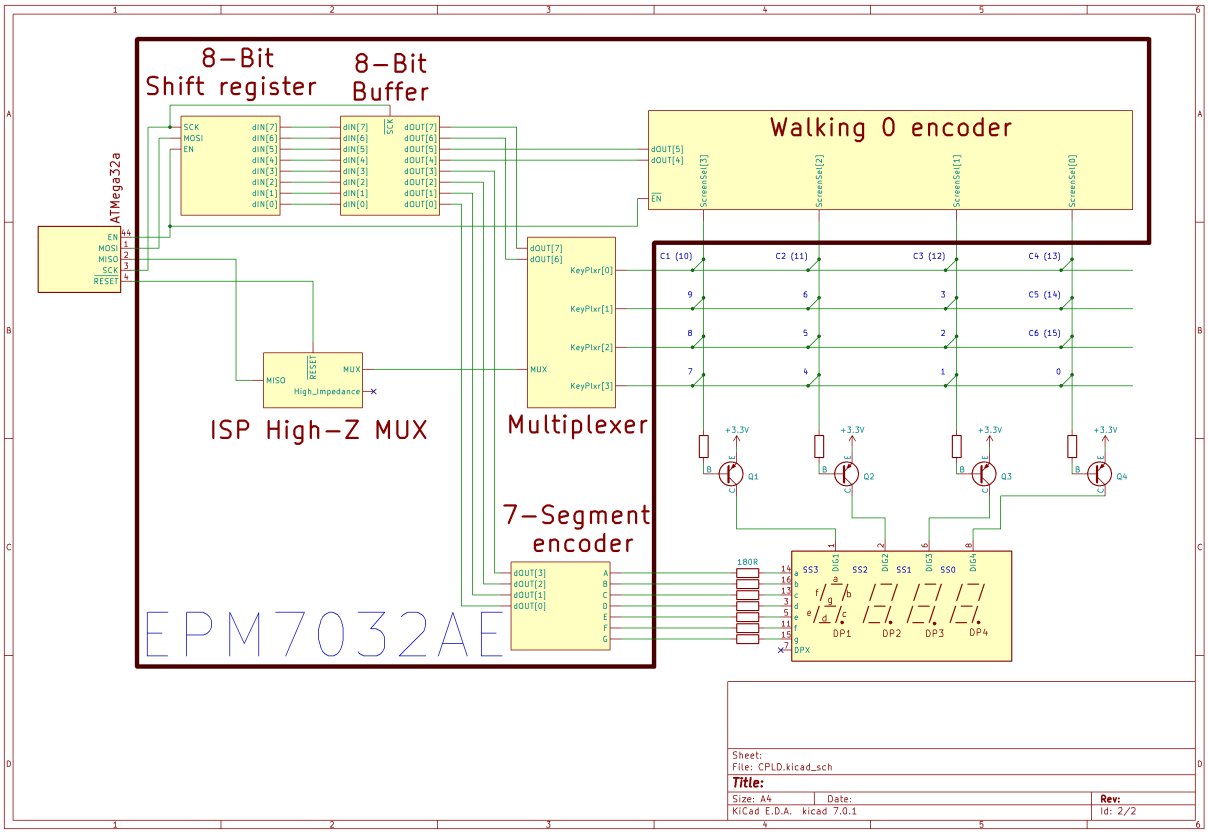


Figure 15: Verilog block diagram

Keep in mind: This system is clocked by the SPI clock, and therefore requires constant clocking to function.

How to test

Build the supporting hardware as described in the schematic found in "Exteral hardware". Create a system which transmits SPI bytes, according to specifications in "How it works". The system will display your desired digits on the selected screens.
You may also use the MISO to implement a 4x4 keypad, which is interpreted by the system creating the SPI bytes. This will not be detailed as to how to implement.

External hardware

Main external system schematic:

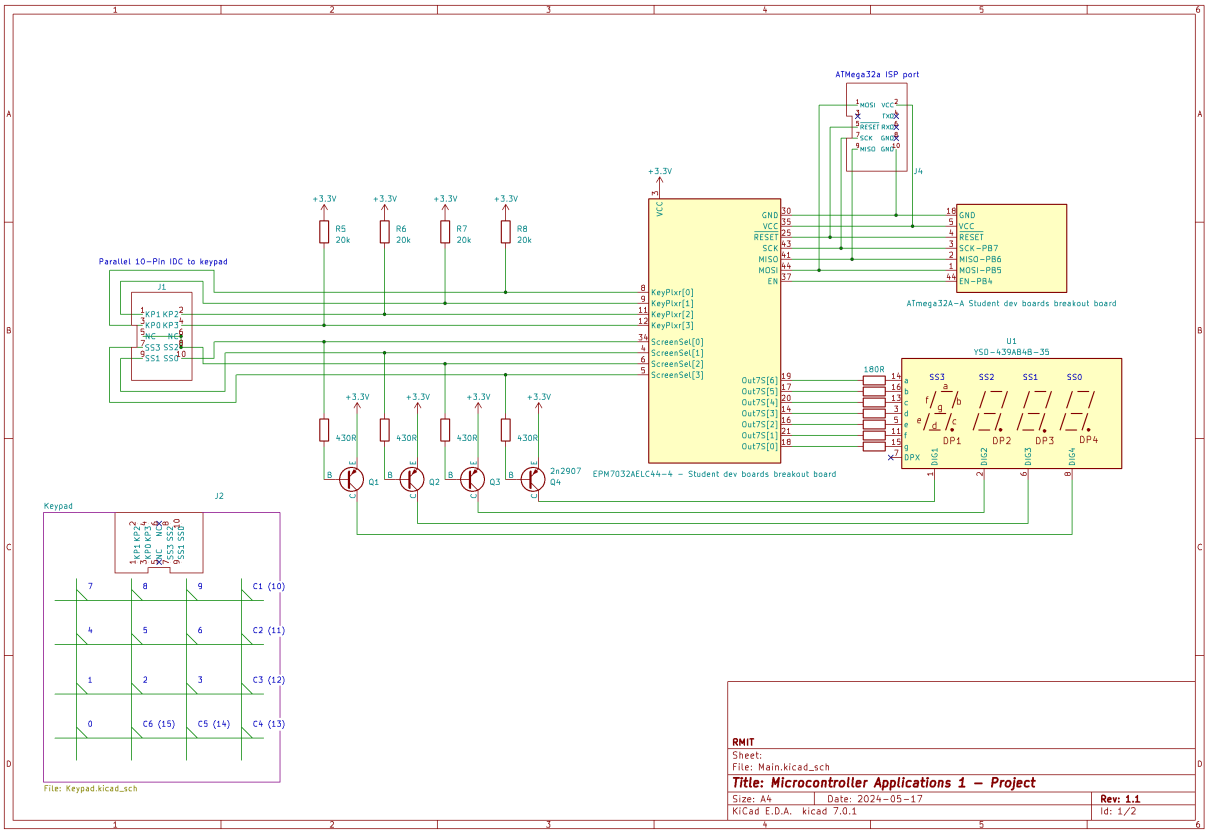


Figure 16: Main schematic

Simple keypad:

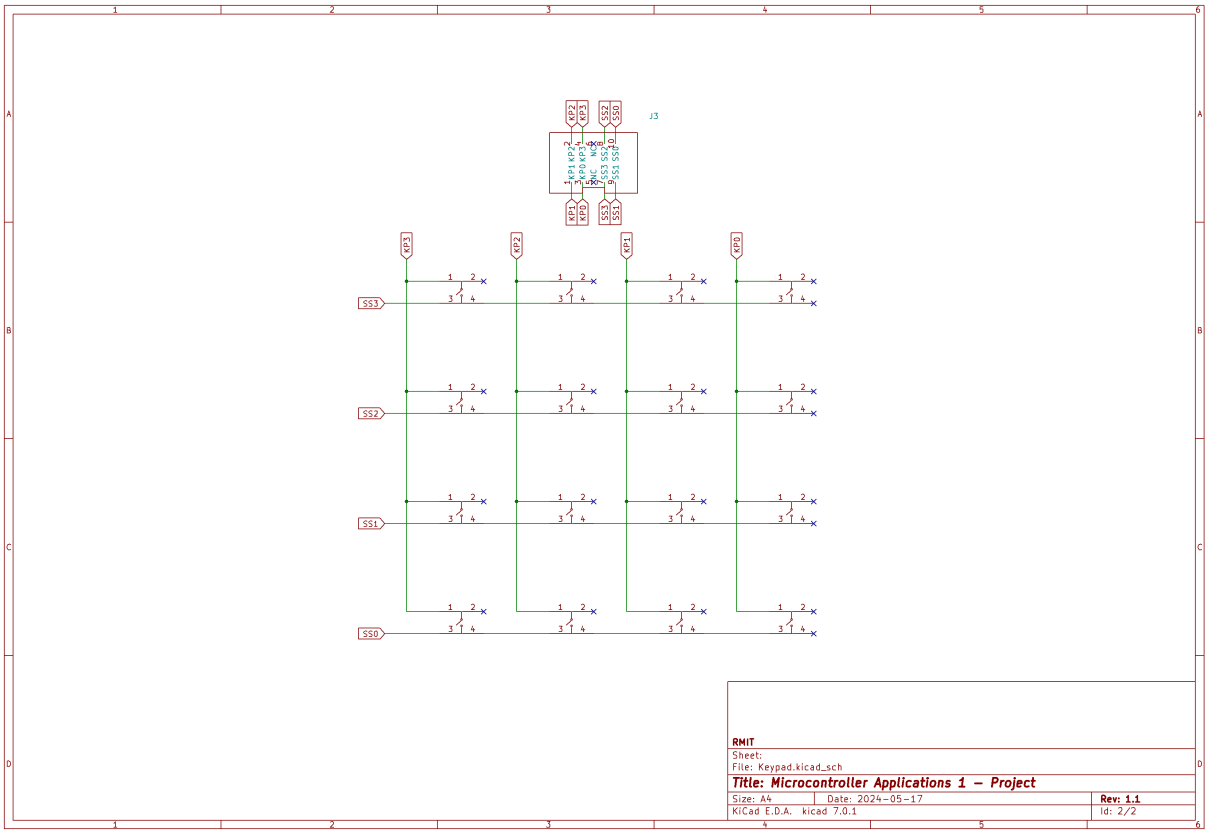


Figure 17: Keypad schematic

Pinout

#	Input	Output	Bidirectional
0		Out7S[0]	ScreenSel[0]
1	MOSI	Out7S[1]	ScreenSel[1]
2	EN	Out7S[2]	ScreenSel[2]
3	RESET	Out7S[3]	ScreenSel[3]
4	KeyPlxr[0]	Out7S[4]	High-Z
5	KeyPlxr[1]	Out7S[5]	
6	KeyPlxr[2]	Out7S[6]	
7	KeyPlxr[3]	MISO	

TT08 SKY130 ROM 'YOLO' Test [457]

- Author: Sylvain Munaut
- Description: Quick and dirty test of some parts of a SKY130 ROM compiler
- GitHub repository
- HDL project
- Mux address: 457
- Extra docs
- Clock: 0 Hz

How it works

FIXME

How to test

FIXME

External hardware

FIXME

Pinout

#	Input	Output	Bidirectional
0	wl[0]	rom_out[0]	bl0_ena
1	wl[1]	rom_out[1]	bl1_ena
2	wl[2]	rom_out[2]	bl_mux_n[0]
3	wl[3]	rom_out[3]	bl_mux_p[0]
4	wl[4]	rom_out[4]	bl_mux_n[1]
5	wl[5]	rom_out[5]	bl_mux_p[1]
6	wl[6]	rom_out[6]	pullup_n
7	wl[7]	rom_out[7]	prechg_n

Traffic-light-sequence [458]

- Author: Shaurya Sharma
- Description: A simple traffic light sequence
- GitHub repository
- Wokwi project
- Mux address: 458
- Extra docs
- Clock: 0 Hz

How it works

Sequence of Traffic light from Red to Red-Yellow to Green to Yellow to Red

How to test

Input 00= Red, Input 01= Red-Yellow, Input 10= Green, Input 11= Yellow

External hardware

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

Pinout

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

i2c peripherals demonstrating address decoding and i2c reads [459]

- Author: Steve Jenson <stevej@gmail.com>
- Description: An i2c peripheral demonstrating address decoding and i2c reads.
- GitHub repository
- HDL project
- Mux address: 459
- Extra docs
- Clock: 0 Hz

How it works

Fnv-1a 32-bit peripheral: send bytes via write requests, get the hash via a read request. Every read request resets the hash.

LZC: send up to 32 bits with write request, read back the number of leading zeroes with a read request.

ZeroOne: Sends the byte 0101_0101

OneZero: Sends the byte 1010_1010

How to test

Fnv-1a: Send a known set of bytes, get a known hash back.

LZC: Send 32 zeros, get the number 32 back. Send 32 1s, get 0 back

ZeroOne/OneZero: make a read request.

External hardware

i2c master device with test code. Arduino test code provided.

#	Input	Output	Bidirectional

Pinout

#	Input	Output	Bidirectional
0			(INT)
1			(RESET)
2			SCL
3			SDA
4			
5			
6			
7			

TT08 SKY130 Shift Register 'YOLO' Test [460]

- Author: Sylvain Munaut
- Description: Quick and dirty test of a custom high density shift register
- GitHub repository
- HDL project
- Mux address: 460
- Extra docs
- Clock: 0 Hz

How it works

FIXME

How to test

FIXME

External hardware

FIXME

Pinout

#	Input	Output	Bidirectional
0	sr_in[0]	sr_out[0]	clk_ph0
1	sr_in[1]	sr_out[1]	clk_ph1
2	sr_in[2]	sr_out[2]	
3	sr_in[3]	sr_out[3]	
4	sr_in[4]	sr_out[4]	
5	sr_in[5]	sr_out[5]	
6	sr_in[6]	sr_out[6]	
7	sr_in[7]	sr_out[7]	

Adder with Flow Control [461]

- Author: Yuri Panchul
- Description: An adder with a separate flow control for each argument and the result
- GitHub repository
- HDL project
- Mux address: 461
- Extra docs
- Clock: 3 Hz

How it works

adder_with_flow_control design contains an adder with a separate flow control for each argument and the result.

The design is an answer to an RTL job interview question described by Yuri Panchul in an article (in Russian) on Habr website. The design is used as a part of systemverilog-homework and basics-graphics-music GitHub repositories. These repos are maintained by engineers and educators associated with the Verilog Meetup community.

In this solution to the interview question, the flow control is implemented using one of the following pipeline register/buffer modules. The choice is specified inside the *adder_with_flow_control.sv* module using the define macro *FLOW_CONTROL_BUFFER*.

- fcb_1_single_allows_back_to_back
- fcb_2_single_half_perf_no_comb_path
- fcb_3_single_for_pipes_with_global_stall
- fcb_4_wrapped_2_deep_fifo
- fcb_5_double_buffer_from_dally_harting

More details about the modules:

fcb_1_single_allows_back_to_back This module is a general-purpose flow-controlled register which allows full back-to-back performance but has a combinational path between down_rdy and up_rdy which can introduce timing problems in deep pipelines.

fcb_2_single_half_perf_no_comb_path This flow-controlled register has no combinational path at all, but cannot sustain a back-to-back stream of data. However, it requires fewer gates than *fcb_4_wrapped_2_deep_fifo* or *fcb_5_double_buffer_from_dally_harting*.

fcb_3_single_for_pipes_with_global_stall This flow-controlled register is suitable if the designer wants to always stall the whole pipeline at once, without parts of it making progress.

fcb_4_wrapped_2_deep_fifo The most high-bandwidth flow-controlled buffer among those that have no combinational path between down_rdy and up_rdy. However this solution occupies the largest area.

fcb_5_double_buffer_from_dally_harting This pipeline buffer is Yuri Panchul's edition of the code derived from *Digital Design: A Systems Approach by William James Dally and R. Curtis Harting. 2012*. It has high bandwidth and no combinational path between down_rdy and up_rdy, but not the highest possible bandwidth for this *adder_with_flow_control* design.

How to test

A self-checking testbench for the design is located in a directory *test_extra* that contains:

- *clean.bash* - a script to delete temporary files produced by *simulate.bash*.
- *simulate.bash* - a script that simulates the design together with a testbench using Icarus Verilog and produces a log *log.txt*.
- *tb.sv* - a self-checking testbench that generates a log, a status **PASS** or **FAIL**, and performance data.

After the manufacturing, the design can be manually tested in the same way it is tested in the testbench. It is important to cover the following scenarios:

- Back-to-back data.
- Argument starvation (either *A* or *B*).
- Backpressure.

External hardware

Buttons and LEDs

Pinout

#	Input	Output	Bidirectional
0	a_vld	a_rdy	a_data[0]
1	b_vld	b_rdy	a_data[1]
2	sum_rdy	sum_vld	a_data[2]
3		sum_data[0]	a_data[3]
4		sum_data[1]	b_data[0]
5		sum_data[2]	b_data[1]
6		sum_data[3]	b_data[2]
7		sum_data[4]	b_data[3]

PS2 Decoder [462]

- Author: Ben Payne
- Description: A PS2 keyboard decoder
- GitHub repository
- HDL project
- Mux address: 462
- Extra docs
- Clock: 25000000 Hz

How it works

This decoder works by first deboucing the inputs to make sure that we get a clean sample of them that is syncronized to our clock. It then looks at the down transistion of ps2_clk and reads the value of ps2_data. It shifts this int oa 11 bit shift register. When ps2_clk remains high for more than 1/2 of the 10kHz ps2_clk cycle it knows that the end of the data has arrived. It then triggers a valid flag to tell the system that something has arrived. The valid flag, witch is exposed on a pin, will trigger the fifo to read the byte of data and it will be stored to retrival but the host. When valid is trigger it will also trigger the interupt pin. The valid pin is a pulse for one system clock cycle, but the interupt will remain set until it is cleared. We also include a data_rdy signal that test the host that there is data to read. This is useful if your interupt handler needs to read multiple bytes.

When the host wants to read a byte, it asserts the chip select (cs) signal when the system clock goes high. This will result in the uio bus being set with the data value. The uio bus will be put into a output state only when cs is asserted, at all other times it will be an input bus (but we never read it...)

How to test

Simply interface a PS2 keyboard to the PS2 clock and data lines. You will need to level shift these signal to the 3.3v of the chip. At this point you can hit keys and they be queued in the fifo. Then you would want to interface a retro computer to the CS, interupt and data lines to read the fifo. This will depend on the system your using, but note you'll need external address decoding logic and for chips like the m68k you'll need to generate the DTACK and other signals elsewhere.

External hardware

Hook up an PS2 PMOD device to level shift the keyboards 5V to 3.3V for this chip. I have a design for this if anyone wants it.

Pinout

#	Input	Output	Bidirectional
0	ps2_clk	valid	data_out[0]
1	ps2_data	interupt	data_out[1]
2	clear_int	data_rdy	data_out[2]
3	cs		data_out[3]
4			data_out[4]
5			data_out[5]
6			data_out[6]
7			data_out[7]

Brailliance [463]

- Author: Akshat B, Evana T, John L, Rittrija M, Riley Gu
- Description: ASCII-to-Braille Converter
- GitHub repository
- HDL project
- Mux address: 463
- Extra docs
- Clock: 5000000 Hz

How it works

Input ASCII signals to 8-bit braille outputs (First two bits are zeroed for redundancy)

How to test

Connect to FPGA and use breadboards + LEDs/Push Pull Solenoid Actuators for product demonstration

External hardware

- Breadboard
- Jumper Wires
- LEDs / Push Pull Solenoid Actuators
- Resistors

Pinout

#	Input	Output	Bidirectional
0	clk	reader1_out[0]	
1	reset	reader1_out[1]	
2	next	reader1_out[2]	
3		reader1_out[3]	
4		reader1_out[4]	
5		reader1_out[5]	
6		reader1_out[6]	
7		reader1_out[7]	

2048 sliding tile puzzle game (VGA) [482]

- Author: Uri Shaked
- Description: Slide numbered tiles on a grid to combine them to create a tile with the number 2048.
- GitHub repository
- HDL project
- Mux address: 482
- Extra docs
- Clock: 0 Hz

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. 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

After resetting the game, you will see a jumping "2048" animation on the screen. Press any of the `ui_in[3:0]` pins to start the game. The game will start with two tiles with the number 2 on the board. Use the `ui_in` pins to move the tiles in the desired direction. The game will end when the board is full and no more moves can be made.

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

Pinout

#	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		R0	debug_data
5		G0	debug_data
6		B0	debug_data
7	debug_mode	HSync	debug_data

Demo by a1k0n [484]

- Author: Andy Sloane
- Description: Tiny Tapeout demo competition entry
- GitHub repository
- HDL project
- Mux address: 484
- Extra docs
- Clock: 48000000 Hz

a1k0n's tinytapeout08 demo compo entry

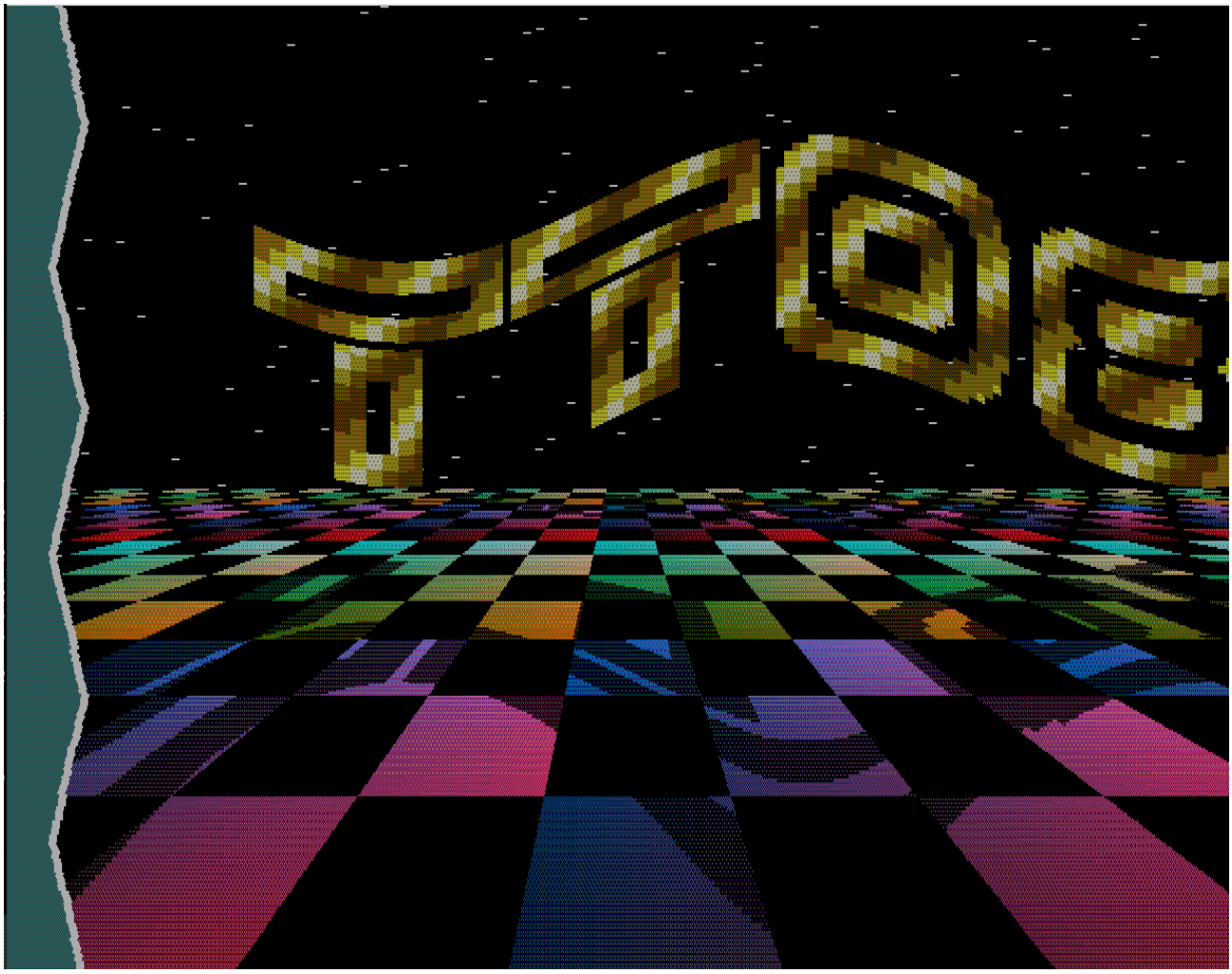


Figure 18: preview

How it works It's a standalone VGA+sound demo that fits in two tiles; you'll just have to see.

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).

I will add more info here after the deadline passes, as the demo is in flux as I try to fit effects into 2 tiles...

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

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

5MHz Ring Oscillator [486]

- Author: Alex Jaeger
- Description: A *very* simple 5MHz ring oscillator
- GitHub repository
- Analog project
- Mux address: 486
- Extra docs
- Clock: 0 Hz

How it works

This is a single-pin analog design project. The design presents a 5MHz clock signal at the output pin ua[0].

How to test

The design will be automatically enabled when selected on the evaluation board.

External hardware

No extra hardware required.

Pinout

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

Analog pins

ua#	analog#	Description
0	4	Oscillator Output

Analog 8 bit 3.3v R2R DAC [488]

- Author: Matt Venn
- Description: A simple 8 bit DAC with a sine waveform driver and 3.3v output
- GitHub repository
- Analog project
- Mux address: 488
- Extra docs
- Clock: 0 Hz

How it works

A simple 8 bit R2R DAC. Driven externally or by an digitally generated sine waveform generator.

3.3v output is achieved with level shifting drivers.

How to test

Drive externally Set the `external data` input high to provide the DAC with external data.

Then drive the 8 inputs and observe the analog output.

Drive with internal sawtooth wave generator Set the `external data` input low to enable the sine generator. A sine wave should be seen on the analog output. Everytime the sine counter is at 0, digital output 0 should go high for one clock.

To change the frequency, set the inputs and then raise the 'load divider' input.

External hardware

A multimeter to measure the output voltage on analog pin 0.

#	Input	Output	Bidirectional

Pinout

#	Input	Output	Bidirectional
0	bit 0	count zero	external data
1	bit 1		load divider
2	bit 2		
3	bit 3		
4	bit 4		
5	bit 5		
6	bit 6		
7	bit 7		

Analog pins

ua#	analog#	Description
0	5	DAC output

Analog Voltage Controlled Oscillator [490]

- Author: Georg Boecherer
- Description: A voltage controlled oscillator with 4 differential stages.
- GitHub repository
- Analog project
- Mux address: 490
- Extra docs
- Clock: 0 Hz

How it works

A voltage controlled oscillator (VCO) using 4 differential stages and a bias controller. At each stage, the outputs are low-pass filtered by nfet transistors connected as capacitors to ground.

The first analog pin `ua[0]` can be used as input to control the frequency by changing the gate voltages of nfet current sources connected to ground. The second analog pin `ua[1]` can be used to monitor the gate voltage generated by the bias controller of the pfet current sources connected to VDD. Alternatively, the second analog pin `ua[1]` can be used as input pin to overwrite the bias controller.

The 4 differential stages provide in total 8 tap signals with phase differences of 360/8. The 8 tap signals are converted to digital signals by buffers and output at the 8 digital output pins `uo[0]` to `uo[7]`. *NB:* the 8 tap signals are ordered by wiring convenience, not by phase.

For further information, see github.com/gbsha/tt08-analog-vco for updated information.

How to test

Connect a control voltage or current to the analog input pin `ua[0]` and observe the oscillating signals at the digital output pins.

External hardware

DC power supply for the control voltage/current, oscilloscope for monitoring the outputs.

Pinout

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

Analog pins

ua#	analog#	Description
0	3	bias control signal
1	2	optional bias control signal

Voltage Controlled LC-Oscillator [492]

- Author: Noritsuna Imamura
- Description: Voltage Controlled LC-Oscillator of 2GHz for Skywater130nm
- GitHub repository
- Analog project
- Mux address: 492
- Extra docs
- Clock: 0 Hz

Voltage Controlled LC-Oscillator

This project is "Voltage Controlled LC-Oscillator of 2GHz for Skywater130nm".

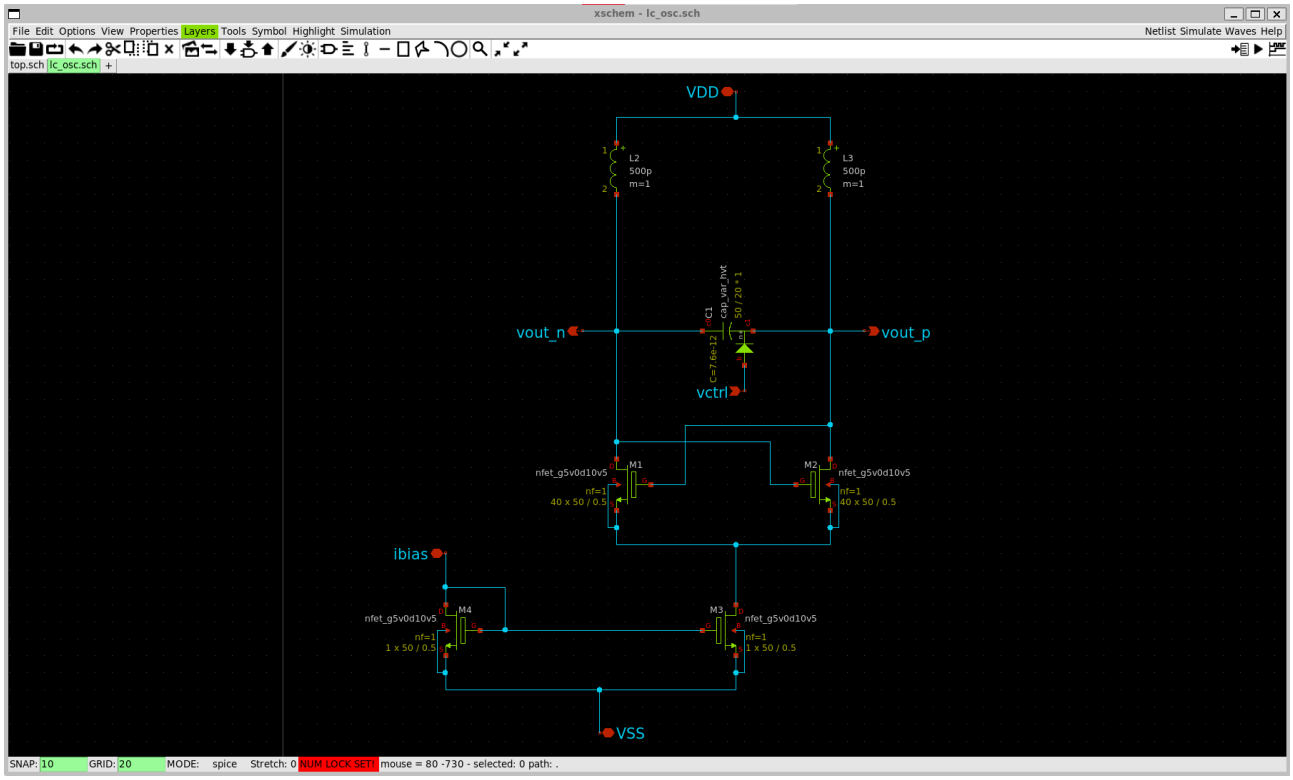


Figure 19: circuit

Circuit

Test Bench

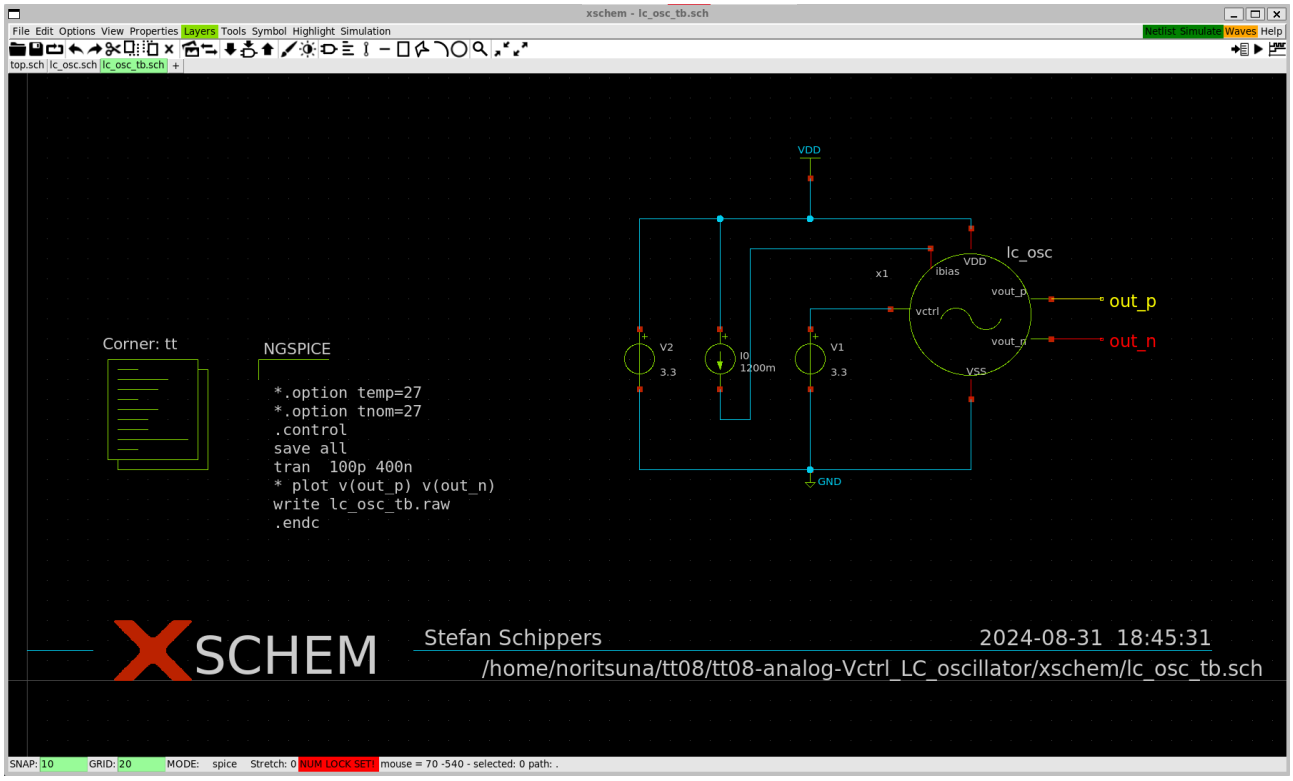
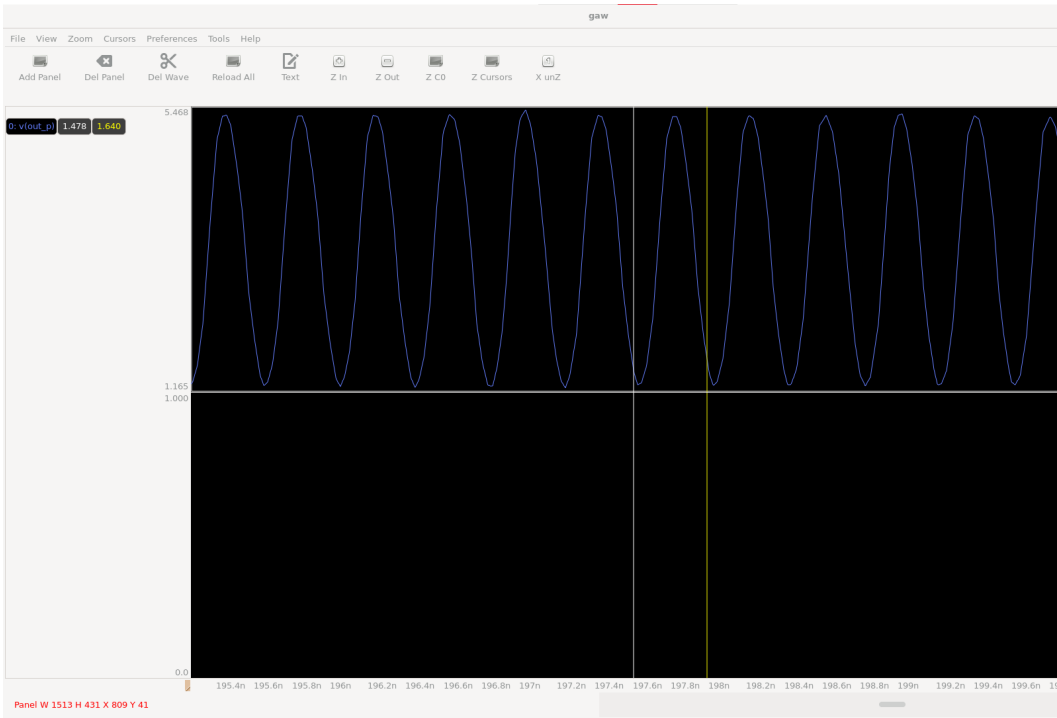


Figure 20: test bench

Result



- 388pSec = 2.6GHz

Layout

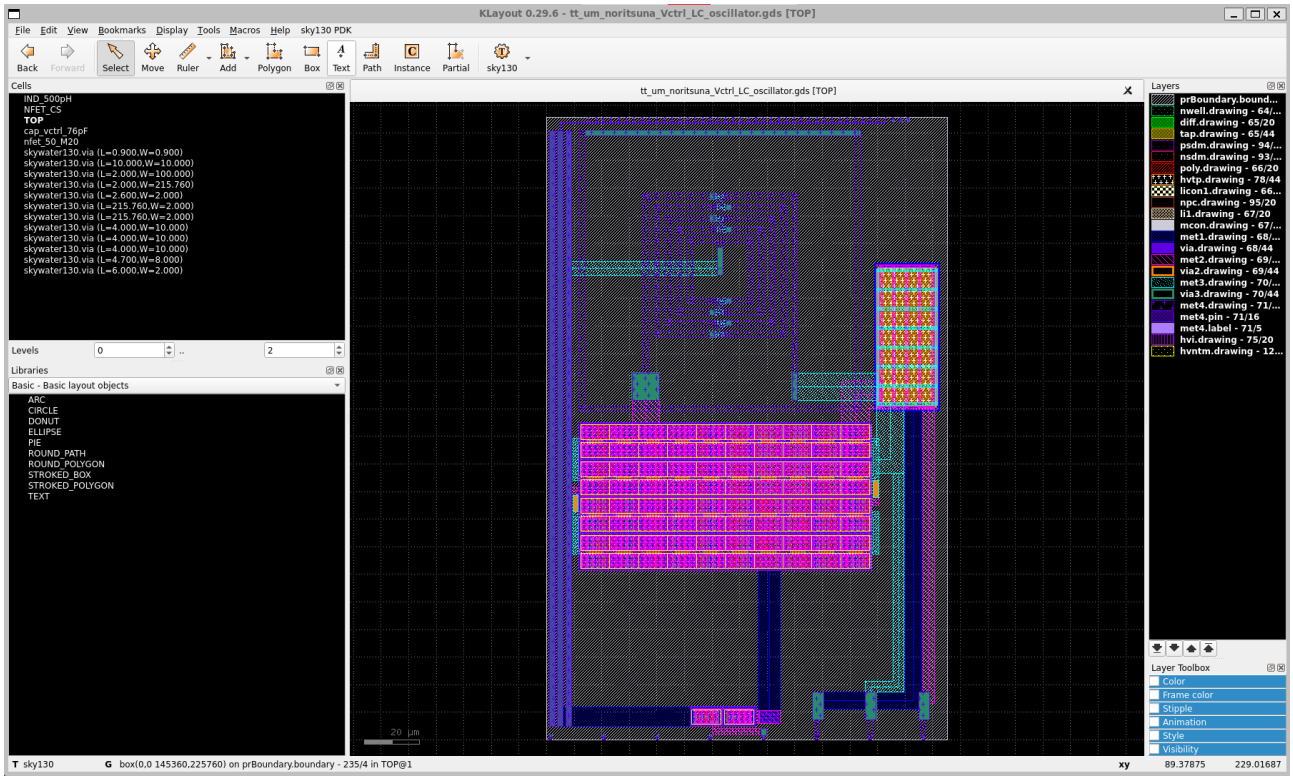


Figure 21: laytout

Tapeout This project made by Tiny Tapeout.

Pinout

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

Analog pins

ua#	analog#	Description
0	4	vout_p

ua#	analog#	Description
1	1	vout_n
2	3	vctrl
3	2	ibias

2-stage Opamp Designs [494]

- Author: Vipul Sharma
- Description: This project contain opamp circuits designed by participants under SSP training program
- GitHub repository
- Analog project
- Mux address: 494
- Extra docs
- Clock: 0 Hz

How it works

This project contains in total 3 circuits: 2 nos. of 2-stage opamp and 1 POR (Power on reset) circuit. These circuits are designed by participants of analog IC design training conducted by SSP (Saudi Semiconductor Program).

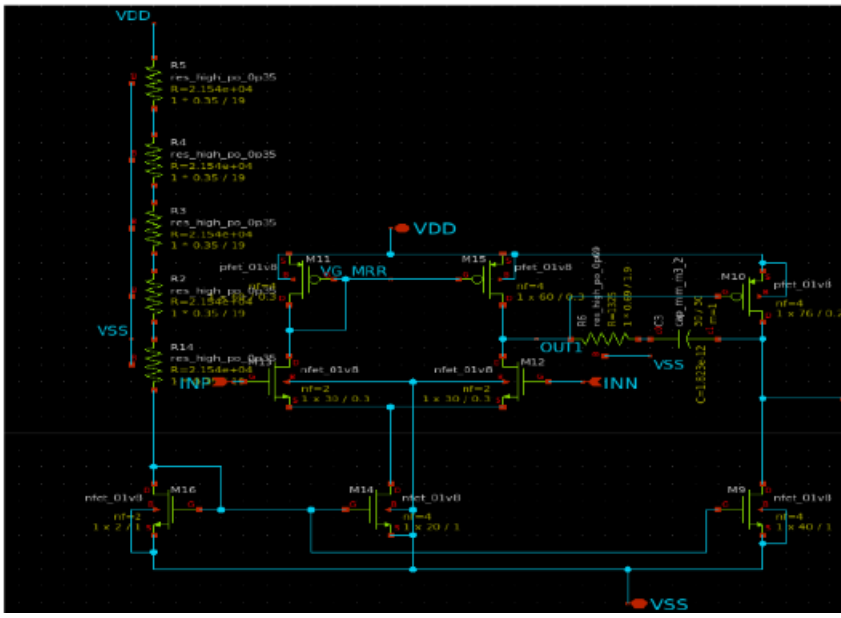
2-stage Opamp A 2-stage Miller operational amplifier (op-amp) circuit comprises two amplification stages, with a Miller capacitor connected between the output of the first stage and the input of the second stage. This arrangement improves stability and bandwidth, making it ideal for high-gain, high-frequency applications. The 2-stage Miller op-amp is frequently utilized in precision analog signal processing, active filters, and high-impedance buffer circuits.

First Stage: An NMOS differential input pair, selected for its high transconductance and speed, amplifies the difference between the input signals.

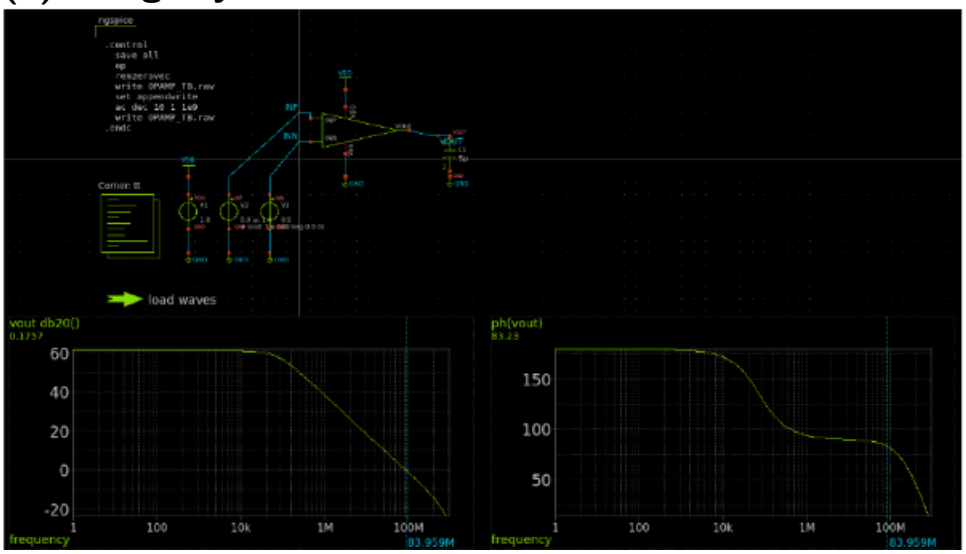
Second Stage: A common-source amplifier further increases the gain.

