Tiny Tapeout 05 Datasheet

Project Repository

https://github.com/TinyTapeout/tinytapeout-05

August 2, 2024

Contents

Chip map

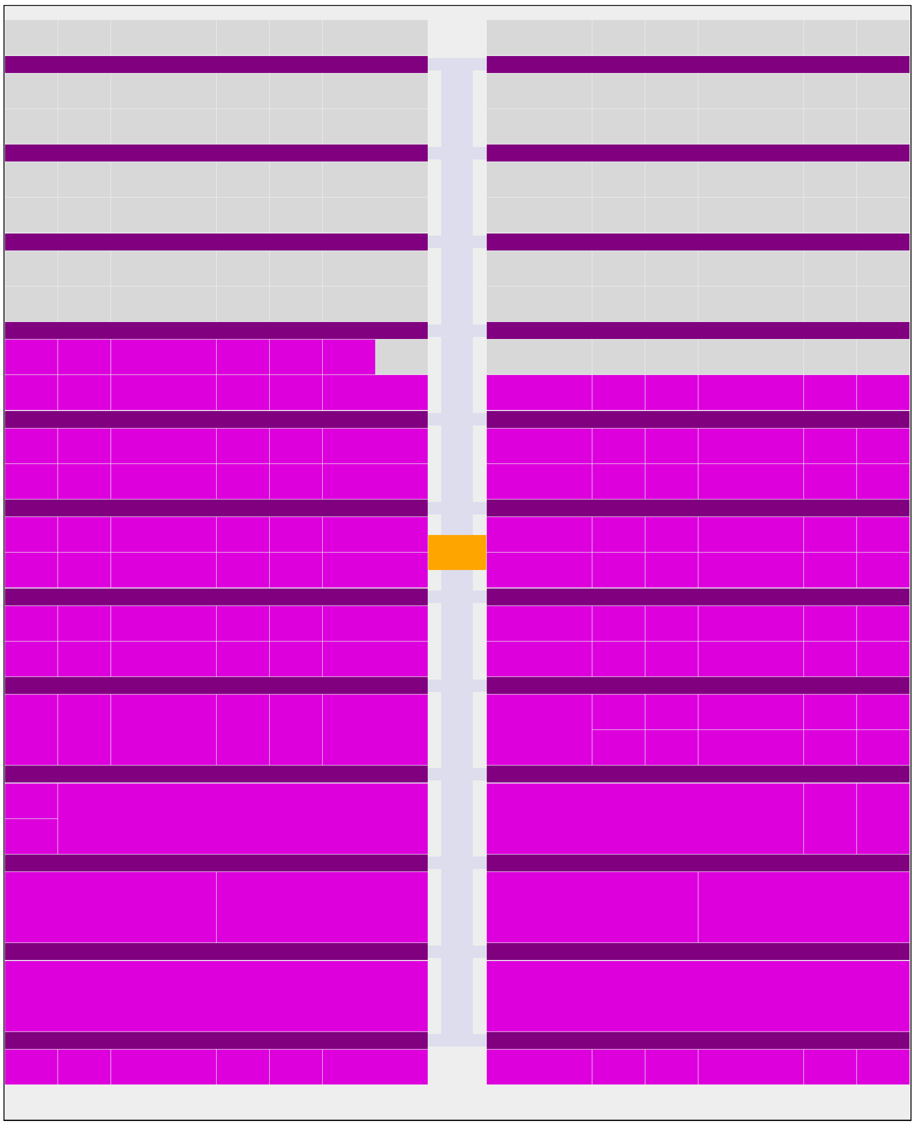


Figure 1: Full chip map

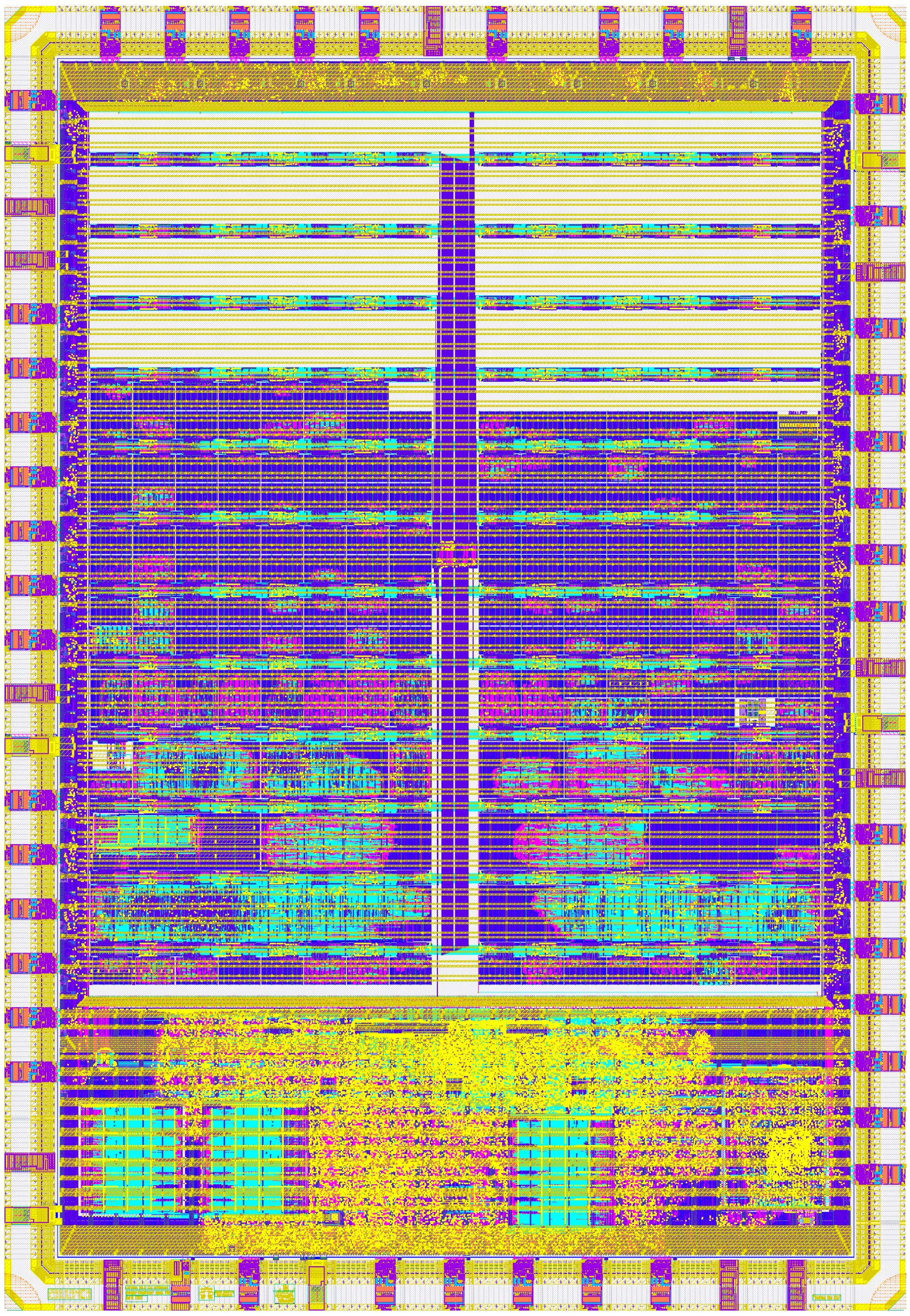


Figure 2: GDS render

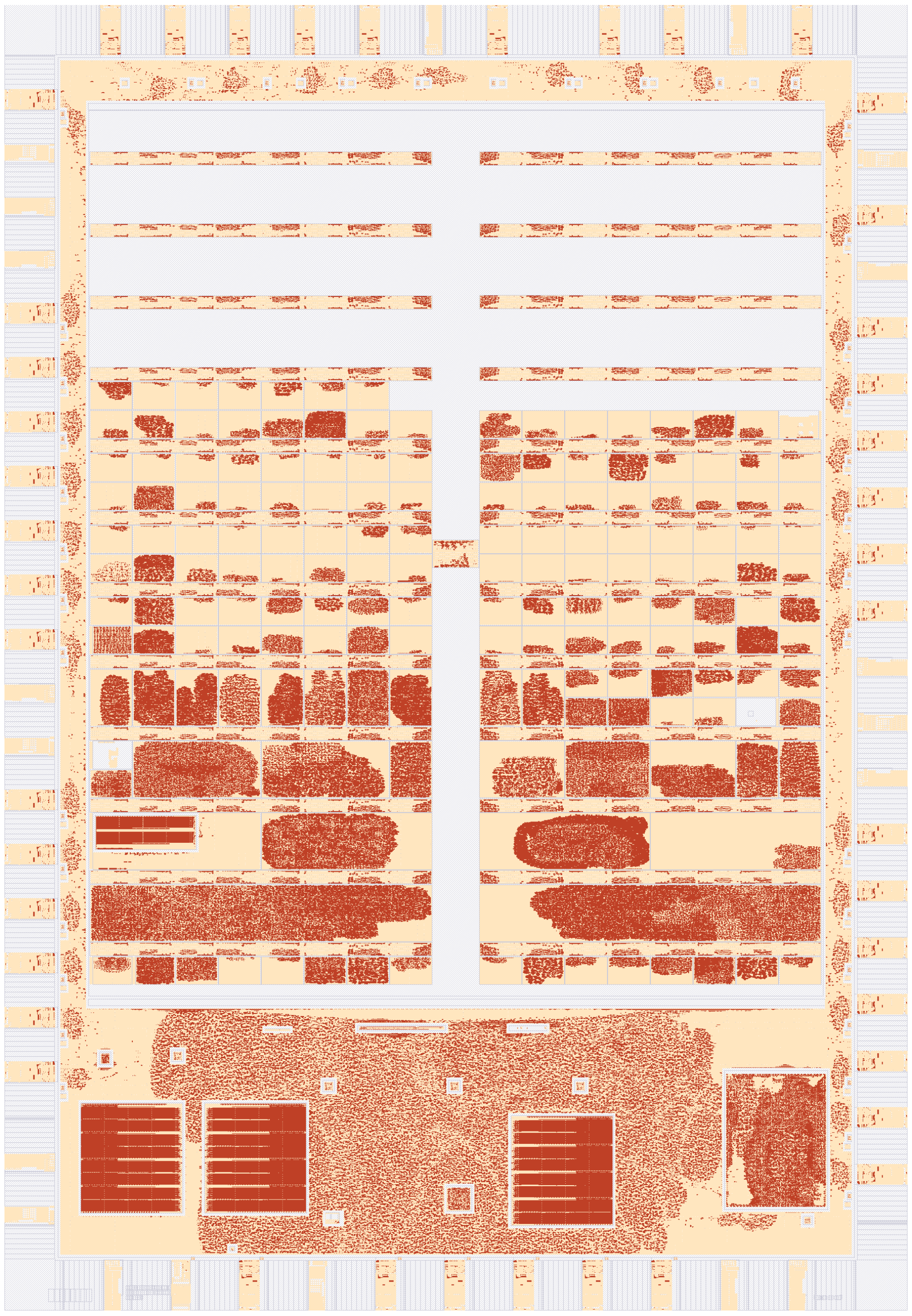


Figure 3: Logic density (local interconnect layer)

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
- External hardware:

How it works

ROM memory that contains information about the Tiny Tapeout chip. The ROM is 8-bit wide and 128 bytes long.