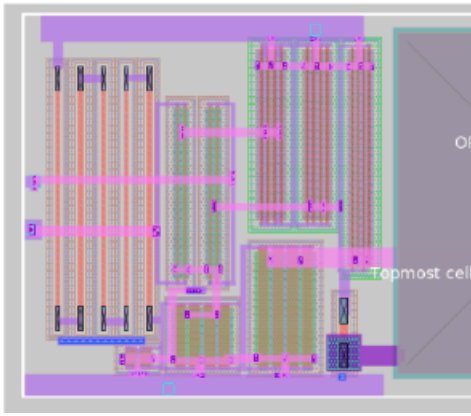
Miller Compensation involves using a capacitor between the output of the first stage and the input of the second stage to stabilize the op-amp by introducing a dominant low-frequency pole, which ensures stability in feedback systems.

In the op-amp design submissions, **Amr Abdelrahman** and **Majid Sami** each presented their designs. The details of their submissions are as follows.

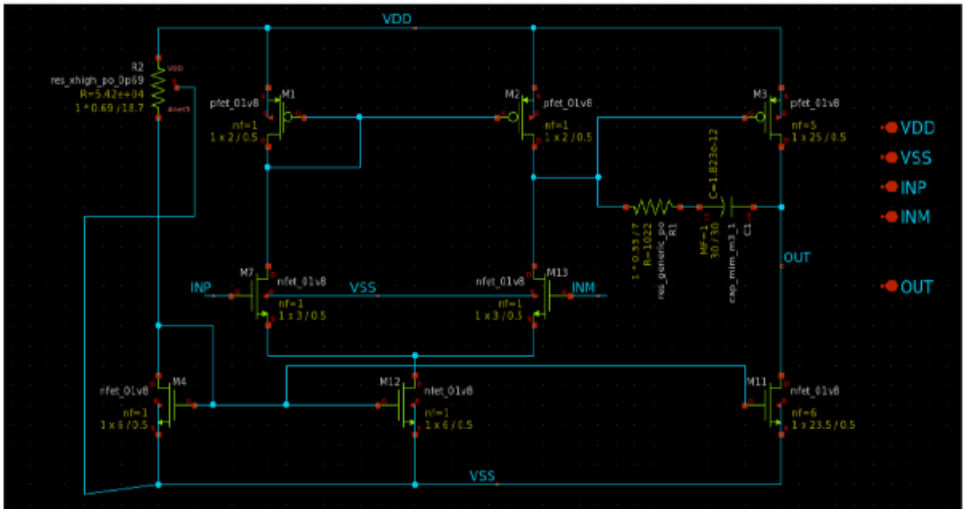


(1) Design by Amr Abdelrahman

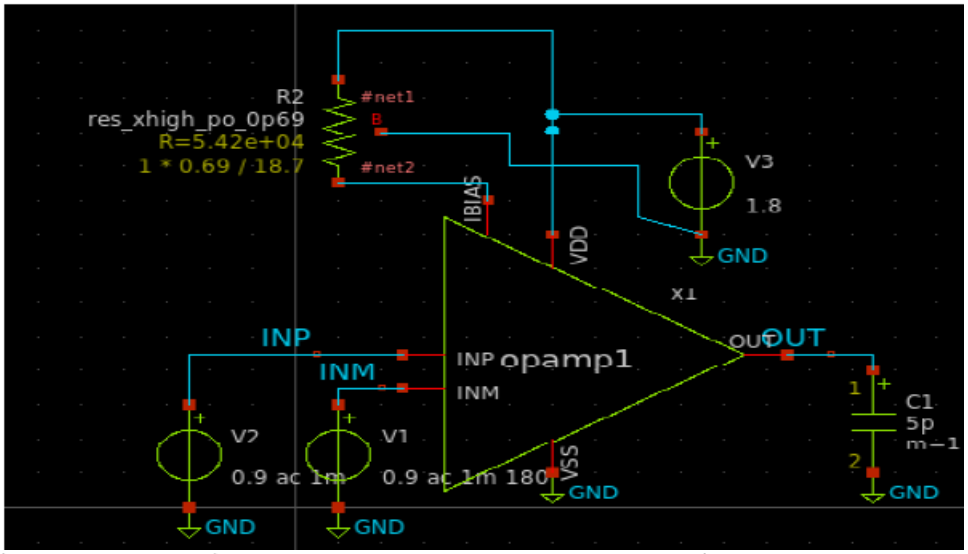


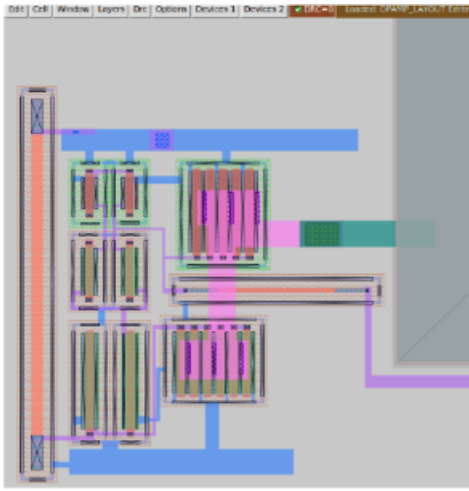


Specification	Target Value	Achieved Value	Unit
Power Supply (V_{DD})	1.8	1.8	V
Negative Supply (V_{SS})	0	0	V
Load Capacitance	5	5	pF
Gain (A_v)	60	61.2	dB
Unity Gain Bandwidth (UGB)	5	82.5	MHz
Slew Rate	10	42	V/μs
Output Range	0.3-1.5	0.21-1.55	V
Input Range	0.8-1.5	0.9-1.5	V
Power Dissipation	¡ 5	0.655	mW
Phase Margin (PM)	-	80	degrees



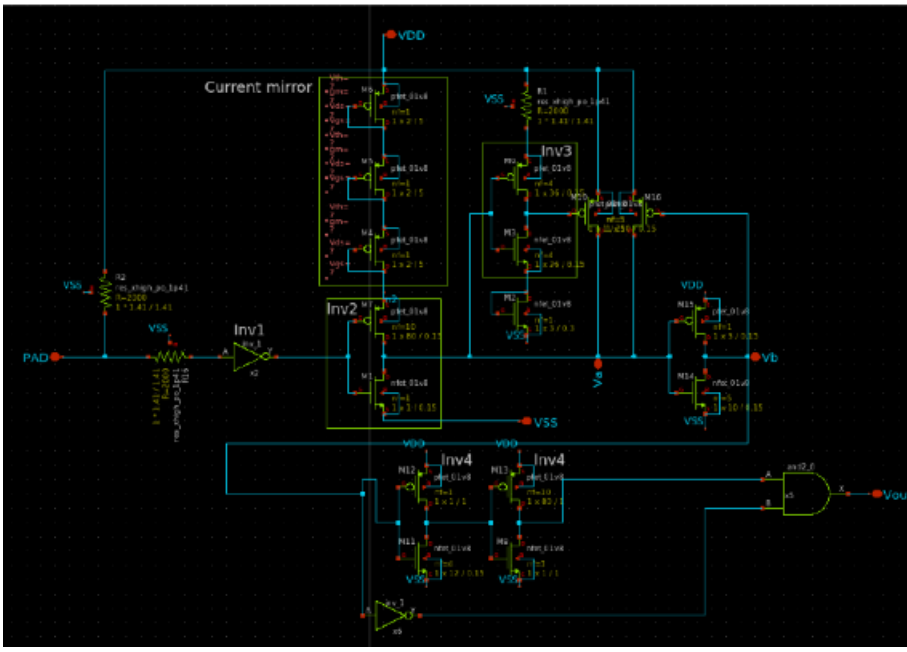
(2) Design by Majid Sami



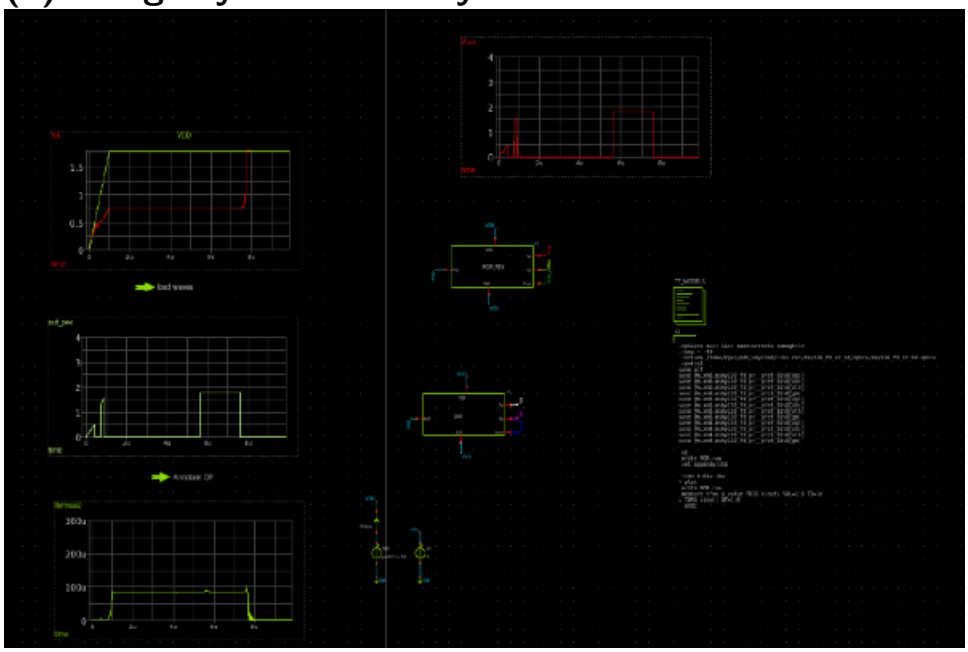


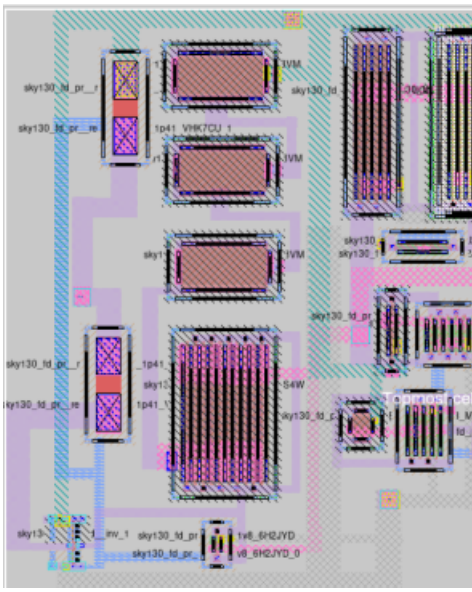
Spics	Result	Comment
VDD	1.8 v	Achieved
VSS	0 v	Achieved
Load Capacitor	5 ZpF	Achieved
Gian	71.9 dB	Achieved
UGB	16 MHz	Achieved
Slew Rate	10.6761	MATLAB Achieved
Output Range	0.25-1.6	Achieved
Input Range	0.2-1.6	Achieved
Power Dissipation	0.21632 mW	Achieved

POR (Power On Reset) Circuit Power-on reset (POR) circuit ensures that electronic systems start up in a known, stable state by generating a reset signal when power is initially applied. This circuit detects when the power supply reaches a sufficient voltage level and holds the reset line active until the voltage stabilizes, preventing erratic behavior and data corruption. POR circuits are crucial in microcontrollers, consumer electronics, and industrial systems, as they guarantee reliable initialization and consistent performance, thereby enhancing system stability and functionality during power-up.The circuit designed without capacitor which yields 30% reduction of chip's area compared to the conventional designs



(3) Design by Khalid Alorayir





How to test

(1) Testing Opamp Circuit

Non-Inverting Configuration

1. **Set Up the Circuit:** Connect the op-amp with the input signal to the non-inverting input (+) and a feedback resistor network to the inverting input (−).
2. **Power the Op-Amp:** Apply the required positive and negative supply voltages.
3. **Apply Input Signal:** Feed a known input signal to the non-inverting input.

4. **Measure Output:** Use an oscilloscope or multimeter to measure the output voltage and verify if **Vout = Vin.(1 + Rf/Rin)** matches the measured output.
5. **Check Frequency Response:** Ensure consistent gain across frequencies.
6. **Evaluate Stability and Linearity:** Look for any oscillations or instability and confirm the output accurately represents the input signal.

Buffer (Voltage Follower) Configuration

1. **Set Up the Circuit:** Connect the op-amp with output to the inverting input ($-$) and the input signal to the non-inverting input ($+$).
2. **Power the Op-Amp:** Apply the necessary supply voltages.
3. **Apply Input Signal:** Feed a known input signal to the non-inverting input.
4. **Measure Output:** Use an oscilloscope or multimeter to measure the output voltage and ensure it closely follows the input with minimal offset.
5. **Check Frequency Response and Stability:** Confirm fidelity of output across different frequencies and ensure stable output without oscillations.
6. **Assess Load Driving Capability:** Test with various loads to verify effective driving.

(2) Testing POR Circuit

1. Power up the circuit such that VDD voltage reaches final 1.8V value from 0V in 1us to 10us seep time.
2. Check the reset signal with an oscilloscope to confirm proper activation and deactivation.
3. Verify that the voltage reaches the threshold and that the reset signal duration is sufficient, especially for rise times of 1µs to 10µs.
4. Test the POR circuit under varying power conditions and ensure correct system initialization post-reset.
5. Power down and repeat the test to ensure consistent performance.

External hardware

Digilent Analog Discovery can be used for various measurements of opamp circuits.

1. **Signal Generation**: Use the Analog Discovery's waveform generator to create test signals for the op-amp and power-on-reset circuits.
2. **Measurement**: Connect the oscilloscope probes to monitor the input and output signals of the op-amp and observe the behavior of the power-on-reset circuit.

3. **Frequency Response**: Analyze the frequency response of the op-amp by sweeping through various frequencies and recording the output using the Analog Discovery's built-in tools.
4. **Transient Analysis**: Measure how the op-amp and power-on-reset circuits respond to transient signals or sudden changes, such as power-up events.
5. **Voltage Levels**: Check the stability and correct operation of the power-on-reset circuit by measuring the voltage levels and timing of the reset pulse.

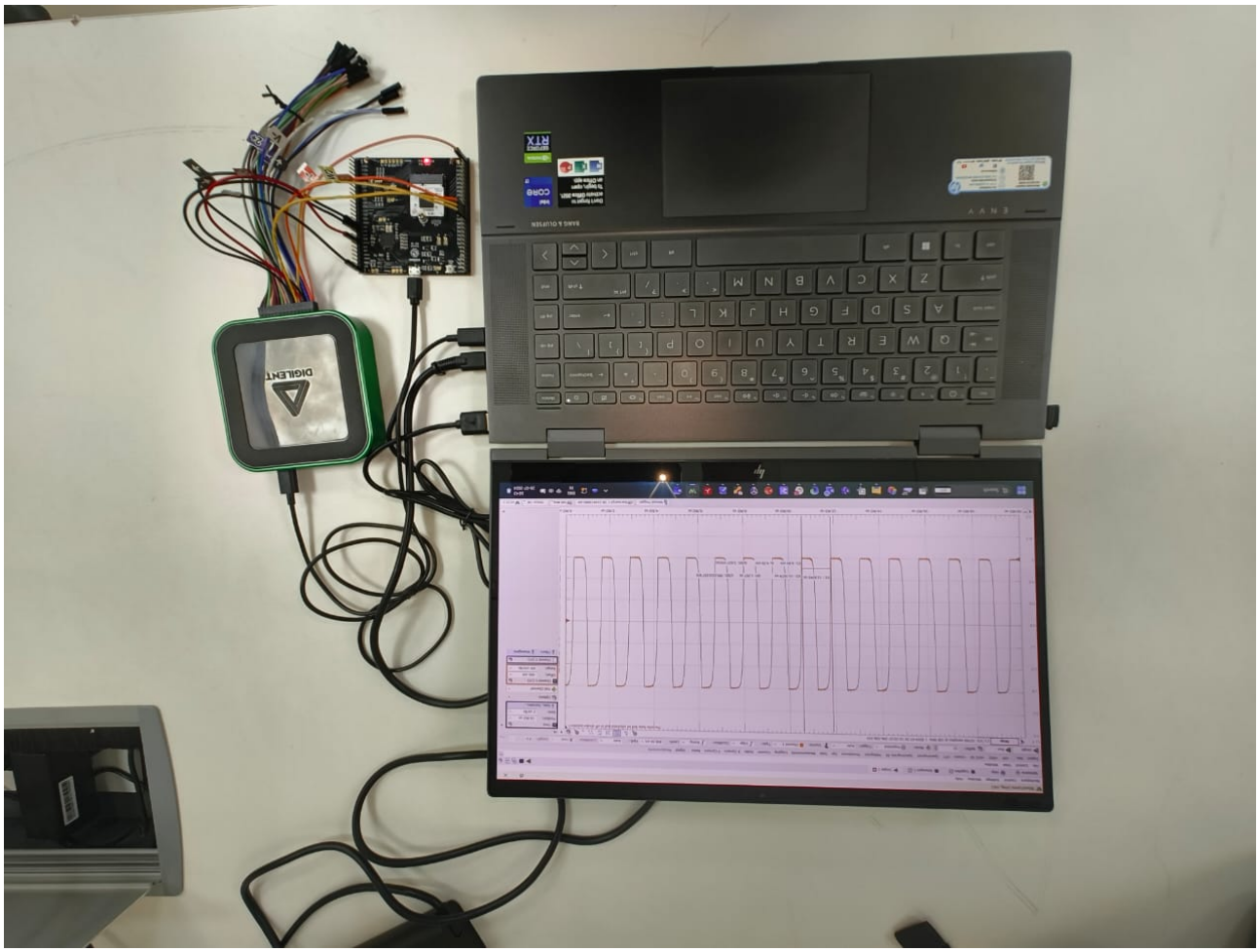


Figure 22: Sample Measurement Setup using Analog Discovery 3

Pinout

#	Input	Output	Bidirectional
0			
1			
2			
3			
4			

#	Input	Output	Bidirectional
5			
6			
7			

Analog pins

ua#	analog#	Description
0	5	Vout_POR
1	0	Va_POR
2	4	OUT_Maj
3	1	OUT_Amr
4	3	INN
5	2	INP

Neural Net ASIC [518]

- Author: Neural Navigators: Linyang Lee, Harsha Ganta, Stephanie Shen, William Li, Kiana Dai
- Description: MNIST Handwriting prediction on a neural network
- GitHub repository
- HDL project
- Mux address: 518
- Extra docs
- Clock: 10000000 Hz

How it works

Neural network

How to test

Test

Pinout

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

Pi Snake [520]

- Author: htfab
- Description: Voltage divider that generates volts
- GitHub repository
- Analog project
- Mux address: 520
- Extra docs
- Clock: 0 Hz

How it works

The blue circle has unit radius and therefore an area of units. We can enclose it by a polygon with exactly 3.3 units of area, adding the parts marked red. Using precision resistors that meander through the blue and red areas respectively we can construct a voltage divider that subdivides the the 3.3 V power and ground rails to achieve a V output.

How to test

Measure the voltage between the single analog output pin and ground. It should read approximately volts.

External hardware

Multimeter or other test equipment

Pinout

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

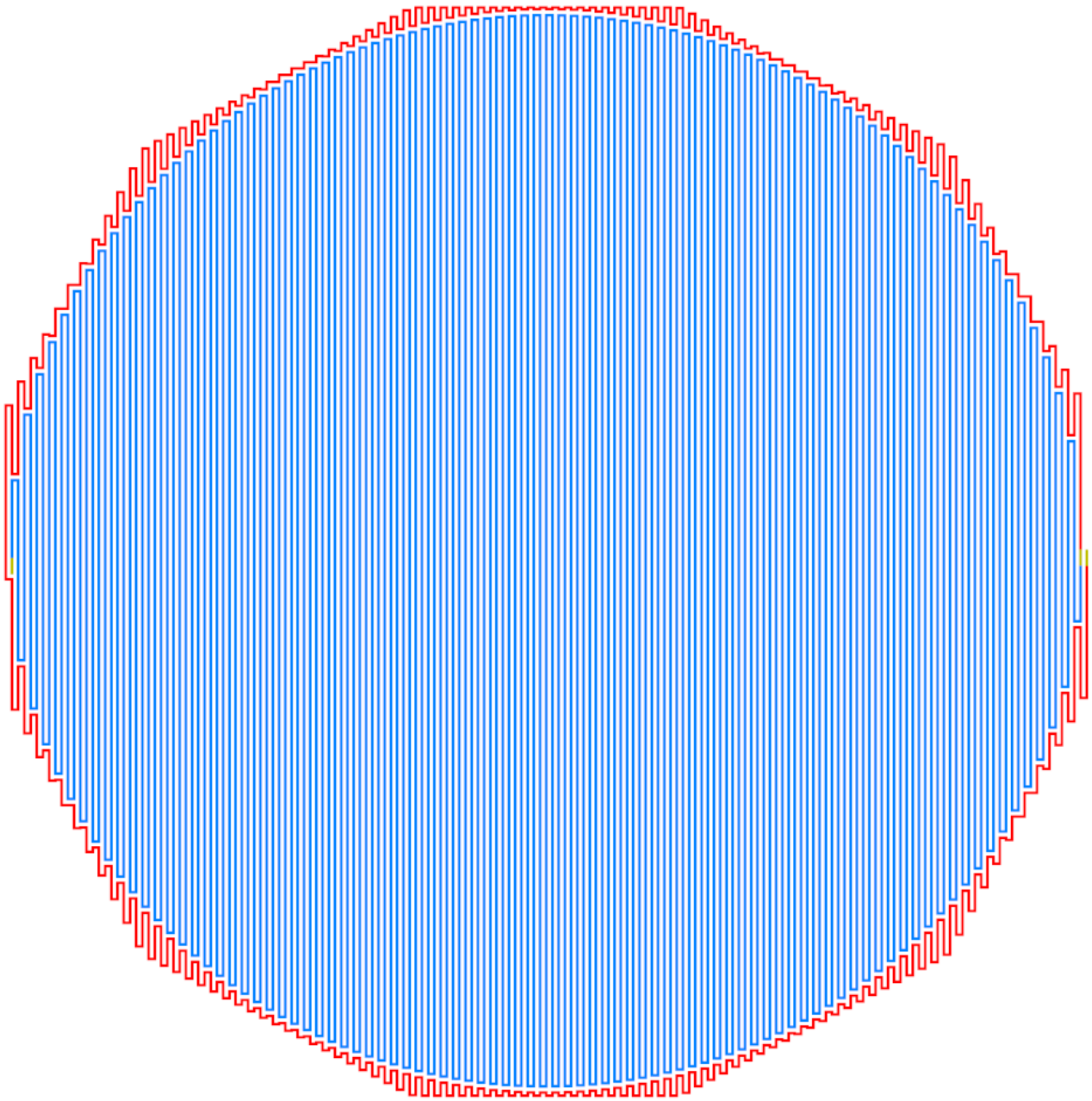


Figure 23: blue line meanders through circle, red line meanders through polygon minus circle

Analog pins

ua#	analog#	Description
0	6	voltage output

5-T OTA [522]

- Author: S Mishra
- Description: A simple 5-Transistor operational amplifier with external bias current
- GitHub repository
- Analog project
- Mux address: 522
- Extra docs
- Clock: 0 Hz

How it works

Apply a VDD of 1.8V and VSS ground. Then apply a 1uA current to BIAS pin. Then apply 0.9V to MINUS pin, while applying a voltage of 0-1.8V on PLUS pin in 10mV steps. Observe the DIFFOUT pin and check that DIFFOUT shoots up to 1.8V when PLUS is greater than MINUS by 10mV.

How to test

Observe the DIFFOUT pin and check that DIFFOUT shoots up to 1.8V when PLUS is greater than MINUS by 10mV.

External hardware

Power Supply, Sourcemeter for currents, Multimeter to measure voltage.

Pinout

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

Analog pins

ua#	analog#	Description
0	10	DIFFOUT
1	7	BIAS
2	9	MINUS
3	8	PLUS

Bandgap Reference [524]

- Author: Asal Golmanesh
- Description: Bandgap 1.2V at 1.8 supply with skywater 130nm from 0 to 80deg
- GitHub repository
- Analog project
- Mux address: 524
- Extra docs
- Clock: 0 Hz

How it works

This project involves designing a bandgap reference circuit using Skywater 130nm CMOS technology to provide a stable 1.17V output across a temperature range of 0 to 80°C. A startup circuit is included to initialize the system upon power-up. The circuit operates at 1.8V and consumes 0.35 mW of total power.

How to test

To test the circuit, apply the specified power supply voltage of 1.8V. Upon power application, the startup circuit should activate, enabling the entire circuit. The following major tests are required to evaluate the bandgap reference (BGR) circuit:

1. Temperature Testing: Use a temperature chamber or hotplate to vary the temperature and measure the reference voltage at different points:
    - Start at room temperature (~25°C).
    - Test at low temperatures (e.g., 0°C).
    - Test at high temperatures (e.g., 80°C).
2. Line Regulation: Vary the supply voltage (e.g., from 1.6V to 2.0V) and measure the output reference voltage to assess the circuit's ability to maintain a stable output.
3. Load Regulation: Apply different loads to the bandgap reference output and measure the change in output voltage to ensure stability under varying load conditions.
4. Power Consumption: Use a multimeter or source-measure unit (SMU) to measure the current consumption of the bandgap circuit.

5. Long-Term Drift: If feasible, operate the circuit for an extended period (e.g., hours or days) under typical conditions and periodically measure the output voltage to verify that the reference voltage does not drift significantly over time.

External hardware

Ensure a proper testing environment with equipment such as oscilloscopes, multimeters, SMUs, temperature chambers, and power supplies. An accurate power supply is essential to provide the required 1.8V (or the specified voltage in your design) to the circuit.

Pinout

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

Analog pins

ua#	analog#	Description
0	10	

Sine Wave Synthesizer [526]

- Author: Maximilian Scherzer
- Description: Generate Sine Wave
- GitHub repository
- Analog project
- Mux address: 526
- Extra docs
- Clock: 0 Hz

How it works

Sine wave synthesizer (SWS), that generates a stepwise approximation of a sinusoid.

How to test

The SWS produces a sinsoid with an output frequnecy depding on the clk.

Drive externally NN

Drive with internal sawtooth wave generator NN

External hardware

Pins (ion,iop) need a 1kohm resistor to gnd.

Pinout

#	Input	Output	Bidirectional
0	reset		
1	dem_dis		
2			
3			
4			
5			
6			

#	Input	Output	Bidirectional
7			

Analog pins

ua#	analog#	Description
0	11	bias
1	6	ib_hi
2	10	ib_lo
3	7	iop
4	9	ion

Simple 8 Bit ALU [576]

- Author: Joseph Johnson
- Description: a very simple ALU
- GitHub repository
- Wokwi project
- Mux address: 576
- Extra docs
- Clock: 1000 Hz

How it works

The project currently works by including a 8-bit shift register connected to the bus. The SI pin can be used to shift data into the register, and the ESO pin determines whether the shift-register will output to the bus. The bus is a conjunction of OR-gates making it one directional (as simply combining the wires seems less possible). There is a tri-state buffer that connects to two registers (register A and B) allowing them to interact with the bus. The direction is controlled using DIRA and DIRB. If the DIRA/DIRB sets the direction to be using the bus as an input, LA/LB determines whether the register should load what is on the bus. Register B is connected to the ALU and may be inverted to form a 2's Compliment substraction using the AS pin. The ALU will thus perform addition or subtraction depending on the state of AS. It will take Register A as the other input and output to the bus if the EAO pin is pulled high.

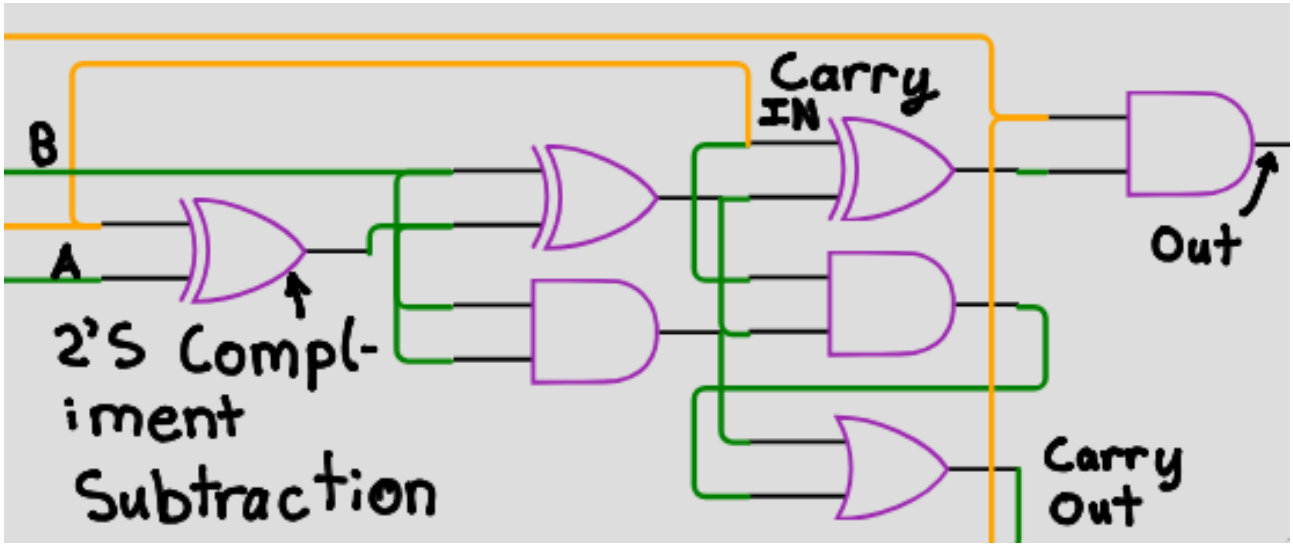


Figure 24: Annotated diagram of ALU

You can see here an annotated diagram of a single portion of the ALU. The wires labeled A and B represent the arbitrarily named inputs (in this case, the input labeled

A actually comes from the B register and vice versa). The original xor gate allows for 2's compliment subtraction with input A. The orange wire labeled carry in comes from the AS pin and determines whether A is being added or subtracted. 2's compliment subtraction requires one be added to the number so we carry one in (this is the LSB, so nothing would be carried in typically). The output on the right is the single digit output. The carry out will be hooked up to the carry in of the next segment to add numbers greater than 1 bit.

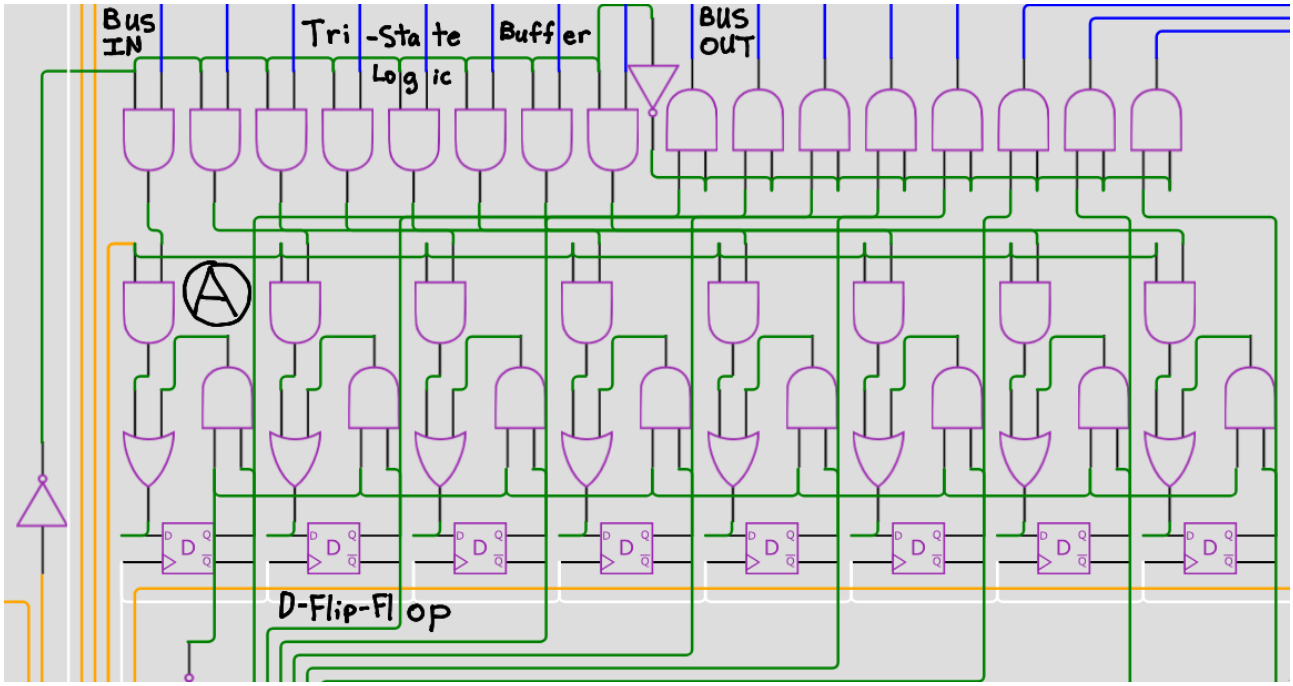


Figure 25: Annotated diagram of register

You should be able to see here an annotated diagram of a single register. The 8 blue wires to the right of "Bus IN" represent the bus input and will only be active if the DIRA pin is set low (because of the invertor towards the bottom left). This is inverted to determine whether the bus should be considered an output and the bus output appears on the top right. The section labeled "A" is a special circuit that does not load any data if the LA pin is not high. Because the D-Flip-Flop circuit will complain if its clock input contains any logic when being converted, a more difficult method to prevent unintended data modification must be used. If the LA pin is not high, the Q pin is piped back to the D pin on each of the flip-flops, retaining the data.

This picture demonstrates the data bus. It uses logical OR gates to connect a series of wires together which are then used to produce output. The bus output is shown as the output of the entire circuit, and can be used to send data between the ALU and registers.

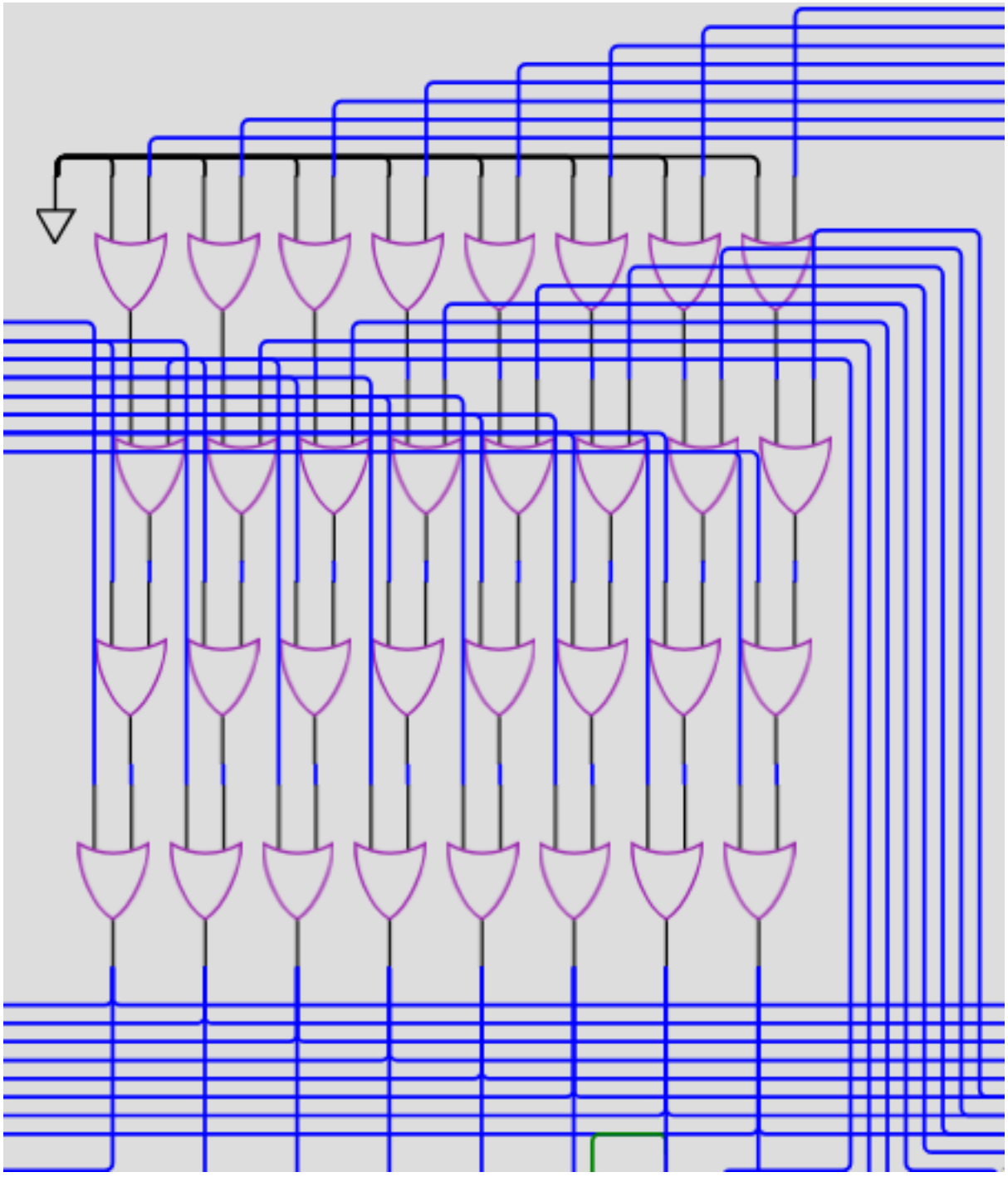


Figure 26: Diagram of Bus

How to test

Input your two numbers MSB first (using the SI pin), pulsing the clock after inputting each individual bit. Set the ESO pin high to output the contents of the shift-register to the bus. Leave DIRA and DIRB on their default to allow information to stream in from the bus (inverting them will output the contents of register A or B to the bus). Set LA or LB appropriately depending on where you would like to load information. Pulse the clock to load information from the bus to any listening registers. Reset LA/LB to prevent accidentally overwriting data from the bus. Whenever Register A and B are set correctly, set the AS pin depending on whether you desire to add or subtract register B. Set the EAO pin high to output the contents of the ALU to the bus. The output pins will mirror the bus and be produced with OUT0 indicating the LSB and OUT7 indicating the MSB.

You will likely wish to shift a few zeros into the shift-register first to ensure it is not full of corrupted data (it likely will be when the chip is initially started).

One fun thing to do is load the value 1 into register A and 0 into register B then set the contents of the ALU to output to the bus and set LB high. This will cause 1 to be added to register B every clock pulse and you should be able to see the output count up. You can do the same thing but backwards if you set the AS pin high.

Pin Location	Pin Name	Expanded Name	Description
0	SI	Serial Input	user may input data serially using this pin
1	ESO	Enable Serial Output	outputs the shift-register to the bus if high
2	DIRA	A Direction	allows register A to act as an input if low and vice versa
3	DIRB	B Direction	allows register B to act as an input if low and vice versa
4	EAO	Enable ALU Output	outputs the contents of the ALU to the bus if high
5	LA	Load A	loads information from the bus to register A if high
6	LB	Load B	loads information from the bus to register B if high
7	AS	Add Subtract	determines whether B is added (low) or subtracted (high)

External hardware

A sequence of 8 LEDs is hooked up to the outputs to display the final number in binary.

Pinout

#	Input	Output	Bidirectional
0	SI	OUT0	
1	ESO	OUT1	
2	DIRA	OUT2	
3	DIRB	OUT3	
4	EAO	OUT4	
5	LA	OUT5	
6	LB	OUT6	
7	AS	OUT7	

4-bit ALU [578]

- Author: Richard Xu, Louis Barbosa, Hallie Ho, Emmy Xu, Gia Bhatia, Emily Chen
- Description: The 4-bit ALU is designed to perform basic arithmetic and logical operations on 4-bit binary numbers
- GitHub repository
- HDL project
- Mux address: 578
- Extra docs
- Clock: 0 Hz

Project Datasheet: 4-Bit ALU

Overview The 4-Bit ALU is designed to perform various arithmetic and logical operations on 4-bit binary numbers. It supports operations such as addition, subtraction, multiplication, division, and several logical operations. Additionally, it includes an encryption function that can be used to encrypt 4-bit inputs using an 8-bit key. This capstone project is from the MIT BWSI Basics of ASICs class.

How it Works The module accepts two 4-bit binary numbers, `a` and `b`, and a 4-bit operation code (`opcode`) that determines the operation to be performed. The results are then output through the `uo_out` wire, while additional status information, such as carry out and overflow, is output through the `uio_out` wire. The `uio_oe` wire controls the enable or disable functionality for the `uio_in` and `uio_out` wires.

Operations

- **ADD**: Adds `a` and `b`, producing a 4-bit result and a carry out.
- **SUB**: Subtracts `b` from `a`, producing a 4-bit result and a borrow indication.
- **MUL**: Multiplies `a` and `b`, producing an 8-bit result.
- **DIV**: Divides `a` by `b`, producing a 4-bit quotient and remainder. Division by zero is handled by returning a zero result.
- **AND**: Performs a logical AND operation on `a` and `b`.
- **OR**: Performs a logical OR operation on `a` and `b`.
- **XOR**: Performs a logical XOR operation on `a` and `b`.
- **NOT**: Performs a logical NOT operation on `a`, with `b` being ignored.
- **ENC**: Encrypts the inputs `a` and `b` using an 8-bit key derived from concatenating `a` and `b`. The encryption function applies an XOR operation between this 8-bit concatenated value and a fixed 8-bit key. The result is an 8-bit encrypted value.

How to Test To test this 4-bit ALU chip, set the values for a and b based on the 4-bit binary values for each, as well as the 4-bit operation code. The output can be up to 8-bits, using the uo pins along with the first 4 bidirectional pins. The uio pin 6 is used for the carry out, and the uio 7 is used for the overflow.

External hardware N/A

Pinout

#	Input	Output	Bidirectional
0	a[0]	result[0]	opcode[0]
1	a[1]	result[1]	opcode[1]
2	a[2]	result[2]	opcode[2]
3	a[3]	result[3]	opcode[3]
4	b[0]	result[4]	
5	b[1]	result[5]	
6	b[2]	result[6]	carry_out
7	b[3]	result[7]	overflow

Morse Code Keyer [580]

- Author: Brady Etz
- Description: Convert a keyed CW input to morse tones and 7-segment character output
- GitHub repository
- HDL project
- Mux address: 580
- Extra docs
- Clock: 12000000 Hz

How it works

Morse Keyer takes a paddle-type dit/dah signal (io[0:1]) and converts it to an auxilliary Morse code output (io[4]) and a buzzer tone (io[5]). The design outputs auxilliary dit/dah signals (io[2:3]) to send to other gear, like a radio. Additionally, it outputs to the demonstration board's seven-segment display (out[7:0]) to reveal the character you just completed as you key.

To use a straight-key input (or press a single button to key Morse code) set in[0] HIGH. For Iambic paddles, set in[0] LOW. To use Iambic keying type A, set in[1] LOW. For Iambic-B, set in[1] HIGH. (https://ag6qr.net/index.php/2017/01/06/iambic-a-or-b-or-does-it-matter/) WPM control is set between 7 WPM and ~100 WPM with in[7:4] via the demo board dip switches. The timing element in this system divides the system clock first with a 512x prescaler, then feeds it into the variable delay below:

Control [7:4]	WPM	Clocks	Timer Preset
4'b0000	110.3	255	12'b000011111111
4'b0001	55.0	511	12'b000111111111
4'b0010	36.7	767	12'b001011111111
4'b0011	27.5	1023	12'b001111111111
4'b0100	22.0	1279	12'b010011111111
4'b0101	18.3	1535	12'b010111111111
4'b0110	15.7	1791	12'b011011111111
4'b0111	13.7	2047	12'b011111111111
4'b1000	12.2	2303	12'b100011111111
4'b1001	11.0	2559	12'b100111111111
4'b1010	10.0	2815	12'b101011111111
4'b1011	9.2	3071	12'b101111111111
4'b1100	8.5	3327	12'b110011111111
4'b1101	7.9	3583	12'b110111111111

Control [7:4]	WPM	Clocks	Timer Preset
4'b1110	7.3	3839	12'b111011111111
4'b1111	6.9	4095	12'b111111111111

WARNING: The auxilliary Morse output MUST NOT be used as a raw radio TX control for a homemade radio. This is because the keying interface must control the transmitted wave shape to maintain acceptable RF bandwidth. Be a good RF neighbor. Always use the provided keyer inputs for your radio. These are typically provided with a 3.5mm TRS jack, with Sleeve = GND, Ring = Dah, and Tip = Dit/Straight.

Most radio systems use active-low / open-collector signaling to protect systems operating at various supply voltages. Please see the External Hardware section for recommendations.

How to test

Set the input clock frequency to 12 MHz. Set the dip switches (in[7:0]) on the dev board to the desired paddle setup and WPM rate. Attach hardware like that shown in the External Hardware section to use.

External hardware

For the best experience, and to use custom radio hardware and paddles, I recommend assembling a companion PCB affixed the bidirectional PMOD.

A /pcb/ directory accompanies the standard Tiny Tapeout directories with the KiCad files.

Please see the schematic below for a screenshot of the recommended application schematic:

Pinout

#	Input	Output	Bidirectional
0	Paddle Selection (1 = Iambic)	7-Seg. Display A	External Dit / Straight In (active-high)

#	Input	Output	Bidirectional
1	Iambic-A/B Type Selection (1 = B)	7-Seg. Display B	External Dah In (active-high)
2		7-Seg. Display C	Aux. Dit Paddle Out (active-high)
3		7-Seg. Display D	Aux. Dah Paddle Out (active-high)
4	WPM Select [0] (LSB)	7-Seg. Display E	Aux. Morse Out (active-high)
5	WPM Select [1]	7-Seg. Display F	Buzzer Tone Out
6	WPM Select [2]	7-Seg. Display G	
7	WPM Select [3] (MSB)	7-Seg. Display .	

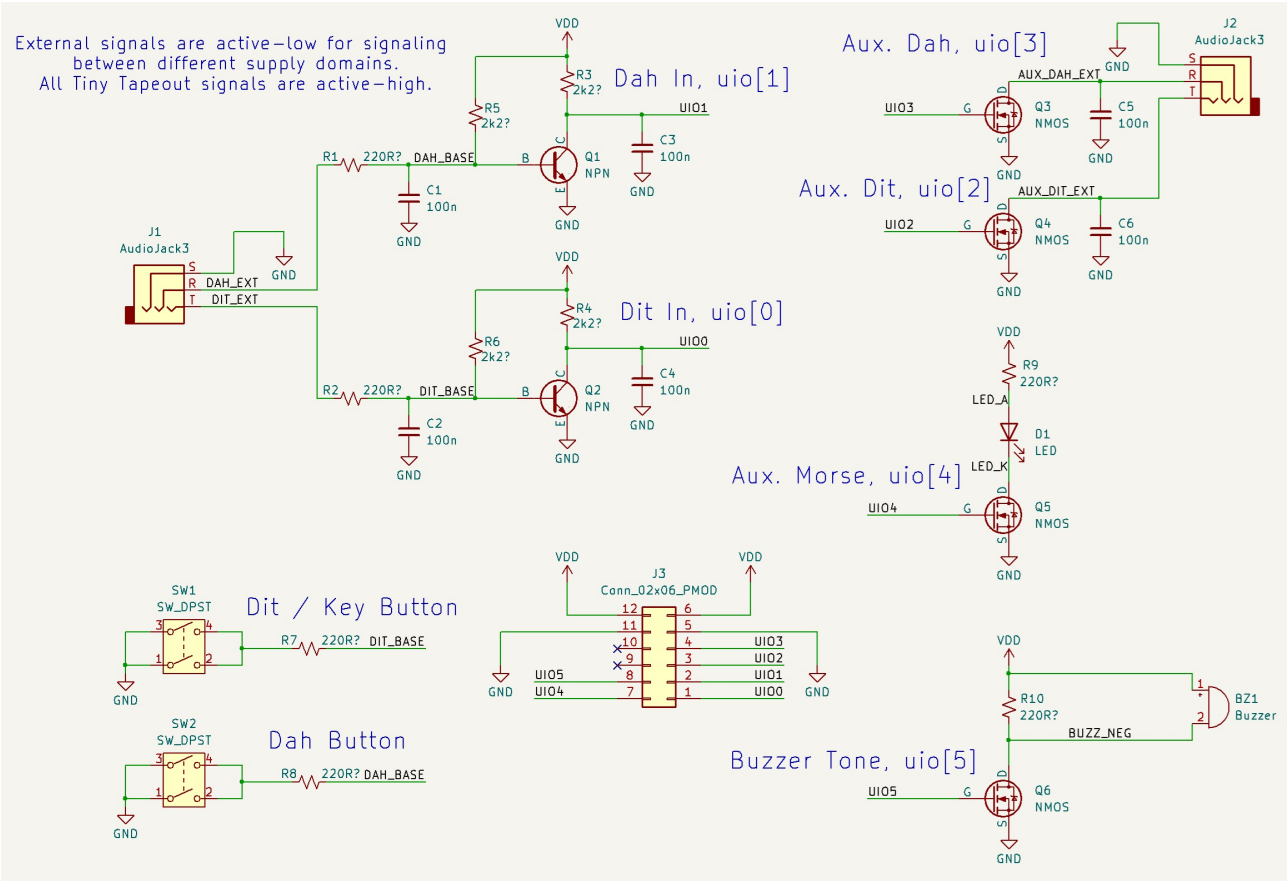


Figure 27: KiCad Application Schematic - 2024 Sep 02

nVious Graphics [582]

- Author: James Ross
- Description: Submission for VGA Demoscene
- GitHub repository
- HDL project
- Mux address: 582
- Extra docs
- Clock: 25175000 Hz

How it works

This is a VGA demo that runs without input, but also accepts 8-bit input on the `ui_io[7:0]` pins to display a virtual 7-segment LED display (with decimal).

How to test

Basic Functionality Plug into a VGA monitor, select this circuit to test, and reset.

External Input To test the user input functionality, connect the `ui_io[7:0]` pins. The idea is that this would be a possibly useful graphical extension to the dozens of existing projects that utilize the 7-segment LED display to show results.

External hardware

Requires the TinyVGA PMOD

Pinout

#	Input	Output	Bidirectional
0	Segment A	R1	
1	Segment B	G1	
2	Segment C	B1	
3	Segment D	VSync	
4	Segment E	R0	
5	Segment F	G0	
6	Segment G	B0	

#	Input	Output	Bidirectional
7	Segment H	HSync	

Tiny PLL [584]

- Author: LegumeEmittingDiode
- Description: 4-channel fractional-N frequency synthesizer
- GitHub repository
- HDL project
- Mux address: 584
- Extra docs
- Clock: 10000000 Hz

How it works

Overview This project showcases `tiny_pll`, a completely self-contained fractional-N frequency synthesizer using less than 6% of the area of a 1x1 TinyTapeout tile. The design goals of this project were as follows:

1. The design should be as simple as possible to reduce the chance of failure.
2. The design should be as small as possible so it can be incorporated into future Tiny Tapeout designs with minimal area overhead.

There are 4 `tiny_pll` instances in this project. Each instance multiplies the frequency of a reference clock by a rational number A/B, where A and B can be between 1 and 15. Such a block has two main use cases: provided to the tile through the mux and GPIO pins

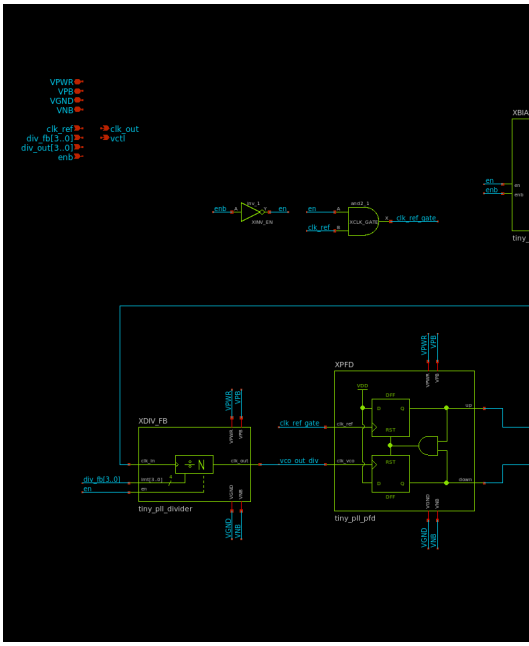
1. Generating several internal clocks from a single off-chip oscillator (e.g., for a large digital design with multiple clock domains)
2. Generating one or more internal clocks at a higher frequency than what can be

`tiny_pll` is designed for a 10 MHz reference input, which implies an output frequency between 67 kHz and 150 MHz. The 4 output clocks are connected to the GPIO pins `uo[3:0]`. In reality, the maximum output frequency is limited by 4 factors:

1. The speed of the Caravel I/O cells, which itself is a factor of the off-chip load capacitance
2. The routing between the TT mux and the I/O cells
3. The speed of the TT mux
4. The routing between the project tile and the TT mux The minimum output frequency is limited to roughly 1 MHz due to the minimum speed of the VCO.

A 1-bit delta-sigma ADC is included to allow measurement of the analog control voltage on `uo[4]`.

This design is inherently mixed-signal due to the analog nature of the PLL. Consequently, the top-level layout is implemented as a custom analog/digital section for the PLL and ADC, surrounded by RTL which implements the control/status registers (CSRs) and various clock buffering and multiplexing functions. Schematics were created using `xschem` and simulated with `ngspice`; custom layout was done using `klayout` with the Efabless `sky130` PDK; digital synthesis and PnR was done using a custom OpenROAD flow; and `magic` and `netgen` were used for LVS, DRC and parasitic extraction.



PLL The top-level schematic of `tiny_pll` is shown below:

The PLL uses a standard fractional-N architecture, where an input and output frequency divider are used to set the frequency multiplication with respect to the reference clock input. The output frequency is `A/B * f_ref`, where `A` is the division ratio of `XDIV_FB`, `B` is the division ratio of `XDIV_OUT` and `f_ref` is the input clock frequency. Documentation for the PLL subcells is included below.

Throughout the schematics, the pins `VPB` and `VNB` are included to connect the bulk terminals of all PMOS and NMOS devices, respectively. This is done to ensure the corresponding terminals of the standard cell instances at each level of hierarchy are propagated to the top level and connected to VPWR and VGND.

Divider Frequency dividers are implemented using a 4-bit binary counter followed by 4 XOR gates to check for equality with a division ratio input `lmt[3..0]`. When the counter output is equal to `lmt`, `div_rstb` is immediately asserted, which resets the

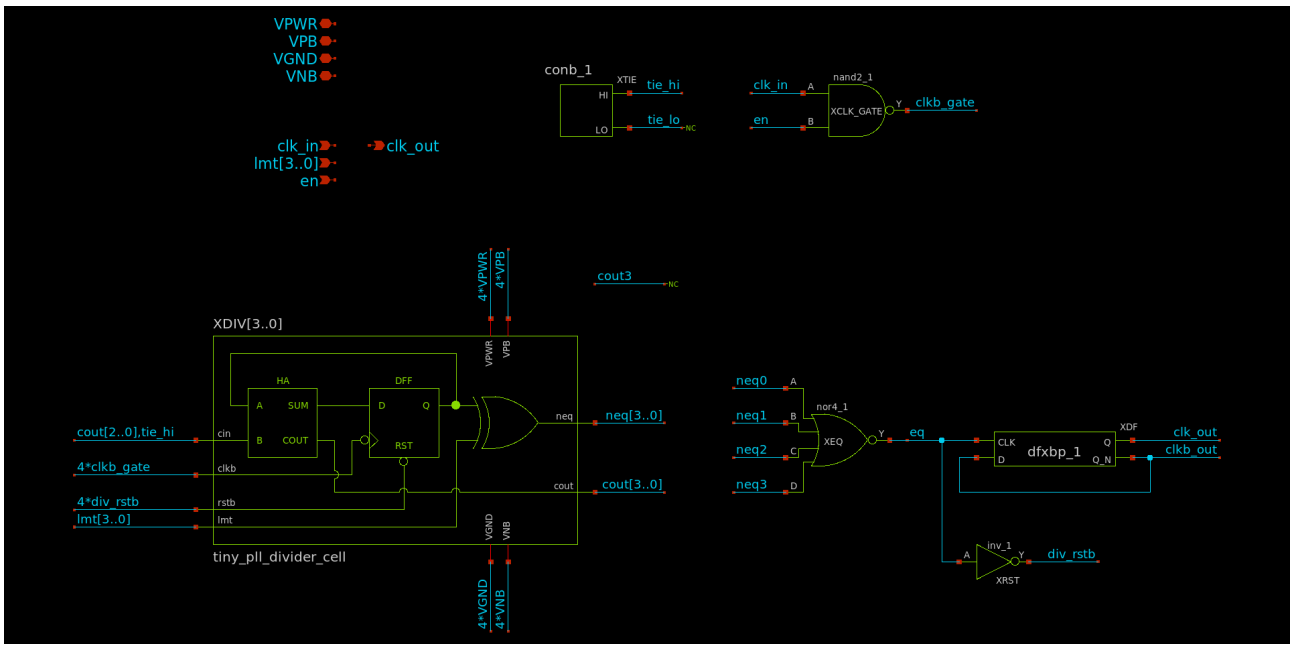


Figure 28: Divider schematic

counter to 0 at the rising edge of `clk_in`. As a result, the maximum division ratio from `clk_in` to eq is 15, when `lmt == 4'b1111`.

Since the counter is reset as soon as its output is equal to the division ratio, a very short pulse is produced at the eq node, with a duration equal to the propagation delay of the counter. This could potentially be a timing concern for `XDF`, but since the counter delay is at least 3 gate delays, the flip-flop was observed to operate as intended across process, voltage and temperature (PVT) in simulation.

The D flip-flop (DFF) at the output is included to ensure an output duty cycle close to 50%. As a result, the actual output frequency is `f_ref / (2*lmt)`, which implies a division ratio from `clk_in` to `clk_out` between 2 and 30.

The tie cell `sky130_fd_sc_hd__conb_1` is used when gates must be connected to VPWR or VGND to avoid potential ESD issues.

Phase-frequency detector (PFD) The PFD is composed of two DFFs, clocked by the divided VCO output and the reference input, respectively. Since the input of both DFFs is tied to 1, each DFF can be implemented using two S-R latches, each of which uses two `nor2` gates. The full PFD thus uses 8 `nor2` gates, one `nand2` and one `inv_1`, which is considerably smaller than using discrete DFF standard cells with the D inputs tied to VPWR.

A NAND followed by an inverter is used instead of a single AND to slightly increase the minimum output pulse width and avoid charge pump glitches.

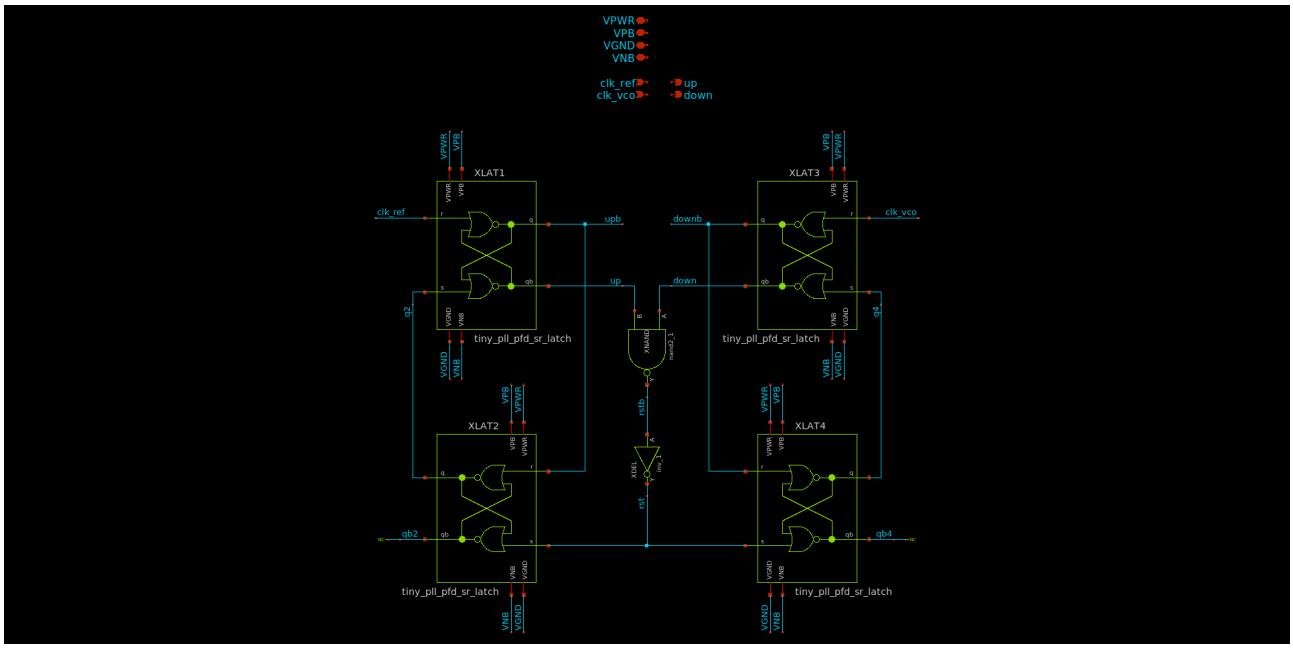


Figure 29: PFD schematic

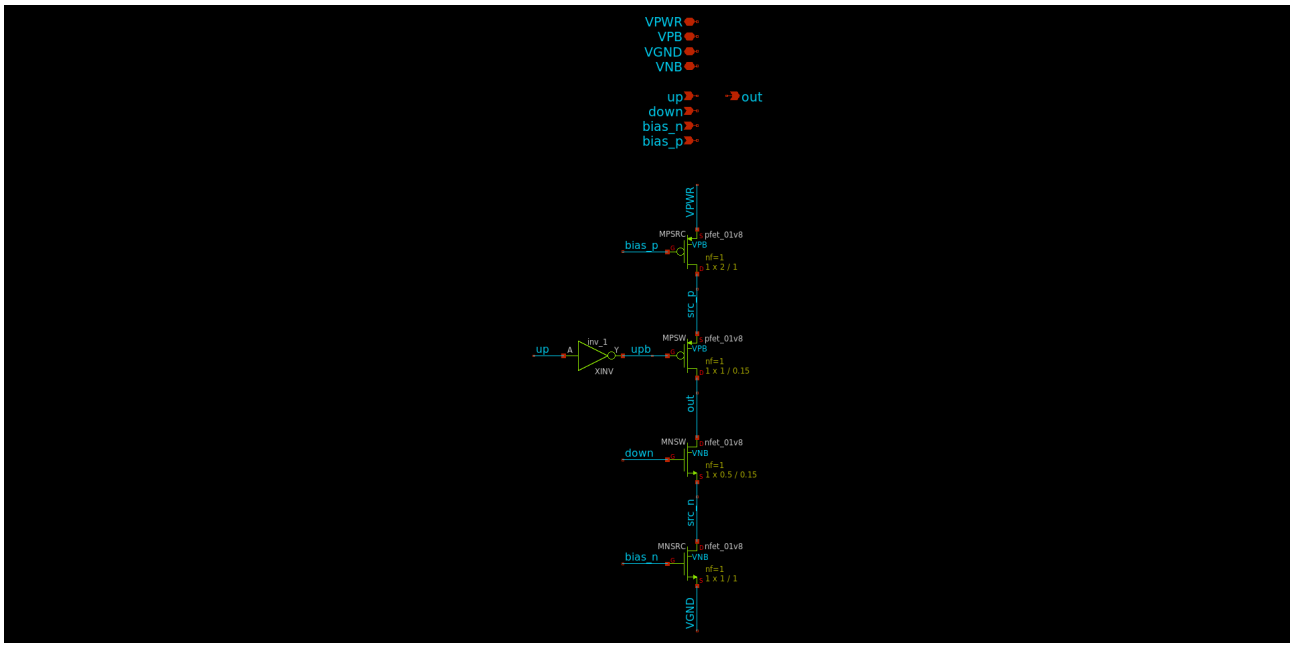


Figure 30: Charge pump schematic

Charge pump The charge pump uses two current sources (`MNSRC` and `MPSRC`), which can be interchangeably switched to the output with the `up` and `down` inputs. The charge pump current is nominally 1 uA and is set by the bias generator. The switches use nearly minimum width to reduce area, and minimum length to reduce capacitance. The PMOS switch uses 2x the W/L of the NMOS switch to ensure roughly equal drain-source saturation voltages (VDSAT).

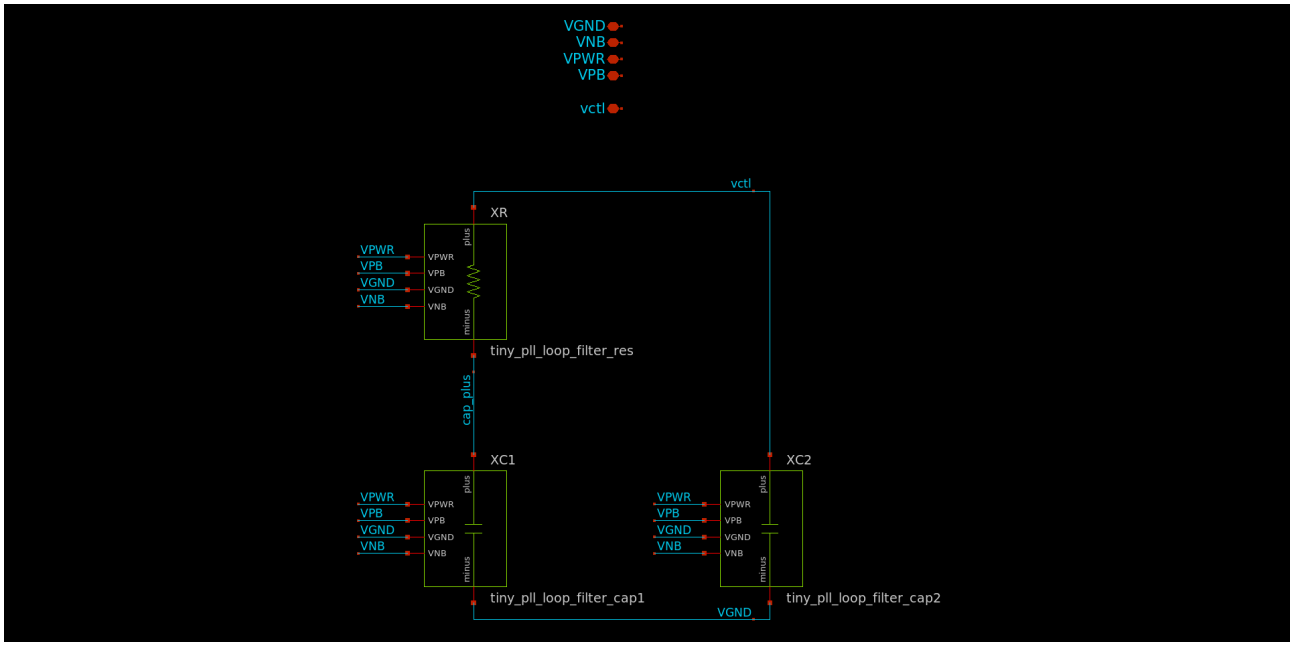


Figure 31: Loop filter schematic

Loop filter The loop filter is implemented using a series R/C combination to compensate the loop transfer function such that a zero is placed below the crossover frequency to ensure stability, and a pole is placed above the crossover frequency to ensure fast settling time. A second capacitor `XC2` is included to reduce ripple in the control voltage, which in turn reduces phase noise at the PLL output. Component values were selected using a linearized model developed using schematic-only simulations of the VCO to determine the voltage-to-frequency gain. The loop bandwidth was chosen to be on the order of 100 kHz, with a phase margin of 65 degrees at an output frequency of 10 MHz. The resulting R/C values are `R = 100 kOhm` and `C1 = 1 pF`.

In reality, the loop characteristics vary significantly across output frequency due to the nonlinear gain of the VCO, which was observed to have a nearly exponential voltage-to-frequency characteristic in simulation. This is likely due to the VCO current sources operating in the subthreshold region, where the ID/VGS characteristic is near-exponential.

The loop filter resistor is implemented using the `urpm` high-resistance poly implant, which is roughly 2 kOhm/square. While e-test values are not provided for this resistor

in sky130, the value is not critical, and significant variations (+/-50%) were observed to result in a stable loop in simulation.

The loop filter capacitors are implemented using NMOS devices with drain and source shorted to VGND. This is due to the significantly higher capacitance density of MOS devices relative to MIM capacitors (~8 vs ~2 fF/um^2). The MOS capacitance is highly nonlinear and increases at high control voltages due to the inversion charge, but again the capacitor value is not critical and this nonlinearity does not cause instability in the feedback loop.

The loop filter consumes nearly 50% of the area of the PLL. Various methods were explored to reduce loop filter area, including:

1. MIM capacitors could be used and placed on top of the other circuit blocks to reduce area
2. A capacitance multiplier could be used to allow a smaller intrinsic capacitance

The MIM capacitor method is possible, but there is some ambiguity in the sky130 design rules as to whether a MIM capacitor can be placed over met1 and the base layers (see capm.10 in the sky130 periphery rules. Additionally, this could result in unwanted noise from the digital blocks coupling into the capacitors, which could degrade phase noise performance. Further, the capacitors would have to be divided up to lie between the power rails on met4 which would increase their area.

A capacitance multiplier was implemented using a 100 fF capacitor with a 10:1 multiplication ratio, but the final layout was the same size as the MOS capacitor implementation and was thus exlcuded from the final design. The capacitance multiplier was additionally seen to have poor high-frequency response compared to a MOS or MIM capacitor, which resulted in unacceptably high control voltage ripple.

Voltage-controlled oscillator (VCO) The VCO is a 3-stage current-starved ring oscillator using standard cell inverters. The current sources are minimum-length to maximize W/L, which in turn minimizes VDSAT, and minimize capacitance. The output resistance of these current sources is irrelevant since it only matters that the oscillator current is limited, and not the particular limit value. A triode device MNCTL is used to control the source/sink current of the VCO. LVT NMOS devices are used to ensure the operating control voltage is somewhere near half supply at an output frequency of 10 MHz, which helps ensure the maximum output frequency can be met across process variations. Four "keeper" devices (MNEN1, MNEN2, MNEN3 and MPEN) are included to disable the circuit with zero static power consumption.

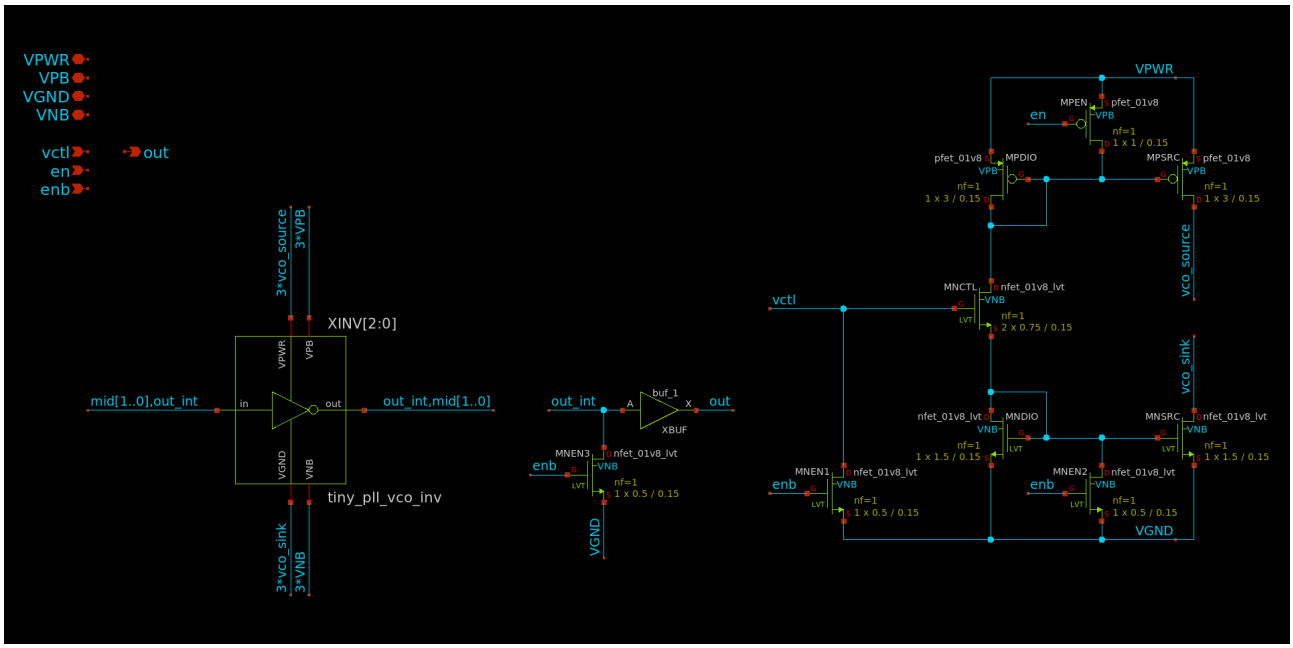


Figure 32: VCO schematic

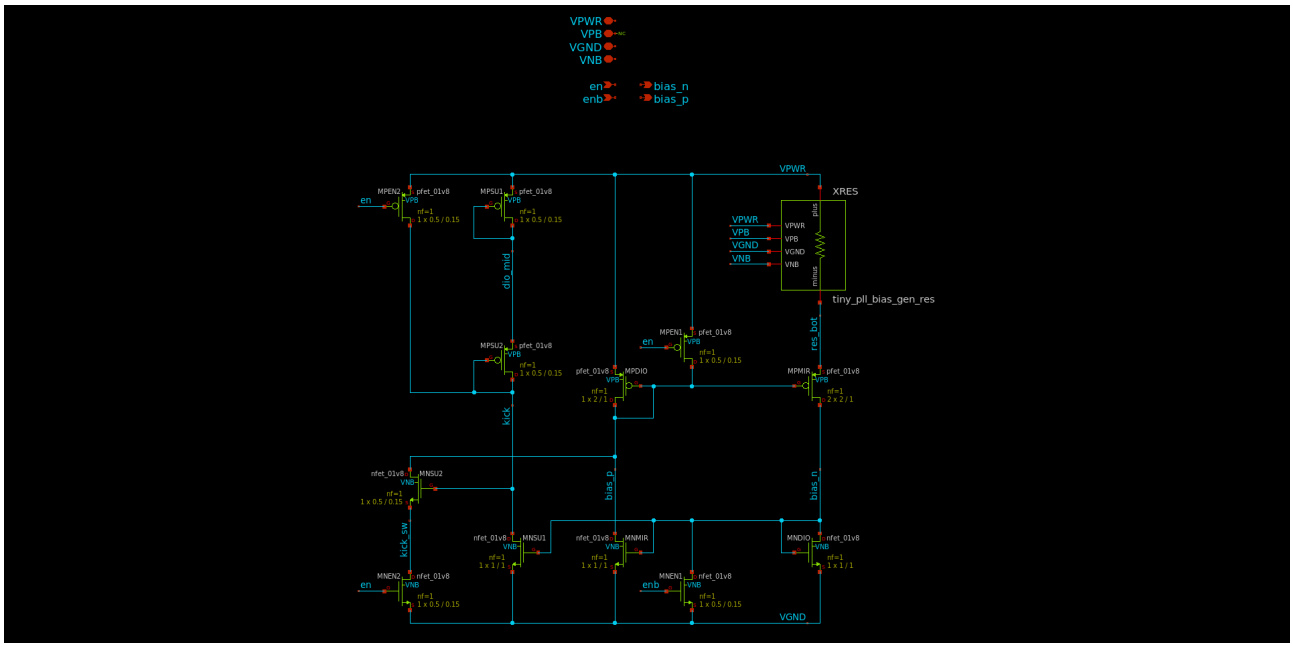


Figure 33: Bias generator schematic

Bias generator The bias generator is a self-biased current mirror, which provides a roughly supply-independent current for the charge pump. The exact current is highly dependent on the poly resistor XRES, but is designed to be nominally 1 uA at 25 degrees C. A startup circuit is included to ensure the bias generator does not fall into an undesirable operating point where IOUT = 0. The diode devices MPSU1 and MPSU2 charge the kick node to VPWR when the circuit is enabled, which pulls bias_p low and establishes a current in the mirror devices. Once the mirror is active, MNSU1 pulls kick low and disables the startup circuit. Multiple "keeper" devices are included to disable the circuit with zero static power consumption.

Pinout

#	Input	Output	Bidirectional
0	csr_data_in[0]: Data input for PLL control registers	clk_out[0]: Channel 0 PLL clock output	clk_csr: Clock input for PLL control registers
1	csr_data_in[1]: Data input for PLL control registers	clk_out[1]: Channel 1 PLL clock output	
2	csr_data_in[2]: Data input for PLL control registers	clk_out[2]: Channel 2 PLL clock output	
3	csr_data_in[3]: Data input for PLL control registers	clk_out[3]: Channel 3 PLL clock output	
4	csr_addr_in[0]: Address input for PLL control registers	adc_out: Channel 0 control voltage ADC output	

#	Input	Output	Bidirectional
5	csr_addr_in[1]: Address input for PLL control registers		
6	csr_addr_in[2]: Address input for PLL control registers		
7	csr_addr_in[3]: Address input for PLL control registers		

simon_cipher [585]

- Author: Simon Cipher
- Description: Bitserial implementation of Simon-128
- GitHub repository
- HDL project
- Mux address: 585
- Extra docs
- Clock: 0 Hz

How it works

This is a bitserial implementation of the SIMON Block Cipher. SIMON is a 128-bit block cipher, see The SIMON and SPECK families of Lightweight Block Ciphers. A bit-serial implementation exchanges throughput for area, thereby creating a compact cipher that is dominated by flip-flops and multiplexer cells. However, the overal design size becomes minimal. A detailed description of the bitserial implementation technique for SIMON is available in SIMON Says, Break the Area Records for Symmetric Key Block Ciphers on FPGAs .

Cell	Count
flip-flop	281
mux	588
other logic	199
TOTAL	1068

The design uses a 3-bit input and a 2-bit output, in addition to clock and reset.

Port	Function
ui[0]	Bitserial Data Input
ui[7:6]	Control Word
uo[0]	Bitserial Data Output
uo[7]	Data Output Valid

The data input is asserted by the control word, and must be valid when the control word indicates a plaintext-loading or key-loading operation.

The data output is asserted by the valid bit, and should be ignored when the data valid bit is 0. The output ciphertext is produced in 128 consecutive clock cycles.

The 2-bit control word defines the operation of the cipher. The LSB is a debug bit study to key-loading process and to verify that the key register was correctly loaded.

Control	Function
00	Idle
01	Load 128-bit plaintext
10	Load 128-bit key (see LIMITATIONS)
11	Encrypt and return ciphertext

LIMITATIONS

This design forces the key bits to 0 upon loading, so that the effective key value of the cipher is always hardcoded to 00000000_00000000_00000000_00000000. This disables the use of the design as a cipher, yet it still demonstrates how a bit-serial architecture can be designed.

How to test

Study the testbench for example test vectors.

External hardware

No external hardware is needed for this project.

Pinout

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

8-Bit Calculator [586]

- Author: Randy Zhu
- Description: ChipCraft Page 157 Lab ID: C-EQUALS
- GitHub repository
- HDL project
- Mux address: 586
- Extra docs
- Clock: 0 Hz

How it works

8-Bit Calculator from ChipCraft Lab ID: C-EQUALS

How to test

Tested with Makerchip simulation.

External hardware

None.

Pinout

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

DemoSiine [587]

- Author: SagarDevAchar
- Description: A Wavy and Rainbowy TT08 Demoscene Submission
- GitHub repository
- HDL project
- Mux address: 587
- Extra docs
- Clock: 25000000 Hz

How it works

The project structure is as shown below:

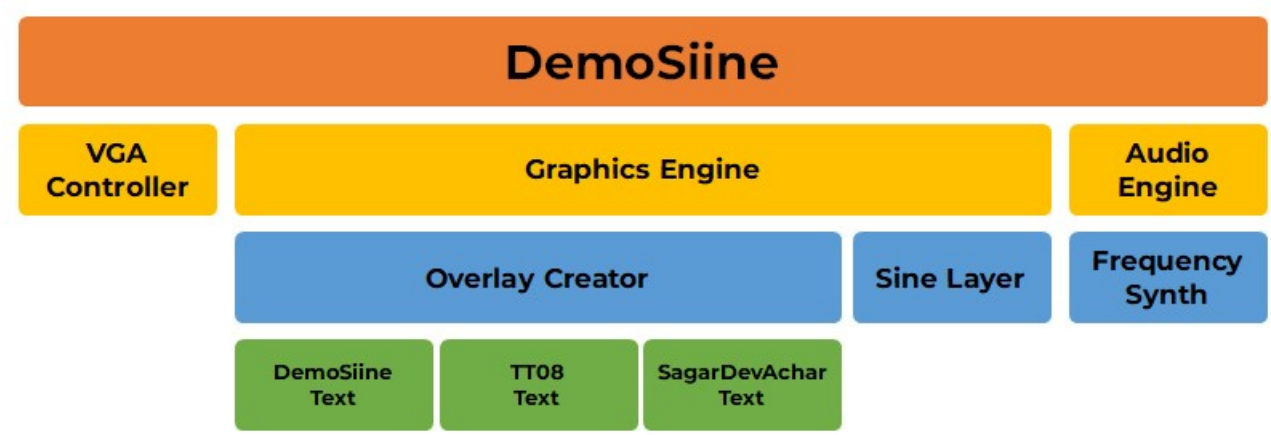


Figure 34: DemoSiine Project Structure

The **Graphics Engine** (driven by the **VGA Controller**, 640x480 @ 60Hz) is an on-demand RGB display pixel generator whose output can be altered using a few input pins. Previews of the different possible display outputs are provided in the last section of this documentation.

The **Audio Engine** drives the **Frequency Synth** to produce a ~28 second looping sound track @ 140 BPM at the output.

External hardware

- Leo's TinyVGA Pmod connected to OUTPUT terminal (`uo_out`)
- Mike's TT Audio Pmod connected to BIDIR terminal (`uio_out`)
- Some switches to the INPUT terminal (`ui_in`)

How to test

- Connect the necessary peripherals
- Provide a 25MHz clock to the top module `tt_um_demosiine_sda`
- Reset the design (if necessary)
- Enjoy the show :)
- Tweak the inputs to customize your show!

Input Configurations

The design takes in 8 digital inputs from the INPUT terminal to modify the on-screen graphics (and audio) to create funky visual effects. All inputs are expected to be LOW to render the output as shown in the default preview as shown below.

The effect of each input pin is presented in the table below:

Input Pin	Parameter	When LOW	When HIGH
`ui_in[7]`	Audio State	Play	Pause
`ui_in[6]`	Animation State	Run	Stop
`ui_in[5]`	Background Style	Black	Rolling RGB
`ui_in[4]`	Overlay Style	Cycle RGB	Rolling RGB
`ui_in[3]`	Overlay State	Enabled	Disabled
`ui_in[2]`	Big Sine State	Enabled	Disabled
`ui_in[1]`	Little Sine State	Enabled	Disabled
`ui_in[0]`	Colour Inversion	Normal	Negative

Previews

Provided below are a some of my favourite previews generated from DemoSiine along with the INPUT configuration which generated them:

Pinout

#	Input	Output	Bidirectional
0	Frame Positive / Negative	Video Red MSB	
1	Enable / Disable Little Sine Layer	Video Green MSB	
2	Enable / Disable Big Sine Layer	Video Blue MSB	
3	Enable / Disable Overlay	Video V-Sync	
4	Toggle Overlay Style	Video Red LSB	

#	Input	Output	Bidirectional
5	Toggle Background Style	Video Green LSB	
6	Run / Stop Animation	Video Blue LSB	
7	Play / Pause Audio	Video H-Sync	Audio Output

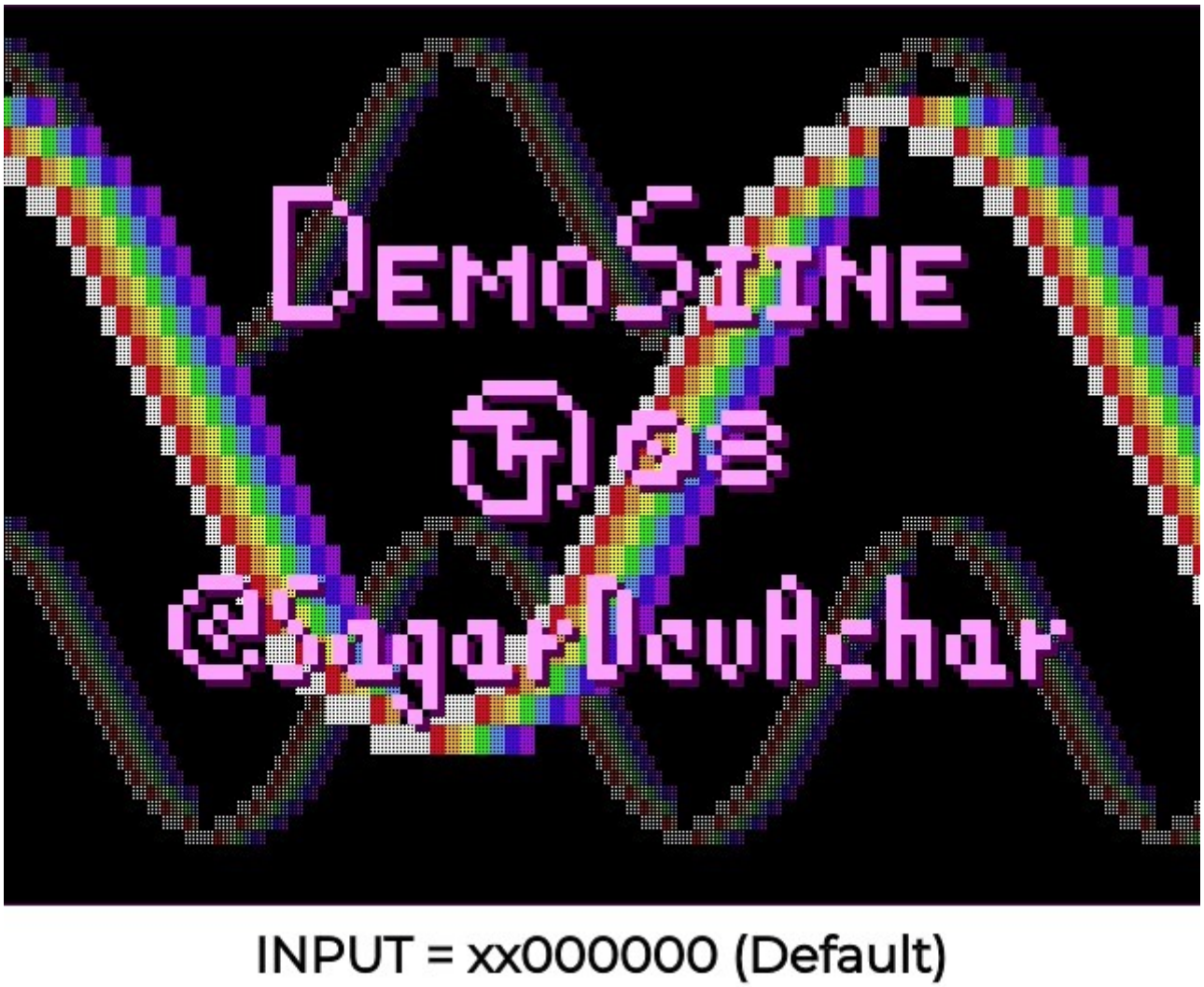


Figure 35: DemoSiine Default Video Output Preview

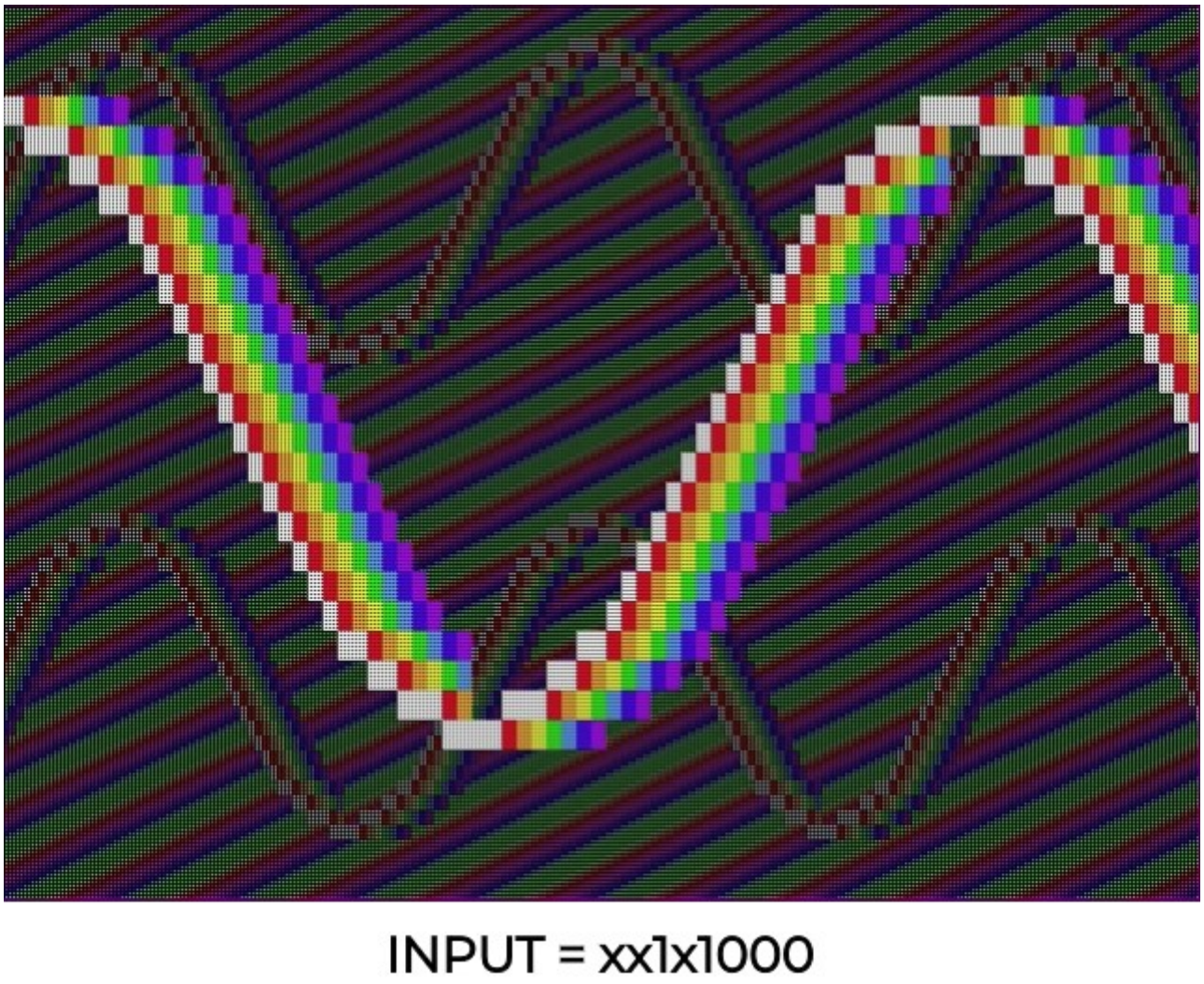


Figure 36: DemoSiine Video Output Preview 2

Figure 37: DemoSiine Video Output Preview 3

Figure 38: DemoSiine Video Output Preview 4

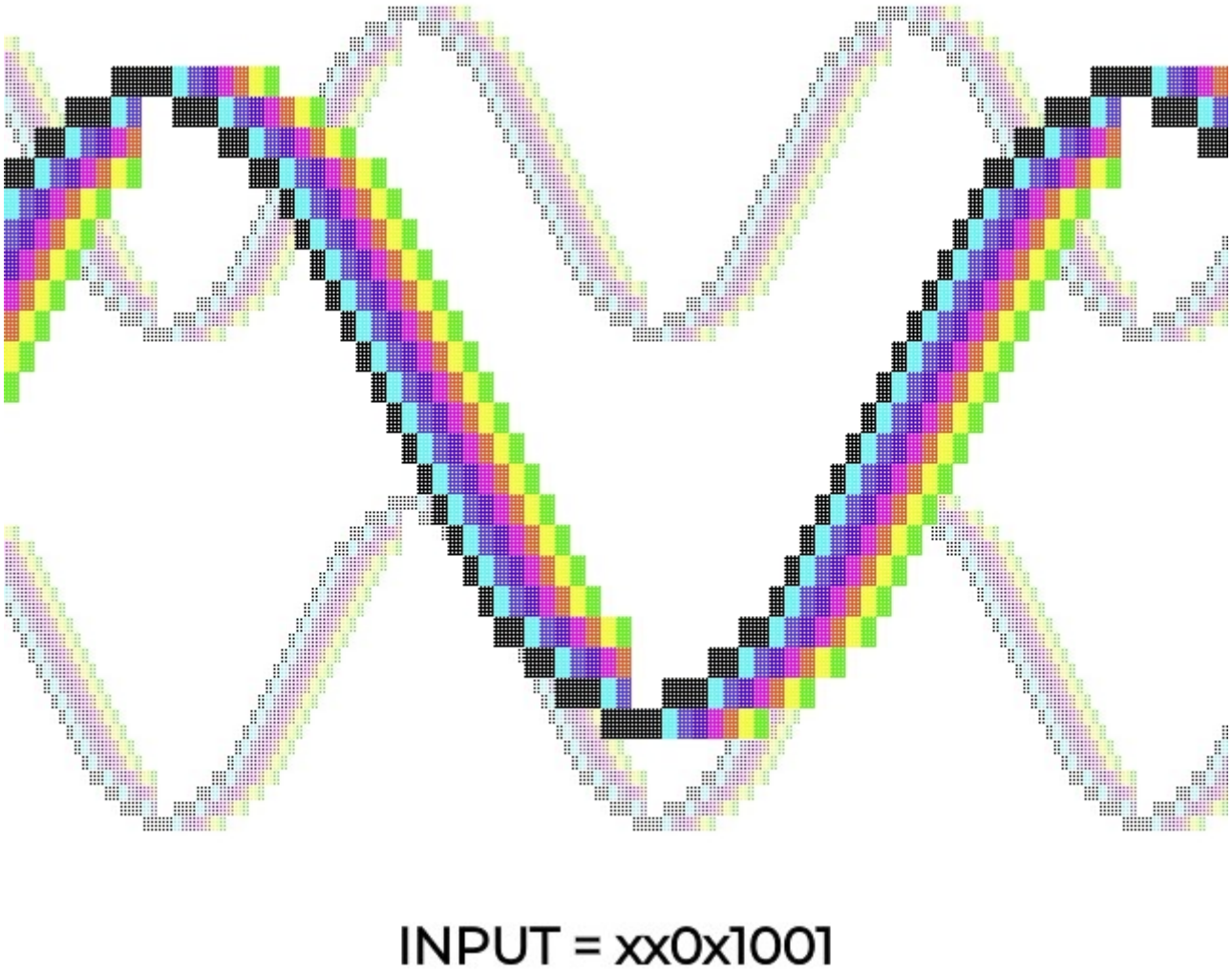


Figure 39: DemoSiine Video Output Preview 5

Figure 40: DemoSiine Video Output Preview 6

HACK CPU [588]

- Author: Dantong LUO, Nour MHANNA, Charbel SAAD
- Description: A 16-bit CPU based on the HACK architecture
- GitHub repository
- HDL project
- Mux address: 588
- Extra docs
- Clock: 12500000 Hz

How it works

The device we developed is a 16-bit CPU based on the HACK architecture. The figure below shows the detailed architecture.