The ROM layout The ROM layout is as follows:

Address	Length	Encoding	Description
0	8	7-segment	Shuttle name (e.g. "tt05"), null-padded
8	8	7-segment	Git commit hash
32	96	ASCII	Chip descriptor (see below)

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	tt05
repo	The name of the repository	TinyTapeout/tinytapeout-05
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=tt05
repo=TinyTapeout/tinytapeout-05
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]	none
1	addr1	data1	none
2	addr2	data2	none
3	addr[3]	data[3]	none
4	addr[4]	data[4]	none
5	addr[5]	data[5]	none
6	addr[6]	data[6]	none
7	addr[7]	data[7]	none

TinyTapeout 05 Factory Test 1

- Author: Sylvain Munaut
- Description: Factory test module
- GitHub repository
- HDL project
- Mux address: 1
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

If `sel` is high, then a counter is output on the output pins and the bidirectional pins (`data_o = counter_o = counter`). If `sel` is low, the bidirectional pins are mirrored to the output pins (`data_o` = `data_i`).

How to test

Set `sel` high and observe that the counter is output on the output pins (`data_o`) and the bidirectional pins (`counter_o`).

Set `sel` low and observe that the bidirectional pins are mirrored to the output pins (`data_o` = `data_i`).

Pinout

#	Input	Output	Bidirectional
0	sel	data_o[0]	data_i[0] / counter_o[0]
1	none	data_o1	data_i1 / counter_o1
2	none	data_o2	data_i2 / counter_o2
3	none	data_o[3]	data_i[3] / counter_o[3]
4	none	data_o[4]	data_i[4] / counter_o[4]
5	none	data_o[5]	data_i[5] / counter_o[5]
6	none	data_o[6]	data_i[6] / counter_o[6]
7	none	data_o[7]	data_i[7] / counter_o[7]

TinyTapeout 05 Loopback Test Module 2

- Author: Sylvain Munaut
- Description: Loopback test module
- GitHub repository
- HDL project
- Mux address: 2
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Input 0 goes to output 0 through 6. Output 7 is input4 & input5 & input6 & input7

How to test

Toggle input 0, measure the time for output 0 to change.

Pinout

#	Input	Output	Bidirectional
0	in0	mirrors in0	none
1	none	mirrors in0	none
2	none	mirrors in0	none
3	none	mirrors in0	none
4	in4	mirrors in0	none
5	in5	mirrors in0	none
6	in6	mirrors in0	none
7	in7	the value of in4 & in5 & in6 & in7	none

Leaky Integrate and Fire Neuron Model [3]

- Author: Miles Segal
- Description: Models the functionality of a leaky integrate and fire neuron, of the style tipically found in spiking neural networks
- GitHub repository
- HDL project
- Mux address: 3
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Uses a set of registers to divide the clock, and then some combinational logic to convert from binary to decimal for the display.

Puts the bottom 8 bits of the counter on the bidirectional outputs.

With all the inputs set to 0, the internal 24 bit compare is set to 10,000,000. This means the counter will increment by one each second.

If any inputs are non zero, then the input will be used as an bits 11 to 18 of the 24 bit compare register. Example: setting the inputs to 00010000 will program 16384 into the compare register. With a 10MHz clock the counter will increment ~610 times per second.

How to test

After reset, the counter should increase by one every second with a 10MHz input clock. Experiment by changing the inputs to change the counting speed.

Pinout

#	Input	Output	Bidirectional
0	compare bit 11	segment a	second counter bit 0
1	compare bit 12	segment b	second counter bit 1
2	compare bit 13	segment c	second counter bit 2
3	compare bit 14	segment d	second counter bit 3
4	compare bit 15	segment e	second counter bit 4

#	Input	Output	Bidirectional
5	compare bit 16	segment f	second counter bit 5
6	compare bit 17	segment g	second counter bit 6
7	compare bit 18	dot	second counter bit 7

Time Multiplexed Neuron Ckt [4]

- Author: Karina Aguilar
- Description: Utilize leaky-integrate-and-fire neurons to make multiple neurons
- GitHub repository
- HDL project
- Mux address: 4
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Apply an input current to the LIF neurons through the switches.

This will add to the membrane potential that decays over time. If the membrane potential exceeds the threshold, then a spike is triggered.

How to test

After reset, the membrane potential will be set to 0.

Then change the inputs to change the current. A higher current should trigger a higher firing rate.

Pinout

#	Input	Output	Bidirectional
0	current bit 11	membrane potential bit a	spike bit 1
1	current bit 12	membrane potential bit b	spike bit 2
2	current bit 13	membrane potential bit c	spike bit 3
3	current bit 14	membrane potential bit d	spike bit 4
4	current bit 15	membrane potential bit e	spike bit 5
5	current bit 16	membrane potential bit f	unspecified
6	current bit 17	membrane potential bit g	unspecified
7	current bit 18	membrane potential bit h	unspecified

SAP-1 Computer [5]

- Author: Brandon Cruz
- Description: Simple as Possible computer into ASIC
- GitHub repository
- HDL project
- Mux address: 5
- Extra docs
- Clock: 10000000 Hz
- External hardware: Oscilloscope

How it works

Originally, Malvino and Brown presented the SAP-1 architecture in a book called Digital Computer Electronics. The design gained massive popularity when it was build as a bread board computer by Ben Eater on a series of YouTube videos. The architecture contains various modules, including the instruction execution set gives the SAP-1 a total of six stages from 0 to 5, repeating all over again.

- Clock
- Program Counter
- Register A
- Register B
- Adder
- Memory
- Instruction Register
- Bus
- Controller This design doesn't have inputs, it is dependent only on the clock that coordinates sequence of the computer's operation. Its operation consists on the communication that that bus provides between modules; the signal load dumps information into a module and the enable signal allows the bus to receive a signal. The bus is 8-bit width since it is an 8 bit computer, and the registers are also 8-bit registers. The computer can only perform addition, whether it is positive numbers or negative numbers (substraction). The signals information is stored within the memory module. There bus operations are coordinated with a series of multiplexers and

The more important module is the controller. It controlls the assertion execution according to the stimuli from the stages. The stages 3 to 5 five depend on the instructions of the operation codes.

How to test

Design Output Reading Section The design is engineered to read the output signal generated from the bus, which contains the information of the add and subtract operations executed by the design. Currently, the only method to read the signals is through an oscilloscope. However, a significant enhancement would be the implementation of a state machine controlling a 3 7-segment display to show the numbers on the 8-bit bus (up to 255).

Pinout

#	Input	Output	Bidirectional
0	n/a	bus[0]	n/a
1	n/a	bus1	n/a
2	n/a	bus2	n/a
3	n/a	bus[3]	n/a
4	n/a	bus[4]	n/a
5	n/a	bus[5]	n/a
6	n/a	bus[6]	n/a
7	n/a	bus[7]	n/a

Current Based Leaky Integrate and Fire Model [6]

- Author: Shatoparba Banerjee
- Description: Implement a current based LIF neuron
- GitHub repository
- HDL project
- Mux address: 6
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Current-based LIF neurons are a simplified abstraction of the behavior of real neurons, and they are often used in large-scale neural network simulations due to their computational efficiency. These models are useful for studying the dynamics of spiking neurons and their role in information processing in the brain.

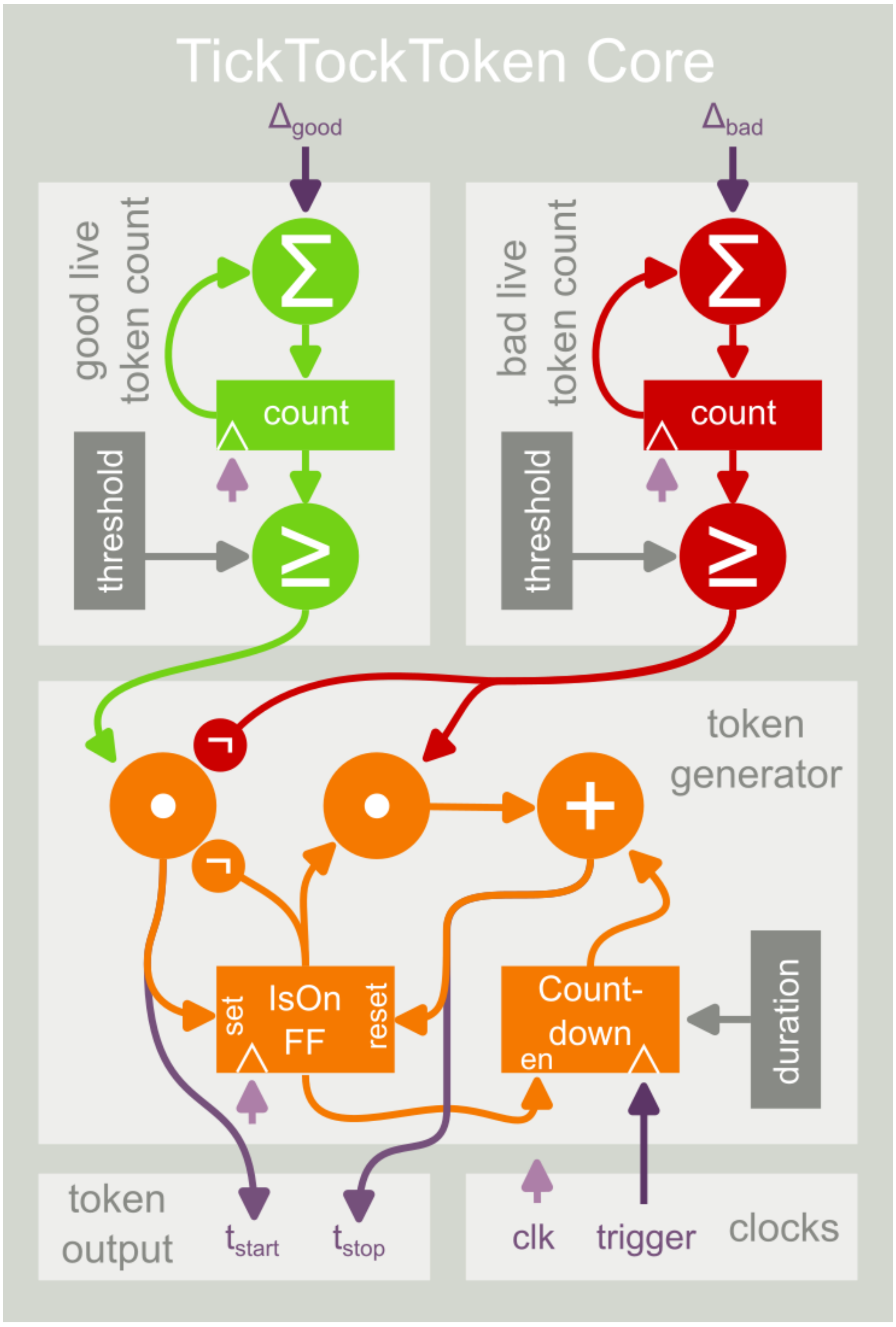
How to test

To test the current-based LIF project, follow these steps: Connect the LIF module to the input switches, 7-segment display, and clock source as specified in the Verilog module. Use input switches to control the input current, and observe the 7-segment display for spike detection, while ensuring the clock signal is appropriately set to provide the desired clock frequency for the simulation.

Pinout

#	Input	Output	Bidirectional
0	current bit 11	membrane potential bit a	spike bit 0
1	current bit 12	membrane potential bit b	unspecified
2	current bit 13	membrane potential bit c	unspecified
3	current bit 14	membrane potential bit d	unspecified
4	current bit 15	membrane potential bit e	unspecified
5	current bit 16	membrane potential bit f	unspecified
6	current bit 17	membrane potential bit g	unspecified
7	current bit 18	membrane potential bit h	unspecified

TickTockTokens [7]



- Author: Johannes Leugering
- Description: Implementation of a processor that uses Tick Tock Tokens for event-based computation.
- GitHub repository
- HDL project
- Mux address: 7
- Extra docs
- Clock: 10000000 Hz

- External hardware: arduino to generate I/O

How it works

Each TickTockToken (ttt) is indicated by two messages, a start and an end event. A ttt-Processor uses these tokens to perform event-based computations in a fashion inspired by Time Petri Nets.

How to test

If I didn't get lazy half-way though, the test-script provided in the repo should run a test model successfully, and the documentation should provide a more through explanation.

Pinout

#	Input	Output	Bidirectional
0	instruction bit 0	data bit 0	token start flag
1	instruction bit 1	data bit 1	token stop flag
2	instruction bit 2	data bit 2	data bit 2
3	instruction bit 3	data bit 3	data bit 3
4	(reserved)	data bit 4	data bit 4
5	(reserved)	data bit 5	data bit 5
6	(reserved)	data bit 6	data bit 6
7	(reserved)	data bit 7	data bit 7

Spiking LSTM Network [8]

- Author: Skye Gunasekaran
- Description: A leaky integrate and fire neuron with adaptive threshold.
- GitHub repository
- HDL project
- Mux address: 8
- Extra docs
- Clock: None Hz
- External hardware: None

How it works

A Leaky Integrate-and-Fire (LIF) neuron is a simple mathematical model used in neuroscience and computational neuroscience to describe the behavior of individual neurons. It provides a simplified yet effective way to simulate the behavior of real neurons. In the neuron, there are two key elements: the current and the threshold. If the current surpasses the threshold, a spike is emitted, otherwise, the spike is 0 (resting). In this spiking LSTM implementation, the neuron's threshold is adaptive, and will increase when the threshold is passed. When the neuron fails to reach the threshold, it will slowy decay back to the initial threshold.

How to test

After applying the reset, the variables should be initialized, and a current can be applied. The testbench will record the current, threshold, and spiking behavior of the neuron. When a higher current is applied, you can see how the threshold increases, and vice versa when a spike is not emitted.

Pinout

#	Input	Output	Bidirectional
0	clk Clock input	uo_out Spike output	uio_in Unused
1	rst_n Reset signal	n/a	uio_out Threshold
2	ui_in Voltage current	n/a	uio_oe Unused
3	ena Unused	n/a	n/a
4	n/a	n/a	n/a
5	n/a	n/a	n/a
6	n/a	n/a	n/a

#	Input	Output	Bidirectional
7	n/a	n/a	n/a

Integrate-and-Fire Neuron. [9]

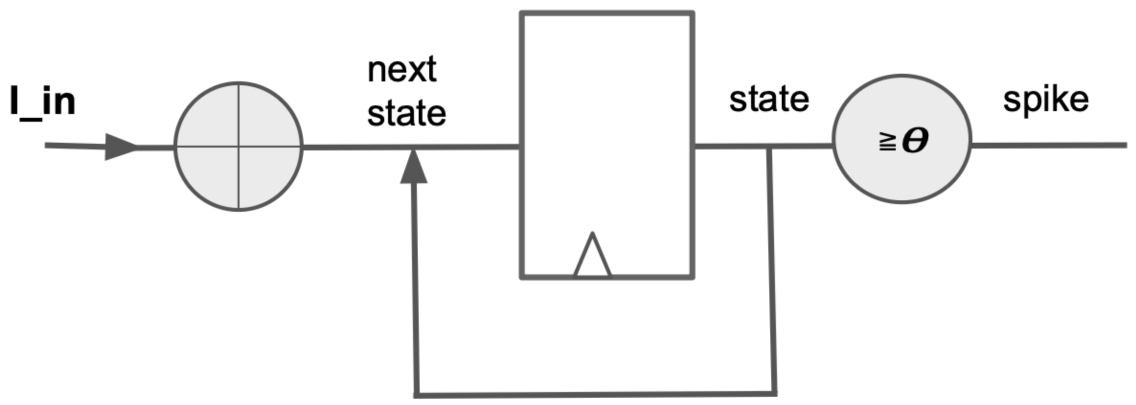


Figure 1: Top-level diagram of the Integrate-and-Fire neuron model.

- Author: Kembay Assel
- Description: Implement an IF neuron in silicon.
- GitHub repository
- HDL project
- Mux address: 9
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Apply an input current injection to the IF neuron using switches. This gets added to a membrane potential. If the membrane potential exceeds the threshold, then it triggers a spike.

How to test

An 8-bit input current is applied to the IF neuron through the designated input (i.e., uio_in). The membrane potential of the IF neuron will respond to the applied input current. Larger currents will lead to a higher membrane potential. The neuron is designed to generate a spike when the membrane potential exceeds a certain threshold.

Pinout

#	Input	Output	Bidirectional
0	current bit 11	membrane potential bit a	spike bit 0
1	current bit 12	membrane potential bit b	unspecified
2	current bit 13	membrane potential bit c	unspecified
3	current bit 14	membrane potential bit d	unspecified
4	current bit 15	membrane potential bit e	unspecified
5	current bit 16	membrane potential bit f	unspecified
6	current bit 17	membrane potential bit g	unspecified
7	current bit 18	membrane potential bit h	unspecified

Neural network on chip [10]

- Author: Faculty of Technical Sciences Cacak, University of Kragujevac
- Description: Neural network built out of perceptrons
- GitHub repository
- HDL project
- Mux address: 10
- Extra docs
- Clock: 50 000 000 Hz
- External hardware:

How it works

Network calculates output based on user provided input and predefined weight parameters of neural network

How to test

Drive inputs to [7:0] ui_in and result of computation of neural network can be obesrved on [7:0] uo_out

Pinout

#	Input	Output	Bidirectional
0	ui_in[0]	uo_out[0]	none
1	ui_in1	uo_out1	none
2	ui_in2	uo_out2	none
3	ui_in[3]	uo_out[3]	none
4	ui_in[4]	uo_out[4]	none
5	ui_in[5]	uo_out[5]	none
6	ui_in[6]	uo_out[6]	none
7	ui_in[7]	uo_out[7]	none

Simple Leaky Integrate and Fire (LIF) Neuron [11]

- Author: Phillip Marlowe
- Description: Given input current, spike when threshold is reached (also assume any files with the letters LFI should be LIF)
- GitHub repository
- HDL project
- Mux address: 11
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Based on input current, calculation is made using it and previous membrane potential. If current membrane potential is above pre-set threshold then spike!

How to test

After reset, input some current and see what happens. Should see an increase on output and possibly a spike eventually.

A current input of 100 after 20 cycles should produce a spike.

Pinout

#	Input	Output	Bidirectional
0	clk_i is the clock	nu_o is the next membrane potential	msb bit of uio_oe is connected to spike_o
1	current_i is the current input to the LFI neuron	spike_o is the single bit to show when the neuron is firing	n/a
2	rst_n is for reset	n/a	n/a
3	n/a	n/a	n/a
4	n/a	n/a	n/a
5	n/a	n/a	n/a
6	n/a	n/a	n/a
7	n/a	n/a	n/a

e Spigot [12]

- Author: diadatp
- Description: Spigot algorithm for calculating the digits of e
- GitHub repository
- HDL project
- Mux address: 12
- Extra docs
- Clock: 350 Hz
- External hardware: 4 x 7447 decoders

How it works

This project implements a bounded spigot algorithm for calculating the digits (currently 31) of e. While there are many ways to calculate the digits of transcendental numbers like e or pi, this spigot algorithm has much lower memory requirements. It however only produces a single digit at a time, and the number of digits produced is precommited at the time of design. For calculating n digits, the algorithm needs at least (n+1) storage locations. Each digit requires (n+1) calculation steps, repeated (n-1) times producing (n-1) digits (first digit 2 is not counted). Each calculation step requires a constant multiply, an add and a divide with remainder. There are many optimizations needed to fit as many digits as possible into a 1x1 tile. The biggest contributor is the storage elements. Some quick modeling revealed that the storage elements need to be about as wide as log(n). The calculation step hardware is shared across all iterations. The intermediate results are never needed outside each calculation and are never stored in memory. The memory access is such that each location is read and written to before moving on to the next. The memory access pattern removes the need for address decoding, replaced with a massive ring of gated shift registers.

How to test

The digits are output on the bidirectional port and the output port in BCD (Binary-coded decimal). A BCD to seven segment decoder will be needed to display the digits. A clock below 500Hz should allow one to see the digits slide across the segment displays.

Pinout

#	Input	Output	Bidirectional
0	none	digit_2[0]	digit_0[0]
1	none	digit_21	digit_01
2	none	digit_22	digit_02
3	none	digit_2[3]	digit_0[3]
4	none	digit_3[0]	digit_1[0]
5	none	digit_31	digit_11
6	none	digit_32	digit_12
7	none	digit_3[3]	digit_1[3]

Continued Fraction Calculator [13]

- Author: Kevin You
- Description: Calculates the continued fraction of the square root of a natural number
- GitHub repository
- HDL project
- Mux address: 13
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

This continued fraction calculator computes the convergents in the standard manner. It computes the coefficients of the continued fraction, and then recursively computes the convergents. It turns out that when the target is the square of a natural number, one can avoid the need of comparisons or taking the floor (except on the first step) and only use addition, multiplication, and integer division. Still, squeezing the design in roughly 1000 cells proved quite difficult, and various simplifications were necessary (such as changing the output from 7-segment to binary).

This calculator, in conjunction with a mobile phone calculator, or paper and pencil, can be used to calculate the fundumental solution of Pell's equation x^2 - Dy^2 = 1. To do this, simply enter D, compute convergents, and verify whether the convergents satisfy Pell's equation P^2 - DQ^2 = 1. The first convergents that satisfy Pell's equation is the fundumental solution. This procedure combines the continued fraction calculator's ability to store various intermediate values and a mobile phone calculator's ability to calculate large numbers.