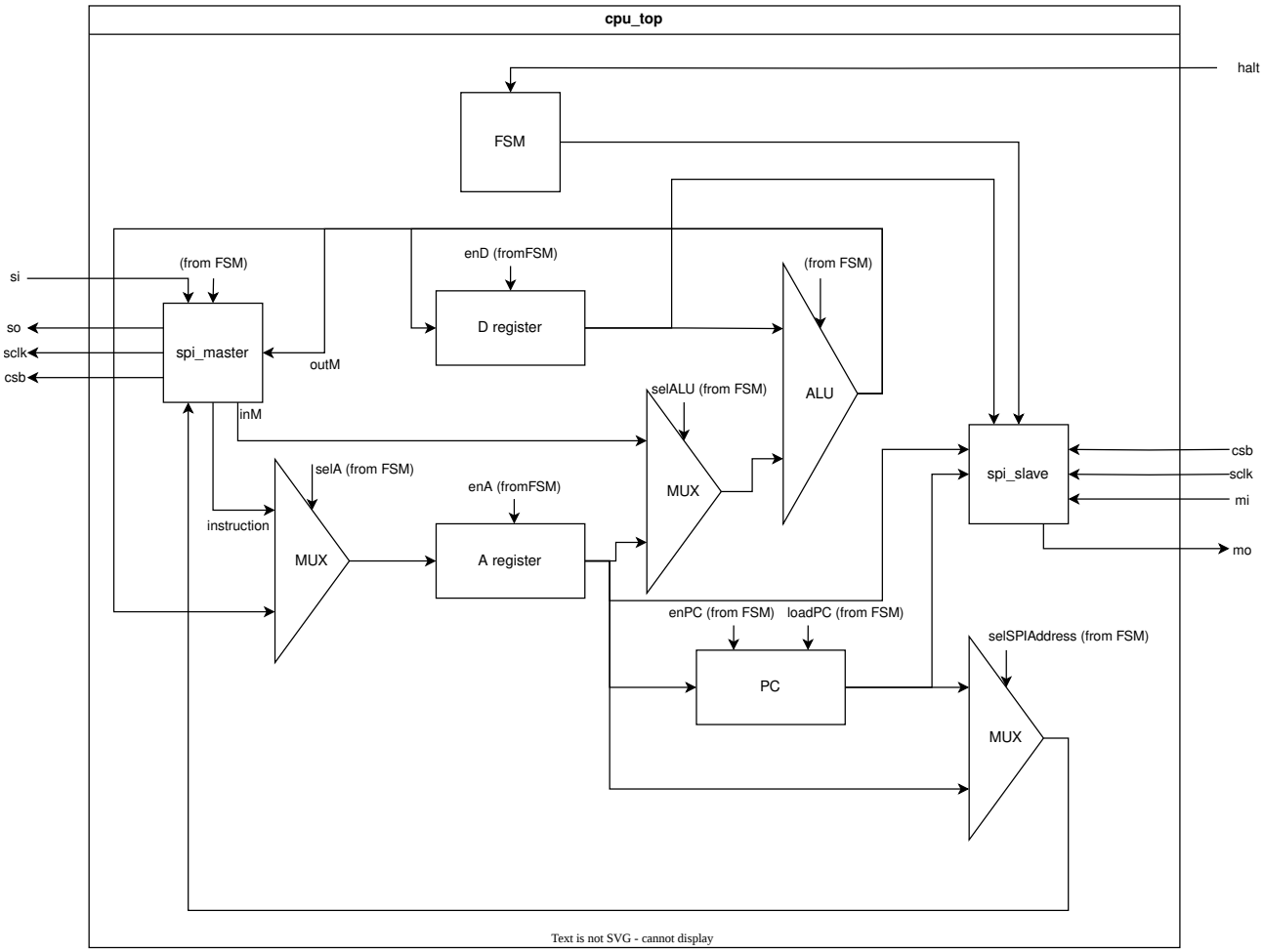


Figure 41: The cpu_top module content

As we can see, it contains three main registers, an ALU, and two SPI modules. Each register has a unique function.

- D register stands as an accumulator.
- A register plays two roles. It first serves as an address register and also as a direct access register.
- PC is the program counter.

The ALU takes two different operands and is driven by 6 control signals, resulting in 18 different operations possible. Control signals can turn an operand to zero, logically reverse it, etc. The figure below shows how it is built.

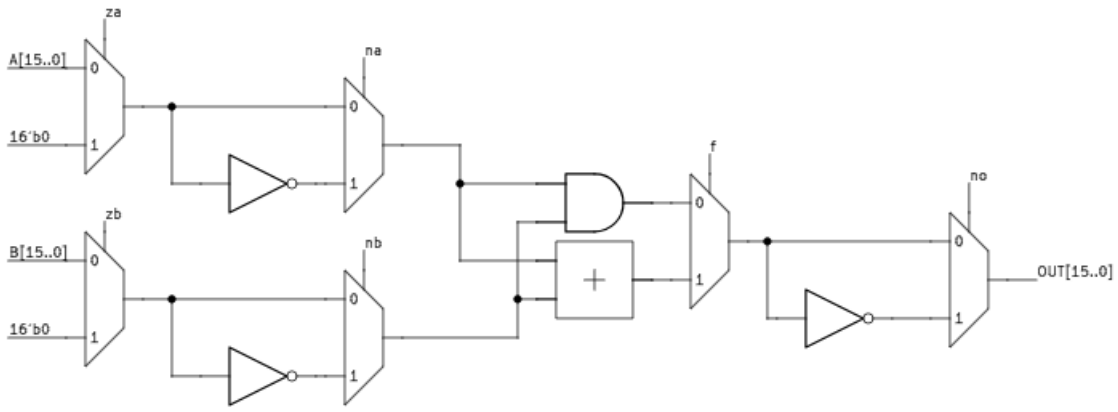


Figure 42: The ALU module

Since we don't have enough space on the chip, we can't include the memory. Moreover, we cannot fetch the 16-bit long instruction and data memory values at the same time because we only have 24 I/O pins. This is why we had to think of another approach. The idea we came up with is to fetch or save a 16-bit word once at a time and use a serial protocol for the transfer. We can see below the state diagram of the CPU.

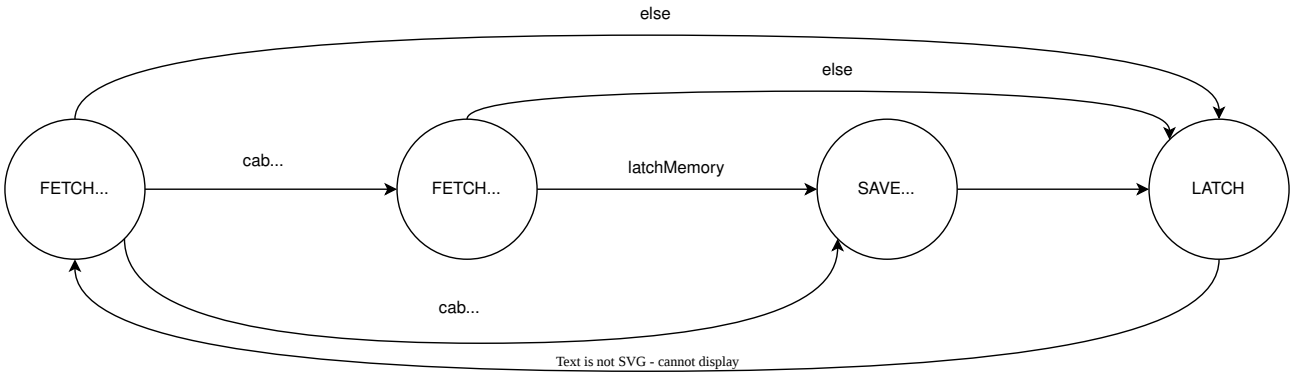


Figure 43: The finite state machine

For the serial communication protocol, we chose SPI since it is one of the simplest to implement. We have to take into account 4 signals:

- MOSI: The signal containing the data transferred from the CPU to the memory.

- MISO: The signal transferring the data from the memory to the CPU.
- CSB: The signal that tells the memory that the CPU needs it.
- SCLK: The clock signal that cadences the transfer.

As shown in the figure below, SPI comes in 4 different modes. We are only going to work with modes 0 and 3 (Flip at the negative edge of the clock then sample on the positive edge).

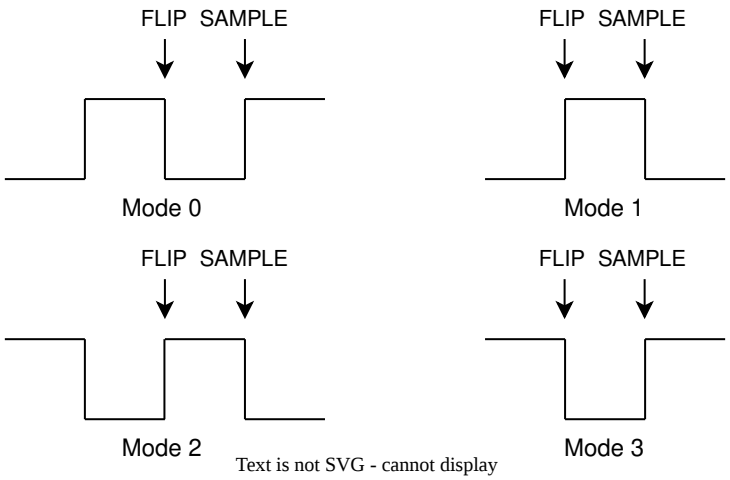
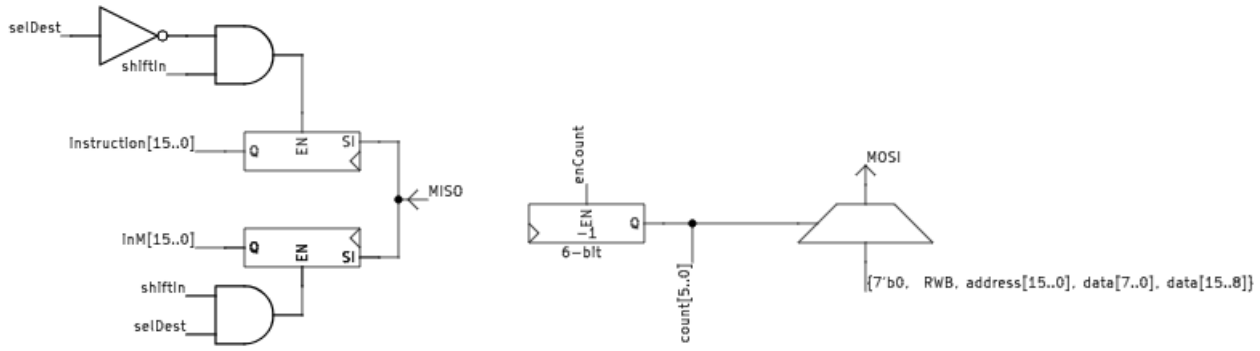


Figure 44: The different SPI modes

The two following figures contain the logic circuits handling the transfer signals. As we can see, the MISO signal is latched in two different shift registers: one for the instruction, the other for the memory data. The MOSI signal is generated via a 40 to 1 multiplexer driven by a counter. The SPI module is monitored by its own FSM.



The final module also processes SPI signals but is used for debugging purposes. This time the SCLK and CSB are driven by the debugging device and the MISO and MOSI signals are inverted. The figure below shows how the module is built.

To communicate, the debugging device sends two bits of data, and depending on these bits, the CPU will output a specific value:

- 0: register D

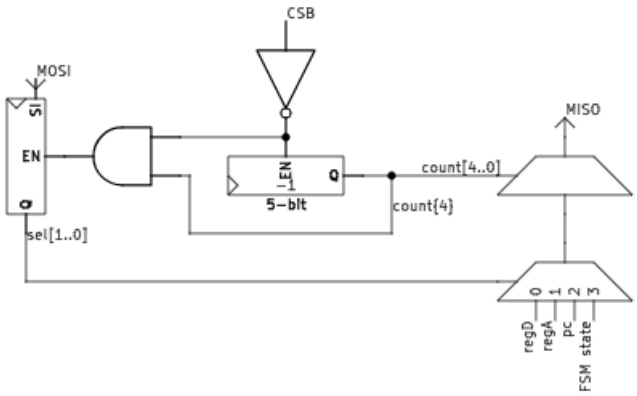


Figure 45: The debuging module

- 1: register A
- 2: Program Counter
- 3: FSM state

How to test

The chip needs to be connected to an SPI RAM. We focused the design around the 23XX512, an SPI RAM developed by Microchip Inc. Just add the binary code to the RAM and provide an adequate 12.5 MHz clock signal. For debugging, a microcontroller with an SPI interface can do the job.

Since the chip is going to be soldered to a debug board with an RP2040 on it, we can use the code provided by MichaelBell to emulate the RAM (Github repsitory). The RP2040 can also be used as a debugger.

External hardware

- 65 KB SPI RAM.
- Microcontroller with SPI interface.

Pinout

#	Input	Output	Bidirectional
0	external halt signal (to use when debugging)		GPIO21 - RAM CS
1			GPIO22 - RAM MOSI
2			GPIO23 - RAM MISO
3			GPIO24 - RAM SCK

#	Input	Output	Bidirectional
4			DEBUG CS
5			DEBUG MOSI
6			DEBUG MISO
7			DEBUG SCK

Munch [589]

- Author: bytex64
- Description: Displays munching squares through VGA PMOD
- GitHub repository
- HDL project
- Mux address: 589
- Extra docs
- Clock: 25175000 Hz

How it works

This generates VGA output for a munching squares animation plus some other stuff, and some simple music. It uses the VGA and audio PMODs listed below.

Clock generation A clock generator module divides the main clock `clk` and `vsync` into two derived clocks - a 393kHz PWM clock for audio output, and a 10Hz "audio tick" clock that drives the pattern sequencer. The clock generator also provides counters for PWM, volume modulation, and audio pattern sequencing.

LFSR An 11-bit LFSR provides a noise source. It is an XNOR type, shifting down towards the LSB and inserting the new bit at MSB. XNOR taps are on bits 0 and 2. Bits 0-5 of the LFSR register are used to provide a noise channel and randomized video noise dithering.

Video The video output is the standard 640x480 @ 60Hz, using a 25.175MHz pixel clock and negative polarity HSYNC/VSYNC. Timing is implemented with a simple two-counter design shamelessly stolen from the Tiny Tapeout VGA playground.

A fixed palette of eight colors is used, and eight brightness levels are created by mixing random bits with the 2-bit per channel brightness levels. Video is output on three layers - the background and layers 0 and 1. A non-black pixel overrides a pixel on any lower layer.

Audio Audio comes from a basic PSG inspired by the SN76489. There are four channels of sound based on 12-bit timers - three pulse channels and one noise channel. The only real difference between a pulse channel and a noise channel is that the pulse channel flips state when the timer counts down, and the noise channel takes a random state from the LFSR. Each channel also has a two-bit volume level.

The 25.175MHz clock is divided by a 6-bit counter to create a 393KHz PWM output. 6-bits gives 64 possible levels. The PWM high period is a simple sum of the four channels' volumes at any given instant (multiplied by two with the low bit dithered from the LFSR). This does mean the PWM will glitch if volume levels change in the middle of a PWM cycle, but that's fine in practice since it's all low-pass filtered anyway.

The four channels are programmed through a sequencer that provides note and volume data to the PSG. The sequencer is clocked by dividing VSYNC by 6, so the sequencer moves through pattern rows at 10Hz, or 600 ticks per minute. Each pattern of 16 ticks represents one measure, four beats, which means the music proceeds at 150 BPM.

The sequencer cycles through pre-programmed patterns of notes. Note timer data is read from an indirected list of notes, then connected directly to the PSG reload values. This does mean the oscillators are not synchronized at note start. Volumes are modulated through a single repeating pattern per channel, indexed from the top two bits of the sequencer div-by-6 clock divider. This means the volume is a three-step pattern cycling at the start of each pattern tick.

Text Generator On-screen text uses a segmented approach, where each segment is defined by a mathematical description of a line segment. Each character is then defined by which segments are off or on, like a multi-segment LED display. So text is generated at full resolution despite its large size; each character is 50x100 pixels.

The text generator is just a sequencer over an input bit stream, indexed by the horizonal and vertical position. In this implementation the input is at most six characters long. The text can be positioned arbitrarily, but for this demo it is fixed.

Stage Sequencer A slower stage clock is derived from the pattern clock. It ticks once every pattern cycle, and drives an overarching "stage sequencer". Each stage counts down for a pre-programmed number of patterns, then switches to the next. The stage number is used in various logic to change the text and colors over time.

Extra outputs In addition to the audio and video, the three highest bits of the internal pattern counter are output on `uio_out[6:4]`. The two highest bits count out the four beats in a pattern, and bit 1 has a negative edge at the beginning of each beat. This could be used for beat synchronization with external systems - I just used it for debugging.

How to test

Set the input clock for 25.175MHz. The Pico/RP2040 can output 25.177MHz on GPOUT0 with a 125MHz main clock and a divider of 4 [integer part] and 247 [fractional part]. This worked on my TV.

Reset, and enjoy. :)

External hardware

- Leo's VGA PMOD
- Tiny Tapeout Audio Pmod

Pinout

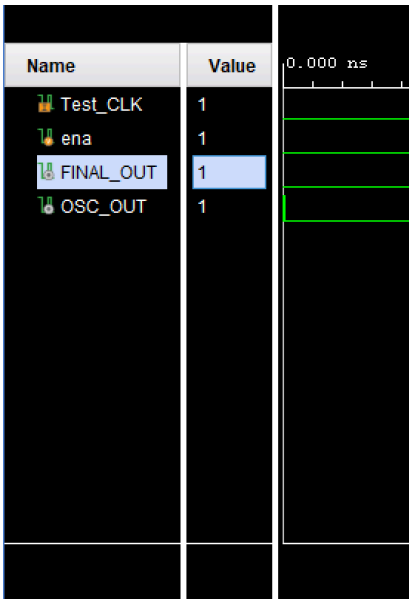
#	Input	Output	Bidirectional
0		R1	
1		G1	
2		B1	
3		VSYNC	
4		R0	beat clock bit 1 (output)
5		G0	beat clock bit 2 (output)
6		B0	beat clock bit 3 (output)
7		HSYNC	audio (output)

Divided Ring Oscillator [590]

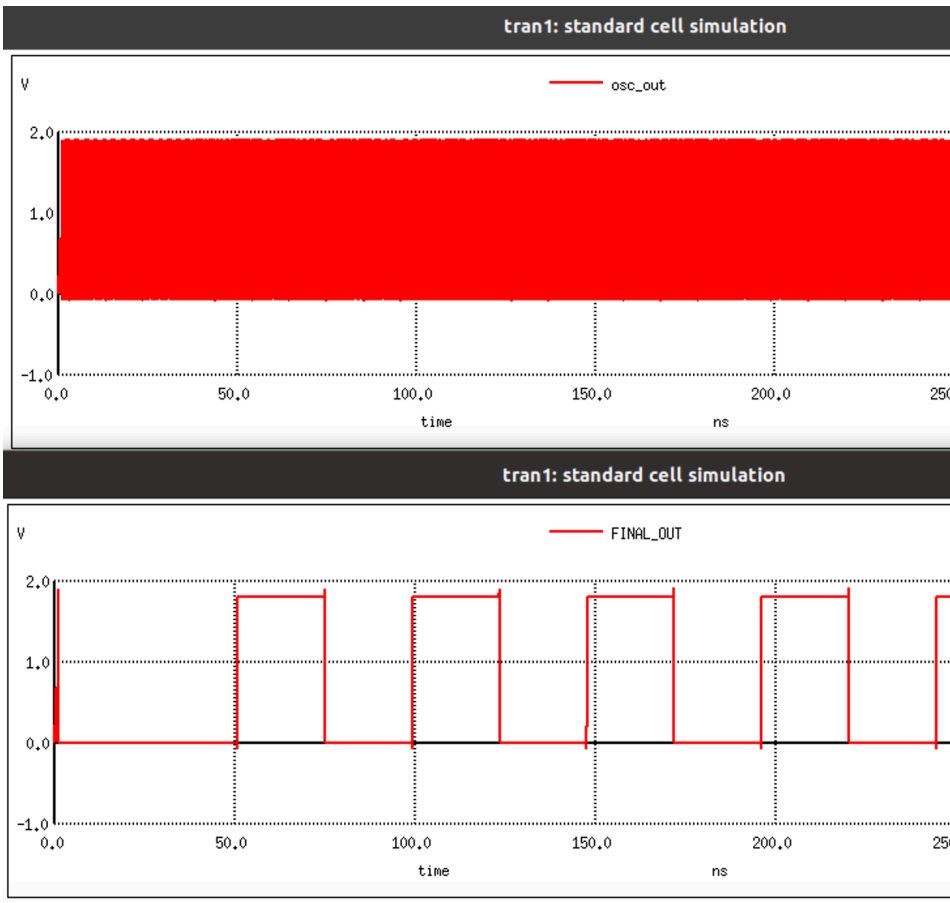
- Author: Ignatius Bezzam, Dhandeep Challagundla, Jarnail Sanghera, Russell Kim
- Description: Ring Oscillator
- GitHub repository
- HDL project
- Mux address: 590
- Extra docs
- Clock: 10000000 Hz

How it works

A ring oscillator working in the GHz range is divided to give an observable output frequency in the 20 MHz range.



Top-Level Complete Mixed-Signal Functionality Verification in Verilog



PEX Sims Verifying Performance

How to test

A supply current of 1-2 mA when enable is high indicates that the ring oscillator is functional. The final output can be observed in the 20 MHz range. Test/debug mode verifies the divider functionality at low frequency. The ring oscillator can be disabled by on-chip signals (ena = low).

External hardware

Oscilloscope (100 MHz), power supply, function generator (10 MHz, digital).

Pinout

#	Input	Output	Bidirectional
0	tst_clk	final_out	n1
1		osc_out	n3
2		ena	
3		clk	

#	Input	Output	Bidirectional
4		rst_n	
5		n2_buf	
6		n4_buf	
7			

cfib Demoscene Entry [591]

- Author: Christian Fibich
- Description: Generates VGA video and PWM audio
- GitHub repository
- HDL project
- Mux address: 591
- Extra docs
- Clock: 50000000 Hz

How it works

My entry to the Tinytapeout Demoscene Competition.

It (pseudo-randomly) generates a soundtrack via PWM and displays a waveform via VGA.

How to test

Connect VGA and PWM Pmod.

Then just apply clock and (asynchronous) reset.

External hardware

The project uses:

- Tiny VGA Pmod via `uo_out[7:0]` (https://github.com/mole99/tiny-vga)
- Mike's audio Pmod via `uio_out[7]` (https://github.com/MichaelBell/tt-audio-pmod)

Pinout

#	Input	Output	Bidirectional
0		r[1]	
1		g[1]	
2		b[1]	
3		vsync	
4		r[0]	

#	Input	Output	Bidirectional
5		g[0]	
6		b[0]	
7		hsync	pwm

PDM Correlator [640]

- Author: Armaan Gomes
- Description: A chip that performs either cross or auto correlatiion on PDM microphone inputs
- GitHub repository
- HDL project
- Mux address: 640
- Extra docs
- Clock: 0 Hz

How it works

It performs an XOR on two input bitstreams and sums the result. The lower this value is the higher correlation. Explain how your project works

How to test

Connect microphones to pins and stuff Explain how to use your project

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	Bit Clock (3.072 Mhz)	PCM Out Mic 0	Delay Select 0
1	LR Clock (48Khz)	PCM Out Mic 1	Delay Select 1
2	PDM Input Mics 0,1	PCM Out Mic 2	Delay Select 2
3	PDM Input Mics 2,3	PCM Out Mic 3	Delay Select 3
4	PDM Input Mics 4,5	PCM Out Mic 4	Delay Select 4
5	PDM Input Mics 6,7	PCM Out Mic 5	Beamformed PCM Output
6		PCM Out Mic 6	Mic Clock
7		PCM Out Mic 7	

PDM Pitch Filter [642]

- Author: Armaan Gomes
- Description: It uses a moving average filter and decimator to filter out a specific frequency
- GitHub repository
- HDL project
- Mux address: 642
- Extra docs
- Clock: 0 Hz

How it works

Explain how your project work This project pitch filters a microphone input stream. Because the bitstream is pdm (1 or -1 at 3.072 Mhz) a sine wave of certain frequencies has a certain length at which its average energy is 0. By making a moving average filter of that length we can eliminate that frequency and its harmonics

How to test

Connect a microphone to the pin and use the spi port to se thte decimator and filter length . Inprogress Explain how to use your project

External hardware

A pdm microphone spi input and clock generator List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Pinout

#	Input	Output	Bidirectional
0	Bit Clock (3.072 Mhz)	PCM Out Mic 0	Delay Select 0
1	LR Clock (48Khz)	PCM Out Mic 1	Delay Select 1
2	PDM Input Mics 0,1	PCM Out Mic 2	Delay Select 2
3	PDM Input Mics 2,3	PCM Out Mic 3	Delay Select 3
4	PDM Input Mics 4,5	PCM Out Mic 4	Delay Select 4
5	PDM Input Mics 6,7	PCM Out Mic 5	Beamformed PCM Output
6		PCM Out Mic 6	Mic Clock

#	Input	Output	Bidirectional
7		PCM Out Mic 7	

16 Mic Beamformer [644]

- Author: Armaan Gomes
- Description: A 0 delays fixed delay and sum beamformer that can utilize up to 16 input microphones
- GitHub repository
- HDL project
- Mux address: 644
- Extra docs
- Clock: 0 Hz

How it works

It does stuff (testing) Explain how your project works

How to test

You can test it (testing) Explain how to use your project

External hardware

You need 16 digital microphones, a clock generator (can be a raspberry pi, microcontroller, etc.), and something that recieves the I2S output (this can be a rapberry pi or most auido output devices). List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Pinout

#	Input	Output	Bidirectional
0	PDM Input Mics 0,1	I2S Out	Bit Clock (3.072 MHz)
1	PDM Input Mics 2,3		LR Clock (48kHz)
2	PDM Input Mics 4,5		
3	PDM Input Mics 6,7		
4	PDM Input Mics 8,9		
5	PDM Input Mics 10,11		
6	PDM Input Mics 12,13		
7	PDM Input Mics 14,15		

VGA Nyan Cat [646]

- Author: Andy Sloane
- Description: Displays the classic nyan.cat animation
- GitHub repository
- HDL project
- Mux address: 646
- Extra docs
- Clock: 25175000 Hz

VGA nyan cat

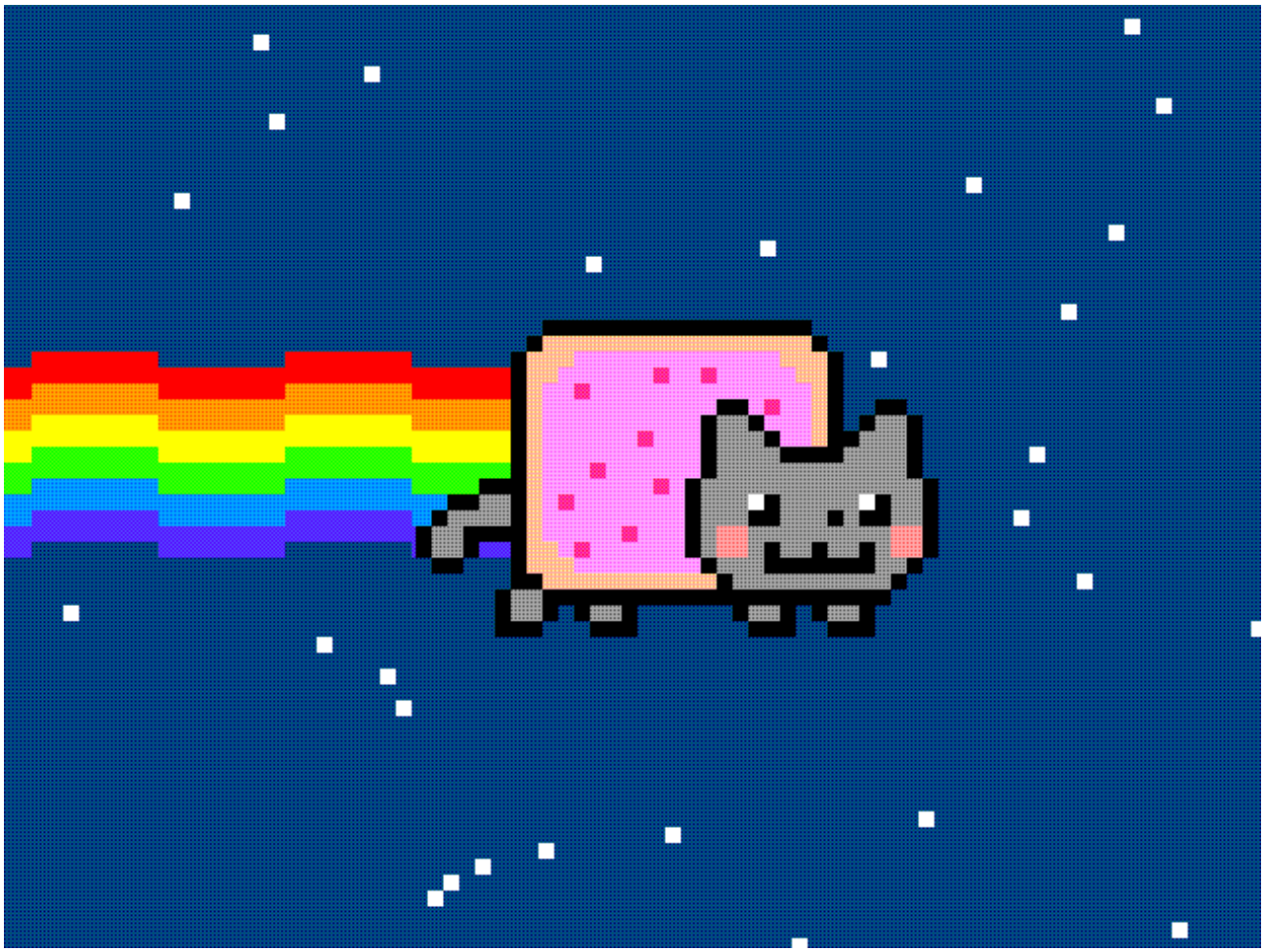


Figure 46: nyancat preview

How it works Outputs nyancat on VGA with music!

Colors and animation are all from the original nyan.cat site, using a 2x2 Bayer dithering matrix which inverts on alternate frames for better color rendition on the Tiny VGA Pmod.

Sound is generated from a MIDI file, split into melody and bass parts. Melody and bass are each square waves mixed with a simple exponential decay envelope, which is then fed to a low-pass filter and then a sigma-delta DAC.

This was designed to fit into 1 tile, and it *almost* did – the cells take up about 93% of 1 tile, but detailed routing doesn't finish. With the deadline approaching I was forced to grow it to 1x2, so I threw in a little easter egg.

How to test Set clock to 25.175MHz or thereabouts, give reset pulse, and enjoy

External hardware 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

Warp [648]

- Author: sylefeb
- Description: Demo on TinyTapeout? Let's do something!
- GitHub repository
- HDL project
- Mux address: 648
- Extra docs
- Clock: 25000000 Hz

Warp

> Please make sure to watch the demo for a few minutes as various effects play out before it loops. At start it waits for a few seconds to ensure VGA sync is achieved.

How it works

Preface This demo is written in Silice, my HDL. Here is the actual source. Silice now fully support TinyTapeout as a build target.

Graphics The core effect is a classical tunnel effect ; however this is normally done with a "huge" pre-computed table having one entry per-pixel. So I thought it'd be challenging and fun to do it while racing the beam! Plus, I really like this effect.

There are several tricks at play: a shallow CORDIC pipeline to compute an *atan* and *length*, and a few precomputed $1/x$ distances to interpolate between – these form keypoint rings along the tunnel. All the effects are then obtained by combining multiple layers in various ways (like a *tunnel effect processor* which registers can be configured for various effects).

The demos uses a lot of dithering (ordered Bayer dithering) given the output is RGB 2-2-2. All computations are grayscale and the RGB lense effect is obtained by delaying the grayscale values using the tunnel distance in R and B.

I also tried to make the logo interesting by deviating from a classical pixelated look. It is composed of tiles, either full or triangular, with a comparator and a bit of logic to do all four possible triangles.

The tunnel viewpoint change is obtained simply by shifting the tunnel center. I was surprised that a simple translation gives such a convincing effect (almost as if the viewpoint was rotating).

The 'blue-orange' tunnel effect is obtained through temporal dithering, one frame being the standard tunnel, the other the rotated tunnel. This gets combined with the RGB lense distortion, achieving the final look.

Audio I am no musician, so making a soundtrack was a challenge for me, but that's something I've always wanted to try. In the end it was a very enjoyable part of the design, and I was surprised at how compact this can be made, the soundtrack using perhaps around 10% of the entire design.

I tried to make a track that matches the spirit and rhythm of the graphics. It is what is is, but I'm happy that there's sound at all!

How to test Plug the VGA+audio PMODs to the board and run. Maybe it works?

Simulation of both audio and video can run on an ECPIX5, with the Diligent VGA PMOD on ports 0,1 and an I2S audio PMOD on port 2 (upper row). The audio also runs on an ULX3S using its DAC (but no video in this case).

External hardware

- VGA PMOD
- Audio PMOD

See https://tinytapeout.com/competitions/demoscene/

Pinout

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

#	Input	Output	Bidirectional
7		HS	Audio

Oscillating Bones [649]

- Author: Uri Shaked
- Description: A stylish ring oscillator built from SkullFET transistors
- GitHub repository
- HDL project
- Mux address: 649
- Extra docs
- Clock: 0 Hz

How it works

A simple yet stylish ring oscillator that uses a chain of 21 SkullFET inverters to generate a square wave output. Based on simulation, the oscillator should have a frequency of around 90 MHz.

How to test

Connect an oscilloscope to the `osc_out` (ou_out pin 0) pin and enjoy the show. You can also observe the divided frequency outputs on `osc_div_2`, `osc_div_4`, and `osc_div_8`.

Simulation results

The following graph shows the output of the oscillator and the divided outputs. It was generated by running `make -C sim` and patiently waiting for the simulation to finish:

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

Note that the simulation results do not include all the parasitics, only the main ones. The actual frequency of the oscillator will probably be lower than the simulated one.

#	Input	Output	Bidirectional

Pinout

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

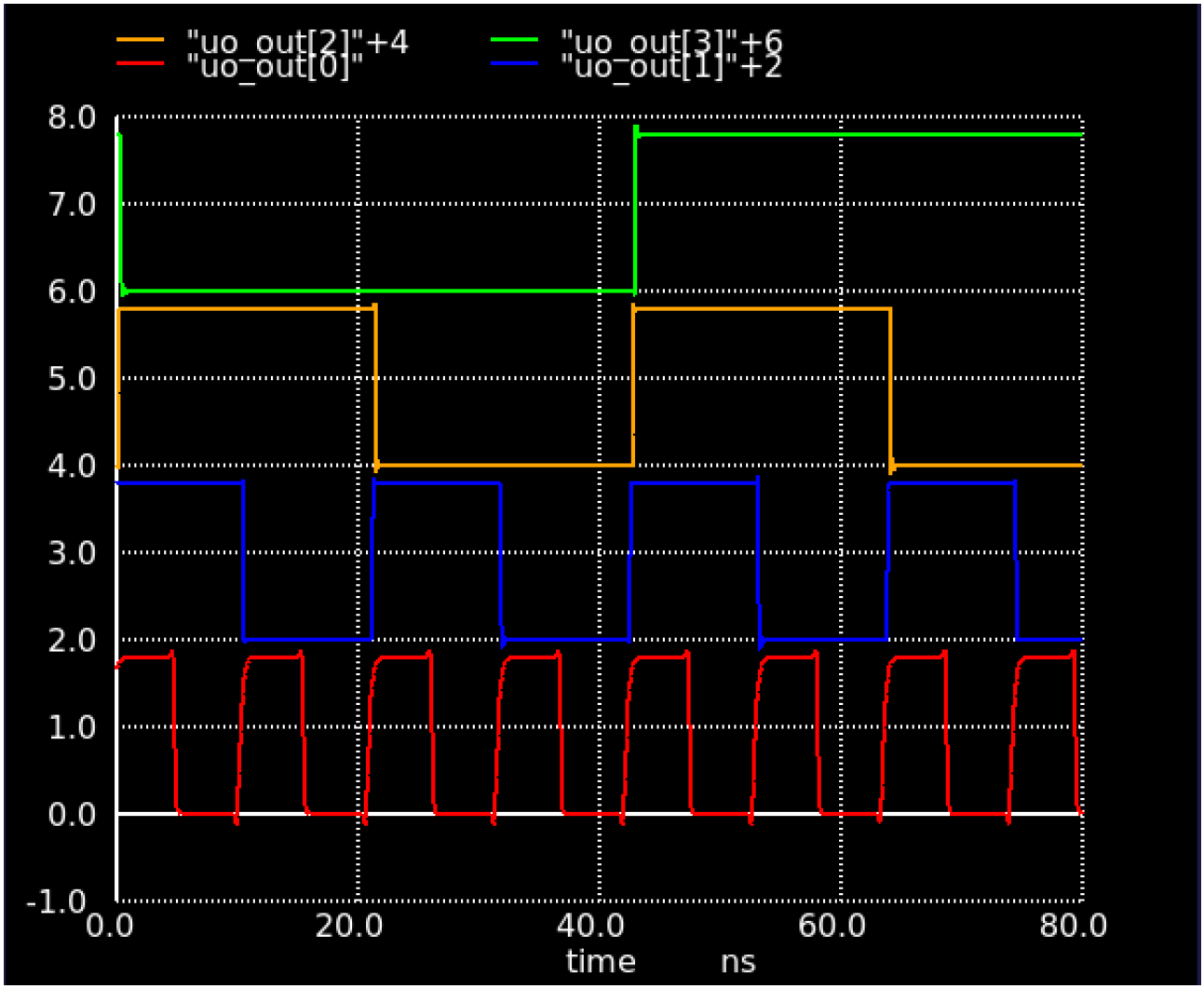


Figure 47: Simulation results

VGA Drop (audio/visual demo) [650]

- Author: ReJ aka Renaldas Zioma, eriQue aka Erik Hemming, Matthias Kampa
- Description: Tiny 8 part Megademo! TBL[Nesnausk]SonikClique
- GitHub repository
- HDL project
- Mux address: 650
- Extra docs
- Clock: 25200000 Hz

How it works

VGA signal generator

How to test

We are learning how VGA and Sky130 works here

External hardware

VGA PMOD

Pinout

#	Input	Output	Bidirectional
0		R1	Audio (PWM)
1		G1	Audio (PWM)
2		B1	Audio (PWM)
3		VSYNC	Audio (PWM)
4		R0	Audio (PWM)
5		G0	Audio (PWM)
6		B0	Audio (PWM)
7		HSYNC	Audio (PWM)

Comm_IC [652]

- Author: Bhavuk
- Description: Communication protcols: UART, SPI, I2C
- GitHub repository
- HDL project
- Mux address: 652
- Extra docs
- Clock: 20000000000 Hz

How it works

Top module for the Comm_IC project. Submitted for the TinyTapeout8 (TT8).
Designed by: Bhavuk
Github ID: Bhavuk-HDL
Date of creation: 04-Sept-2024
Code version: V01

This project combines three different communication protocols, namely:

1. UART: Universal Aynchronous Receiver Transmitter
2. SPI: Serial Peripheral Interface
3. I2C: Inter Integrated Circuit

To communicate with this project, there is 'data_en' signal.
data_en should be low by default. When it gets high e receive 4 bit data from data_in (MSB first) based on the clk rising edge.

First 4-bits of data bits will decide the comm. protocol and readwrite.
data_in = 4'bab_cd:
ab: 00-> Read
ab: 11-> Write
cd: 00-> UART
cd: 01-> SPI
cd: 10-> I2C
ab: 10-> Use previous settings: valid only in 'write mode'.

Second 4-bits will have two directions: 'read mode' or 'write mode'.
Read mode: data will be read from the comm protocol and interrupt will be set to '0'.
Write mode: if cd was set to '11' in the last cycle, we use previous settings for the comunication. Otherwise we use fresh settings.
Next few 4-bit sequences will be used to send the data to resp. module.

How to test

Refer to the test_bench folder in src for test cases.

External hardware

Not applicable

Pinout

#	Input	Output	Bidirectional
0	UART_RX	UART_TX	SDA_out
1	MISO	SEN	new_uart
2	data_en	SCLK	data_out[0]
3		MOSI	data_out[1]
4		SCL	data_out[2]
5		busy_uart	data_out[3]
6		busy_spi	error_i2c
7		busy_i2c	

Sea Battle [654]

- Author: Yuri Panchul
- Description: Sea Battle is a VGA game with sprites for the Tiny Tapeout Demoscene competition.
- GitHub repository
- HDL project
- Mux address: 654
- Extra docs
- Clock: 23000000 Hz

How it works

Sea Battle is a VGA game with sprites for the Tiny Tapeout Demoscene competition.

The *Sea Battle* design is used as a part of basics-graphics-music GitHub repository of Verilog examples, which is maintained by the Verilog Meetup community.

The game uses two keys, *left* and *right*, to control a torpedo. Pressing any key starts the movement. The goal is to hit the moving target.

The design is supposed to work on a 23 MHz frequency and connect to a VGA display using a Tiny VGA board with 2 bits per color channel.

How to test

The design was tested on several FPGA boards and has no self-checking Verilog testbench for simulation. We just hope it is going to work on ASIC silicon as is.

External hardware

Buttons and a Tiny VGA connector.

Pinout

#	Input	Output	Bidirectional
0	Key right	VGA red [1]	
1	Key left	VGA green [1]	
2		VGA blue [1]	
3		VGA vsync	

#	Input	Output	Bidirectional
4		VGA red [0]	
5		VGA green [0]	
6		VGA blue [0]	
7		VGA hsync	

Bouncy Capsule [704]

- Author: htfab
- Description: Demoscene project featuring... well, a bouncy capsule
- GitHub repository
- HDL project
- Mux address: 704
- Extra docs
- Clock: 25000000 Hz

How it works

This is an entry to the Tiny Tapeout demoscene competition

How to test

- Attach the standard PMODs
- Run the clock at 25 (or 25.175) MHz
- Reset the design
- Sit back and enjoy
- Optionally change the input switches

External hardware

- Tiny VGA PMOD
- TT Audio PMOD (or MuseLab's Audio PMOD)

Pinout

#	Input	Output	Bidirectional
0	Pause kinematics	Tiny VGA R1	PDM audio out
1	Reset kinematics	Tiny VGA G1	PDM audio out
2	Mute sound	Tiny VGA B1	PDM audio out
3	Kill sound	Tiny VGA VSync	PDM audio out
4	Hide background	Ting VGA R0	PDM audio out
5	Hide text	Tiny VGA G0	PDM audio out
6	Lock colors	Tiny VGA B0	PDM audio out
7	No re-orientation	Tiny VGA HSync	PDM audio out

FSK Modem +HDLC +UART (PoC) [706]

- Author: Darryl Miles
- Description: FSK Modem w/ HDLC transciever + UART (PoC digital side)
- GitHub repository
- HDL project
- Mux address: 706
- Extra docs
- Clock: 0 Hz

How it works

This is a proof-of-concept design to sketch out the TT_UM digital interface for a later project design that will attempt to incorporate both analogue and digital aspects of the basic skeleton shown in this project.

The design is based on the classic circa 1988 model design used in Amateur Radio Packet systems by G3RUH. The initial specification is looking to achieve data rates of between 4800 and 64000 baud, but the design maybe able to service audio 1200 baud packet radio as well.

The design is 1-data-bit per symbol.

The original TNC (Terminal Node Controller) was a Z80 CPU and 8530 Serial Communications Controller. So inline with this I expect to provide an 8-bit CPU (as a future TT project) as a companion to this so the two items taken together should be able to form a complete communications solution of a capable TNC. This is an area I spent a significant amount of my teenage youth understanding and experimenting with that gave me a good grounding in all the digital electonic, radio and computer/CPU theory/practice that is still in use today.

The original PCB board design used:

- a x16 master TX CLOCK line of the data rate.
- was based on 12v audio interface/opamps, and 74HC TTL logic
- was capable of the range of baud rates with minor modifications, the most used speed in my experience is 9600 baud
- the TX DAC was 4 x 8-bit samples per bit, with the waveform lookup using a 12bit address that can see previous bit information sent
- EPROM were used directly to provide waveforms, these have a number of jumper set modes to allow compensation for non-linear responses at the TX-AUDIO and RX-AUDIO

Due to the need to perform ROM lookups, this is operating in 4 phases sharing 6-bit output from module, and 4-bit input to module. The 4 phases cover a sequence of:

- TX nibble low (6bit address)
- TX nibble high (6bit address)
- RX nibble low (6bit address)
- RX nibble high (6bit address) It is not clear if this arrangement a good choice. There is also a programmable latency on the reply, of zero-cycles or one-cycle, the shifts the expectation of the result.

I also need to validate the DAC 8bit loading scheme prevents any chirping (visibily to DAC of partially loaded data, due to multiplex timing differences) of the data because it is loaded in 2 halves.

The master clock (CLK pin) due all the above, it is ncessary to run the clock pin at x4 the x16 of the original design.

data rate baud	master clock (CLK)	tx clock	tx sample clock
4,800	307,200	76,800	19,200
9,600	614,400	153,600	38,400
19,200	1,228,800	307,200	76,800
38,400	2,457,600	614,400	153,600
64,000	4,096,000	1,024,000	256,000
76,800	4,915,200	1,228,800	307,200

Table is in Hz or Baud

The master clock (pin CLK) is driven at x64 the synchrnous data rate. The tx clock rate is derrived from this 'CLK divide-by-4'

The UART clocking is also derived from CLK, and each side (uart RX and uart tx) can be individually configured to be 1:1 or 2:1 the synchronous data rate:

- Uart TX x1 = data rate x1
- Uart TX x2 = data rate x2
- Uart RX x1 = data rate x8 (due to majority voter, 8 sample buffer)
- Uart RX x2 = data rate x16 (due to majority voter, 8 sample buffer)

As you can see maybe there is some headroom for faster transmission speeds within a TT project, before needing to increase DAC resolution and explore 4FSK/6FSK/QAM etc...

There are 3 main functional areas with the design:

The type of FSK modem is 2FSK (dual tone) outputting continious wave.

Upper Digital (included here) This incorpotates a full-duplex HDLC frame processor attached to a UART (ttl interface), the UART process encodes the frame in format similar to KISS format used by TNCs, with a few modifications.

Lower Digital (included here) This manages the receiver clock recovery PLL circuit and interface, the original designs used EPROM lookup tables with 12bit address (which has visibility on at least the previous encoded bit) and provides an 8bit data output.

The data outputs are then fed into a respective 8bit DAC

The receiver has a PLL lock detector which is used to provide DCD (Data Carrier Detect) signal. While the hardware design is capable of full-duplex operation it is often used in Amateur Radio situations in a half-duplex situation with a carrier sense channel sharing algorithym.

Lower Analogue (not includes in this PoC design, see next iteration) The parts that are missing from the design:

- 8bit DAC for transmit waveform shaping, using 4 samples per bit
- opamp for transmit audio anti-aliasing (low-pass filter?) circuit to remove harmonic noise from the output audio
- 8bit DAC for receiver clock recovery feedback, using 16 samples per bit.
- opamp for receive audio signal interface, this maybe moved to an external board due to needing to protect the TT IC from over voltage from being attached to usuall 12v equipment or maybe 36v when using some ex-commerial radio trancievers. This may have been a comparator circuit (unsure at this time), fed into a DFF to synchronise the incoming data to the x16 (of datarate) clock recovery timing
- 2 x opamp to provide PLL lock detection (unsure how this works atm), I would guess it can detect when the signal is being centered and has been centered for some number of samples, maybe via slow capacitance charge up when the UP/DOWN line is managing to meet an approximate 50%/50% duty cycle per x16 clock recovery tick.
- 2 x opamp to provide zero-crossing detection, this is used to provide the PLL its feedback mechanism (the UP/DOWN line) to advance or retard the edge alightment.

It is hoped all items can be incorporated into the same design using the analogue GDS facility with TT and connected to the respective lower digital signal.

At this time we bring out the interconnection points (between analogue and lower digital) to the external interface of TT and we provide a configuration mechnism to

be externally or internally driven/internally sourced. This should allow for a significant level of simulation and experimentation by users of the project to understand and explore FSK/PLL theory by picking a testing configuration combintation, being full-duplex it should be able to loop-back at various levels to understand each part better. While also providing those with a Ham Radio license to try out on air communicating with their local users or AMSAT.

Have fun... 73s de G7LED

How to test

When the final design is completed, there should be a number of visible and testable aspects available to observe the working of various functions.

I am not expecting this PoV project to yeild good result due to the limited time spent on it just before submission deadlines for TT06.

Check back with the repo for a testing regime.

External hardware

At this PoC stage, testing with RP2040 and FPGA external boards to validate the electrical interface acrhetecture makes sense and provided the most options.

Pinout

#	Input	Output	Bidirectional
0	Rx Data	UART TX	Rx Clock (bidi)
1	Tx Data	UART CTS	Up/Down (bidi)
2	UART RTS	UART DCD	TableAddr[0]
3	TableData[0]	Rx Error	TableAddr[1]
4	TableData[1]	Tx Error	TableAddr[2]
5	TableData[2]	Sending	TableAddr[3]
6	TableData[3]		TableAddr[4]
7	UART RX	Tx Clock Stobe	TableAddr[5]

UART [708]

- Author: Darryl Miles
- Description: UART
- GitHub repository
- HDL project
- Mux address: 708
- Extra docs
- Clock: 0 Hz

How it works

Docs to follow.

How to test

Docs to follow.

External hardware

Standard Tiny Tapeout PCB. The IC is a UART DTE.

Trying for:

- TxD on UO_OUT[4] for OUT4 on GPIO13 with RP2040 UART0 (main set)
- RxD on UO_IN[3] for IN3 on GPIO12 with RP2040 UART0 (main set)
- RTS on UO_OUT[5] for OUT5 on GPIO14 with RP2040 UART0 (main set)
- CTS on UO_IN[6] for IN6 on GPIO19 with RP2040 UART0 (adjacent set)

Pinout

#	Input	Output	Bidirectional
0	altclk		busData0
1	busMode0		busData1
2	busMode1		busData2
3	rxd	dtr	busData3
4	dsr	txd	busData4
5	dcd	rts	busData5
6	cts	intTx	busData6

#	Input	Output	Bidirectional
7	ri	intRx	busData7

donut [710]

- Author: Daniel Endraws
- Description: Showing a Donut
- GitHub repository
- HDL project
- Mux address: 710
- Extra docs
- Clock: 50350000 Hz

How it works

Each ellipse is hand crafted to create a donut.

How to test

Connect the PMOD VGA.

External hardware

TinyVGA PMOD

Pinout

#	Input	Output	Bidirectional
0		R[1]	
1		G[1]	
2		B[1]	
3		vsync	
4		R[0]	
5		G[0]	
6		B[0]	
7		hsync	

RO [712]

- Author: Arna Roy
- Description: Implementation of simple RO
- GitHub repository
- HDL project
- Mux address: 712
- Extra docs
- Clock: 20000000 Hz

How it works

The tt_um_roy1707018 module integrates two essential components:

Ring Oscillator-Based Buffer System which essentially a True Random Number Generator or TRNG (ro_buffer_counter) S-Box Cryptographic Component (ascon_sbox) Ring Oscillator-Based Buffer System (ro_buffer_counter) This module contains a buffer driven by two control signals: ro_activate_1: Controls the first set of ring oscillators (bit 0 of ui_in). ro_activate_2: Controls the second set of ring oscillators (bit 1 of ui_in). It also includes a 3-bit signal (bits 2 to 4 of ui_in) that selects a specific output from the buffer. The module comprises a total of 16 ring oscillators, split into two sets of 8. A 64-bit shift register within the submodule stores the last 64 bits of these oscillators' outputs. The selection bits determine which specific set of 8 values from the shift register is presented as the 8-bit output, which is then processed and connected to uo_out.

S-Box Cryptographic Component (ascon_sbox) The second submodule implements an S-Box, a crucial non-linear substitution step used in cryptographic algorithms like ASCON. This S-Box is activated by bit 7 of ui_in and receives bits 2 to 6 of ui_in as input, producing a 5-bit output.

Final Output The final output, uo_out, is the result of a bitwise XOR operation between TRNG and the S-Box. This combination effectively merges the functionalities of both components into a single output signal.

How to test

In the simulation level, from the testbench we sent different values to the input to see if the ring oscillators or SBOX are working correctly or not.

External hardware

No external hardware is needed for this design.

Pinout

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

CMOS design of 4-bit Signed Adder Subtractor [714]

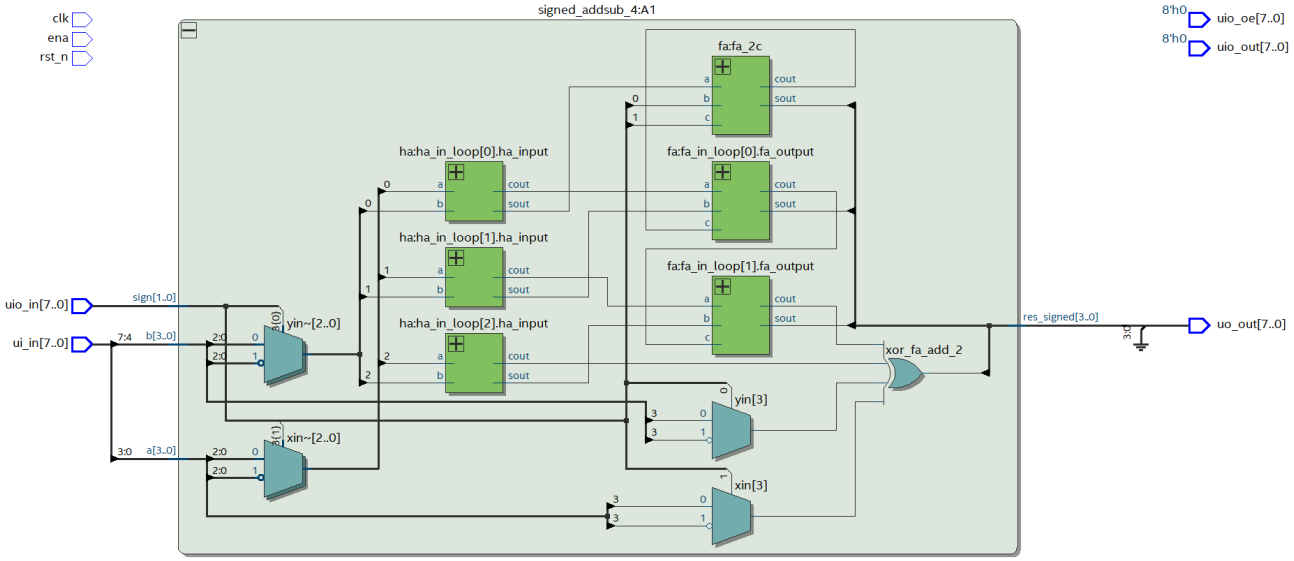
- Author: Vivek Chiranjit
- Description: The project is a signed binary 4-bit adder-subtractor module.
- GitHub repository
- HDL project
- Mux address: 714
- Extra docs
- Clock: 0 Hz

How it works

The project is a signed binary 4-bit adder-subtractor module. The module is constructed using muxes, half adders and full adders.

Depending on the sign[1:0] bits, the circuit can perform the following operations:

sign[1:0]	Operation
00	A + B
01	-A + B
10	A - B
11	-A - B



How to test

The signed_addsub_tb testbench includes extensive test cases for the 4-bit Signed adder-subtractor circuit. The desing has been tested using QuestaSim.

External hardware

None

Pinout

#	Input	Output	Bidirectional
0	a0	s0	sign0
1	a1	s1	sign1
2	a2	s2	
3	a3	s3	
4	b0		
5	b1		
6	b2		
7	b3		

VGA Screensaver with Tiny Tapeout Logo [716]

- Author: Uri Shaked
- Description: Tiny Tapeout Logo bouncing around the screen (640x480, TinyVGA Pmod)
- GitHub repository
- HDL project
- Mux address: 716
- Extra docs
- Clock: 0 Hz

How it works

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

Figure 48: 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.

External hardware

TinyVGA PMOD

Pinout

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

Patater Demo Kit Waggling Rainbow on a Chip [718]

- Author: Jaeden Amero
- Description: A 6-bit Waggling Rainbow demo
- GitHub repository
- HDL project
- Mux address: 718
- Extra docs
- Clock: 25175000 Hz

How it works

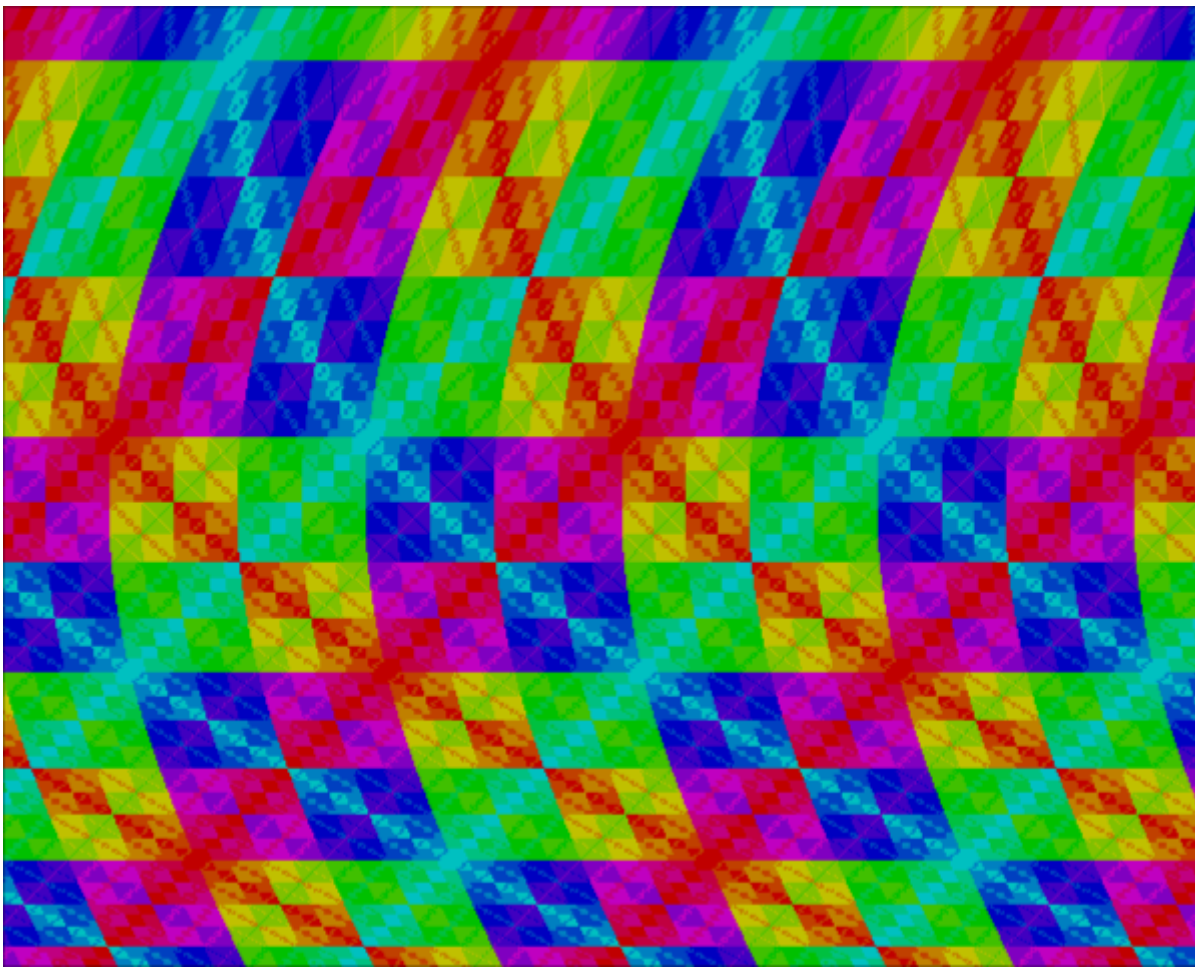
This design outputs a waggling 6-bit rainbow demo on VGA.

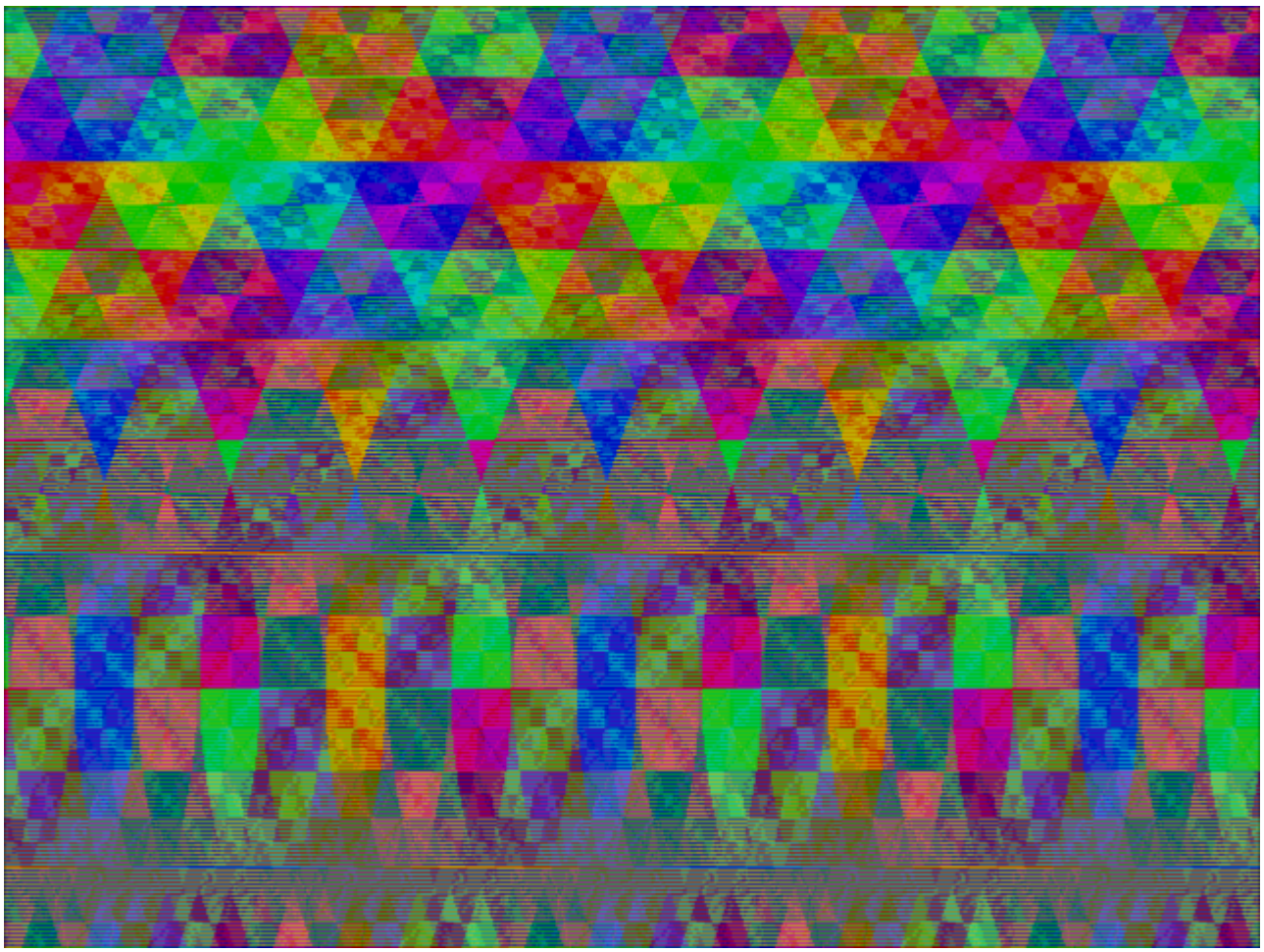
The demo will change effect based on inputs on `ui_in`.

Pin	Pin Name	Setting	Effect
`ui_in[0]`	`DUAL_EN`	Dual mode	Horizontally flips the image each scan line
`ui_in[1]`	`HWAVE_EN`	H wave	Enables use of horizontal waves
`ui_in[4:2]`	`P0_OFF_{2,1,0}`	P0 offset	Controls the speed of the H wave
`ui_in[7:5]`	`P1_OFF_{2,1,0}`	P1 offset	Controls the speed of the V wave

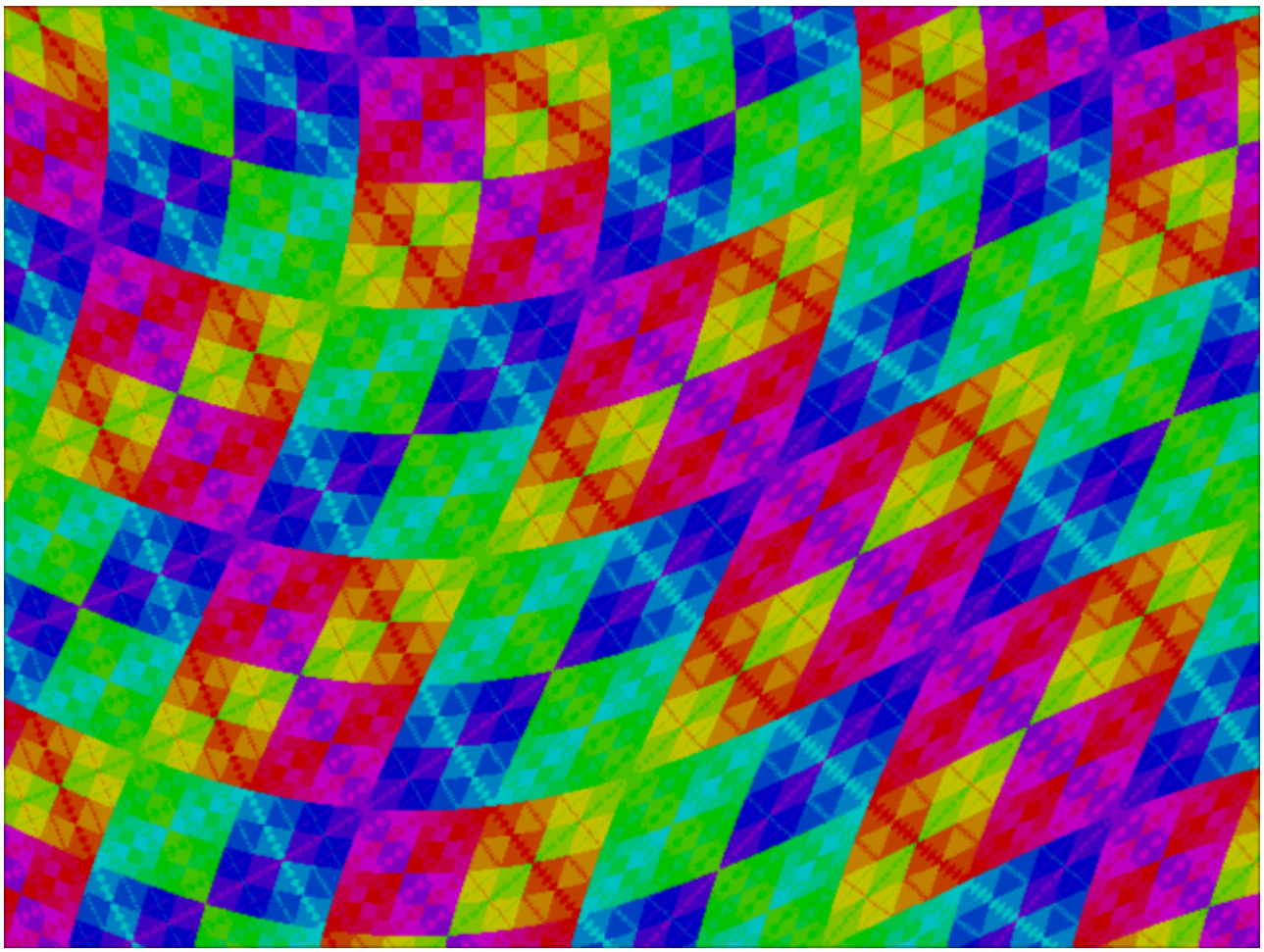
Screenshots

Default





Dual Mode



H Wave

Video A video of a different (software rather than hardware) implementation, of the waggling rainbow effect can be found at https://www.youtube.com/watch?v=AxT45_7WZUQ.

How to test

If wanting to test without hardware, use the VGA playground. Copy and paste the contents of the entire `src/project.v` file into the playground's text editor, replacing all previous content. Then, change the name of the module from `tt_um_patater_demokit` to `tt_um_vga_example` and the simulator will start running in your browser.

If testing with hardware, use a TinyVGA PMOD. Clock the design with 25.175 MHz as described in `info.yaml` (25.157 MHz is standard for 60 Hz 640x480 VGA video).

If testing with lower level simulation tools, an incomplete cocotb test bench (`test/test.py`) is provided. Passing the tests in the cocotb bench is no guarantee that the design will work.

External hardware

External hardware required:

- TinyVGA PMOD

Pinout

#	Input	Output	Bidirectional
0	DUAL_EN	R1	
1	HWAVE_EN	G1	
2	P0_OFF_0	B1	
3	P0_OFF_1	VSync	
4	P0_OFF_2	R0	
5	P1_OFF_0	G0	
6	P1_OFF_1	B0	
7	P1_OFF_2	HSync	

TT08 Pachelbel's Canon demo [768]

- Author: Mike Bell
- Description: Tiny Tapeout visuals with the classic Canon soundtrack
- GitHub repository
- HDL project
- Mux address: 768
- Extra docs
- Clock: 36000000 Hz

How it works

The project plays Pachelbel's Canon along with some fun visuals.

How to test

Set the inputs to 0, clock at 36MHz.

External hardware

Tiny Tapeout Audio Pmod in the bidir Tiny VGA Pmod in the output

Pinout

#	Input	Output	Bidirectional
0		R[1]	
1		G[1]	
2		B[1]	
3		vsync	
4		R[0]	
5		G[0]	
6		B[0]	
7		hsync	PWM output

Sequential Shadows [TT08 demo competition] [770]

- Author: Toivo Henningsson
- Description: My contribution to the TT08 demo competition
- GitHub repository
- HDL project
- Mux address: 770
- Extra docs
- Clock: 50400000 Hz

Intro

Curly / Medieval presents

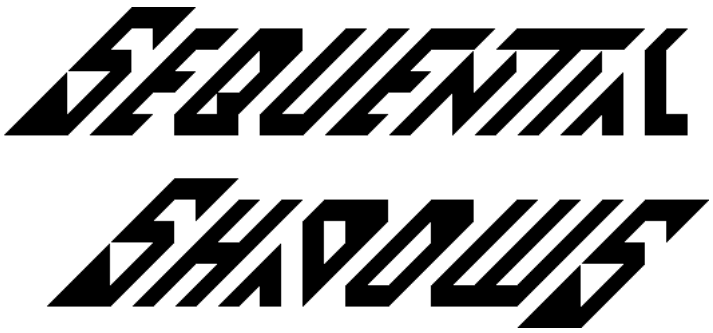


Figure 49: Sequential Shadows logo

my contribution to the Tiny Tapeout 8 demo competition. Code, graphics, and music by Curly (Toivo Henningsson) of Medieval.

How to test

Plug in a TinVGA compatible Pmod on the TT08 demo board's out Pmod. Plug in a Pmod compatbile with Mike's audio Pmod compatible Pmod on the TT08 demo board's bidir Pmod. Set all inputs to zero to get the default behavior. **Warning: The default behavior includes some flashing ligts.** Set `v_bass_off` and `v_drums_off` (keep `ui_in` at 3 instead of 0) to remove flashing. The demo starts directly after reset.

This demo is best viewed with the monitor rotated 90 degrees, with the left side facing down.

Inputs There is no guarantee that changing the inputs after reset is released works as intended, but it probably does. Some of the inputs provide options on how the demo is run:

- `v_bass_off`: Setting this high reduces flashing, but also turns off the bass in some parts.
- `v_drums_off`: Setting this high reduces flashing, but also turns off the drums in some parts.
- `v_bass_low`: Setting this high keeps the bass at its default octave during the entire demo, and increases flashing.
- `pause`: While this is high, the demo is paused and the sound is turned off.
- `step_frame`: While this is high, the the demo advances one frame per cycle. Used for testing.

External hardware

This project needs

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

Pinout

#	Input	Output	Bidirectional
0	v_bass_off	R1	
1	v_drums_off	G1	
2	v_bass_low	B1	
3	pause	vsync	
4		R0	
5		G0	
6		B0	
7	step_frame	hsync	audio_out

TinyMandelbrot [772]

- Author: Gerrit Grutzeck
- Description: A mandelbrot generator
- GitHub repository
- HDL project
- Mux address: 772
- Extra docs
- Clock: 0 Hz

How it works

The project has two parts, first a module to generate a Mandelbrot. Second, a VGA driver, which fetches the data from a framebuffer, which is eamulated by the RP2040.

How to test

RP2040 Mode For this the mode pin has to be selected. Then the configuration should be shifted into the project. Finally the render can be started and the data received via a logic analyzer or the RP2040.

VGA Mode For this the RP2040 has to be programmed with a special firmware to emulate the framebuffer. The the VGA mode has to be selected. Then the configuration should be shifted into the project. Finally the render can be started. After the rendering is finished, the Mandelbrot should be displayed via VGA.

External hardware

To the output Pmod connector the TinyVGA Pmod should be connected, if the VGA mode is used.

Pinout

#	Input	Output	Bidirectional
0	serial enable	R[1] or ctr[0]	write data[0]
1	serial data	G[1] or ctr[0]	write data[1]
2	serial clock	B[1] or ctr[0]	write data[2]
3	output select	vsync or ctr[0]	write data[3]

#	Input	Output	Bidirectional
4	frame data[0]	R[0] or new counter	reset write pointer
5	frame data[1]	G[0]	write data
6	frame data[2]	B[0]	reset read pointer
7	frame data[3]	hsync	read

Sprite Bouncer with Looping Background Options [774]

- Author: Jacob Mack
- Description: Sprite bouncer hardware that supports multiple background options and sprites.
- GitHub repository
- HDL project
- Mux address: 774
- Extra docs
- Clock: 25000000 Hz

How it works

Sprite ROM, background control registers, and audio ROM are configured using SPI

How to test

Configure background, sprite image, and sfx on bounce using SPI

External hardware

Audio Pmod and Tiny VGA Pmod

Pinout

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

“SQUARE-1”: VGA/audio demo [778]

- Author: Zachary Catlin
- Description: On video: munching squares. On audio: the logistic map.
- GitHub repository
- HDL project
- Mux address: 778
- Extra docs
- Clock: 25200000 Hz

How to test

Assuming the ASIC is connected to the TT demo board and suitable interface electronics have been connected (see “External hardware”), select the `tt_um_zec_square1` project to get started. If `rst_n` is not automatically set to logic high upon selection, you’ll need to manually disable the reset. Enable the reset again when you’re done.

If not using the demo board, you’ll need to supply the ASIC with a 25.175 MHz or 25.200 MHz clock, do the appropriate interactions with the project-selection logic to select `tt_um_zec_square1`, and use the pinout to connect to video and audio output devices. Note: y1 and y0 are the high-order and low-order bits (respectively) of color component y.

The video part of the demo repeats with a cycle time of ~8.5 seconds, while the audio part repeats with a cycle time of just under 2 minutes.

External hardware

Assuming the ASIC is connected to the TT demo board, VGA output is obtained by connecting a Tiny VGA Pmod or compatible module to the OUTPUT Pmod connector, and audio output is obtained by connecting a Tiny Tapeout Audio Pmod to the BIDIR Pmod connector.

How it works

SQUARE-1 contains a VGA-compatible video demo and an independent audio demo, described separately below.

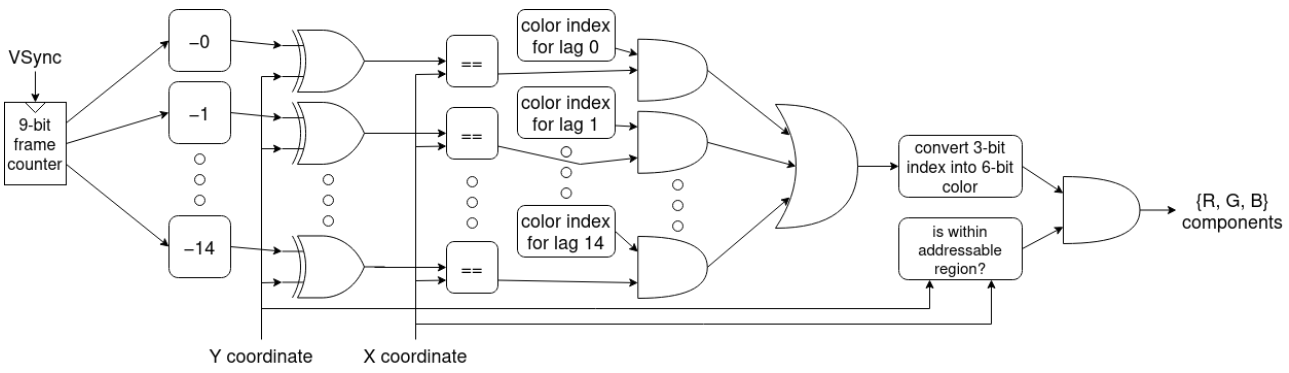
Video While the demoscene dates to the mid-1980s, people have been making aesthetically-interesting graphics with a tiny amount of code for much longer. One of the earliest "display hacks" is munching squares, first implemented *c.* 1962 on MIT's PDP-1 (hence this demo's name). The original version has feedback and user-configurability (see Norbert Landsteiner's write-up for more details and a PDP-1 emulator), but a simple variant requires only two N-bit variables—`t`, a frame counter, and `y`, a row counter, used thus:

```
t ← 0
loop
  wait for end of frame
  t ← (t + 1) mod 2^N
  for y ← 0 to 2^N-1
    plot (t XOR y, y)
```

As the algorithm has so little state and involves simple operations, a "racing the beam" implementation requires little silicon area. SQUARE-1 uses $N = 9$ and accepts that the bottom bit of the square gets lost off the 640×480 screen.

However, a simple implementation of the algorithm would not *look* much like the original version! PDP-1 munching squares uses a Type 30 point display, which was built around a radar-scope CRT using P7 phosphor. P7 is actually a combination of two substances—a bright, short-persistence (decay constant ~20 microseconds) far-blue phosphor excited by the electron beam, and a dimmer, long-persistence (main decay constant ~100 milliseconds, but with a long tail lasting several seconds) yellow phosphor excited by the light from the blue phosphor. As a result, the plotted points have a white or blue-white appearance, then become yellow and visibly fade away.

Fortunately, since each frame only has one point in each line, and said point is different in each frame, it's easy to parallelize an emulation of persistence, which is done in `src/project.v`, which conceptually works like this:



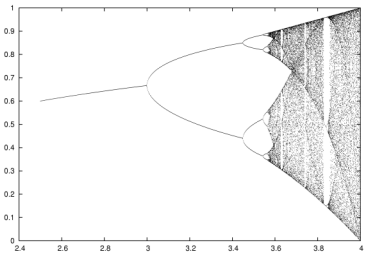
Apart from the VSync/HSync/coordinate-generating module, it's almost entirely combinational logic. SQUARE-1 simulates 14 frames (~1/4 second) of persistence prior to

the current frame—not quite a Type 30, but enough to get the feel of the thing on modern displays.

Audio The audio demo is a sonification of the logistic map. To give a quick overview, the following iteration:

$$x_{i+1} \leftarrow r x_i (1 - x_i)$$

maps values of $x \in (0, 1)$ to values in $(0, 1)$ when $r \in (1, 4)$. When $r \in (1, 3]$, the sequence of x_i values converges to a single value (the *attractor*), but much more interesting behavior happens when $r \in (3, 4)$:

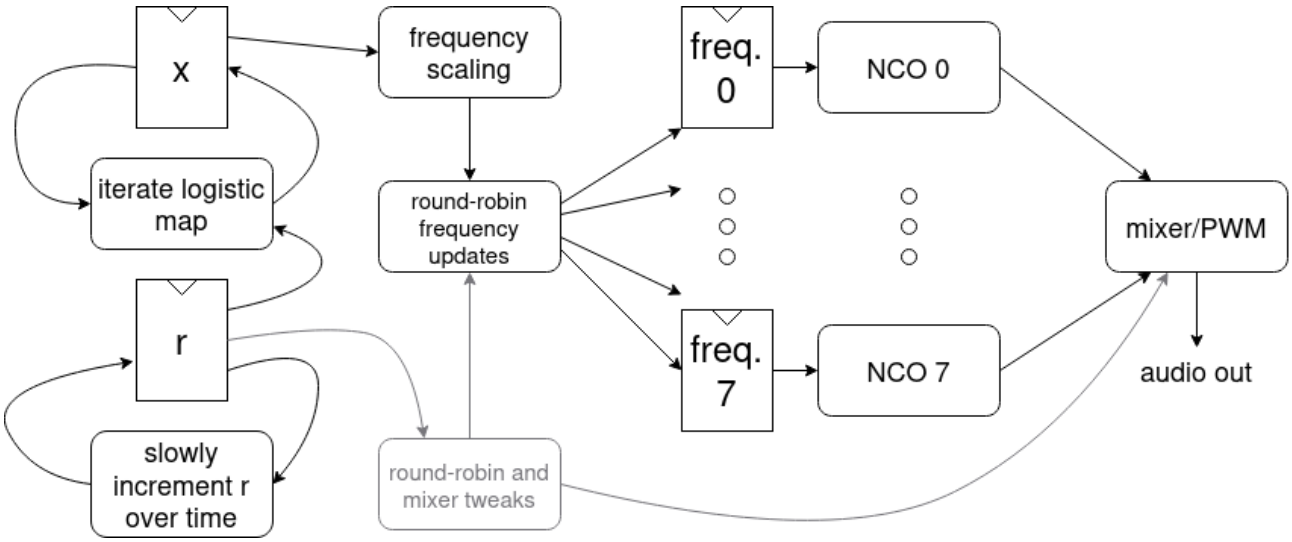


Credit: Ap on en.wikipedia.org

First, the attractor becomes a period-2 cycle, then period-4, -8, -16... and then it exhibits chaotic behavior. That iteratively applying a quadratic polynomial would result in such behavior came as quite a surprise back in the 1960s, and to this day the logistic map is a popular demonstration of mathematical chaos in a simple system.

So, what does it mean to turn the logistic map into a sound? The way SQUARE-1 does it, values of x_i at a given r are scaled from $(0, 1)$ to approximately $(200, 1200)$ Hz, which are then used as the frequencies of an ensemble of 8 square-wave generators. The square waves are then added together and used as the input to a PWM generator, the last providing the sound output. r is varied to cover the range $[\frac{17}{16}, 4)$ over a period of ~2 minutes, varying faster over $r < 3$ to get to the good stuff sooner.

Finally, over a few small portions of the chaotic region, we change the number of square-wave generators that get frequency updates and get mixed together. The reason is that within the chaotic region, there are islands of periodicity, the largest of which have attractors of period 3, 5, and 6. Tweaking the number of active generators to be a multiple of the period leads to better-sounding results within the islands.



Greetz

Eh, I'm not *that* social...

...Hi, Mom! Hi, Dad!

Well, also, thanks to the organizers of the TT08 demoscene competition for finally inspiring me to get off my rear and go sculpt some silicon. Thanks as well to the open source EDA and silicon communities for making all this feasible.