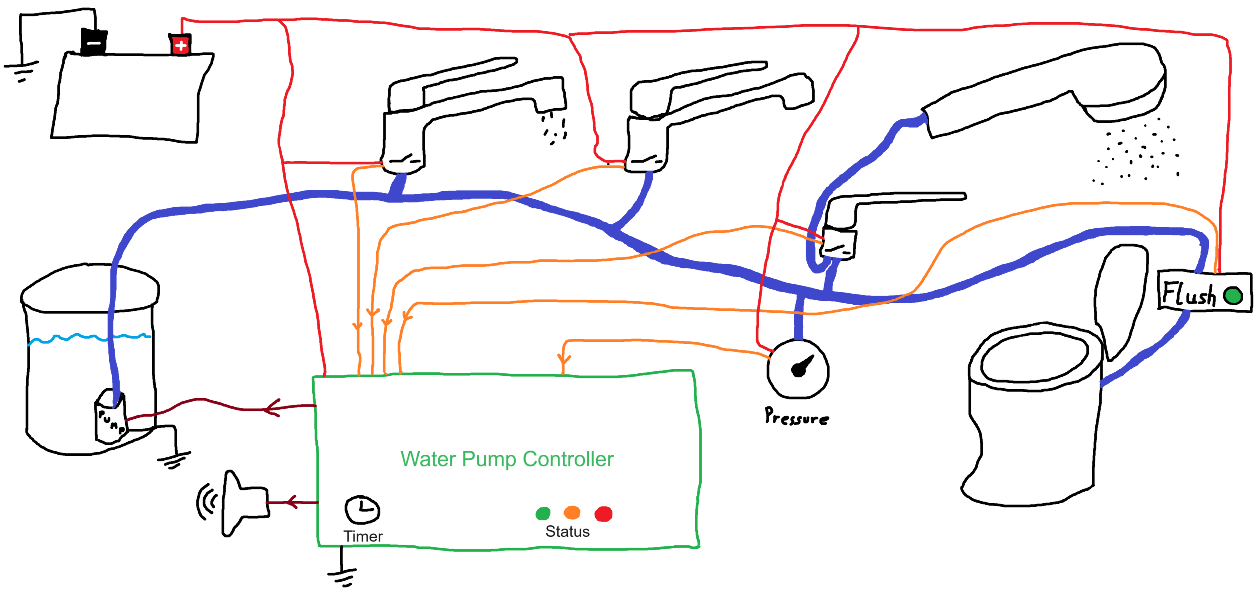
How to test

Enter 14 bit binary number D input via switches, press button 0 to generate the next convergents P and Q, where sqrt(D) ~ P/Q. Press button 1 to read through the values of P and Q in order of P[15:8], P[7:0], Q[15:8], Q[7:0].

Pinout

#	Input	Output	Bidirectional
0	button	status LED	switch
1	button	status LED	switch
2	switch	status LED	switch
3	switch	status LED	switch
4	switch	status LED	switch
5	switch	status LED	switch
6	switch	status LED	switch
7	switch	status LED	switch

Water Pump Controller [14]



- Author: Hendrik
- Description: Controller for a camping van water pump with multiple tap switches and timer
- GitHub repository
- Wokwi project
- Mux address: 14
- Extra docs
- Clock: 32768 Hz
- External hardware: 32768Hz clock (does not need to be prcise, actually), power on reset, controlled water supply system (tap switches, pump), optionally LEDs and Buzzer for controller states

How it works

The water pump controller is intended to replace the wiring based water pump system of typical camping vans with a bathroom and a kitchen and addresses some problems these systems can have.

In such a system, water taps are usually equipped with switches that signal the need for water as soon as the tap is slightly opened so that a pump can be activated to pressurize the pipes.

The controller has six inputs so that each tap (e.g sink in the kitchen, sink in the bathroom, toilet flushing, shower in bathroom, external shower) can use a dedicated input. It has another input for a high pressure switch that would turn off the pump if the pressure rises when all taps are actually closed.

The main feature is a timer that can give an reminder using a buzzer when the water is running for an untypically long period and automatically switch off the pump at some point as well. When the switch off time was reached the buzzer signal can help to indicate which switch is still active.

How to test

For testing the circuit, the outputs (including the pump output) can be connected to LEDs or as in the test board to a 7-segment display. In the test board the pump output corresponds to the top segment. The inputs can be connected to DIP switches. The clock should be set to 32768 Hz (2^15 Hz). The reset signal should provide a power-on reset and optionally a manual reset that might be handy for testing. In the minimal setup, the last three bidirectional I/O pins should be connected via separate resistors to GND. Connecting them directly to GND should be okay as well for a quick test. The pins can be outputs that should only be driven to low in this case, but do not connect them directly to VCC. The other I/O pins can be left open or connected to GND to avoid floating pins. The connection to GND can be done directly or via pull-down resistors to plan ahead for more tests with additional circuitry.

The first test is about enabling the pump while not making use of the timer:

- Keep the input 6 (DIP 7) low to disable the timer
- Keep the input 7 (high pressure switch) low
- Set any combination of tap switches (inputs 0 - 5) high
- The pump output (top segment) should be on
- The pump LED and ActiveNormal LED (right hand segments) should be on
- With all tap switches off all outputs should be off as well

The second test is to verify the high pressure switch:

- Set input 7 (the high pressure switch) high
- Set any combination of tap switches high
- Select any state for the timer enable pin (input 6)
- The pump output should be off
- The pump LED, ActiveLEDs and buzzer can be on, depending on the state of the controller

A simple test for the timer with default values:

- Set input 6 high to enable the timer feature
- Keep input 7 low to see the pump output

- Set any combination of tap switches (input 0 - 5) high
- Wait
- After 128s, the LEDs outputs should change from ActiveNormal to ActiveWarning (bottom right to bottom on the 7-segment display) and the RunLong LED (top left for 7-segment display) should be activated
- At the same time the buzzer should be activated every two seconds. LEDs would blink dim (center segment).
- After another 32s, the pump and pump LED should be turned off and the other LEDs should go from ActiveWarning to ActiveHalted (bottom to bottom left segment).
- The buzzer should emit a sequence corresponding to the first active tap input every 16 seconds. Again for LEDs that would be a dim blinking sequence every 16s.
- Set all tap switches to low
- All outputs should be off
- Activate any tap switch
- Pump (and related LEDs) should be on again, buzzer should be off

Testing the configuration feature requires additional external circuits. Please refer to the testing section of the README of the github project (https://github.com/fahek/water-pump-controller-tto5#more-advanced-tests).

Pinout

#	Input	Output	Bidirectional
0	TapA	Pump	CFG0
1	TapB	PumpEnabled	CFG1
2	TapC	ActiveNormal	CFG2
3	TapD	ActiveWarning	CFG3
4	TapE	ActiveHalted	CFG4
5	TapF	RunLong	TimerScaleConfig
6	EnableTimeout	Buzzer	WarningTimeConfig
7	PressureHigh	BuzzerHaltedOnly	TimeoutConfig

Event Denoising Circuit [15]

- Author: Emily Lee
- Description: Implementing a Denoiser for Event Based Data in Silicon
- GitHub repository
- HDL project
- Mux address: 15
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

The 8 switches correspond to the tuple data input (L-R): Switches 1 & 2: Value of x (2 bits) Switches 3 & 4: Value of y ' ' Switches 5 & 6: Value of p ' ' Switches 7 & 8: Value of t ' '

The denoiser is implemented as a debouncer. A high or low event will only be output if the data remains the same for 5 clock cycles. Due to white/thermal noise in an event camera, a cluster of coordinates that have no movement may incorectly spike high and the result would be a singular bright bit. The debouncing avoids this by first ensuring the event is consistent before outputting.

How to test

Switches (L-R) 5 & 6 corresponds to the input value of the polarity of the tuple. If a high event is wanted - switch 5 should be low and switch 6 should be high. If a low event is wanted = switch 5 must be low, and switch 6 must be low. This will cause the chip to output the debounced tuple corresponding to a high or low event.

Pinout

#	Input	Output	Bidirectional
0	bit 1 of x	bit 1 of x	Unused
1	bit 0 of x	bit 0 of x	Unused
2	bit 1 of y	bit 1 of y	Unused
3	bit 0 of y	bit 0 of y	Unused
4	bit 1 of p - no real use	bit 1 of p	Unused
5	toggles event to be passed through if high	bit 0 of p	Unused

#	Input	Output	Bidirectional
6	bit 1 of t	bit 1 of t	Unused
7	bit 0 of t	bit 0 of t	Unused

7 segment seconds (Verilog Demo) [32]

- Author: Matt Venn, cloned by Cedric Honnet
- Description: Count up to 10, one second at a time.
- GitHub repository
- HDL project
- Mux address: 32
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Uses a set of registers to divide the clock, and then some combinational logic to convert from binary to decimal for the display.

Puts the bottom 8 bits of the counter on the bidirectional outputs.

With all the inputs set to 0, the internal 24 bit compare is set to 10,000,000. This means the counter will increment by one each second.

If any inputs are non zero, then the input will be used as an bits 11 to 18 of the 24 bit compare register. Example: setting the inputs to 00010000 will program 16384 into the compare register. With a 10MHz clock the counter will increment ~610 times per second.

How to test

After reset, the counter should increase by one every second with a 10MHz input clock. Experiment by changing the inputs to change the counting speed.

Pinout

#	Input	Output	Bidirectional
0	compare bit 11	segment a	second counter bit 0
1	compare bit 12	segment b	second counter bit 1
2	compare bit 13	segment c	second counter bit 2
3	compare bit 14	segment d	second counter bit 3
4	compare bit 15	segment e	second counter bit 4
5	compare bit 16	segment f	second counter bit 5

#	Input	Output	Bidirectional
6	compare bit 17	segment g	second counter bit 6
7	compare bit 18	dot	second counter bit 7

Frequency Encoder/Decoder [33]

- Author: Hannah Cohen-Sandler
- Description: Encodes data into frequency variations and then decodes it back into its original form.
- GitHub repository
- HDL project
- Mux address: 33
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Utilizes an array of inputs and outputs to connect to switches, 7-segment displays, and enable bidirectional paths.

The bottom 7 bits of the second counter are linked to the bidirectional output.

The clock is generated using a Phase-Locked Loop.

The Frequency Encoder encodes data input from switches to a pulse output and uses the PLL output to enable the encoding operation.

The Frequency Decoder is connected to bidirectional inputs and decodes the pulse signal based on the PLL output, resulting in a data output.

How to test

Confirm that the system begins in a reset state with rst_n set to 0. Transition the system out of reset by setting rst_n to 1. Set the constant current input signal (ui_in) to a specific value to simulate different input scenarios. Activate the chip design by setting the ena signal to 1. Alter the clk clock signal frequency value and observe how the changes affect the design's behavior. Experiment with various inputs, clock frequencies, and enabling/disabling operations verify the design and accuracy of the encoding, decoding, and pulse counting.

#	Input	Output	Bidirectional

Pinout

#	Input	Output	Bidirectional
0	current bit 11	segment a	second counter bit 0
1	current bit 12	segment b	second counter bit 1
2	current bit 13	segment c	second counter bit 2
3	current bit 14	segment d	second counter bit 3
4	current bit 15	segment e	second counter bit 4
5	current bit 16	segment f	second counter bit 5
6	current bit 17	segment g	second counter bit 6
7	current bit 18	segment h	second counter bit 7

UART Greeter with RNN Module [34]

- Author: Jonathan Zentgraf
- Description: Sends 'Hello' over UART and fills die space with metastability
- GitHub repository
- Wokwi project
- Mux address: 34
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

The UART transmitter is just a shift register with hardcoded initial values. The output of the shift register is fed back into itself in an infinite loop. The "RNN" is a few flip-flops feeding into each other to use up die space. :)

How to test

Testing UART is simple:

1. Connect the UART output to a microcontroller or scope.
2. Set load/enable low (load).
3. Set output enable high.
4. Set load/enable high (enable).
5. Observe as the string "Hello\n" is sent over UART.

The RNN module is trained on random Wokwi wiring, and might be smarter than a single human neuron. It probably detects something we mortals cannot comprehend, and is tied to inputs 0-3 and outputs 0-3. It may be fun to drive these with a very fast clock.

Pinout

#	Input	Output	Bidirectional
0	RNN input 0	RNN output 0	none
1	RNN input 1	RNN output 1	none
2	RNN input 2	RNN output 2	none
3	RNN input 3	RNN output 3	none

#	Input	Output	Bidirectional
4	none	none	none
5	none	none	none
6	Shift register load (low) / enable (high)	UART output enabled	none
7	UART output enable	UART output	none

WS2812B LED strip driver [35]

- Author: Ciro Cattuto
- Description: Drives a WS2812B LED strip with random colors for each refresh
- GitHub repository
- Wokwi project
- Mux address: 35
- Extra docs
- Clock: 20000000 Hz
- External hardware: WS2812B strip of arbitrary lengthd

How it works

This project drives a strip of WS2812B RGB LEDs, periodically updating the strip with random color values. The project consists of three main modules:

- a linear feedback 16-bit shift register to generate a stream of pseudo-random bits
- a 5-bit synchronous increasing counter, wrapping to 0 when the counter reaches 25. WHen driven by a 20 MHz clock source, the counter generates the 1.25 us pulses required by the WS2812B protocol. The duration of the high phase of the pulse is controlled by the random bit stream generated above.
- a 16-bit ripple counter increasing at the end of each pulse, used to divide the pulse frequency and generate the LED strip refresh signal

How to test

Set the clock frequency to 20 MHz and connect OUT2 to the DIN signal of a WS2812B LED strip. Optionally connect to IN6 the DOUT signal of the last LED of the strip. Press and release the reset button. The strip should light up with random colors, updating at a frequency controllable using the SW3 and SW4 switches.

Pinout

#	Input	Output	Bidirectional
0	clock source selection	shift register output	none
1	external clock source	shift register clock	none
2	refresh freq sel (low)	WS2812B LED strip input	none
3	refresh freq sel (high)	LED strip overflow	none

#	Input	Output	Bidirectional
4	none	LED strip refresh	none
5	none	none	none
6	WS2812B LED strip output	none	none
7	shift register input	none	none

Tiny Tapeout 5 Workshop [36]

- Author: Rob Campbell KG6HUM
- Description: Tiny Tapeout 5 Workshop
- GitHub repository
- Wokwi project
- Mux address: 36
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

First 3 output bits are a binary counter. Can be preset with first 3 input bits. Other input bits pass through to the output.

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	none	segment a	none
1	none	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

Tiny Tapeout 1 [37]

- Author: James Bryant
- Description: A description
- GitHub repository
- Wokwi project
- Mux address: 37
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	none	segment a	none
1	none	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

Supercon Workshop [38]

- Author: Caleb Hensley
- Description: Example of logic gates: AND, NAND, OR, XOR
- GitHub repository
- Wokwi project
- Mux address: 38
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Input 0 and 1 input to an AND gate and output to output 0. Input 2 and 3 input to a NAND gate and output to output 1. Input 4 and 5 input to a OR gate and output to output 2. Input 6 and 7 input to a XOR gate and output to output 3.

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	none	segment a	none
1	none	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

Matrix Multiplier [39]

- Author: Erik Mercado
- Description: Multiple Matrices.
- GitHub repository
- HDL project
- Mux address: 39
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

tt_um_matrix_multiplier:

This module performs a multiplication of two 2x2 matrices, where each element is an unsigned 2-bit number. The matrices are input via 8-bit wires (ui_in and uio_in), where every 2 bits represent an element. Error checking is done to ensure that each element is within the range [0, 2]. If there's an error, the output is set to zero; otherwise, the multiplication result is returned via uo_out and uio_out. The module also provides a uio_oe output signal that serves as an output enable for the resultant matrix.

tb (testbench):

This is the simulation testbench for the tt_um_matrix_multiplier module. It toggles a clock signal, initializes input values, and instantiates the tt_um_matrix_multiplier. The testbench is set up to generate VCD files, allowing for waveform viewing using tools like GTKWave.

test.py:

This Python script uses the cocotb framework to test the matrix multiplication functionality. Helper functions are provided to convert 2x2 matrices to binary representations and vice-versa. A list of test matrices and expected results is present. For each test case, the script inputs matrices, waits for the multiplication result, and checks against the expected result. The test concludes by logging a success message if all test cases pass.

How to test

Synthesize and Implement: Use an FPGA toolchain to synthesize the Verilog code and implement it on a suitable FPGA. Simulation: Use a simulator compatible with

Verilog (like ModelSim or Icarus Verilog) to run the testbench (tb.v). You can view the generated VCD file with a tool like GTKWave to visualize the waveform.

Cocotb Test: Setup the cocotb environment and necessary dependencies. Use the test.py script to run the cocotb test. Monitor the test output to ensure that all matrix tests pass.

Pinout

#	Input	Output	Bidirectional
0	compare bit 11	segment a	second counter bit 0
1	compare bit 12	segment b	second counter bit 1
2	compare bit 13	segment c	second counter bit 2
3	compare bit 14	segment d	second counter bit 3
4	compare bit 15	segment e	second counter bit 4
5	compare bit 16	segment f	second counter bit 5
6	compare bit 17	segment g	second counter bit 6
7	compare bit 18	dot	second counter bit 7

Clock Divider [40]

- Author: Joey Castillo
- Description: Divides the clock input eight times, with CLK/2 on OUT7, CLK/4 on OUT6, etc.
- GitHub repository
- Wokwi project
- Mux address: 40
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	none	CLK/256	none
1	none	CLK/128	none
2	none	CLK/64	none
3	none	CLK/32	none
4	none	CLK/16	none
5	none	CLK/8	none
6	none	CLK/4	none
7	none	CLK/2	none

Binary Counter [41]

- Author: Chinchilla
- Description: The Just Kidding flip flop has been changed to a binary counter
- GitHub repository
- Wokwi project
- Mux address: 41
- Extra docs
- Clock: 10 Hz
- External hardware: none

How it works

It didn't last time (3). Counts. With 1 and 0.

How to test

turn on.

Pinout

#	Input	Output	Bidirectional
0	none	segment a	none
1	none	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

ring osc test [42]

- Author: Bob Poekert
- Description: simple ring oscillator
- GitHub repository
- Wokwi project
- Mux address: 42
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

This outputs a square wave at… some frequency on pins 0-3, where the pins are phase shifted by… some frequency.

How to test

Just apply power.

Pinout

#	Input	Output	Bidirectional
0	none	segment a	none
1	none	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

7 segment clock with 4 digits [43]

- Author: Kumar Abhishek
- Description: Multi mode clock.
- GitHub repository
- HDL project
- Mux address: 43
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Uses a set of registers to divide the clock, and then some combinational logic to convert from binary to decimal for the display.

Puts the bottom 8 bits of the counter on the bidirectional outputs.

With all the inputs set to 0, the internal 24 bit compare is set to 10,000,000. This means the counter will increment by one each second.

If any inputs are non zero, then the input will be used as an bits 11 to 18 of the 24 bit compare register. Example: setting the inputs to 00010000 will program 16384 into the compare register. With a 10MHz clock the counter will increment ~610 times per second.

How to test

After reset, the counter should increase by one every second with a 10MHz input clock. Experiment by changing the inputs to change the counting speed.

Pinout

#	Input	Output	Bidirectional
0	compare bit 11	segment a	second counter bit 0
1	compare bit 12	segment b	second counter bit 1
2	compare bit 13	segment c	second counter bit 2
3	compare bit 14	segment d	second counter bit 3
4	compare bit 15	segment e	second counter bit 4
5	compare bit 16	segment f	second counter bit 5

#	Input	Output	Bidirectional
6	compare bit 17	segment g	second counter bit 6
7	compare bit 18	dot	second counter bit 7

test001 [44]

- Author: dmitry
- Description: Just a Test Tiny TapeOut
- GitHub repository
- Wokwi project
- Mux address: 44
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	none	segment a	none
1	none	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

Hodgkin-Huxley Chip Design [45]

- Author: Ethan Mulle
- Description: Implements the Hodgkin-Huxley model
- GitHub repository
- HDL project
- Mux address: 45
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Uses a set of registers to divide the clock, and then some combinational logic to convert from binary to decimal for the display.

Puts the bottom 8 bits of the counter on the bidirectional outputs.

With all the inputs set to 0, the internal 24 bit compare is set to 10,000,000. This means the counter will increment by one each second.

If any inputs are non zero, then the input will be used as an bits 11 to 18 of the 24 bit compare register. Example: setting the inputs to 00010000 will program 16384 into the compare register. With a 10MHz clock the counter will increment ~610 times per second.

How to test

After reset, the counter should increase by one every second with a 10MHz input clock. Experiment by changing the inputs to change the counting speed.

Pinout

#	Input	Output	Bidirectional
0	compare bit 11	segment a	second counter bit 0
1	compare bit 12	segment b	second counter bit 1
2	compare bit 13	segment c	second counter bit 2
3	compare bit 14	segment d	second counter bit 3
4	compare bit 15	segment e	second counter bit 4
5	compare bit 16	segment f	second counter bit 5

#	Input	Output	Bidirectional
6	compare bit 17	segment g	second counter bit 6
7	compare bit 18	dot	second counter bit 7

Character Selector [46]

- Author: Dakota W Winslow
- Description: A circuit to output latin characters on a 7-segment display
- GitHub repository
- Wokwi project
- Mux address: 46
- Extra docs
- Clock: 1000 Hz
- External hardware:

How it works

This project displays a user-selctable character on the 7-segment display. Input DIPs 1-6 are used to select the character to display. Inputs 7 and 8 are not connected. The display is driven one segment at a time, so a high clock rate is required to see thew character. Characters are a mix of upper and lower case, preferring whichever is more recognizable. See the wikipedia page on 7-segment display representations for reference [https://en.wikipedia.org/wiki/Seven-segment_display_character_representations].