Pinout

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

Sequential Shadows Deluxe [TT08 demo competition] [782]

- Author: Toivo Henningsson
- Description: My contribution to the TT08 demo competition, extended version
- GitHub repository
- HDL project
- Mux address: 782
- Extra docs
- Clock: 50400000 Hz

Intro

Curly / Medieval presents

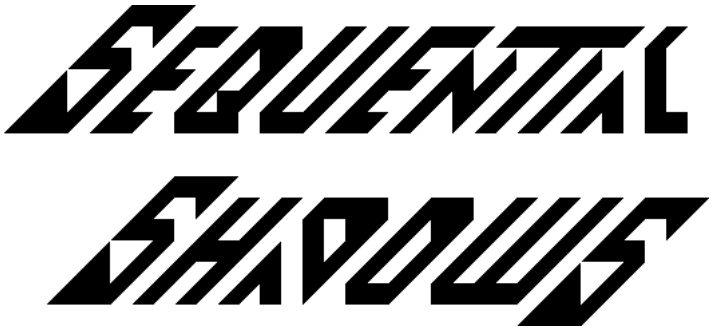


Figure 50: Sequential Shadows logo

my contribution to the Tiny Tapeout 8 demo competition. Code, graphics, and music by Curly (Toivo Henningsson) of Medieval.

This is the deluxe version, with Pmod VGA RGB444 output support and a few changes from the original, in 2x2 tiles compared to the original's 1x2.

How to test

Plug in a TinVGA compatible Pmod on the TT08 demo board's out Pmod. Plug in a Pmod compatbile with Mike's audio Pmod compatible Pmod on the TT08 demo board's bidir Pmod. Set all inputs to zero to get the default behavior. **Warning: The default behavior includes some flashing ligts.** Set `v_bass_off` and `v_drums_off` (keep `ui_in` at 3 instead of 0) to remove flashing. The demo starts directly after reset.

This demo is best viewed with the monitor rotated 90 degrees, with the left side facing down.

Inputs There is no guarantee that changing the inputs after reset is released works as intended, but it probably does. Some of the inputs provide options on how the demo is run:

- `v_bass_off`: Setting this high reduces flashing, but also turns off the bass in some parts.
- `v_drums_off`: Setting this high reduces flashing, but also turns off the drums in some parts.
- `v_bass_low`: Setting this high keeps the bass at its default octave during the entire demo, and increases flashing.
- `pause`: While this is high, the demo is paused and the sound is turned off.
- `step_frame`: While this is high, the the demo advances one frame per cycle. Used for testing.
- `rgb444_mode`: Setting this high sets the output to RGB444 mode instead of the default RGB222
- `pmod_vga_pinout`: Setting this high enables the alternative Pmod VGA pinout.
  - The `t_` outputs are used when `pmod_vga_pinout` is low. This fits the TinyVGA Pmod pinout. (`p_` only outputs are not driven.)
  - The `p_` outputs are used when `pmod_vga_pinout` is high. This fits the Pmod VGA pinout.
- `logo_shadow_off`: When high, removes the logo's shadow (like in the non-deluxe version).

If using A Pmod VGA as output, set `rgb444_mode` unless you want the original RGB222 experience.

For the demo competition, set `pmod_vga_pinout` and `rgb444_mode` if you have a Pmod VGA, and please consider if you can still hook up the sound. Don't set any of the other inputs.

External hardware

This project needs

- either
  - a TinVGA VGA Pmod.
  - Mike's audio Pmod.
- or a Pmod VGA

- There is no ready option to output the audio in this case, but it's still present on the same pins, so you may be able to get it out with some creative wiring, and e g feed it to Mike's audio Pmod.

The choice of pinout is controlled by the `pmod_vga_pinout` input.

Pinout

#	Input	Output	Bidirectional
0	v_bass_off	t_R1 / p_R0	p_G0
1	v_drums_off	t_G1 / p_R1	p_G1
2	v_bass_low	t_B1 / p_R2	p_G2
3	pause	t_vsync / p_R3	p_G3
4	rgb444_mode	t_R0 / p_B0	p_hsync
5	pmod_vga_pinout	t_G0 / p_B1	p_vsync
6	logo_shadow_off	t_B0 / p_B2	audio_out_n
7	step_frame	t_hsync / p_B3	audio_out

Wirecube [832]

- Author: Leo Moser
- Description: A demo for the Tiny Tapeout demoscene competition - see for yourself!
- GitHub repository
- HDL project
- Mux address: 832
- Extra docs
- Clock: 50350000 Hz

How it works

The documentation will be updated after the competition has concluded.

How to test

Connect a Tiny VGA to the output Pmod port, set the clock frequency to two times 25.175 MHz = 50.350 MHz, make sure `ui_in` is set to `0x00` and enjoy the show!

External hardware

- Tiny VGA Pmod

Pinout

#	Input	Output	Bidirectional
0	toggle background bit 0	R1	
1	toggle background bit 1	G1	
2	toggle background bit 2	B1	
3	toggle cube bit 0	VS	
4	toggle cube bit 1	R0	
5	toggle cube bit 2	G0	
6	toggle speed bit 0	B0	
7	toggle speed bit 1	HS	

RGBW Color Processor [834]

- Author: Enrico Sanino
- Description: Color processor for RGBW LEDs, with generation of hue, tint and intensity based on a color index. Is also a direct SPI to 4 channels PWM converter.
- GitHub repository
- HDL project
- Mux address: 834
- Extra docs
- Clock: 66000000 Hz

How it works

Color generator for RGBW LEDs, with generation of hue, tint and intensity based on a color index. Is also a direct SPI to 4 PWM channels converter, making it flexible to any different kind of use. The system block diagram is as follow:

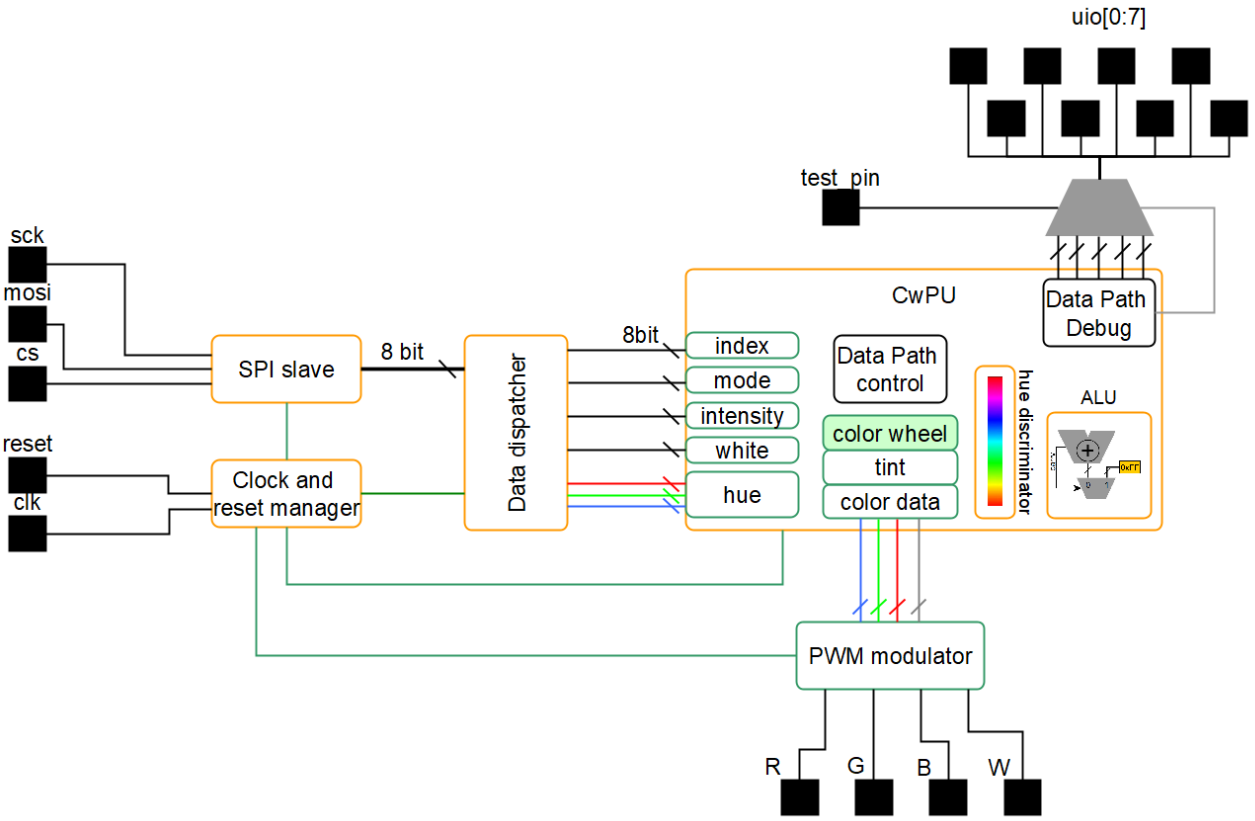


Figure 51: block diagram image

It is an SPI slave in Mode 0, with SPI protocol consisting of 8 byte long command, discriminated with a preamble sequence (see Protocol and Test for the description).

This payload is unpacked in different data: red, green, blue, white, bypass mode, intensity, color index. This data is then provided to the color wheel processor. If the bypass mode is activated, the RGBW info from the red, green, blue and white SPI bytes is directly provided as a PWM output in the respective channels.

If bypass mode is not active, only the white, intensity and color index are considered, from which the hue (RGB data) is generated based on the index, then a tint (hue + white) and then the intensity is applied, forming the final color. This is then applied to the PWM outpus to the respective channels.

When bypass mode is not active (color wheel mode), then there is a latency proportional to the "rotation" of the color wheel, i.e. lower the number lower the latency. This is the laterncy of the color wheel processing unit (CwPU), after which the desired complete color is output on the PWM channels.

Debug pins A debug enable pin, when asserted, will output on the uio pins different internal signals of the CwPU while in operation. This is just to check the internal signals in case the tapeout goes wrong, and for curiosity purposes for fidelity against the gate level simulation.

PWM modulator The PWM modulator has a period of t_pwm = t_clk_presc * 256, and a resolution of 1/256 steps. The t_clk_presc is the prescaled clock, t_clk_presc = t_clk * 2. Each update is synchronous to the period, hence any change in the duty cycle will happens to the next PWM period without generating artifacts.

Clock and reset maanger The clock and reset manager will issue a precaled clock to the whole system by a factor of 2, except for the multiplicator, which has to run twice as fast w.r.t. the system. A toggle on the reset pin will reset the whole system at the next reset *release*. Meaning, to reset the system, the reset (active low) must go to LOW, then it must be deasserted to HIGH. By doing this, the clock must be always present (sync reset).

When reset is deasserted (HIGH), the manager will start and will keep the rest of the system in reset state for the next 128 t_clk cycles (main clock from the pin). This will guarantee that the whole system will be correctly initialized.

Therefore any SPI transaction can take place after at least 128 clock cycles after reset condition is deasserted, otherwise one SPI packed would be lost.

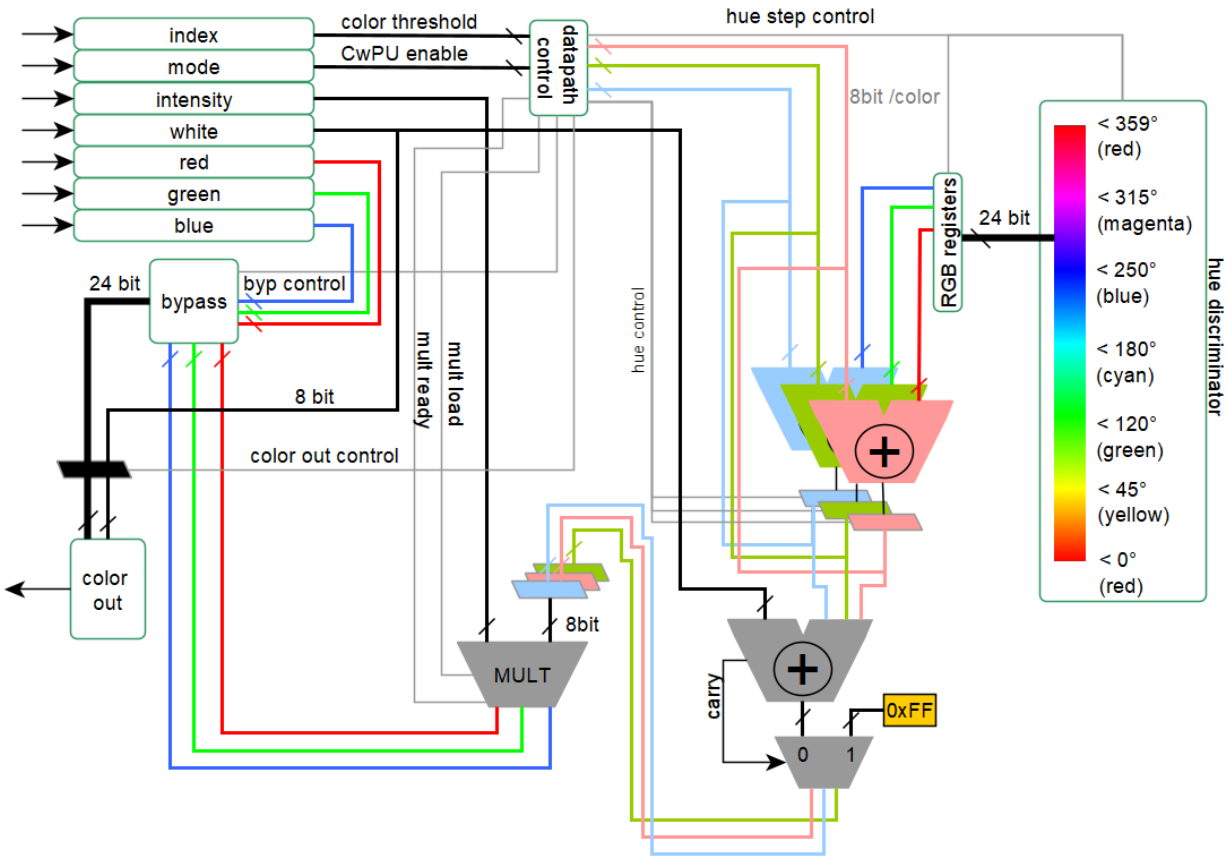


Figure 52: SCwPU datapath

Color wheel processor The logic datapath of the CwPU is shown below:

The CwPU has all the data width of 8 bit, and the energy intensive color discrimination path is active when non in bypass mode only. When active will take the index. Starting from zero, increments the hue progression and compares against this index (i.e. rotates the color wheel) to process at run time with no LUT, the corresponding requested hue. During the rotation, the RGB internal values will also change, increasing and decreasing the hue components to sweep all the combinations to match the requested one. The final value will be used for the next step, which is the tint.

The next step is the sum of the white component, generating a tint, a white adjusted color. It will sum the white up to the maximum value, and the value is output to the intensity multiplier. Also the white is output to the multiplier. This is to not only output an RGB to emualate the white, but to increase the color rendering index (CRI) by allowing to use a single output that can be connected to a pure white generator/phosphor based white LED.

The multiplication for the intensity then takes place with a single multiplicator unit, hence the local control takes care of the data load and synchonization, with 2 clock cycles per operation. Since the multiplicator goes twice as fast, the CwPU has not additional wait states, resulting in 1 CwPU clock cycle delay. Also the white is multiplied. After this step, the output data of each component (R, G, B and W) are 16bit, but the 8 LSB are truncated, generating a final 24 bit color information and 8 bit white.

This data is used by the 4 channel PWM modulator.

When in bypass mode, the CwPU will only replicate the same RGBW info in input to the PWM modulator input in one clock cycle.

SPI protocol SPI is Mode 0 as shown in this timing diagram, highlighting the preable and first byte transfer:

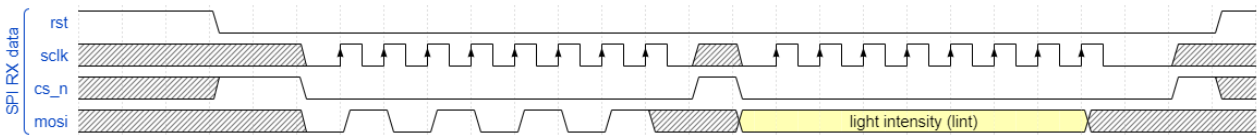


Figure 53: SPI transaction image, bit detail

While a whole packet must be compliant with the following diagram:

Which contains:

1. preamble: 0x55
2. intensity: 0x00 - 0xFF
3. color index: 0x00 - 0xFF

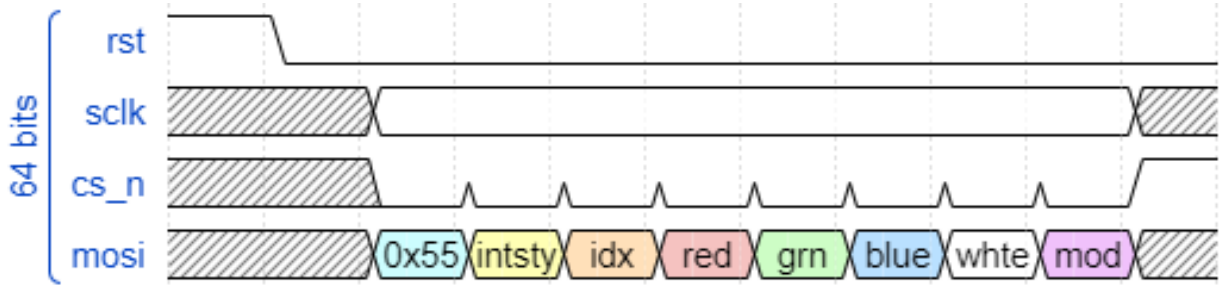


Figure 54: SPI transaction image, whole packet structure

4. red: 0x00 - 0xFF
5. green: 0x00 - 0xFF
6. blue: 0x00 - 0xFF
7. white: 0x00 - 0xFF
8. bypass mode: 0xA4 for the color generation, 0x21 bypass

Note that in between each byte is mandatory to toggle the CS signal, since in reality a full transaction is interpreted as a 8 individual single byte transactions. Therefore, if the bus gets corrupted, sending any data without preamble with more than 8 bytes, will ensure a clean bus state ready to be synchronized again. Otherwise a reset is an alternative.

How to test

This is normally tested with a micropython script to be interpreted directly from the REPL interface of the TT08 demoboard (see https://tinytapeout.com/guides/get-started-demoboard/). To test the design simply setup the demoboard, and run the script in the test folder. It means it can be simply copy/pasted into the REPL terminal.

To see an output, is suggested to wire some LEDs to the output of the demoboard being careful to not overload the output pins. If you don't know what you are doing, then is better to get like 4x of these for the 4 LEDs tindie.com/products/aleadesigns or any other LED controller that **won't load** more than 4mA on the TT08 chip output pads (see pad spec here).

A custom PMOD will come soon to ease the LED test.

With the RP2040 no input wiring is needed, and the output will be:

uo_out[0] -> Red LED

uo_out[1] -> Green LED

uo_out[2] -> BLue LED

uo_out[3] -> White LED

What to expect on the outputs

Given the HUE ternary (r,g,b) processed from the index by the CwPU, the final color is *RGBW = ((r,g,b)+w)intensity*, having a PWM signal per each color channel.

So the white and intensity have a direct impact regardless the hue generated.

The output "color equation" with bypass is *RGBW = spi(red, green, blue, white)* with NO intensity, NO automatic white. In this mode, the data provided via SPI is the data taken by the PWM modulator as is.

External hardware

While we're working at a PMOD right now, the external hardware are 4 LEDs, one per each color, connected to the outputs. Be aware that the outputs cannot take more than 4mA!!! So a dedicated circuit is needed (but will be provided soon). Stay tuned.

To control the design, no external controller is needed since it uses the internal RP2040 of the demoboard, see the documentation here of the test and the REPL script here. Alternatively, a custom firmware and another dedicated python script is provided with the relative STM32 based project, briefly documented here.

Pinout

#	Input	Output	Bidirectional
0		red_pwm	test_out_0
1		green_pwm	test_out_1
2		blue_pwm	test_out_2
3	test_pin	white_pwm	test_out_3
4	cs_n		test_out_4
5	sck		test_out_5
6	mosi		test_out_6
7	clk_div_en		test_out_7

Stochastic Multiplier, Adder and Self-Multiplier [836]

- Author: Ciecen Lestari, Chih-Kuan Ho, David Parent
- Description: Multiplier, Adder and Self-Multiplier using stochastic computing
- GitHub repository
- HDL project
- Mux address: 836
- Extra docs
- Clock: 50000000 Hz

How it works

Design Details

The Stochastic Multiplier, Adder and Self-Multiplier is a digital logic design implementing stochastic arithmetic operations—addition, multiplication, and self-multiplication—using serial 9-bit inputs and outputs. These inputs and outputs are buffered by 1 bit.

Stochastic computing makes use of probability to hold information. LFSRs (Linear feedback serial registers) are used to generate pseudo-random numbers, which are then used by SNGs (Stochastic Number Generators) to generate the stochastic bitstream. The probability of each bit in the stream being a 1 gives the value held by the bitstream, and this is then manipulated by the operations.

Using the bipolar representation, which can represent positive and negative numbers, multiplication can be implemented with an XNOR gate, while addition can be implemented with a MUX.

The design's held inputs are reset every 2^17+1 clock cycle period. The module uses serial-to-parallel input and parallel –to-serial output.

The purpose of having this design is to build the basic blocks of stochastic computing, and future work may include applying these to build circuits like the digital QIF neuron or other circuits.

Figure 55: image

REFERENCES USED

General Stochastic Computing Design:

A. Alaghi, W. Qian, and J. P. Hayes, "The Promise and Challenge of Stochastic Computing," IEEE Trans. Comput. Aided Des. Integr. Circuits Syst., vol. 37, no. 8, pp. 1515–1531, Aug. 2018, doi: 10.1109/TCAD.2017.2778107.

B. R. Gaines, "Stochastic computing," in Proceedings of the April 18-20, 1967, spring joint computer conference, in AFIPS '67 (Spring). New York, NY, USA: Association for Computing Machinery, Apr. 1967, pp. 149–156. doi: 10.1145/1465482.1465505.

Pins Utilization

Input Pins:
ui_in[0] for serial input of 9bit (+1 bit buffer) probability with 1 bit buffer.
ui_in[1] for serial input of 9bit (+1 bit buffer) probability with 1 bit buffer.

Output Pins:
uo_out[0] for serial output of 9bit (+1 bit buffer) probability result of multiplier.
uo_out[1] for serial output of 9bit (+1 bit buffer) probability result of adder.
uo_out[2] for serial output of 9bit (+1 bit buffer) probability result of self-multiplier.
uo_out[3] signals the inner reset of the clk_counter of the module (not rst_n).

Figure 56: image

Multiplier: Input 0 * Input 1

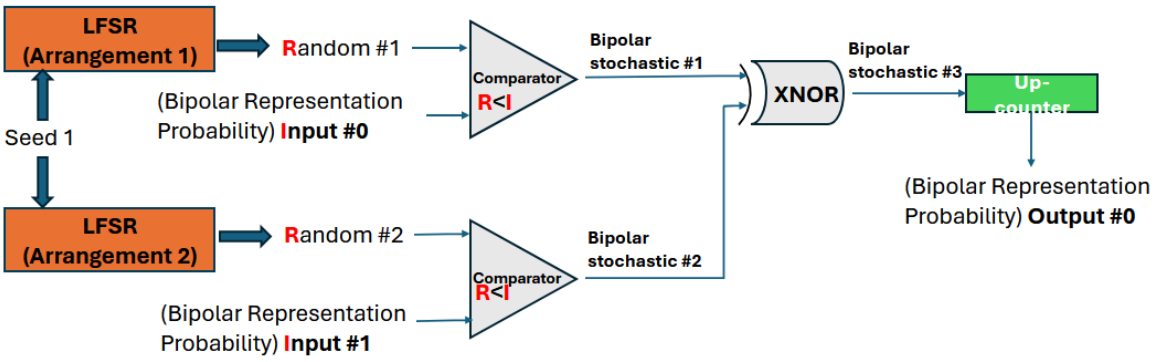


Figure 57: image

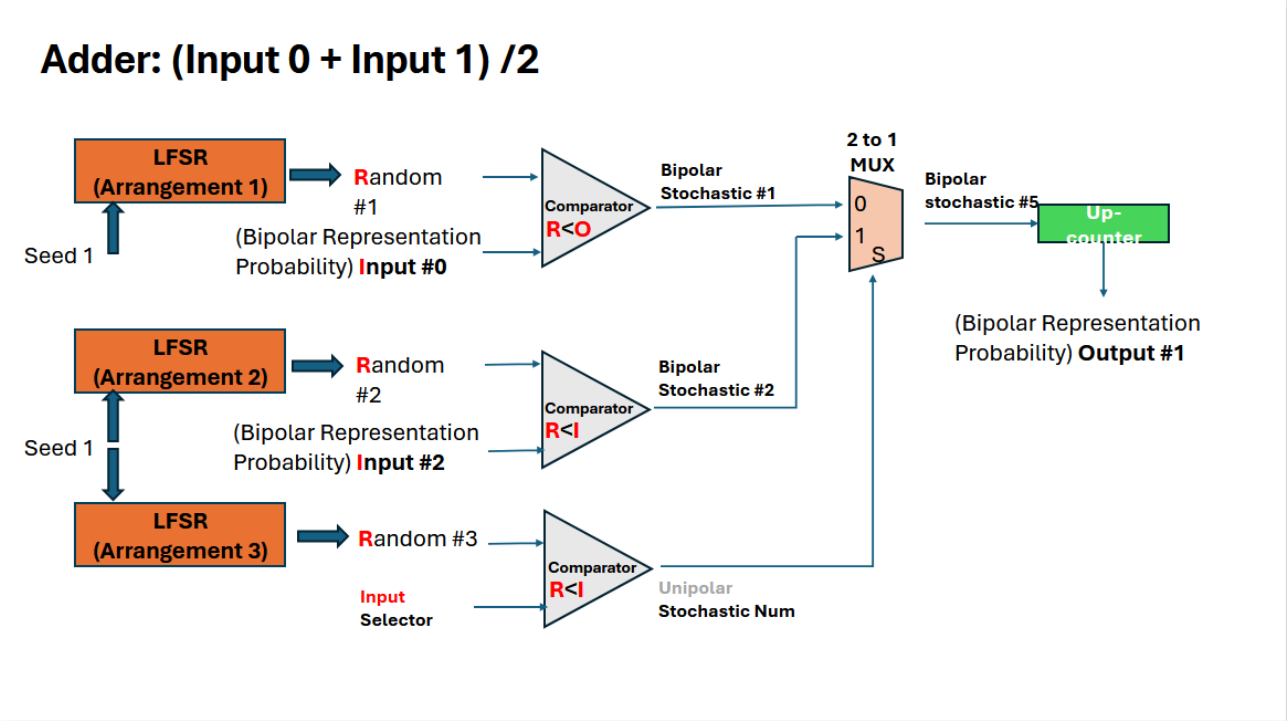


Figure 58: image

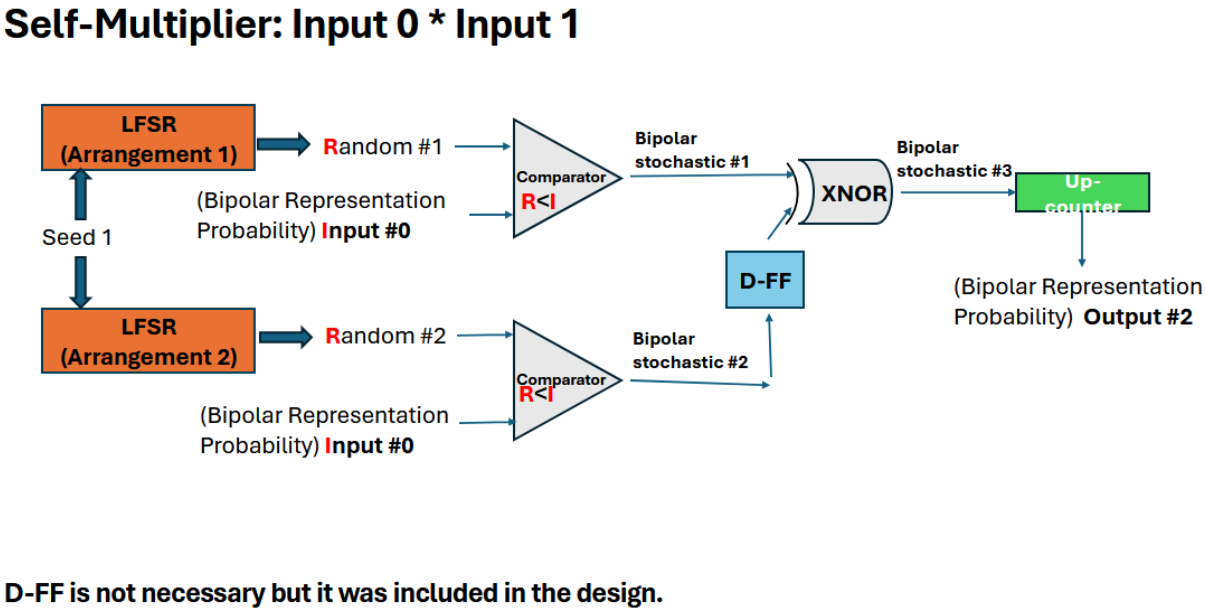


Figure 59: image

Gross, W. J., & Gaudet, V. C. (Eds.). (2019). Stochastic Computing: Techniques and Applications (1st ed. 2019.). Springer International Publishing. https://doi.org/10.1007/978-3-030-03730-7

Qian, W. (2011). Digital yet deliberately random: Synthesizing logical computation on stochastic bit streams (Order No. 3466985). Available from ProQuest Dissertations & Theses Global: The Sciences and Engineering Collection. (885872145). Retrieved from http://search.proquest.com.libaccess.sjlibrary.org/dissertations-theses/digital-yet-deliberately-random-synthesizing/docview/885872145/se-2

LFSR Design in Stochastic Computing:

Jason H. Anderson, Yuko Hara-Azumi, and Shigeru Yamashita. 2016. Effect of LFSR seeding, scrambling and feedback polynomial on stochastic computing accuracy. In Proceedings of the 2016 Conference on Design, Automation & Test in Europe (DATE '16). EDA Consortium, San Jose, CA, USA, 1550–1555. https://dl.acm.org/doi/abs/10.5555/2971808.2972171

Digital QIF neuron:

E. J. Basham and D. W. Parent, "Compact digital implementation of a quadratic integrate-and-fire neuron," 2012 Annual International Conference of the IEEE Engineering in Medicine and Biology Society, San Diego, CA, USA, 2012, pp. 3543-3548, doi: 10.1109/EMBC.2012.6346731.

keywords: {Mathematical model;Clocks;Equations;Vectors;Computational modeling;Field programmable gate arrays;Neurons},

How to test

Input 2 repeating streams of 9 bits (+1 bit buffer) that represent the numbers to be multiplied/added. The self multiplier only processes input from the 1st stream. Read the serial output result, which is also 9bits (+1 bit buffer).

External hardware

ADALM2000

#	Input	Output	Bidirectional

Pinout

#	Input	Output	Bidirectional
0	serial_input_1	serial_output_mul	
1	serial_input_2	serial_output_add	
2		serial_output_smul	
3		clk_counter_reset	
4			
5			
6			
7			

DL float MAC [838]

- Author: Ananya P & Nidhi M D
- Description: MAC unit for 16 bit DL float data type
- GitHub repository
- HDL project
- Mux address: 838
- Extra docs
- Clock: 40000000 Hz

Design Description

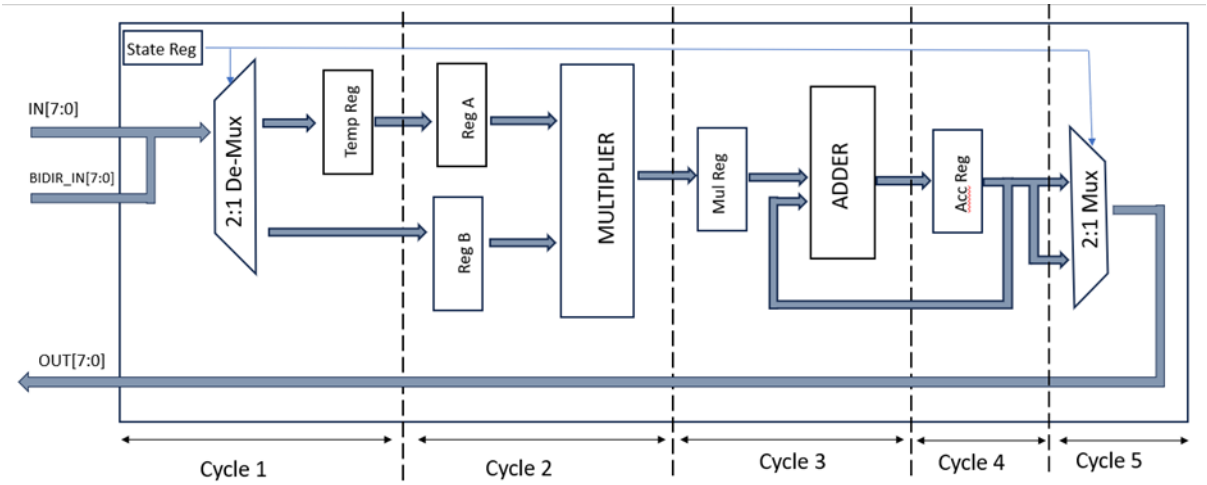


Figure 60: image

The digital design is a 5 stage pipelined architecture implementation of MAC Operation for 16 bit DLFloat numbers. DLFloat is a 16-bit floating-point format designed for deep learning training and inference, where speed is prioritized over precision.

Details of DLFloats:

Sign bit: 1 bit

Exponent width: 6 bits

Significand precision: 9 bits

Bias exponent: 31

Value	Binary format
Max normal	S. 111110.111111111
Min normal	S. 000001.000000000

Value	Binary format
Zero	S. 000000.000000000
Infinity-Nan (combined)	S. 111111.111111111

Work Flow Details:

- The two 16 bit DLFloat input operands are supplied through the ui_in and uio_in (input)pins over two clock cycles getting stored in two registers.

- In the MAC module, the first stage involves multiplying the two inputs, followed by addition of the multiplication result and the accumulated value. The accumulated value in the MAC module starts at zero upon reset.

- After the MAC operation, the 16-bit accumulated result is pushed through uo_out pins over two clock cycles. First the msb 8 bits are pushed out followed by lsb bits.

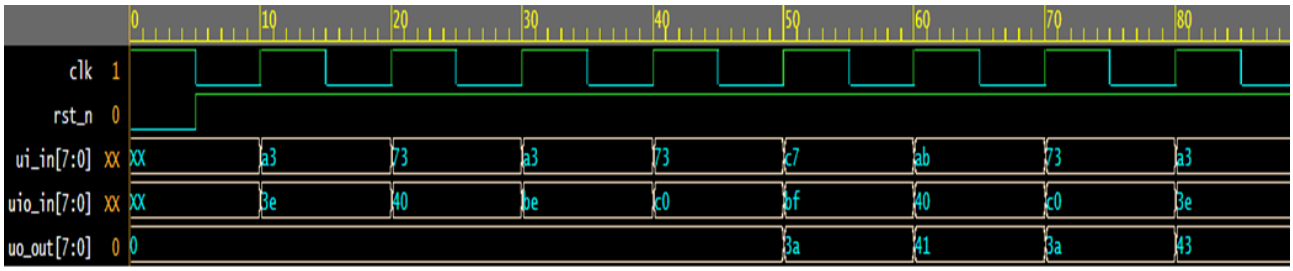


Figure 61: image

This arrangement helps in achieving a pipelined architecture where after 5 clock cycles from reset the output values can be pushed out in every cycle.

Here the addition and multiplication follows the IEEE754 algorithm and the MAC operation incorporates handling the special cases like inf, NaN ,subnormals, zero and a full 16 bit precision range.

The Multiplier and Adder blocks also handle overflow and underflow cases with a saturation logic where upon overflow the result is pushed to the largest number that can be represented in the DLFloat format and similarly with underflow the result is pushed to smallest number with the exception that in Multiplier the underflow is pushed to zero to not affect the accumulated results.

How to test

The DLFloat inputs are fed as binary/hexadecimal equivalent of the binary floating point format. The outputs can be read in similar manner

External hardware

An FPGA is required to drive the inputs to the device and needs to be programmed to capture and display the 16-bit result, which arrives as 8 bits over two clock cycles.

Pinout

#	Input	Output	Bidirectional
0	FP16 in[0]	FP16 out[0]/FP16 out[8]	FP16 in[8]
1	FP16 in[1]	FP16 out[1]/FP16 out[9]	FP16 in[9]
2	FP16 in[2]	FP16 out[2]/FP16 out[10]	FP16 in[10]
3	FP16 in[3]	FP16 out[3]/FP16 out[11]	FP16 in[11]
4	FP16 in[4]	FP16 out[4]/FP16 out[12]	FP16 in[12]
5	FP16 in[5]	FP16 out[5]/FP16 out[13]	FP16 in[13]
6	FP16 in[6]	FP16 out[6]/FP16 out[14]	FP16 in[14]
7	FP16 in[7]	FP16 out[7]/FP16 out[15]	FP16 in[15]

Frequency Counter SSD1306 OLED [840]

- Author: Pawel Sitarz (embelon)
- Description: Simple Frequency Counter displaying result on SSD1306 SPI OLED
- GitHub repository
- HDL project
- Mux address: 840
- Extra docs
- Clock: 1000000 Hz

How it works

Project measures frequency on ui[0] input by counting pulses during 100ms periods. Measured frequency is then displayed on graphical 128x32 pixels OLED display in form of emulated 7-segment display.

How to test

Internal logic needs 1MHz clock (to be generated by RPi Pico)

- Connect PMOD OLED display to see measurement
- Connect unknown frequency signal to be measured to ui[0]

External hardware

Freqquency is displayed on 128x32 OLED display with SSD1306 controller: PMOD OLED

Pinout

#	Input	Output	Bidirectional
0	clk_x - measured frequency input	OLED nRST	
1		OLED nVBAT	
2		OLED nVDC	
3		OLED nCS	
4		OLED Data/Command	
5		OLED CLK	
6		OLED Data Out	

#	Input	Output	Bidirectional
7			

schoolRISCV CPU with Fibonacci program [842]

- Author: Stanislav Zhelnio, Alexander Romanov, Yuri Panchul and Mike Kuskov
- Description: A minimalistic SoC with a schoolRISCV educational CPU and a ROM memory with a program that computes the Fibonacci numbers.
- GitHub repository
- HDL project
- Mux address: 842
- Extra docs
- Clock: 50000000 Hz

How it works

A minimalistic SoC with a schoolRISCV educational CPU and a ROM memory with a program that computes the Fibonacci numbers.

schoolRISCV was originally designed by Stanislav Zhelnio and Alexander Romanov (HSE MIEM) by a suggestion from Yuri Panchul. The goal was to create the simplest possible CPU suitable for the introductory Verilog and FPGA classes. The design was based on a textbook *Digital Design and Computer Architecture* by David Harris and Sarah Harris. Later on Yuri Panchul and Mike Kuskov (Innopolis) adopted the design for the GitHub repositories systemverilog-homework and basics-graphics-music. Now these repos are maintained by the engineers and educators associated with the Verilog Meetup community.

How to test

SystemVerilog testbench A self-checking testbench for the design is located in a directory *test_extra* that contains:

- *clean.bash* - a script to delete temporary files produced by *simulate.bash*.
- *simulate.bash* - a script that simulates the design together with a testbench using Icarus Verilog, producing *log.txt*. Before the simulation, the script compiles assembly *program.s* using the RARS instruction set simulator (ISS) that generates a file *program.hex*. This *program.hex* is used to fill the ROM for both simulation and synthesis.
- *tb.sv* - a self-checking testbench that generates a log and the status *PASS* or *FAIL*.

cocotb testbench The cocotb testbench just runs the simulation for 300 clock cycles checking that the value of the lowest two bits of the dedicated outputs *uo_out* is equal to 01 at the end, which corresponds to self-diagnostics *PASS* and not *FAIL*.

Post silicon After the manufacturing, the design can be manually tested by resetting, driving a clock, and observing the outputs. If the LED connected to the bit 0 of the dedicated outputs (uo_out) turns on (*PASS*) and the LED connected to bit 1 turns off (*FAIL*) the design probably works.

Furthermore, you can drive a slow 3 Hz clock and observe the LEDs connected to the bidirectional signals uio_out. Those pins are configured as outputs and they output the lowest 4 bits of the CPU program counter (PC) and the lowest 4 bits of the RISC_V architecture register *a0* (register 10) that contains the currently computed Fibonacci number.

External hardware

LEDs.

Pinout

#	Input	Output	Bidirectional
0		Test pass	CPU reg a0[0]
1		Test fail	a0[1]
2			a0[2]
3			a0[3]
4			Program Counter pc[0]
5			pc[1]
6			pc[2]
7			pc[3]

VGA Mandelbrot [844]

- Author: Mike Bell
- Description: Mandelbrot on VGA, racing the beam
- GitHub repository
- HDL project
- Mux address: 844
- Extra docs
- Clock: 100000000 Hz

How it works

The Mandelbrot fractal is computed "racing the beam" and displayed through the TinyVGA Pmod.

One iteration of the computation is done over two clock cycles, and a maximum iteration depth of 14 iterations is used. The design is clocked at 100MHz, allowing four clock cycles per 25MHz pixel clock. This means one value is computed every 7 pixels, giving a result like this:

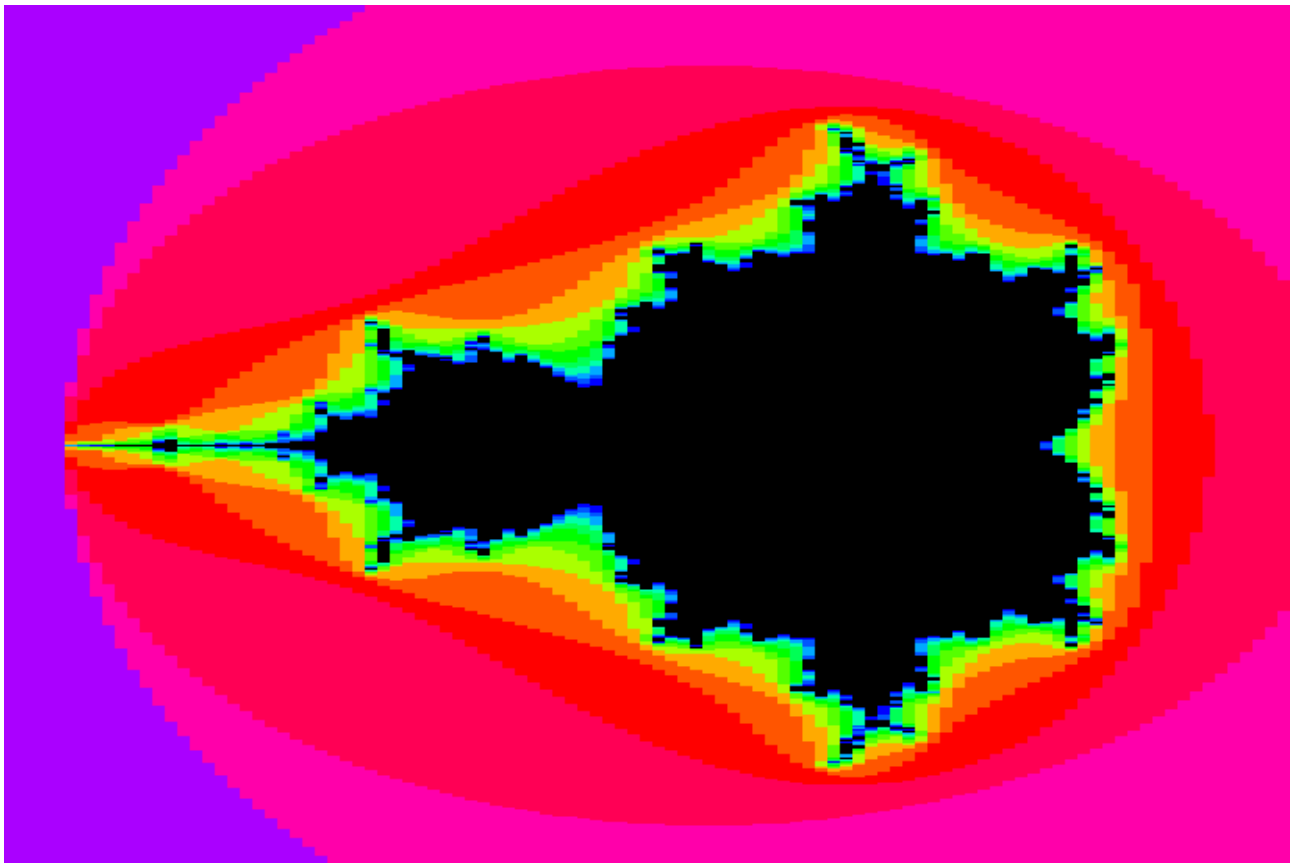


Figure 62: The Mandelbrot set

The computation uses 16-bit fixed point arithmetic. The multiplications are approximated to save area, giving a possible error in the least significant bit. This gives at least 14-bit accuracy on each iteration.

The output image is at a 720x480 resolution (~103x480 Mandelbrot pixels).