How to test

Connect the clock line to the ocillator (or press the clock button REALLY fast). Then, use thew DIP switches to enter a 6-digit binary number corresponding to the character to be displayed. 0-25 for a-z, then 26:[space], 27:[_], 28:[-], 29:[.], 30:[!], 31:["]. If only one segment is displayed, make sure your clock is set properly!

Pinout

#	Input	Output	Bidirectional
0	dip0	segment a	none
1	dip1	segment b	none
2	dip2	segment c	none
3	dip3	segment d	none
4	dip4	segment e	none
5	dip5	segment f	none
6	not connected	segment g	none
7	not connected	dot	none

Intructouction to PRBS [47]

- Author: Chih-Kuan Ho and David Parent
- Description: This is a simple design used to verify the design flow, so that we can teach lower division college studdnts IC desgin.
- GitHub repository
- Wokwi project
- Mux address: 47
- Extra docs
- Clock: 10k Hz
- External hardware:

How it works

This takes a 4 bit LSFR confugred as a PRBS=$X^{3+X}2+1$ Reset sets four DFF to zero to make sure the osiclation starts. This uses XNOR because there was a synth warning The outpus are for a 7 segment displant and the last out put is for the PRBS

How to test

Set the clock, pulse reser and it it should givne RBS stream.

Pinout

#	Input	Output	Bidirectional
0	Clock	segment a	none
1	Reset	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	segment PRBS	none

tto5 Supercon Project [64]

- Author: Ryan Young
- Description: quick full adder design
- GitHub repository
- Wokwi project
- Mux address: 64
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	none	segment a	none
1	none	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

Delta Modulation Spike Encoding [65]

- Author: John Madden
- Description: Delta Modulation for Spiking Neural Networks (SNN) based on snnTorch's implementation.
- GitHub repository
- HDL project
- Mux address: 65
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

A spike is generated if the difference between the current and previous data inputs is greater than a specified input threshold. The design is meant to mimic the implementation of delta modulation in the snnTorch python package. Each clock cycle is treated as an input/output, therefore there can be consecutive spikes that appear to be constantly high.

The input parameter, `off_spike`, enables spike generation when negative threshold is exceeded. A negative spike is represented by `spike[1] = 0`. A positive spike is represented by `spike[0] = 1`.

All numerical inputs and outputs are unsigned 4-bit integers. You are able to (1) input the `data` value, the input (2) the `threshold` for a spike to be generated, and (3) a value for the previous register for debugging.

The previous data register is included to be facilitate debugging with the ability to read the current value in the register and force the register to a specific value.

How to test

The module is intended to have a digital input, such as an ADC with a parallel output that is directly fed into the `data` input with a shared `clk` signal. The `threshold` is meant to be tied to a constant value. The module outputs through `spikes` net.

For simpler testing, the input does not need to be matched to the `clk`. With the `threshold` set, `data` can be changed and spikes can be viewed on an oscilloscope.

Pinout

#	Input	Output	Bidirectional
0	input threshold bit 0	spike bit 0	parameter off_spike
1	input threshold bit 1	spike bit 1	input load_prev bit
2	input threshold bit 2	nc, constant output low	nc, constant output low
3	input threshold bit 3	nc, constant output low	nc, constant output low
4	input data bit 0	reg prev bit 0	input force_prev bit 0
5	input data bit 1	reg prev bit 1	input force_prev bit 1
6	input data bit 2	reg prev bit 2	input force_prev bit 2
7	input data bit 3	reg prev bit 3	input force_prev bit 3

GameOfLife [66]

- Author: Eric Moderbacher
- Description: a single cell's logic for conways game of life
- GitHub repository
- Wokwi project
- Mux address: 66
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	none	segment a	none
1	none	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

Reflex Game [67]

- Author: Alan
- Description: Reflex Game where
- GitHub repository
- HDL project
- Mux address: 67
- Extra docs
- Clock: 10 Hz
- External hardware:

How it works

It doesn't. boom.

How to test

Make a test bench. boom.

Pinout

#	Input	Output	Bidirectional
0	none	segment a	none
1	none	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

Logic Gates Tapeout [68]

- Author: Alexandre Ney Guimaraes
- Description: TesteX
- GitHub repository
- Wokwi project
- Mux address: 68
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	none	segment a	none
1	none	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

Stream Cipher w/ LSR (8 bit key) [69]

- Author: Fiona Fisher
- Description: Uses a stream cipher and linear shift register to encrypt a message.
- GitHub repository
- HDL project
- Mux address: 69
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Holds an internal linear shift register of eight bytes.

When encrypting, each increment of IO_0 takes in the byte currently on the inputs, XORs it with the most recent byte stored in the LSR, and then puts it into the LSR.

When not encrypting, the message can be viewed by putting an index zero to seven on the inputs. The output will either be the encrypted message or the decrypted message, based on I0_2.

The LSR can only be reset with the rst_n signal. If more than sixteen bytes are inputted into the LSR without resetting, encrypted bytes will be lost, meaning the decryption of the last byte will not be accurate.

You can toggle between encryption and viewing the message with IO_1. You do not have to finish inputting the message before viewing the current encryption.

How to test

Set IO_1 to high to indicate encryption. Place a number on the input. Set the IO_0 to high to put it into the LSR. Set the IO_0 to low before adding the next number. Repeat up to seven times.

Set IO_1 to low to view the message. Use IO_2 to toggle between viewing the message encrypted (high) or decrypted (low). Use the input to indicate the index of the message you want to view.

Reset to place a new message on the LSR.

Pinout

#	Input	Output	Bidirectional
0	bit0	bit0	inc (indicate that input to encryption is valid)
1	bit1	bit1	encrypt
2	bit2	bit2	view (high shows encrypted message, low shows decrpted message)
3	bit3	bit3	none
4	bit4	bit4	none
5	bit5	bit5	none
6	bit6	bit6	none
7	bit7	bit7	none

tt5modifyd [70]

- Author: HMaxMax
- Description: triple or gate
- GitHub repository
- Wokwi project
- Mux address: 70
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	none	segment a	none
1	none	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

ALU Chip [71]

- Author: Devan Grover & Siddharth Kunisetty
- Description: ALU Chip that outputs 7 Segment
- GitHub repository
- HDL project
- Mux address: 71
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

This is a simple, 4 bit ALU that outputs its result on a 7 Segment Display.

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	alu_in_1 [7:4] - First input into the ALU	7 Segment Out [6:0] - Output to the 7 Segment display	alu_out [7:4] (OUT) - Output BCD value of operation
1	alu_in_2 [3:0] - Second input into the ALU	None [7] - NC	alu_op_in [3:0] (IN) - Input operation for the ALU (ADD, SUBTRACT, AND, OR, EQUALS, NOT, GT, LT)
2	n/a	n/a	n/a
3	n/a	n/a	n/a
4	n/a	n/a	n/a
5	n/a	n/a	n/a
6	n/a	n/a	n/a
7	n/a	n/a	n/a

Tapeout Test [72]

- Author: bignug13
- Description: For Supercon 2023: Some logic gates that add things
- GitHub repository
- Wokwi project
- Mux address: 72
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Inputs 1-4 (Values: 1,2,4,8) and Inputs 5-8 (Values: 1,2,4,8) are added together and reflected in Outputs 1-5 (Values: 1,2,4,8,16).

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	none	segment a	none
1	none	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

Calculator chip [73]

- Author: Rylan Morgan
- Description: calculator
- GitHub repository
- HDL project
- Mux address: 73
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Basic ALU. Use the input pins to specify an 8 bit number and output pins to view result. IO pin 0 is the enter pin, assert high to enter command/value. IO pins 4-1 are used to select the command for the ALU. IO pins 7-5 are for flags. Ops: 0x0: add 0x1: subtract 0x2: bitwise or 0x3: bitwise and 0x4: bitwise xor 0x5: left shift by 1 0x6: right shift by 1 (logic) 0x7: right shift by 1 (arithmetic) 0x8: 2s compliment negate 0x9: bitwise invert 0xA: reverse bitpatern 0xB: unused 0xC: unused 0xD: unsigned input < output 0xE: unsigned input > output 0xF: input == output

How to test

enter a bunch of numbers and ops, should work great

Pinout

#	Input	Output	Bidirectional
0	NumIn [7:0] number in	NumOut [7:0] output	OpIn [3:0] what op to run
1	Reset	n/a	Enter enter command
2	Clock	n/a	Flags [2:0] overflow, negative, and zero flag
3	n/a	n/a	n/a
4	n/a	n/a	n/a
5	n/a	n/a	n/a
6	n/a	n/a	n/a
7	n/a	n/a	n/a

Shifty Snakey [74]

- Author: poynting
- Description: Shift register snake demo
- GitHub repository
- Wokwi project
- Mux address: 74
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	none	segment a	none
1	none	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

Synth [75]

- Author: Gyanepsaa Singh
- Description: Sound synthesizer
- GitHub repository
- HDL project
- Mux address: 75
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Basic sound synthesizer module: generates sound signal, modulates it with an ASDR envelope, and outputs it.

Pinout

#	Input	Output	Bidirectional
0	select triangle or sawtooth waveform	sound output	2-bit attack
1	None	n/a	2-bit decay
2	sampling frequency clock	n/a	2-bit sustain
3	hold a note	n/a	2-bit release
4	4 frequency selection bits	n/a	n/a
5	n/a	n/a	n/a
6	n/a	n/a	n/a
7	n/a	n/a	n/a

Sawtooth Generator [76]

- Author: Mooneer Salem
- Description: Generates sawtooth waves for use as audio.
- GitHub repository
- HDL project
- Mux address: 76
- Extra docs
- Clock: 50000000 Hz
- External hardware:

How it works

This project increments a counter from 0 to 25,000,000 and back to zero again. The current value of this counter is then passed into a PDM modulator to generate the output. Filtering it with a low pass filter (designed for use in the audio range, recommended cutoff ~30 KHz) and then amplifying the result will result in usable audio.

How to test

Add a suitable RC low pass filter to output pin 7. This can be probed by an oscillocope as-is. To listen to the audio, the output of the RC filter should be attached to a suitable audio amplifier.

Pinout

#	Input	Output	Bidirectional
0	none	PCM sawtooth wave output (bit 9)	PCM sawtooth wave output (bit 9)
1	none	PCM sawtooth wave output (bit 10)	PCM sawtooth wave output (bit 10)
2	none	PCM sawtooth wave output (bit 11)	PCM sawtooth wave output (bit 11)
3	none	PCM sawtooth wave output (bit 12)	PCM sawtooth wave output (bit 12)
4	Frequency left shift amount (in bits) – bit 0	PCM sawtooth wave output (bit 13)	PCM sawtooth wave output (bit 13)

#	Input	Output	Bidirectional
5	Frequency left shift amount (in bits) – bit 1	PCM sawtooth wave output (bit 14)	PCM sawtooth wave output (bit 14)
6	Frequency left shift amount (in bits) – bit 2	PCM sawtooth wave output (bit 15)	PCM sawtooth wave output (bit 15)
7	Frequency left shift amount (in bits) – bit 3	PDM sawtooth wave output (needs LPF)	PDM sawtooth wave output (needs LPF)

Blinking A [77]

- Author: Ariella Eliassaf
- Description: Blink an A on the 7segment display
- GitHub repository
- Wokwi project
- Mux address: 77
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	none	segment a	none
1	none	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

Supercon 2023 [78]

- Author: Alec Probst
- Description: Supercon 2023 Tiny Tapeout Submission. Displays a white pixel and blue background through VGA. Makes use of Cutout1's VGA Flappy bird code.
- GitHub repository
- HDL project
- Mux address: 78
- Extra docs
- Clock: 25MHz Hz
- External hardware: A VGA adaptor

How it works

Correctly times the signal Digital outputs for VGA

How to test

Connect to a VGA connector. Use a D2A Converter.

Pinout

#	Input	Output	Bidirectional
0	none	R1	none
1	none	G1	none
2	none	B1	none
3	none	vsync	none
4	none	R0	none
5	none	G0	none
6	none	B0	none
7	none	hsync	none

Sparsity Aware Matrix Vector Multiplication [79]

- Author: Test
- Description: Count up to 10, one second at a time.
- GitHub repository
- HDL project
- Mux address: 79
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Uses a set of registers to divide the clock, and then some combinational logic to convert from binary to decimal for the display.

Puts the bottom 8 bits of the counter on the bidirectional outputs.

With all the inputs set to 0, the internal 24 bit compare is set to 10,000,000. This means the counter will increment by one each second.

If any inputs are non zero, then the input will be used as an bits 11 to 18 of the 24 bit compare register. Example: setting the inputs to 00010000 will program 16384 into the compare register. With a 10MHz clock the counter will increment ~610 times per second.

How to test

After reset, the counter should increase by one every second with a 10MHz input clock. Experiment by changing the inputs to change the counting speed.

Pinout

#	Input	Output	Bidirectional
0	compare bit 11	segment a	second counter bit 0
1	compare bit 12	segment b	second counter bit 1
2	compare bit 13	segment c	second counter bit 2
3	compare bit 14	segment d	second counter bit 3
4	compare bit 15	segment e	second counter bit 4
5	compare bit 16	segment f	second counter bit 5

#	Input	Output	Bidirectional
6	compare bit 17	segment g	second counter bit 6
7	compare bit 18	dot	second counter bit 7

Ring Oscillator and Clock Source Switch [96]

- Author: Dave Cox
- Description: A series of NOT gates with whip outs to measure self oscillation, and a clock switch
- GitHub repository
- Wokwi project
- Mux address: 96
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

There are two functional blocks - a ring oscillator with multiple taps, and a glitchless clock switch.

How to test

To test the oscillator - Input 0 is input to first inverter in the oscillator. One of the inverted outputs either 0 (slowest), 1 (mid), or 2 (fastest) should be connected to input 0. To test the clock switch - input 1 selects between clock0 (on input 2) and clock 1 (on input 3). Selected CLK appears on OUT3

Pinout

#	Input	Output	Bidirectional
0	0 Osc In	0 OSC OutSlow/segment a	none
1	1 ClkSel	1 OSC OutMid/segment b	none
2	2 CLK0 in	2 OSC OutFast/segment c	none
3	3 Clk1 in	3 ClkOut/segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

Matrix Vector Multiplication (Verilog Demo) [97]

- Author: Aled dela Cruz
- Description: Multiplies inputted vector by
- GitHub repository
- HDL project
- Mux address: 97
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Used to multiply a vector by a matrix. The matrix is initially filled with all zeros, but can be set to 0s by the user. The first input switch, when flipped, will initialize a matrix multiplication between the current set vector and the empty matrix. If the user wants to set the matrix, it can done line by line using the second bit. Flipping the second bit causes the current 6 right most bits to be set to a certain value in the matrix. NO current functionality to know which row of the matrix is set

How to test

After reset, the counter should increase by one every second with a 10MHz input clock. Experiment by changing the inputs to change the counting speed.

Pinout

#	Input	Output	Bidirectional
0	compare bit 11	segment a	second counter bit 0
1	compare bit 12	segment b	second counter bit 1
2	compare bit 13	segment c	second counter bit 2
3	compare bit 14	segment d	second counter bit 3
4	compare bit 15	segment e	second counter bit 4
5	compare bit 16	segment f	second counter bit 5
6	compare bit 17	segment g	second counter bit 6
7	compare bit 18	dot	second counter bit 7

IDK WHAT TO DO [98]

- Author: Benjamin Meyer
- Description: Help me
- GitHub repository
- Wokwi project
- Mux address: 98
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	none	segment a	none
1	none	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

7-segment display logic system [99]

- Author: Abrez Hussain, Dean Xavier Batres, Nathan Chau
- Description: 7 segment display counter
- GitHub repository
- Wokwi project
- Mux address: 99
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Binary counter connected to the clock.

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	none	segment a	none
1	none	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

Hardware Trojan Example [100]

- Author: Jeremy Hong
- Description: Simple hardware trojan circuit described by Ryan Cornateanu in a medium article
- GitHub repository
- Wokwi project
- Mux address: 100
- Extra docs
- Clock: 0 Hz
- External hardware: No external hardware required, just TinyTapeout Carrier Board

How it works

Based off of medium article by Ryan Cornateanu: "Hardware Trojans IUnder a Microscope https://ryancor.medium.com/hardware-trojans-under-a-microscope-bf542acbcc29

How to test

Use DIP switches as input, 1- 4 is for normal circuit that would be considered "secure", and 5 - 8 for compromised circuit with embedded Hardware Trojan

Pinout

#	Input	Output	Bidirectional
0	r0_normal	Output from r0_normal - r3_normal circuit	none
1	r1_normal	segment b not used	none
2	r2_normal	segment c not used	none
3	r3_normal	segment d not used	none
4	r0_trojan	segment e not used	none
5	r1_trojan	segment f not used	none
6	r2_trojan	Output from r0_trojan - r3_trojan circuit	none
7	r3_trojan	dot not used	none

Analog Clock [101]

- Author: Justin Hui
- Description: LED controller for an Analog Clock taking a 1Hz internal clk input
- GitHub repository
- Wokwi project
- Mux address: 101
- Extra docs
- Clock: 1 Hz
- External hardware: 16 leds

How it works

increments seconds Counter for 60 sec. increments min Counter for 60 min. increments hour counter for 12 hours

all daisy chained.

The LEDs will show the hour and the last 15min increment

How to test

connect leds to each output pin as described below

RST will set the time to 11:59.

Input Pins 7/8 are used to set the time, by toggling those it should increment the internal clock by 1 min/hour. You should see the hour output update immediately. The min output will only change once the next 15min increment passes

Input Pin 1 will stop the clock when high

Pinout

#	Input	Output	Bidirectional
0	none	hour 12 led	hour 8 led
1	stop the clock	hour 1 led	hour 9 led
2	none	hour 2 led	hour 10 led
3	none	hour 3 led	hour 11 led
4	none	hour 4 led	0 min
5	none	hour 5 led	15 min

#	Input	Output	Bidirectional
6	increment min counter by 1	hour 6 led	30 min
7	increment hour counter by 1	hour 7 led	45 min

7 segment display [102]

- Author: Shravyasai Koushik
- Description: Converts binary input up until 9 and some alphabets into 7 segment display
- GitHub repository
- Wokwi project
- Mux address: 102
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Based on a simple logic circuit consisting of OR and AND gates.

How to test

Utilise first 4 switches as binary input from 0-15.

Pinout

#	Input	Output	Bidirectional
0	red switch board	7 segment display	none
1	none	none	none
2	none	none	none
3	none	none	none
4	none	n/a	none
5	none	n/a	none
6	none	n/a	none
7	none	n/a	none

W_Li_10/28 [103]

- Author: Wendi Li
- Description: The circuit controls the seven segment display to diplay the authors initial and the date the circuit is designed
- GitHub repository
- Wokwi project
- Mux address: 103
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	none	segment a	none
1	none	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

Supecon Gate Play [104]

- Author: Adam Chasen
- Description: One of each with some flippy floppies
- GitHub repository
- Wokwi project
- Mux address: 104
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	none	segment a	none
1	none	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

ECE 183 - Integrate and Fire Network Chip Design [105]

- Author: Manju Shettar
- Description: Count up to 10, one second at a time.
- GitHub repository
- HDL project
- Mux address: 105
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

This project emulates a two-layer neural network using a series of integrate and fire neurons defined in Verilog. The neurons accumulate incoming spikes and when their cummulative signal surpasses a defined threshold, they generate a spike and reset their potential.

The network is composed of two neuron layers. The first input layer accepts two 16-bit inputs, corresponding to external stimuli or current, which the neurons of the first layer will processes. Based on the internal states of these neurons, they may or may not fire to generate spikes.

The spikes that are generated from the first layer are used as input to the second layer. In this model, a spike is defined asw a binary high signal, translated into a 16-bit value to mimic the input current standard. If there is no spike, there is no input current (zero).

The second layer defines the neural network's output. Each neuron in the second layer may generate a spike, which is represented again with a bit.

How to test

Testing involves applying different input stimuli (current) and observing spiking states.

By changing the values of 'input1' and 'input2', and observing the output layers, we can observe different spiking outputs from the neural network.