How to test

Provide a 100MHz clock.

The image position and zoom can be configured using the input and bidir pins.

in[2:0] control the configuration to set, and {io[7:0], in[7:3]} specify a signed value when setting a register.

These values should only be updated during vsync.

Ctrl	Value
0	Enable demo mode (Zooms in and out repeatedly)
1	Set X coordinate for top-left of screen to value / 2^10
2	Set Y coordinate for top-left of screen to value / 2^11
3	No action
4	Set X increment per column to value[9:0] / 2^13
5	Set Y increment per column to value[9:0] / 2^13
6	Set X increment per row to value[7:0] / 2^13
7	Set Y increment per row to value[7:0] / 2^13

Note there are 103 columns and 480 rows displayed.

External hardware

Tiny VGA Pmod in the output socket.

Pinout

#	Input	Output	Bidirectional
0	Ctrl 0	R[1]	Input 5
1	Ctrl 1	G[1]	Input 6
2	Ctrl 2	B[1]	Input 7
3	Input 0	vsync	Input 8

#	Input	Output	Bidirectional
4	Input 1	R[0]	Input 9
5	Input 2	G[0]	Input 10
6	Input 3	B[0]	Input 11
7	Input 4	hsync	Input 12

Rounding error [846]

- Author: Edwin Török
- Description: Competition entry
- GitHub repository
- HDL project
- Mux address: 846
- Extra docs
- Clock: 25250000 Hz

How it works

Idea This started out as an attempt to implement a ray tracer in 2 TT tiles. However, there isn't enough room for a proper one, precision has to be limited, which leads to unavoidable rounding errors.

So embrace rounding errors, and make them the primary feature!

The end result doesn't resemble a 3D scene, or a sphere, or in fact not even a properly rounded circle, but it has rounding errors! And that is the goal of this project now!

HardCaml The RTL was written using HardCaml, an OCaml DSL that emits Verilog. For convenience the generated Verilog is committed into the source tree, so no additional tools are needed.

I used registers with asynchronous reset, in theory it should be better for an area constrained design.

VGA signal generation

ModeLine VGA signal timing is described in "3. DMT Video Timing Parameter Definitions" in "VESA Display Monitor Timing Standard Version 1.0, Rev. 13", and is implemented in `src/generator/modeline.ml`. Examples on how to implement them on an FPGA are available in several places.

The code supports several resolutions, however to conserve area for the demo I've chosen only 640x480@59.94Hz, which has negative `hsync`/`vsync` polarities. This resolution would need a 25.175 MHz pixel clock, however that can't be produced exactly by the TT08 board, it can only approximate it using a PWM. Therefore, the design is configured to run at the nearest frequency that can be exactly generated:

25.25 MHz, which should be within the 0.5% acceptable by the standard. The ModeLine implemented is: `ModeLine &quot;640x480_59.94&quot; 25.175 640 656 752 800 480 490 492 525 -hsync -vsync`. (This has 59.94 refresh rate and not 60Hz due to the standard preferring NTSC and its 1.001 adjustment).

The design itself runs off the VGA pixel clock, as I didn't want to deal with potential clock domain crossing issues.

Counters There are 2 counters: one for H, and one for V synchronization pulses. When the H counter overflows it enables and increments the V counter for 1 cycle. This is implemented in `generator/vga.ml`, together with waveform expectation tests.

Both H and V counters start out in the visible area for convenience (we can directly use these counters as x/y coordinates, without needing to perform arithmetic in the circuit), then blank the colour signals for the duration of the front porch, synchronization signal and back porch. Although the monitor would recognize the hsync+vsync low as the start of a frame, this is equivalent, but offset by a few clocks.

R, G, B colours The demo supports 2-bit colours, and as usual these would be sRGB colours, not a linear scale. So we define an internal table indexed by 3 bits representing a linear RGB value, mapping to the sRGB bits.

A register is used for the output, both to avoid logic glitches becoming visible to the monitor, and to provide a reg to reg path that `OpenSTA` can use to compute setup/hold times.

Generating the colours When test mode is used (pin `ui[0]` set to `1`) the design outputs vertical colour bars with a white-black-white border. This doesn't have rounding errors, everything is sharp.

In normal mode (pin `ui[0]` is `0`) the "rounding error graphics" is rendered, see below.

Ray marching For an explanation of how ray marching works, see this ray marching tutorial. The "scene" is represented using signed distance functions. The "eye" Z coordinate is animated between 3.5 and 4.5 in 256 steps, where each frame is one step.

CORDIC Fixed point arithmetic with 9 bits of precision is used in the HDL, with the exponent tracked by the generator code to reduce register width (though this is not as good as tracking it in hardware, but that'd require more area). Vector normalization is implemented using the CORDIC implementation provided by HardCaml, configured to use 10 bits, and a limited number of iterations (4) to fit into the desired area. This works by rotating the vector until its angle is 0, and then rotating a second unit vector to match the rotation of the original. Or equivalently transform the original from rectangular to polar coordinates, overwrite the length with 1, and convert back from polar to rectangular. CORDIC is defined for 2D in the library, and I define a 3D wrapper based on rectangular to spheric coordinate conversions, although there would be ways to directly compute a 3D version of CORDIC, that is not implemented here.

This is implemented in `src/vecmath`.

GLSL ES "emulation" The low level operations are wrapped by a higher level embedded DSL that allows writing code quite similar to GLSL ES, with a very small number of operators: arithmetic (+, -, *, /), comparison (==, `&lt;&gt;`), `abs`, `min`, `max`, `clamp`, `length`, `distance`, `dot`, `normalize`, `reflect`.

Unfortunately the full renderer didn't fit into 2 tiles, so had to comment out quite a lot of the "GLSL" code (only 1 step of ray marching, no clamping, very simple gradient approximation), what is remaining does not resemble a sphere, or in fact it doesn't even look 3D.

OpenLane configuration The target density had to be increased to 98% to fit, and the setup slack margin setting had to be increased, see config.json. There are max slew and max fanout violations at 100C and 1.6V, but that shouldn't prevent the design from working at 25C and 1.8V.

The design was simulated using both tt-vgaviz and vgasim, although had to adjust the modeline for vgasim to recognize the standard one. A simple cocotb test which checked vsync/hsync generation was added post submission.

Lack of audio Audio is enabled, but is only a very simple test signal based on hsync and vsync.

Simulating There is a `src/sim/vgasim.ml`, which generates a `demo.v` compatible with vgasim, this uses a different resolution though. `vgasim` has to be called with `-g 640x480`, and `videomode.h` needs to be edited to use `480 490 492 525` (don't know why it wants 521, that doesn't seem to be the standard timing).

Alternatively the cocotb test in test/ can be run with make -B WAVES=1, and then tt-vgaviz can be used: tt-vgaviz tb.vcd (actually in FST format).

How to test

Configuration

- Provide a 25.25 MHz clock on the clk pin (RP2040 should be able to provide this with no jitter). Or if you can try 25.175 MHz instead, but this will have some jitter. YMMV.
- Power the design with at least 1.8V

Main demo

- Set pin ui[0] to 0 to run the default demo.
- Reset the design
- You should see circles moving slowly and large rounding errors:

Test mode

- Set pin ui[0] to 1 to show a test image with color bars.
- Reset the design again if desired
- You should see:

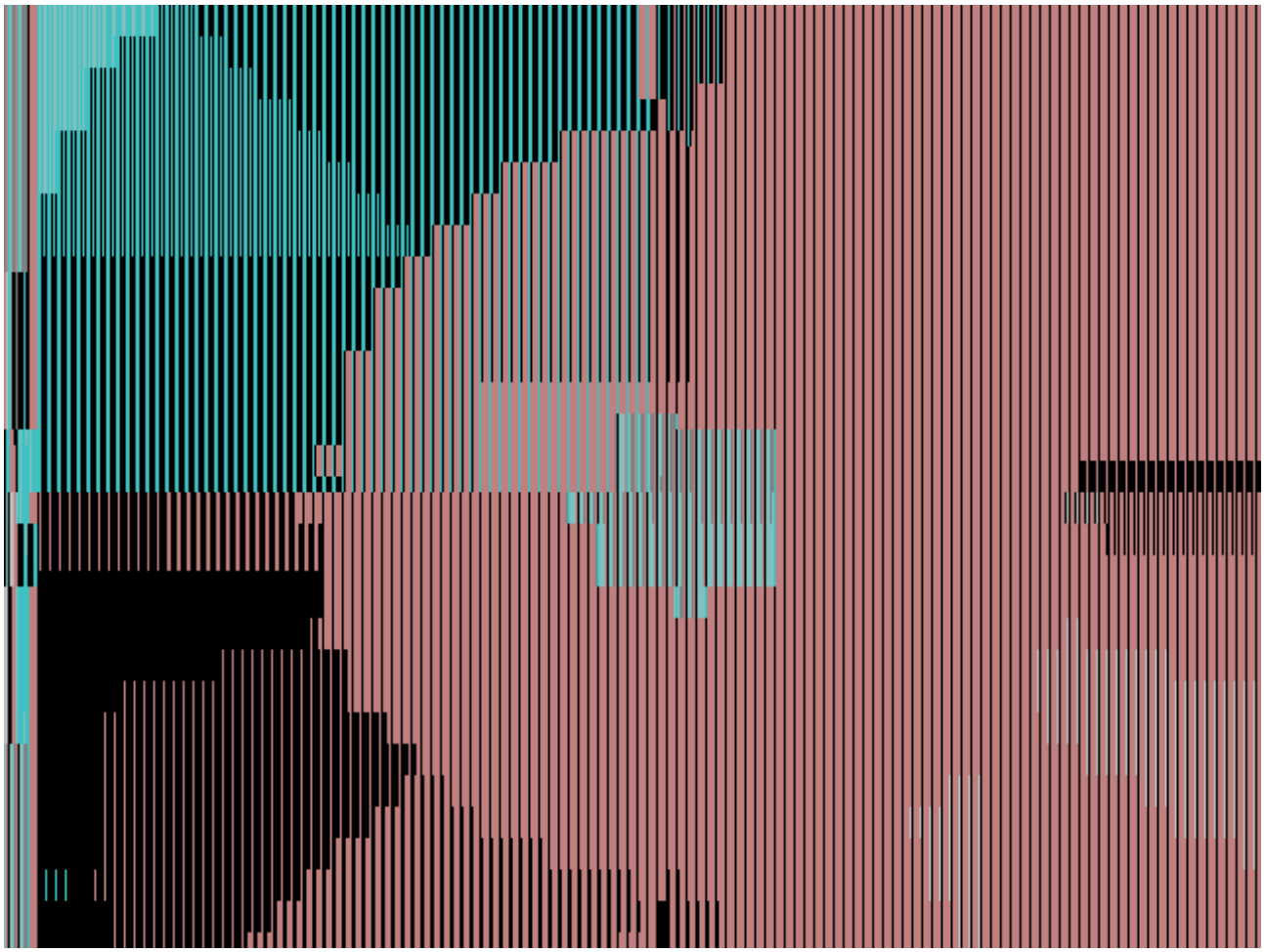
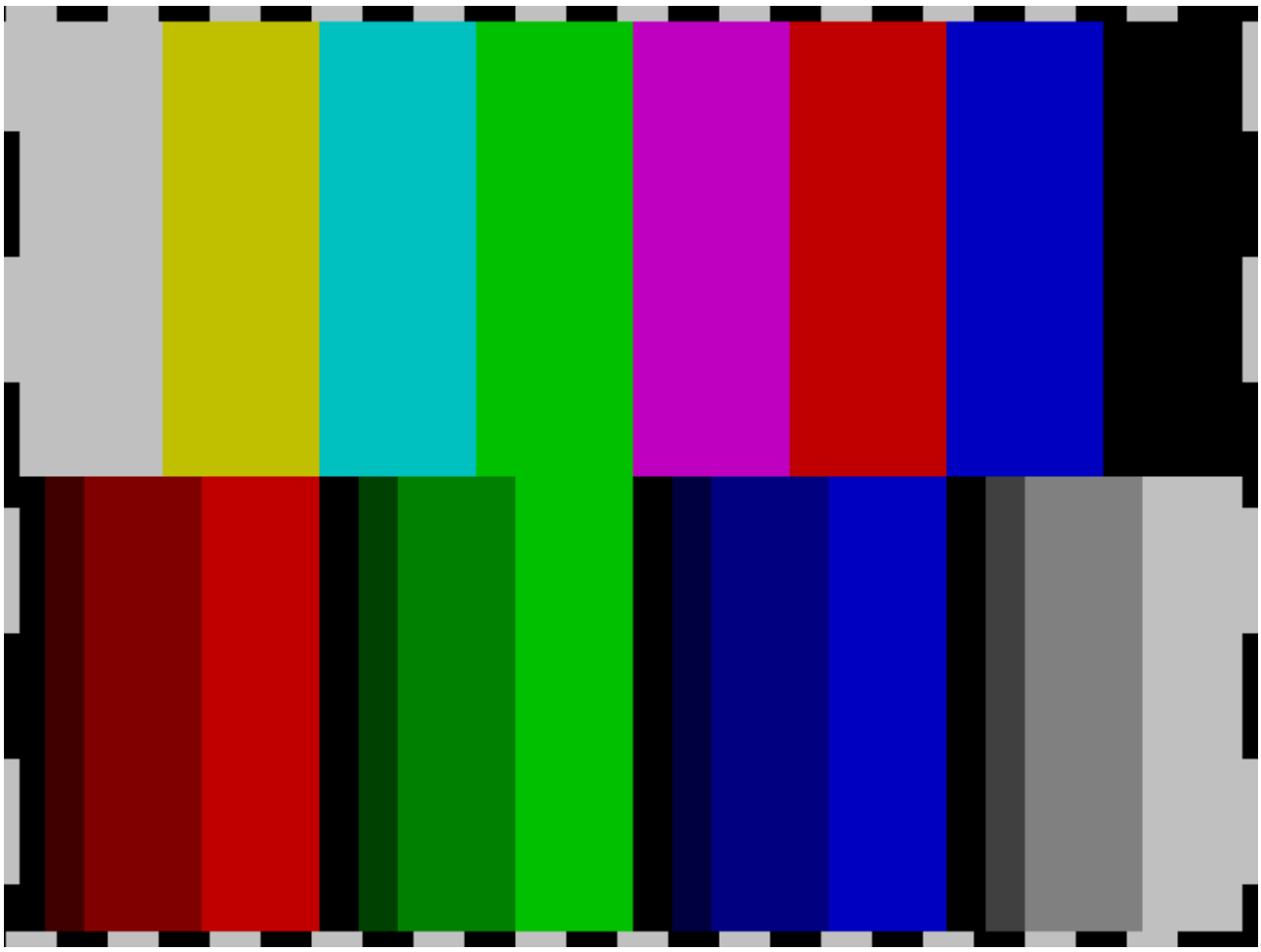


Figure 63: circles



The “audio” out is connected, but is not expected to result in anything audible.

External hardware

Connect according to the Demoscene rules

- VGA output using Leo's VGA PMOD on pins `uo[0-7]`, connected to a monitor supporting 640x480 resolution.
- Audio output using Mike's audio PMOD on `uio[7]`

Pinout

#	Input	Output	Bidirectional
0	test mode (0=no, 1=yes)	r1	
1		g1	
2		b1	
3		vsync	

#	Input	Output	Bidirectional
4		r0	
5		g0	
6		b0	
7		hsync	PWM output

INTERCAL ALU [897]

- Author: Rebecca G. Bettencourt
- Description: An ALU for the five operators of the INTERCAL programming language.
- GitHub repository
- HDL project
- Mux address: 897
- Extra docs
- Clock: 0 Hz

How it works

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

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

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

This ALU has two 32-bit registers, B and A (in no particular order). (These may also be thought of as four 16-bit registers, AL, AH, BL, and BH.) To write a byte to a register, set A0 and A1 to the byte address, set S0 LOW for the A register or HIGH for the B register, set S1 through S3 LOW, set the bidirectional I/O pins to the byte

value, set /WE LOW, then set /WE HIGH again. (Do not set S1 through S3 HIGH when writing, or else something unpredictable will happen, most likely nothing.)

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

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

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

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

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

These operators perform their respective logical operations on all pairs of adjacent bits, the result from the first and last bits going into the first bit of the result. The effect is that of rotating the operand one place to the right and ANDing, ORing, or XORing with its initial value. Thus, #&77 (binary = 1001101) is binary 0000000000000100 = 4, #V77 is binary 1000000001101111 = 32879, and #?77 is binary 1000000001101011 = 32875.

Operations 2, 4, and 6 work on the 16-bit halves of the A register independently, while operations 3, 5, and 7 work on the 32-bit whole of the A register.

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

The interleave operator takes two 16-bit values and produces a 32-bit result by alternating the bits of the operands. Thus, #65535$#0 has the 32-bit binary form 101010....10 or 2863311530 decimal, while #0$#65535 = 0101....01 binary = 1431655765 decimal, and #255$#255 is equivalent to #65535.

Operation 8 returns the interleave of the lower halves of A and B, while operation 9 returns the interleave of the upper halves of A and B. (Should the chip fabrication process allow for it, operation 8½ will, of course, return the interleave of the middle halves of A and B.)

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

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

To help understand the select operator, the INTERCAL manual also provides a helpful circuitous diagram.

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

How to test

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

S	A	B	F
MINGLE16L (8)	0	256	65536
MINGLE16L (8)	65535	0	2863311530
MINGLE16L (8)	0	65535	1431655765
MINGLE16L (8)	255	255	65535
SELECT16 (10)	51	21	5 *
SELECT16 (10)	179	201	9
SELECT16 (10)	201	179	17
SELECT16 (10)	179	179	31
SELECT16 (10)	201	201	15
AND16 (2)	77		4
OR16 (4)	77		32879
XOR16 (6)	77		32875

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

- Not found in the INTERCAL manual.

External hardware

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

Further reading

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

Pinout

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

4-bit minicomputer ALU [899]

- Author: Mike McCann
- Description: this design provides basic arithmetic and logic functions
- GitHub repository
- HDL project
- Mux address: 899
- Extra docs
- Clock: 0 Hz

How it works

The project is a 4-bit ALU section that is usfull in mini and micro computer CPUs.

How to test

This device can be tested by inputting data on the two input ports (A/B), a function code (F0, F1, F2) and observing the output on pins d0, d1, d2, d3.

External hardware

This project was tested uising an Altera FPGA (EP2C20F484C7).

Pinout

#	Input	Output	Bidirectional
0	da0	d0	NEG_ZERO
1	da1	d1	ci_left
2	da2	d2	ci_right
3	da3	d3	COM
4	db0	co_left	F0
5	db1	co_right	F1
6	db2	EQU	F2
7	db3	ZERO	

Hardware UTF Encoder/Decoder [901]

- Author: Rebecca G. Bettencourt
- Description: Converts Unicode code points between UTF-8, UTF-16, and UTF-32.
- GitHub repository
- HDL project
- Mux address: 901
- Extra docs
- Clock: 0 Hz

How it works

This project contains hardware logic to convert between the UTF-8, UTF-16, and UTF-32 encodings for Unicode text.

It will detect and raise an error signal on overlong encodings, out of range code point values, and invalid byte sequences.

(You can optionally disable range checking if you wish to use the original UTF-8 spec that supports values up to 0x7FFFFFFF.)

Basic operation

- In the initial state, all dedicated inputs should be set HIGH.
- At any time, set /RESET (rst_n) LOW and pulse CLK to reset all inputs and outputs to initial state.
- At any time, set /ROUT (input 0) LOW and pulse CLK to seek to the beginning of the output.
- You can set ERRS or /PROPS (input 1) HIGH to get an error status on the dedicated outputs.
- You can set ERRS or /PROPS (input 1) LOW to get character properties on the dedicated outputs.
- You can set CHK (input 2) HIGH to raise an error signal when the code point value is out of range (0x110000).
- You can set CHK (input 2) LOW to ignore out of range code point values and encode/decode values up to 0x7FFFFFFF.
- You can set CBE (input 3) HIGH to specify big endian order for UTF-32 and UTF-16 input and output.
- You can set CBE (input 3) LOW to specify little endian order for UTF-32 and UTF-16 input and output.

Inputting UTF-32

1. Set READ or /WRITE (input 4) LOW.
2. Set /CIO (input 5, character I/O) LOW.
3. Set bidirectional I/O to the first byte of the UTF-32 word and pulse CLK.
4. Set bidirectional I/O to the second byte of the UTF-32 word and pulse CLK.
5. Set bidirectional I/O to the third byte of the UTF-32 word and pulse CLK.
6. Set bidirectional I/O to the fourth byte of the UTF-32 word and pulse CLK.
7. Set /CIO (input 5, character I/O) HIGH.
8. Set READ or /WRITE (input 4) HIGH.
9. If READY (output 0) is HIGH and ERROR (output 5) is LOW, the input and output are both valid.
10. If READY (output 0) is LOW or ERROR (output 5) is HIGH, the input was out of range (0x110000 or, if CHK is LOW, 0x80000000).

Inputting UTF-16

1. Set ERRS or /PROPS (input 1) LOW.
2. Set READ or /WRITE (input 4) LOW.
3. Set /UIO (input 6, UTF-16 I/O) LOW.
4. Set bidirectional I/O to the first byte of the first UTF-16 word and pulse CLK.
5. Set bidirectional I/O to the second byte of the first UTF-16 word and pulse CLK.
6. If HIGHCHAR (output 3) is LOW, skip to step 9.
7. Set bidirectional I/O to the first byte of the second UTF-16 word and pulse CLK.
8. Set bidirectional I/O to the second byte of the second UTF-16 word and pulse CLK.
9. Set /UIO (input 6, UTF-16 I/O) HIGH.
10. Set READ or /WRITE (input 4) HIGH.
11. Set ERRS or /PROPS (input 1) HIGH.
12. If READY (output 0) is HIGH and ERROR (output 5) is LOW, the input and output are both valid.
13. If RETRY (output 1) is HIGH, the first word was a high surrogate but the second word was not a low surrogate. The output will be the high surrogate only; the last word will need to be processed again.

Inputting UTF-8

1. Set READ or /WRITE (input 4) LOW.
2. Set /BIO (input 7, byte I/O) LOW.
3. Set bidirectional I/O to the current byte of the UTF-8 sequence and pulse CLK.

4. Repeat step 3 until READY (output 0) or ERROR (output 5) is HIGH.
5. If READY (output 0) is HIGH and ERROR (output 5) is LOW, the input and output are both valid.
6. If RETRY (output 1) is HIGH, the UTF-8 sequence was truncated (not enough continuation bytes). The output will be the truncated sequence only; the last byte will need to be processed again.
7. If INVALID (output 2) is HIGH, the UTF-8 sequence was a single continuation byte or invalid byte (0xFE or 0xFF).
8. If OVERLONG (output 3) is HIGH, the UTF-8 sequence was an overlong encoding.
9. If NONUNI (output 4) is HIGH, the UTF-8 sequence was out of range (0x110000).

Outputting UTF-32

1. Set READ or /WRITE (input 4) HIGH.
2. Set /CIO (input 5, character I/O) LOW.
3. Pulse CLK and read the first byte of the UTF-32 word from the bidirectional I/O.
4. Pulse CLK and read the second byte of the UTF-32 word from the bidirectional I/O.
5. Pulse CLK and read the third byte of the UTF-32 word from the bidirectional I/O.
6. Pulse CLK and read the fourth byte of the UTF-32 word from the bidirectional I/O.
7. Set /CIO (input 5, character I/O) HIGH.
8. If the UTF-32 word is within range, the input and output are both valid.
9. If the UTF-32 word is not within range, then the input was either incomplete or invalid.

Outputting UTF-16

1. Set READ or /WRITE (input 4) HIGH.
2. If UEOF (output 6) is HIGH, then the input was either incomplete or invalid.
3. Set /UIO (input 6, UTF-16 I/O) LOW.
4. Pulse CLK and read the next byte of the UTF-16 sequence from the bidirectional I/O.
5. Repeat step 4 until UEOF (output 6) is HIGH.
6. Set /UIO (input 6, UTF-16 I/O) HIGH.

Outputting UTF-8

1. Set READ or /WRITE (input 4) HIGH.
2. If BEOF (output 7) is HIGH, then the input was either incomplete or invalid.
3. Set /BIO (input 7, byte I/O) LOW.
4. Pulse CLK and read the next byte of the UTF-8 sequence from the bidirectional I/O.
5. Repeat step 4 until BEOF (output 7) is HIGH.
6. Set /BIO (input 7, byte I/O) HIGH.

Error status

When ERRS or /PROPS (input 1) is HIGH, the dedicated outputs will be:

#	Name	Meaning
0	READY	The input and output are complete sequences.
1	RETRY	The previous input was invalid or the start of another sequence and was ignored. Process the output, reset, and try the previous input again.
2	INVALID	The input and output are invalid.
3	OVERLONG	The UTF-8 input was an overlong sequence.
4	NONUNI	The code point value is out of range (0x110000). (This is set independently of the CHK input; the CHK input only changes whether this counts as an error.)
5	ERROR	Equivalent to (RETRY or INVALID or OVERLONG or (NONUNI and CHK)).

If all of these outputs are LOW, the accumulated input is incomplete and more input is required (underflow).

Character properties

When ERRS or /PROPS (input 1) is LOW, the dedicated outputs will be:

#	Name	Meaning
0	NORMAL	The code point value is valid and not a C0 or C1 control character, surrogate, private use character, or noncharacter.
1	CONTROL	The code point value is valid and a C0 or C1 control character (0x00-0x1F or 0x7F-0x9F).
2	SURROGATE	The code point value is valid and a UTF-16 surrogate (0xD800-0xDFFF).
3	HIGHCHAR	The code point value is valid and either a high surrogate (0xD800-0xDBFF) or a non-BMP character (0x10000).
4	PRIVATE	The code point value is valid and either a private use character (0xE000-0xF8FF, 0xF0000) or the high surrogate of a private use character (0xDB80-0xDBFF).
5	NONCHAR	The code point value is valid and a noncharacter (0xFDD0-0xFDEF or the last two code points of any plane).

If all of these outputs are LOW, there is no valid code point in the output.

How to test

The `test.py` file covers a comprehensive set of test cases which are listed in a separate file to avoid bloating the TT08 manual.

External hardware

Any device that needs to process Unicode text.

Pinout

#	Input	Output	Bidirectional
0	/ROUT	READY; NORMAL	I/O LSB
1	ERRS, /PROPS	RETRY; CONTROL	I/O
2	CHK	INVALID; SURROGATE	I/O
3	CBE, /CLE	OVERLONG; HIGHCHAR	I/O
4	READ, /WRITE	NONUNI; PRIVATE	I/O
5	/CIO	ERROR; NONCHAR	I/O
6	/UIO	UEOF	I/O
7	/BIO	BEOF	I/O MSB

RGB Mixer demo5 [903]

- Author: Matt Venn
- Description: Zero to ASIC demo project
- GitHub repository
- HDL project
- Mux address: 903
- Extra docs
- Clock: 10000000 Hz

How it works

Debounce the inputs, drive an encoder module, and output a PWM signal for each encoder.

How to test

Twist each encoder and the LEDs attached to the outputs should change in brightness.

External hardware

Use 3 digital encoders attached to the first 6 inputs.

Pinout

#	Input	Output	Bidirectional
0	enc0 a	pwm0	
1	enc0 b	pwm1	
2	enc1 a	pwm2	
3	enc1 b		
4	enc2 a		
5	enc2 b		
6	debug bit 0		
7	debug bit 1		

Universal Binary to Segment Decoder [905]

- Author: Rebecca G. Bettencourt
- Description: Decodes various binary codes to various segmented displays.
- GitHub repository
- HDL project
- Mux address: 905
- Extra docs
- Clock: 0 Hz

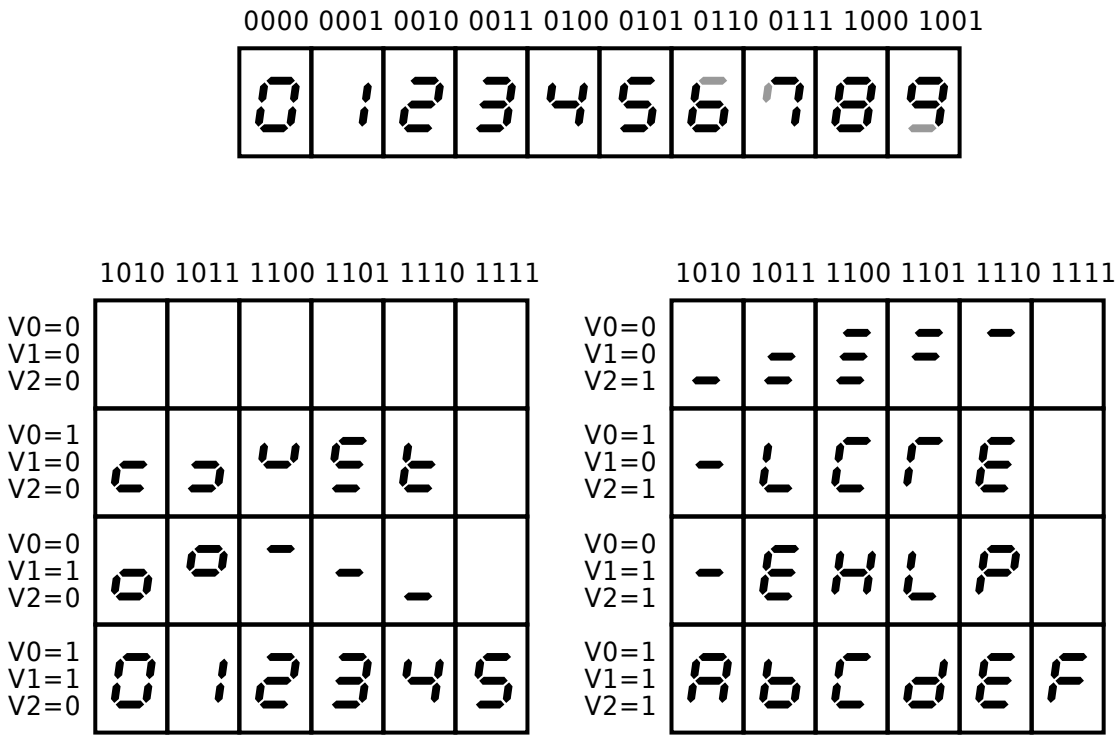
How it works

This project is composed of four modules:

- A BCD to seven segment decoder with a wide variety of options for customizing the appearance of digits
- An ASCII to seven segment decoder with two different "fonts"
- A dual BCD to Cistercian numeral decoder
- A BCV (binary-coded *vigesimal*) to Kaktovik numeral decoder

BCD to seven segment decoder

This mode converts a decimal digit in BCD to its representation on a standard seven segment display. There are inputs that affect the display of the digits 6, 7, and 9, and eight different options for handling out-of-range values. These inputs allow this decoder to match the behavior of just about any other BCD to seven segment decoder, making it *universal*.



The signals used in this mode are:

- **/AL** - Active low. If HIGH, outputs will be HIGH when lit. If LOW, outputs will be LOW when lit.
- **/BI** - Blanking input. If LOW, all segments will be blank regardless of other inputs, including **/LT**.
- **/LT** - Lamp test. When **/BI** is HIGH and **/LT** is LOW, all segments will be lit.
- **/RBI** - Ripple blanking input. If the BCD value is zero and **/RBI** is LOW, all segments will be blank.
- **V0**, **V1**, **V2** - Selects the output when the BCD value is out of range.
- **X6** - When HIGH, the extra segment **a** will be lit on the digit 6.
- **X7** - When HIGH, the extra segment **f** will be lit on the digit 7.
- **X9** - When HIGH, the extra segment **d** will be lit on the digit 9.
- **A**, **B**, **C**, **D** - BCD input from least significant bit **A** to most significant bit **D**.
- **a**, **b**, **c**, **d**, **e**, **f**, **g** - Outputs for a seven segment display.
- **/RBO** - Ripple blanking output. HIGH when BCD value is nonzero or **/RBI** is HIGH.

The pin assignments in this mode are:

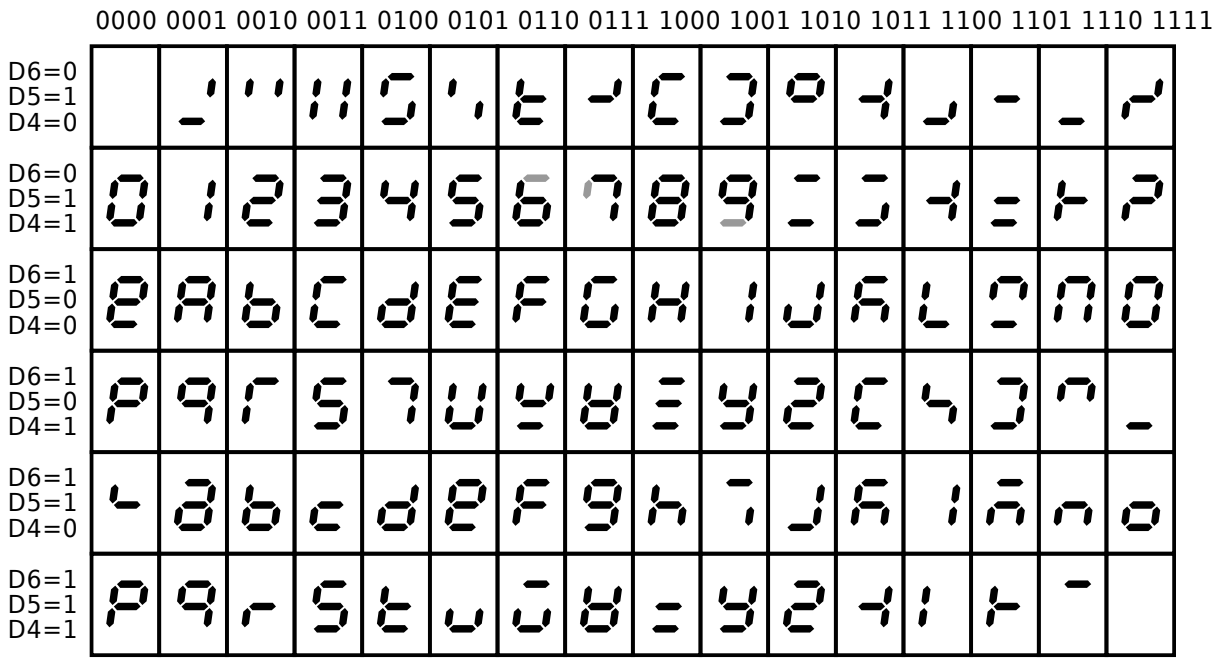
	Dedicated Input	Dedicated Output	Bidirectional
0	A	Segment a	Input - X6

	Dedicated Input	Dedicated Output	Bidirectional
1	B	Segment b	Input - X7
2	C	Segment c	Input - X9
3	D	Segment d	Input - /LT
4	V0	Segment e	Input - /BI
5	V1	Segment f	Input - /AL
6	V2	Segment g	Input - LOW
7	/RBI	/RBO	Input - LOW

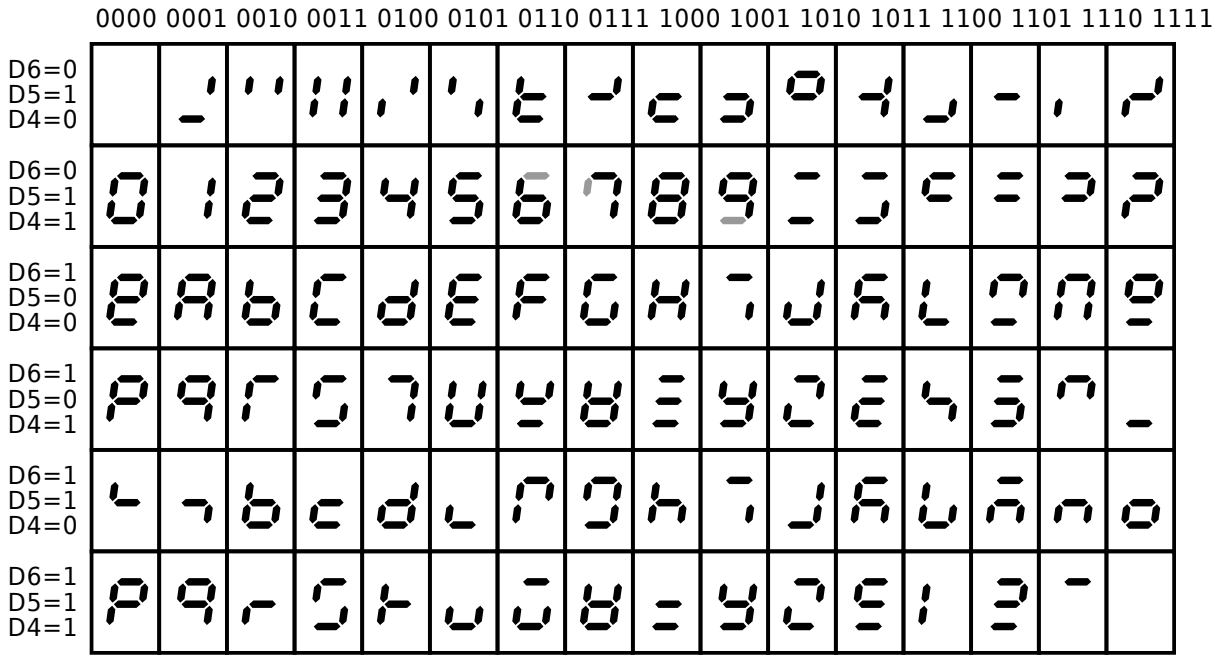
ASCII to seven segment decoder

This mode converts an ASCII character to a representation on a standard seven segment display. Like with the BCD decoder, there are inputs that affect the display of the digits 6, 7, and 9. There are also two choices of "font" and the option to display lowercase letters as uppercase or as lowercase.

FS=0:



FS=1:



The signals used in this mode are:

- **/AL** - Active low. If HIGH, outputs will be HIGH when lit. If LOW, outputs will be LOW when lit.
- **/BI** - Blanking input. If LOW, all segments will be blank regardless of other inputs.
- **FS** - Font select. Selects one of two "fonts."
- **LC** - Lower case. If LOW, lowercase letters will appear as uppercase.
- **X6** - When HIGH, the extra segment **a** will be lit on the digit 6.
- **X7** - When HIGH, the extra segment **f** will be lit on the digit 7.
- **X9** - When HIGH, the extra segment **d** will be lit on the digit 9.
- **D0**…**D6** - ASCII input from least significant bit **D0** to most significant bit **D6**.
- **a**, **b**, **c**, **d**, **e**, **f**, **g** - Outputs for a seven segment display.
- **/LTR** - Letter. LOW when the input is a letter (A…Z or a…z).

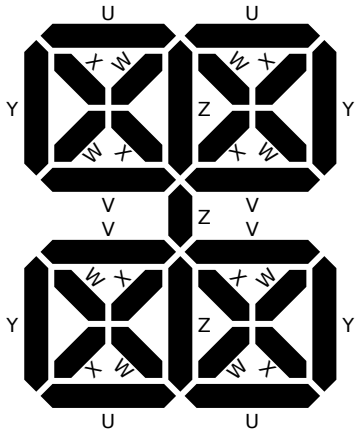
The pin assignments in this mode are:

	Dedicated Input	Dedicated Output	Bidirectional
0	D0	Segment a	Input - X6
1	D1	Segment b	Input - X7
2	D2	Segment c	Input - X9
3	D3	Segment d	Input - FS
4	D4	Segment e	Input - /BI

	Dedicated Input	Dedicated Output	Bidirectional
5	D5	Segment f	Input - /AL
6	D6	Segment g	Input - HIGH
7	LC	/LTR	Input - LOW

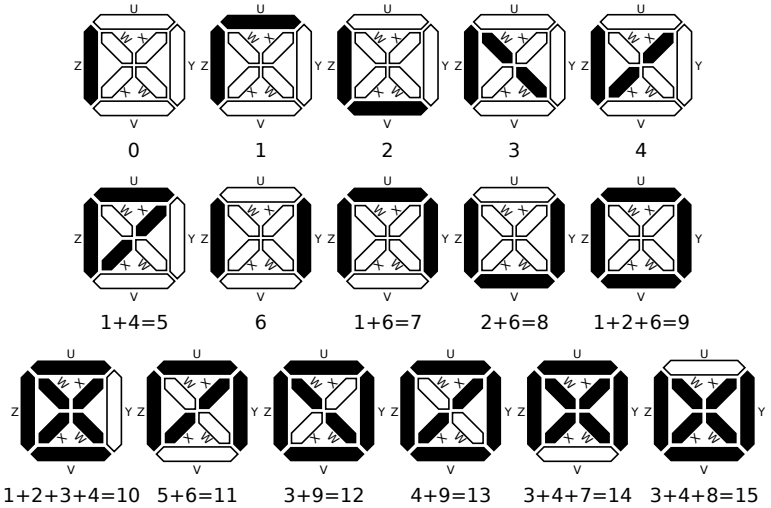
Dual BCD to Cistercian numeral decoder

This mode converts two decimal digits in BCD to their representations on the segmented display for Cistercian numerals shown below.



The patterns produced for each input value are shown below.

Patterns as seen in top right (units) position:



The signals used in this mode are:

- **/AL** - Active low. If HIGH, outputs will be HIGH when lit. If LOW, outputs will be LOW when lit.
- **/BI** - Blanking input. If LOW, all segments will be blank regardless of other inputs, including **/LT1** and **/LT2**.
- **/LT1** - Lamp test for digit 1. When **/BI** is HIGH and **/LT1** is LOW, all segments for digit 1 will be lit.
- **/LT2** - Lamp test for digit 2. When **/BI** is HIGH and **/LT2** is LOW, all segments for digit 2 will be lit.
- **A1**, **B1**, **C1**, **D1** - BCD input for digit 1 from least significant bit **A1** to most significant bit **D1**.
- **A2**, **B2**, **C2**, **D2** - BCD input for digit 2 from least significant bit **A2** to most significant bit **D2**.
- **U1**, **V1**, **W1**, **X1**, **Y1** - Outputs for digit 1 on a Cistercian segmented display.
- **U2**, **V2**, **W2**, **X2**, **Y2** - Outputs for digit 2 on a Cistercian segmented display.

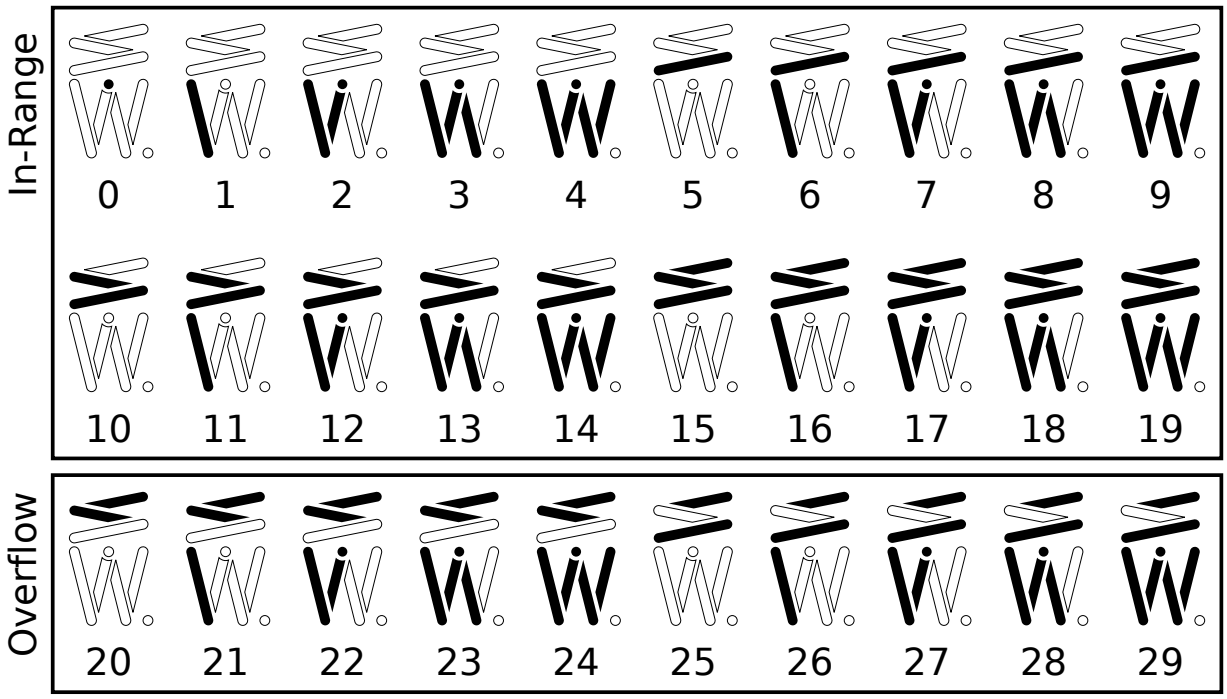
The pin assignments in this mode are:

	Dedicated Input	Dedicated Output	Bidirectional
0	A1	Segment U1	Output - Y1
1	B1	Segment U2	Output - Y2
2	C1	Segment V1	Input - /LT1
3	D1	Segment V2	Input - /LT2
4	A2	Segment W1	Input - /BI
5	B2	Segment W2	Input - /AL
6	C2	Segment X1	Input - LOW
7	D2	Segment X2	Input - HIGH

BCV to Kaktovik numeral decoder

This mode converts a *vigesimal* (base 20) digit in BCV (binary-coded vigesimal) to its representation on the segmented display for Kaktovik numerals shown below.

The patterns produced for each input value are shown below.



The signals used in this mode are:

- **/AL** - Active low. If HIGH, outputs will be HIGH when lit. If LOW, outputs will be LOW when lit.
- **/BI** - Blanking input. If LOW, all segments will be blank regardless of other inputs, including **/LT**.
- **/LT** - Lamp test. When **/BI** is HIGH and **/LT** is LOW, all segments will be lit.
- **/RBI** - Ripple blanking input. If the BCV value is zero and **/RBI** is LOW, all segments will be blank.
- **/VBI** - Overflow blanking input. If the BCV value is out of range and **/VBI** is LOW, all segments will be blank.
- **A**, **B**, **C**, **D**, **E** - BCV input from least significant bit **A** to most significant bit **E**.
- **a**, **b**, **c**, **d**, **e**, **f**, **g**, **h** - Outputs for a Kaktovik segmented display.
- **/RBO** - Ripple blanking output. HIGH when BCV value is nonzero or **/RBI** is HIGH.
- **V** - Overflow. HIGH when BCV value is out of range (greater than or equal to 20).

The pin assignments in this mode are:

	Dedicated Input	Dedicated Output	Bidirectional
0	A	Segment a	Output - h
1	B	Segment b	Output - V
2	C	Segment c	

	Dedicated Input	Dedicated Output	Bidirectional
3	D	Segment d	Input - /LT
4	E	Segment e	Input - /BI
5		Segment f	Input - /AL
6	/VBI	Segment g	Input - HIGH
7	/RBI	/RBO	Input - HIGH

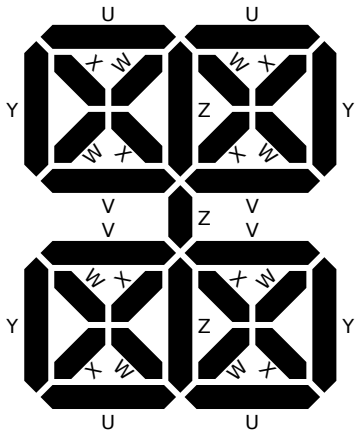
How to test

The `test` directory includes extensive tests for each of the four modules.

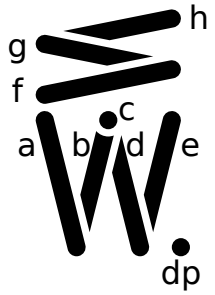
External hardware

For the BCD and ASCII modes, a standard seven-segment display is used.

For the Cistercian mode, a segmented display like the one below is used. There are design files for such a display here.



For the Kaktovik mode, a segmented display like the one below is used. There are design files for such a display here.



Pinout

#	Input	Output	Bidirectional
0	A; D0; A1; A	Segment a; U1; a	X6; X6; Y1; h
1	B; D1; B1; B	Segment b; U2; b	X7; X7; Y2; V
2	C; D2; C1; C	Segment c; V1; c	X9; X9; /LT1; -
3	D; D3; D1; D	Segment d; V2; d	/LT; FS; /LT2; /LT
4	V0; D4; A2; E	Segment e; W1; e	/BI (blanking input)
5	V1; D5; B2; -	Segment f; W2; f	/AL (active low)
6	V2; D6; C2; /VBI	Segment g; X1; g	M0 (mode select)
7	/RBI; LC; D2; /RBI	/RBO; /LTR; X2; /RBO	M1 (mode select)

Simon Says memory game [907]

- Author: Uri Shaked
- Description: Repeat the sequence of colors and sounds to win the game
- GitHub repository
- HDL project
- Mux address: 907
- Extra docs
- Clock: 50000 Hz

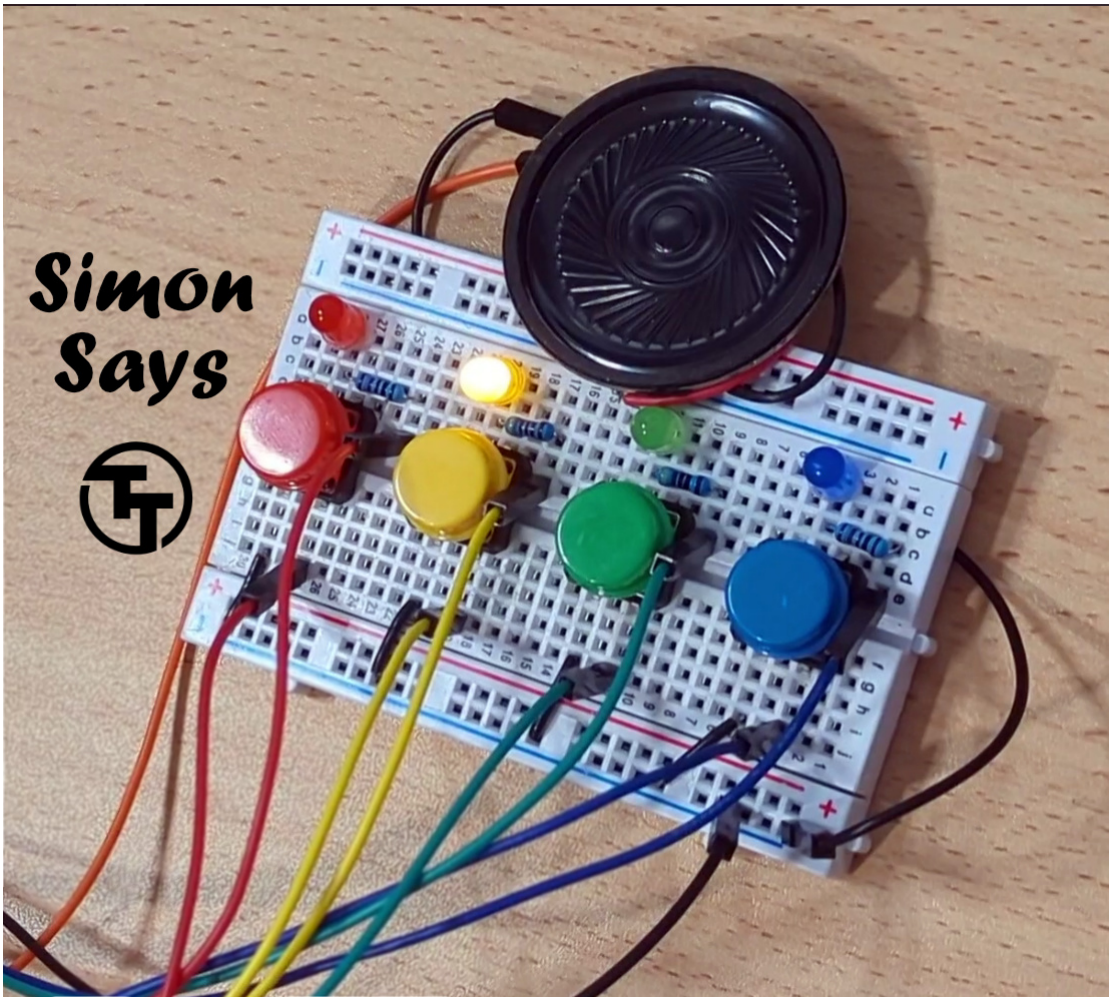


Figure 64: Simon Says Game

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/397436605640509441 (including wiring diagram).

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

#	Input	Output	Bidirectional
2	btn3	led3	seg_c
3	btn4	led4	seg_d
4	seginv	speaker	seg_e
5		dig1	seg_f
6		dig2	seg_g
7			

Asynchronous Multiplier [909]

- Author: Tommy Thorn
- Description: An asynchronous multiplier
- GitHub repository
- HDL project
- Mux address: 909
- Extra docs
- Clock: 50000000 Hz

How it works

This design emits a sequence of `r = x^2+x`, for x=0,1,2,... on the outputs using the handshake protocol (tie ack to req to get free running sequence). Well, in truth, we use 26-bits of internal precision, but we only have 15-bits for outputs, we what is actually emitted is `r ^ (r &gt;&gt; 15)`.

The very naive algorithm (with the body unrolled once) is

```
x = 0
loop:
  x = x + 1
  a = b = c = x
  while b != 0:
    if (b & 1) == 1:
      c += a
    a *= 2
    b /= 2
    if (b & 1) == 1:
      c += a
    a *= 2
    b /= 2
  output (c)
```

which was hand translated (roughly following Introduction to Asynchronous Circuit Design) into a token flow graph:

Note, I use a simpler, less expensive, construction for the conditional iteration as having independent control-flow for the trivial condition is overkill.

The graph was realized using four-phase bundled data. Alas, I'm still working on the timing analysis, so the inserted delays are (hopefully) way oversized.

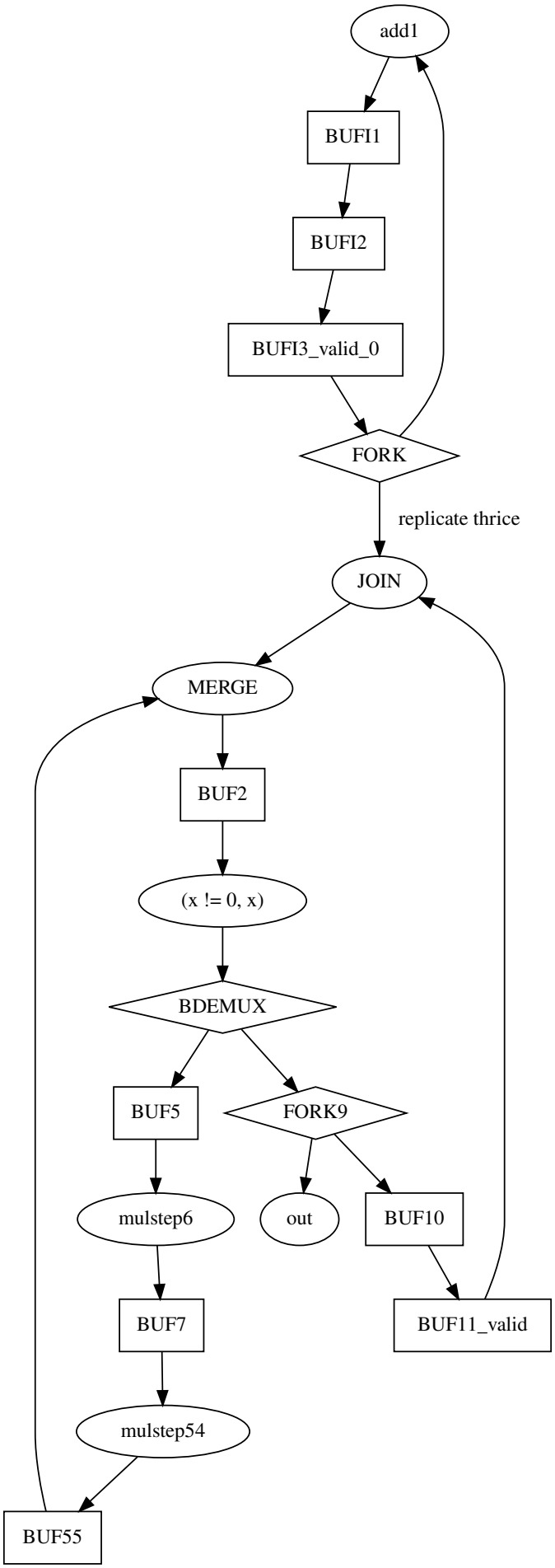


Figure 65: token-flow graph

How to test

The data is presented using the standard 4-phase (RTZ) protocol (idle, Req, Req+Ack, Ack, idle, ...). To get a continuous stream, simply tie ack to req. The values expected are 0, 2, 6, ..., x(x+1)

External hardware

A logic analyzer is convenient to pick up the values on the outputs, but default RP2040 works fine.

Pinout

#	Input	Output	Bidirectional
0	ack	req	result_7
1		result_0	result_8
2		result_1	result_9
3		result_2	result_10
4		result_3	result_11
5		result_4	result_12
6		result_5	result_13
7		result_6	result_14

Supermic [910]

- Author: Armaan Gomes, Asmi Sawant, Ria Saheta, Vikhaash Kanagavel Chithra, Morgan Packard, Sanjay Ravishankar
- Description: A 8 channel customizable beamforming signal processor
- GitHub repository
- HDL project
- Mux address: 910
- Extra docs
- Clock: 0 Hz

How it works

Cool stuff makes cool stuff happen Explain how your project works

How to test

Plug cool stuff into the chip and it will output cool stuff Explain how to use your project

External hardware

You need some cool microphones and a cool clock generator and a cool i2s reciever List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Pinout

#	Input	Output	Bidirectional
0	Bit Clock (3.072 Mhz)	PCM Out Mic 0	Delay Select 0
1	LR Clock (48Khz)	PCM Out Mic 1	Delay Select 1
2	PDM Input Mics 0,1	PCM Out Mic 2	Delay Select 2
3	PDM Input Mics 2,3	PCM Out Mic 3	Delay Select 3
4	PDM Input Mics 4,5	PCM Out Mic 4	Delay Select 4
5	PDM Input Mics 6,7	PCM Out Mic 5	Beamformed PCM Output
6		PCM Out Mic 6	Mic Clock
7		PCM Out Mic 7	

VGA Tiny Logo (1 tile) [911]

- Author: Renaldas Zioma
- Description: Large 480x480 pixels Tiny Tapeout logo with bling and dithered colors crammed into 1 tile!
- GitHub repository
- HDL project
- Mux address: 911
- Extra docs
- Clock: 25175000 Hz

How it works

Compressed VGA Logo

How to test

Connect to VGA monitor

External hardware

TinyVGA PMOD, VGA monitor

Pinout

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

Dice [961]

- Author: ZHU QUANHAO
- Description: after you press the button the system will generate a random number from 0-F
- GitHub repository
- HDL project
- Mux address: 961
- Extra docs
- Clock: 50000000 Hz

How it works

It genreate number by inverter ring

How to test

press the button to capture number

External hardware

2 LED display

Pinout

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

Lab and Lectures SoC [963]

- Author: Aloke Kumar Das
- Description: A tiny SoC comprising of a cpu, memory and SPI protocol
- GitHub repository
- HDL project
- Mux address: 963
- Extra docs
- Clock: 50 Hz

How it works

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

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

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

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

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

and check if zero, skip the next instructio Indirect addressing mode of all the above instruction are also supported

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

How to test

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

External hardware

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

Pinout

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

#	Input	Output	Bidirectional
7	keyboard 7	display 7	cpu display flag

asic design is my passion [965]

- Author: Nicholas Junker
- Description: baby's first asic - cheeky little text meme
- GitHub repository
- HDL project
- Mux address: 965
- Extra docs
- Clock: 25175000 Hz

How it works

Real, real bad graphic design & fun shapes bouncing around on the screen.

How to test

Hook up to VGA monitor using the TinyTapeout VGA module.

External hardware

Tiny VGA Pmod peripheral!

Pinout

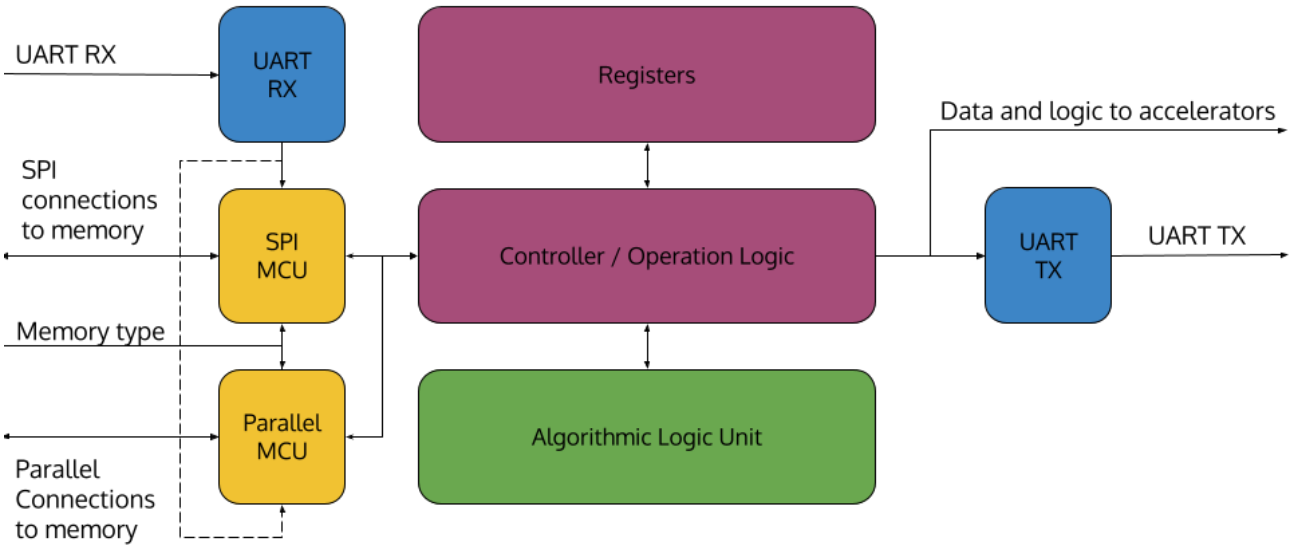
#	Input	Output	Bidirectional
0		R1	
1		G1	
2		B1	
3		VSync	
4		R0	
5		G0	
6		B0	
7		HSync	

Zoom Zoom [966]

- Author: Justin T, Andrew H, Simon Y, Kellen Y, Vallabh A, Nicole C
- Description: Custom Cpu with custome external memory bus and sha-3 and CORDIC accelerators
- GitHub repository
- HDL project
- Mux address: 966
- Extra docs
- Clock: 60000000 Hz

What is Zoom Zoom?

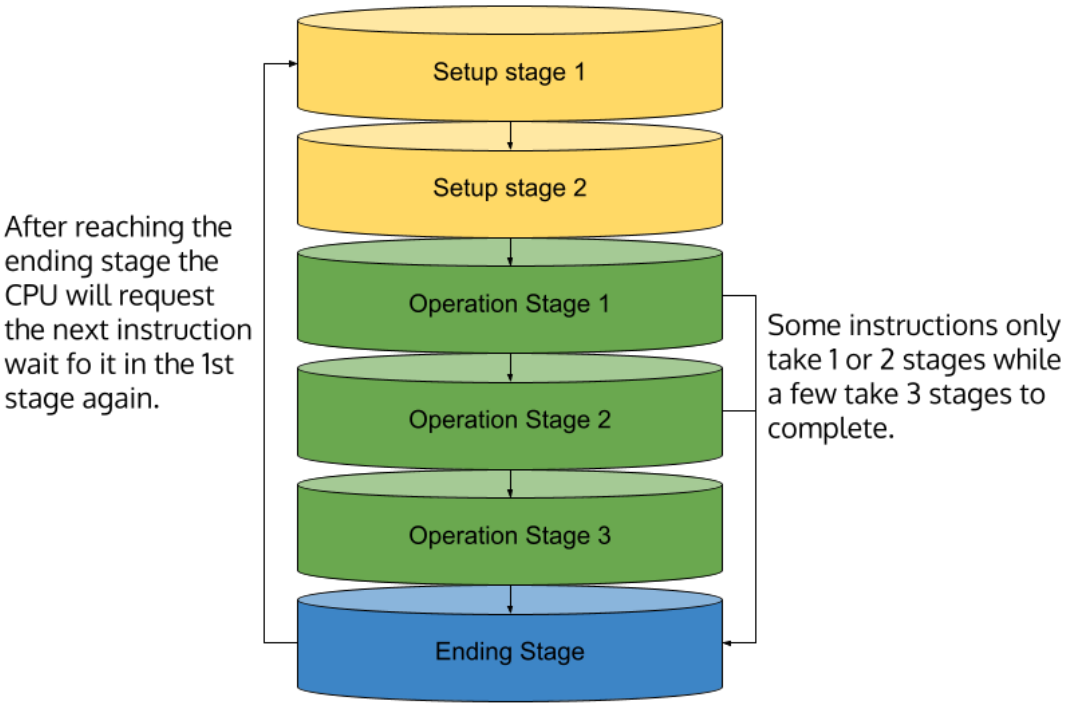
Zoom Zoom is a custom, 16-bit, barebones CPU. We store memory externally using either a custom parallel connection or SPI. We also have a simple UART protocal implemented on the CPU as well as numerous accelerators(that may not be included in the final design due to size constraints).(Link to Document with helpful coding info)



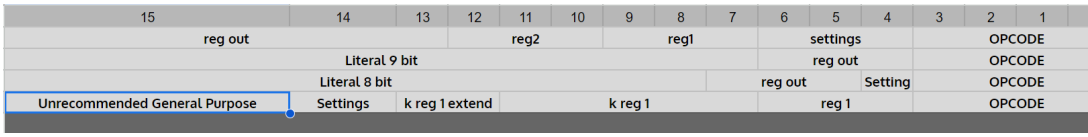
Detailed List of Features

- Custom Architecture and ISA
  - 16-bit instructions
  - 5 types of instructions
- 6 general purpose registers
  - 1 flag register
  - 1 zero register
- UART Interface
- SPI and Custom Parallel Memeroy Interface
  - 16 bit memory address
  - supports up to 65536 memory addresses(2^16)
- Flexible design easy integration of accelerators as instructions

The Architecture



Instruction Layout



General Instructions

Instruction	Name	Type	Opcode	Settings	Description
nop	No Operation	0	0000		
ld	Load	A	0101	0	reg out = mem[mem[inst addr + 1]]
ldr	Load Register	A	0101	1	reg out = mem[reg1]
str	Store	A	0110	0	mem[mem[inst addr + 1]] = reg2
strr	Store Register	A	0110	1	mem[reg1] = reg2
ldi	Load Immediate	L	0111		reg out = L9[7:15]

ALU Instructions

Instruction	Name	Type	Opcode	Settings	Description
add	Add	A	0001	000	reg out = reg1 + reg2
sub	Subtract	A	0001	001	reg out = reg1 - reg2
mult	Multiply	A	0001	010	reg out = reg1[0:7] * reg 2[0:7]
nand	NAND	A	0001	011	reg out = !(reg1 & reg2)
addi	Add Immediate	I	0010	0	reg out = register 2 + L8[0:7]
multi	Multiply Immediate	I	0010	1	reg out = register 2 * L8[0:7]
shl	Shift Legt	A	0001	100	reg out = reg1 « 0
shr	Shift Right	A	0001	101	reg out = reg1 » 0

Branching Instructions

Instruction	Name	Type	Opcode	Settings	Description
jmp	Jump	A	0100	000	inst addr = reg1
jmpz	Jump if Zero	A	0100	001	reg_out = inst addr; if (ZF) { inst addr = reg1 }
jmpg	Jump if Greater	A	0100	010	reg_out = inst addr; if (GF) { inst addr = reg1 }
jmpe	Jump if Equal	A	0100	111	reg_out = inst addr; if (EF) { inst addr = reg1 }
jmpl	Jump if Less	A	0100	011	reg_out = inst addr; if (!GF) { inst addr = reg1 }

Instruction	Name	Type	Opcode	Settings	Description
jmpm	Jump if Memory Flagged	A	0100	100	reg_out = inst addr; if (MF) { inst addr = reg1 }
jmpu	Jump if UART Flagged	A	0100	101	reg_out = inst addr; if (UF) { inst addr = reg1 }
jmpi	Jump Immediate	A	0100	110	inst addr = mem[inst addr + 1]

Programming the CPU

> :warning: **Memory Address 769 is reserved**: The Assembler does not give a warning currently!

To assemble, we use custoasm with installation instructions here. We recommend installation via rust's package manager by running `cargo install customasm`. You can then compile an assembly file by running `customasm -o &lt;outputfilename&gt; &lt;filename&gt;`. The format for the assembly file is to add `#include &quot;x3q16_ruleset.asm&quot;` to the top of each .asm file as well as that file which is located here. Instruction memory and General Purpose are all located in the same place. Thus, to store general values in memory, just jump to wherever you store it in memory.

Accelerators

> :warning: **Many are still a work in progress or aren't supported by the assembler**

Keccakf1600 Approximately 50% of the computational time for the Kyber Algorithm is hashing needed for random number generation. The Kyber algorthm uses SHA-3 and SHAKE algorithms to generate cryptographically secure random polynomials and numbers. Both of these algorithm rely on the keccakf1600 state permutation which target to accelerate. More information on the keccak algorithm can be found here and the kyber algorithm here.

The branch `keccak_integration` holds a complete state permuation accelerator however this is not included in main since it's too big to fit for tinytapeout. A smaller accelerator is currently being worked on.

How to test

Generate the binary file from test/x3q16 and load it into memory. Reset the chip and see if anything is written in memory.

External hardware

Either a SPI ram chip or a MCU emulator of parallel storage with custom protocol

Pinout

#	Input	Output	Bidirectional
0	lower_byte_in	write_enable	DATA0
1	upper_byte_in	register_enable	DATA1
2	rx	read_enable	DATA2
3	IN3	lower_bit	DATA3
4	IN4	tx	DATA4
5	IN5	upper_bit	DATA5
6	IN6	OUT6	DATA6
7	IN7	OUT7	DATA7

Crispy VGA [967]

- Author: James Meech
- Description: The scrolling VGA example from the vga playground but as you set more inputs high it gets successively more crispy
- GitHub repository
- HDL project
- Mux address: 967
- Extra docs
- Clock: 0 Hz

How it works

This project "Crispy VGA" takes as input the output of a standard tiny tapeout VGA project. Crispy VGA then adds a programmable amount of random noise to the VGA signal and passes it through to the output. The uio_in[0] input sets the noise on the hsync signal. The uio_in[1] input sets the noise on the B signal. The uio_in[2] input sets the noise on the G signal. The uio_in[3] input sets the noise on the R signal. The uio_in[4] input sets the noise on the vsync. The uio_in[5] signal sets the noise level applied to the R, G, and B wires to high or low. The uio_in[0:5] inputs set the succesively increasing noise levels on the audio signal.

How to test

Plug an existing tiny tapeout VGA project into the input of this design. Plug the output of this design into a standard VGA input monitor. Power up both tiny tapeout boards and select the appropriate control bits for the level of noise that you want to see on the output VGA signal.

External hardware

You will need a VGA input monitor and a computer that can output a VGA signal or a second tiny tapeout ASIC with a working VGA design that follows the standard pinout. You will also need two tiny tapeout VGA adapters and two VGA cables.

Pinout

#	Input	Output	Bidirectional
0	R[1] vga input	R[1] vga input	Crispy input bit 0 that toggles the noise on the hsync signal on or off. Also adds one bit of noise to audio.
1	G[1] vga input	G[1] vga input	Crispy input bit 1 toggles the noise on the B signal on or off. Also adds one bit of noise to audio.
2	B[1] vga input	B[1] vga input	Crispy input bit 2 toggles the noise on the G signal on or off. Also adds one bit of noise to audio.
3	vsync vga input	vsync vga input	Crispy input bit 3 toggles the noise on the R signal on or off. Also adds one bit of noise to audio.
4	R[0] vga input	R[0] vga input	Crispy input bit 4 that toggles the noise on the vsync signal on or off. Also adds one bit of noise to audio.
5	G[0] vga input	G[0] vga input	Crispy input bit 5 that sets the noise level applied to the R, G, and B wires to high or low. Also adds one bit of noise to audio.
6	B[0] vga input	B[0] vga input	Audio input bit
7	hsync vga input	hsync vga input	Audio output bit

Calculator [969]

- Author: JING Shuangyu
- Description: A calculator do basic calculation
- GitHub repository
- HDL project
- Mux address: 969
- Extra docs
- Clock: 10000000 Hz

How it works

the calculator can support addition, subtraction, multiplication and division on positive integer number.

How to test

The project can be tested by entern input through the keypad and then check whether the display shows the desire output.

External hardware

The calculator need a 4x4 matrix keypad for input and a 3-digit seven segment display to show the calculated result.

Pinout

#	Input	Output	Bidirectional
0	ROW_1	sseg_A	0
1	ROW_2	sseg_B	E_1
2	ROW_3	sseg_C	E_2
3	ROW_4	sseg_D	E_3
4		sseg_E	COL_1
5		sseg_F	COL_2
6		sseg_G	COL_3
7		sseg_dp	COL_4

mulmul [970]

- Author: JJ Wong
- Description: Small 4-bit vector multiplication engine
- GitHub repository
- HDL project
- Mux address: 970
- Extra docs
- Clock: 0 Hz

How it works

Write the registers and vector length and accumulator value (optional) into the chip's registers using the read and write opcodes, then run the system with the run opcode. The vectors will be multiplied and summed together in two clock cycles and output an 8-bit word.

Input words are 4 bits wide. Write the length of the 4-bit vectors you want to multiply into address 0. The vectors should be in words 1-32. Word 1 will be multiplied by word 17, etc. The result will be accumulated into words 33-34 (8 bits).

How to test

You can run the testbench tests in the test dir.

External hardware

Will be programmed by RP2040. No other external hardware.

Pinout

#	Input	Output	Bidirectional
0	addr[0]	out[0]	data[0]
1	addr[1]	out[1]	data[1]
2	addr[2]	out[2]	data[2]
3	addr[3]	out[3]	data[3]
4	addr[4]	out[4]	state[0]
5	addr[5]	out[5]	state[1]
6	op[0]	out[6]	

#	Input	Output	Bidirectional
7	op[1]	out[7]	

DDR throughput and flop aperature test [971]

- Author: Eric Smith
- Description: Grab data on every edge of clock with varying pos pulse width
- GitHub repository
- HDL project
- Mux address: 971
- Extra docs
- Clock: 0 Hz

How it works

Badly probably.

Use a positive edge detector on the clock and its compliment. Or together those dectors to get 2 positive pulses per period or a 2x clock. Vary clk 2x pos pulse width by varying number of inv per detect.

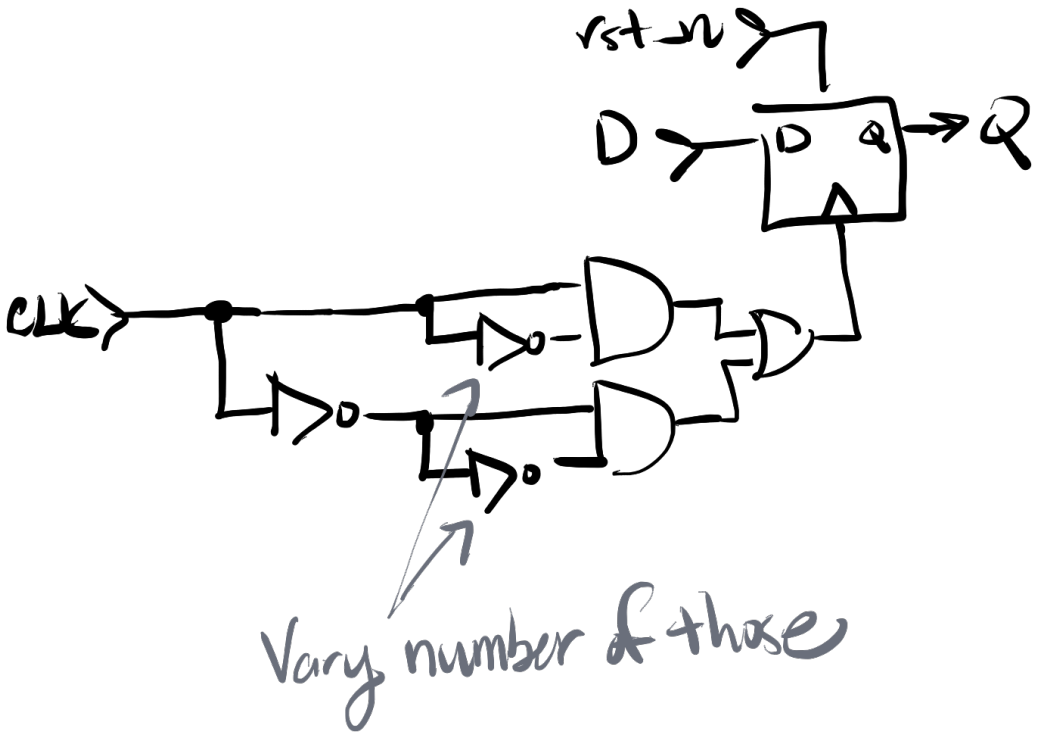


Figure 66: Concept Diagram

How to test

Carefully.

External hardware

Analog Discovery 3

Pinout

#	Input	Output	Bidirectional
0	pulse = 1 inv	q for pulse = 1 inv	
1	pulse = 3 inv	q for pulse = 3 inv	
2	pulse = 5 inv	q for pulse = 5 inv	
3	pulse = 7 inv	q for pulse = 7 inv	
4		q for normal flop	
5		1	
6		1	
7		clk	

VGA Scroller [973]

- Author: FavoritoHJS
- Description: Scrolls across a very pixelated cityscape
- GitHub repository
- HDL project
- Mux address: 973
- Extra docs
- Clock: 25000000 Hz

How it works

The terrain is based on an LFSR, using the deterministic randomness of one to generate each layer of the city.

How to test

Set Clock to 25.18MHz, and use a Tiny VGA carrier board for video.

External hardware

This project requires a Tiny VGA carrier board to display video.

Pinout

#	Input	Output	Bidirectional
0		Rh	
1		Gh	
2		Bh	
3		vsync	
4		Rl	
5		Gl	
6		Bl	
7		hsync	

DDC [974]

- Author: Armaan Gomes
- Description: Converts I2S input to PDM output
- GitHub repository
- HDL project
- Mux address: 974
- Extra docs
- Clock: 0 Hz

How it works

It uses an inverted cic filter and modulator to convert an i2s signal to pdm Explain how your project works

How to test

Can and I2s output and a pdm input deive Explain how to use your project

External hardware

I2s Output device and pdm input device

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

Pinout

#	Input	Output	Bidirectional
0	Bit Clock (3.072 Mhz)	PCM Out Mic 0	Delay Select 0
1	LR Clock (48Khz)	PCM Out Mic 1	Delay Select 1
2	PDM Input Mics 0,1	PCM Out Mic 2	Delay Select 2
3	PDM Input Mics 2,3	PCM Out Mic 3	Delay Select 3
4	PDM Input Mics 4,5	PCM Out Mic 4	Delay Select 4
5	PDM Input Mics 6,7	PCM Out Mic 5	Beamformed PCM Output
6		PCM Out Mic 6	Mic Clock
7		PCM Out Mic 7	

Glyph Mode [975]

- Author: James Ross
- Description: Submission for VGA Demoscene
- GitHub repository
- HDL project
- Mux address: 975
- Extra docs
- Clock: 25175000 Hz

How it works

This is a standalone VGA demo that runs with or without input. It will accept two pins `ui_io[0]` and `ui_io[1]` for palette color selection:

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

How to test

Plug into a VGA monitor and select this circuit to test

External hardware

Requires the TinyVGA PMOD

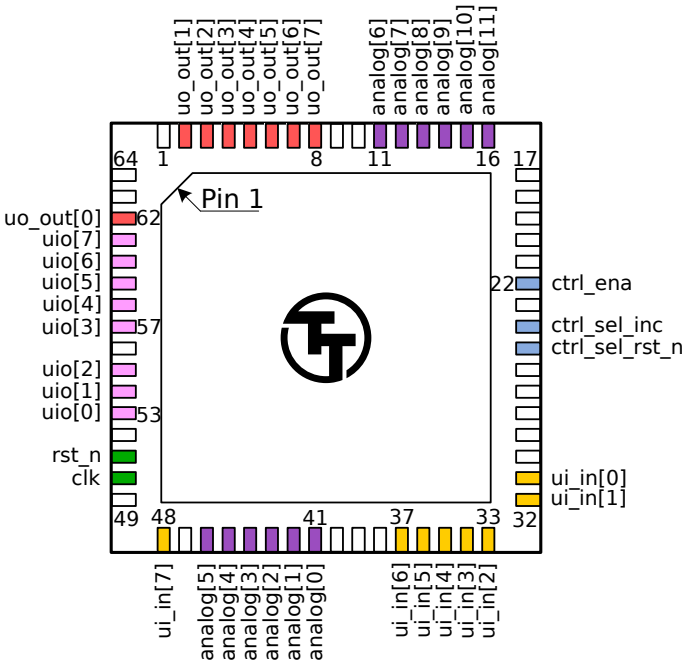
Pinout

#	Input	Output	Bidirectional
0	Palette 0	R1	
1	Palette 1	G1	
2		B1	
3		VSync	
4		R0	

#	Input	Output	Bidirectional
5		G0	
6		B0	
7		HSync	

Pinout

The chip is packaged in a 64-pin QFN package. The pinout is shown below.



Bottom View

Figure 67: Pinout

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

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:

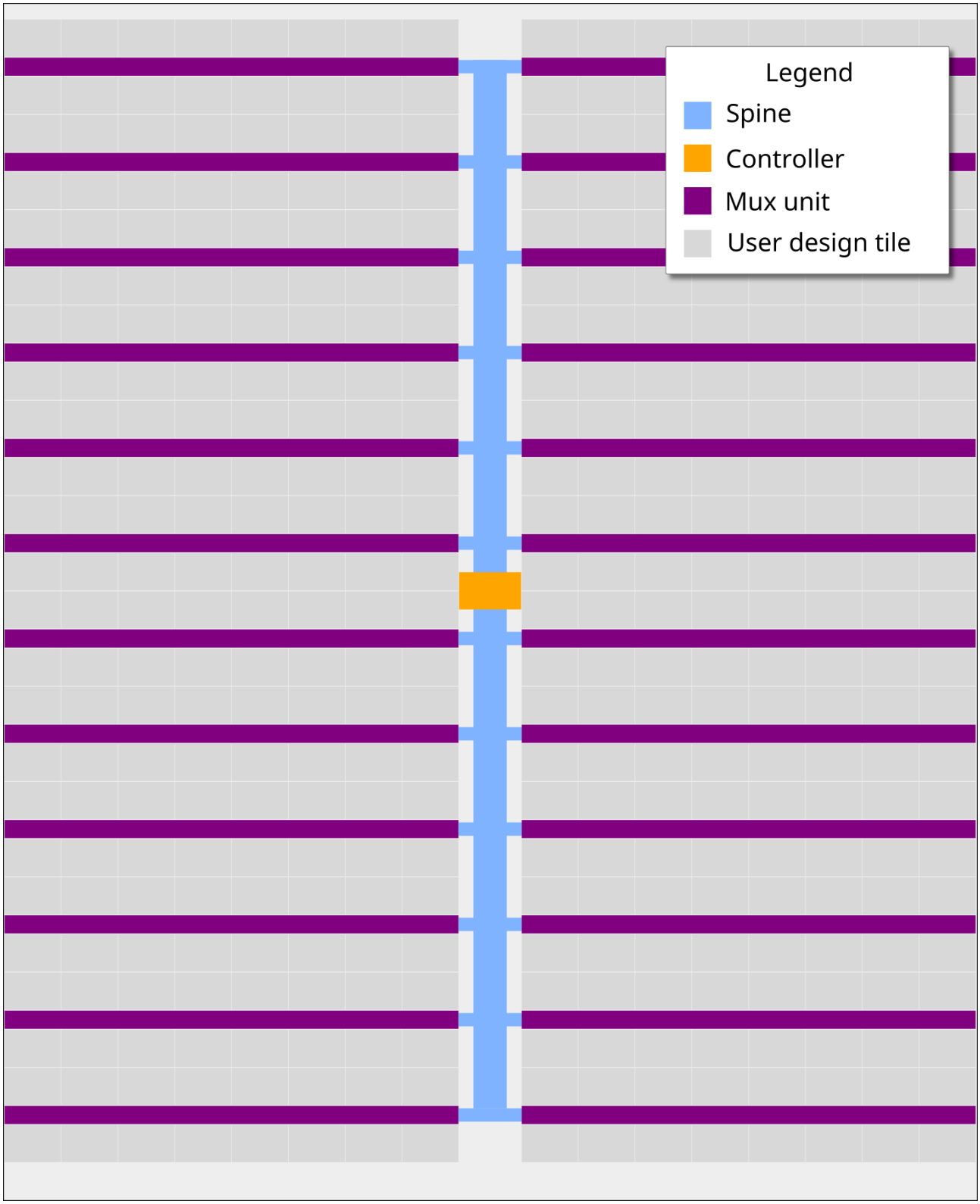


Figure 68: Mux Diagram

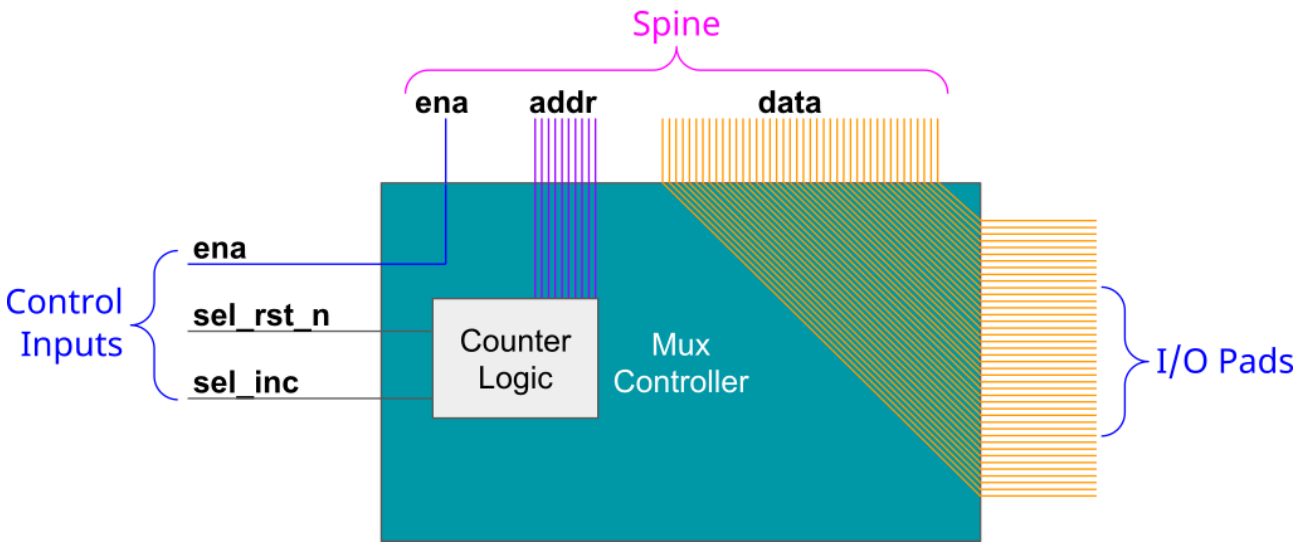


Figure 69: Mux Controller Diagram

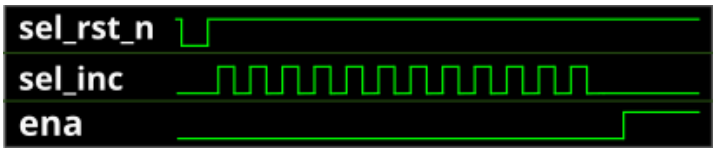


Figure 70: Mux signals for activating the design at address 12

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/3643478076
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

mprj_io pin	Function	Signal	QFN64 pin
0	Input	ui_in[0]	31
1	Input	ui_in[1]	32
2	Input	ui_in[2]	33
3	Input	ui_in[3]	34
4	Input	ui_in[4]	35

mprj_io pin	Function	Signal	QFN64 pin
5	Input	ui_in[5]	36
6	Input	ui_in[6]	37
7	Analog	analog[0]	41
8	Analog	analog[1]	42
9	Analog	analog[2]	43
10	Analog	analog[3]	44
11	Analog	analog[4]	45
12	Analog	analog[5]	46
13	Input	ui_in[7]	48
14	Input	clk †	50
15	Input	rst_n †	51
16	Bidirectional	uio[0]	53
17	Bidirectional	uio[1]	54
18	Bidirectional	uio[2]	55
19	Bidirectional	uio[3]	57
20	Bidirectional	uio[4]	58
21	Bidirectional	uio[5]	59
22	Bidirectional	uio[6]	60
23	Bidirectional	uio[7]	61
24	Output	uo_out[0]	62
25	Output	uo_out[1]	2
26	Output	uo_out[2]	3
27	Output	uo_out[3]	4
28	Output	uo_out[4]	5
29	Output	uo_out[5]	6
30	Output	uo_out[6]	7
31	Output	uo_out[7]	8
32	Analog	analog[6]	11
33	Analog	analog[7]	12
34	Analog	analog[8]	13
35	Analog	analog[9]	14
36	Analog	analog[10]	15
37	Analog	analog[11]	16
38	Mux Control	ctrl_ena	22
39	Mux Control	ctrl_sel_inc	24
40	Mux Control	ctrl_sel_rst_n	25
41	Reserved	(none)	26
42	Reserved	(none)	27
43	Reserved	(none)	28

† 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.

Sponsored by

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 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