Pinout

#	Input	Output	Bidirectional
0	{'input1[15:0]': "16-bit stimulus input to the first layer's first neuron."}	{'output_layer21': 'Spiking status of the first neuron in the second layer'}	{'uio_in': 'Reserved for future use.'}
1	{'input2[15:0]': "16-bit stimulus input to the first layer's second neuron."}	{'output_layer2[0]': 'Spiking status of the second neuron in the second layer'}	n/a
2	{'clk': 'Clock signal.'}	n/a	n/a
3	{'rst_n': "Reset signal; when low, it resets the neurons' current and output spikes."}	n/a	n/a
4	n/a	n/a	n/a
5	n/a	n/a	n/a
6	n/a	n/a	n/a
7	n/a	n/a	n/a

tto5 [106]

- Author: kl
- Description: tto5
- GitHub repository
- Wokwi project
- Mux address: 106
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

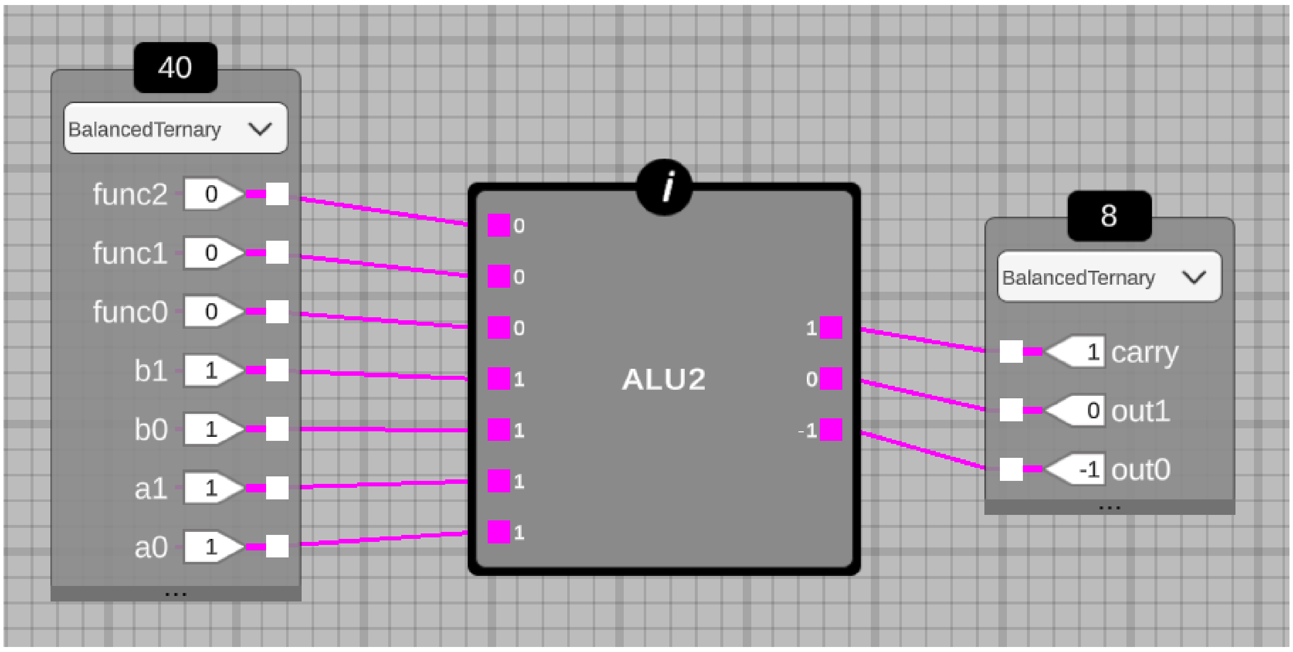
How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	none	segment a	none
1	none	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

REBEL-2 Balanced Ternary ALU [107]



- Author: Ole Christian Moholth and Steven Bos
- Description: This 2-trit balanced ternary ALU is part of the REBEL-2 balanced ternary logic CPU
- GitHub repository
- HDL project
- Mux address: 107
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

WIP. This balanced ternary ALU has several operations based on a novel REBEL-2 ISA. Operations include MIN,MAX,ADD,SUB,MUL,CMP,SHFT and can be done trit-wise or word-wise. This project is designed, generated and verified with Mixed Radix Circuit Synthesizer (MRCS).

How to test

There are many automated test included for MRCS. A verilog testbench for FPGA and FPGA constraint file is WIP.

Pinout

#	Input	Output	Bidirectional
0	ui_in[7] func2H	uo_out[7] carryH	uio_in[7] b0H
1	ui_in[6] func2L	uo_out[6] carryL	uio_in[6] b0L
2	ui_in[5] func1H	uo_out[5] out1H	uio_in[5] a1H
3	ui_in[4] func1L	uo_out[4] out1L	uio_in[4] a1L
4	ui_in[3] func0H	uo_out[3] out0H	uio_in[3] a0H
5	ui_in2 func0L	uo_out2 out0L	uio_in2 a0L
6	ui_in1 b1H	uo_out1 unused	uio_in1 unused
7	ui_in[0] b1L	uo_out[0] unused	uio_in[0] unused

Stochastic Multiplier [108]

- Author: David Parent
- Description: Creates a PRBS stream whre the probability of a 1 is the multiplication of two, two bit vectors.
- GitHub repository
- Wokwi project
- Mux address: 108
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

See: P. K. Gupta and R. Kumaresan, "Binary multiplication with PN sequences," IEEE Trans. Acoust., vol. 36, no. 4, pp. 603–606, Apr. 1988, doi: 10.1109/29.1564.

How to test

Set A and B and clock. Toggle reset low to make sure PRBS gen starts.

Pinout

#	Input	Output	Bidirectional
0	CLK	PRBS of A*B	none
1	RESET	PRBS of A	none
2	A	PRBS of B	none
3	B	PRBS	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

7 segment seconds - count down [109]

- Author: Jeff DiCorpo
- Description: Count down from 9, one second at a time.
- GitHub repository
- HDL project
- Mux address: 109
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Uses a set of registers to divide the clock, and then some combinational logic to convert from binary to decimal for the display.

Puts the bottom 8 bits of the counter on the bidirectional outputs.

With all the inputs set to 0, the internal 24 bit compare is set to 10,000,000. This means the counter will increment by one each second.

If any inputs are non zero, then the input will be used as an bits 11 to 18 of the 24 bit compare register. Example: setting the inputs to 00010000 will program 16384 into the compare register. With a 10MHz clock the counter will increment ~610 times per second.

How to test

After reset, the counter should increase by one every second with a 10MHz input clock. Experiment by changing the inputs to change the counting speed.

Pinout

#	Input	Output	Bidirectional
0	compare bit 11	segment a	second counter bit 0
1	compare bit 12	segment b	second counter bit 1
2	compare bit 13	segment c	second counter bit 2
3	compare bit 14	segment d	second counter bit 3
4	compare bit 15	segment e	second counter bit 4
5	compare bit 16	segment f	second counter bit 5

#	Input	Output	Bidirectional
6	compare bit 17	segment g	second counter bit 6
7	compare bit 18	dot	second counter bit 7

TT05 Submission [110]

- Author: Alexander Whittemore
- Description: I don't know what this project does yet but hopefully it's cool.
- GitHub repository
- Wokwi project
- Mux address: 110
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	none	segment a	none
1	none	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

Leaky Integrate-and-Fire Neuron [111]

- Author: Mariana_Huerta
- Description: Implement a LIF neuron in 130 nm CMOS
- GitHub repository
- HDL project
- Mux address: 111
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Start by applying an input current injection to the LIF neuron.

This gets added to a membrane potential which decays by a factor beta over time.

When the membrane potential exceeds the threshold, a spike is triggered.

How to test

Reset the circuit to set the membrane potential to 0.

The inputs can be changed to vary the current. A higher current will result in a higher spike rate.

Pinout

#	Input	Output	Bidirectional
0	current bit 11	membrane potential bit a	spike bit 0
1	current bit 12	membrane potential bit b	unspecified
2	current bit 13	membrane potential bit c	unspecified
3	current bit 14	membrane potential bit d	unspecified
4	current bit 15	membrane potential bit e	unspecified
5	current bit 16	membrane potential bit f	unspecified
6	current bit 17	membrane potential bit g	unspecified
7	current bit 18	membrane potential bit h	unspecified

Count via LFSR [128]

- Author: Eric Smith
- Description: Count via LFSR and display on 7 segment
- GitHub repository
- Wokwi project
- Mux address: 128
- Extra docs
- Clock: 1 Hz
- External hardware:

How it works

after sync reset on io[0], send some clocks. increment count on posedge clock

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	reset_n	segment a	none
1	none	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

I2C BERT [130]

- Author: Darryl Miles
- Description: I2C Bit Error Rate Test
- GitHub repository
- HDL project
- Mux address: 130
- Extra docs
- Clock: 10000000 Hz
- External hardware: I2C Controller/RP2040

How it works

This text will be updated nearer the scheduled TT05 redistribution time (early 2024) along with the project github README.md and gh-pages documentation. Please regenerate your documentation.

Issue synchronous reset, ensure interface inputs are set to zero. Power-on-reset configuration is possible via the input pins, see documentation.

This design is an I2C peripheral that implements an 8-bit ALU over I2C. The purpose of the ALU is to allow pattern testing to occur and read back the accumulated result.

There are a few clocking modes, the default uses SCL pin as per the standard.

Connection to I2C interface:

- uio2 = SDA (should be direct to RP2040 pin with capable mode)
- uio[3] = SCL (shouid be direct to RP2040 pin with capable mode)

When in open-drain mode the standard pull-up resistor is in the order of 4k7 to 10k and no more than 400pF capacitance on lines. Higher speeds my require attention to those metrics for your setup. The project is peripheral only and does not drive SCL. So open-drain or push-pull can be used by the controller no matter the mode setup in this project.

Power-on-reset configuration (set all zero for standard mode):

- ui_in1 sets CLOCKMUX to use divider
- ui_in2 sets PUSHPULL I2C bus mode (by default open-drain is in use)
- ui_in[3] activates DIV12 divider setup on reset (default is 10Mhz for 10Khz)
- {uio_in[7:0], ui_in[7:4]} is the DIV12 value to use

The design is based around a high-speed clock, at default speed of 10MHz with

Other than the default divider setup for CLOCKMUX mode there is no restriction upon the system clock used, other than trying to operate at low ratios of system-clock:SCL. The design has been simulated from “6:1” upto 1000000:1. Maybe lower than 6:1 are possible.

How to test

RP2040 code is expected to be provided to conduct testing based on simulation expectations.

Pinout

#	Input	Output	Bidirectional
0	i2cSampleDivisor bit0	segment a	none
1	i2cSampleDivisor bit1	segment b	none
2	none	segment c	I2C SCL (bidi)
3	none	segment d	I2C SDA (bidi)
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	powerOnSense (out)

tt05-loopback tile with input skew measurement [132]

- Author: Eric Smith
- Description: Count up to 10, one second at a time.
- GitHub repository
- HDL project
- Mux address: 132
- Extra docs
- Clock: 10000000 Hz
- External hardware: programmable delay lines on inputs

How it works

Need to write this

How to test

Need to write this

Pinout

#	Input	Output	Bidirectional
0	compare bit 11	segment a	second counter bit 0
1	compare bit 12	segment b	second counter bit 1
2	compare bit 13	segment c	second counter bit 2
3	compare bit 14	segment d	second counter bit 3
4	compare bit 15	segment e	second counter bit 4
5	compare bit 16	segment f	second counter bit 5
6	compare bit 17	segment g	second counter bit 6
7	compare bit 18	dot	second counter bit 7

Flappy VGA [134]

- Author: Daniel Robinson
- Description: A simple flappy bird clone with a button input and 640x480 VGA output. 25MHz clock required.
- GitHub repository
- HDL project
- Mux address: 134
- Extra docs
- Clock: 25000000 Hz
- External hardware: Some kind of VGA adapter needed. Compatible with Tiny VGA PMOD. Also needs a debounced button that goes low when pressed on ui_in[0].

How it works

There are three main modules in the design. The vgaControl module takes in the clock and outputs the horizontal and vertical sync signals, and provides the current pixel coordinate to the bitGen module. The gameControl module takes the button input and updates the game state (bird position, pipe position, hole position, score, etc) each frame. The bitGen module takes in the game state and pixel coordinate and outputs the color that the current pixel should be.

How to test

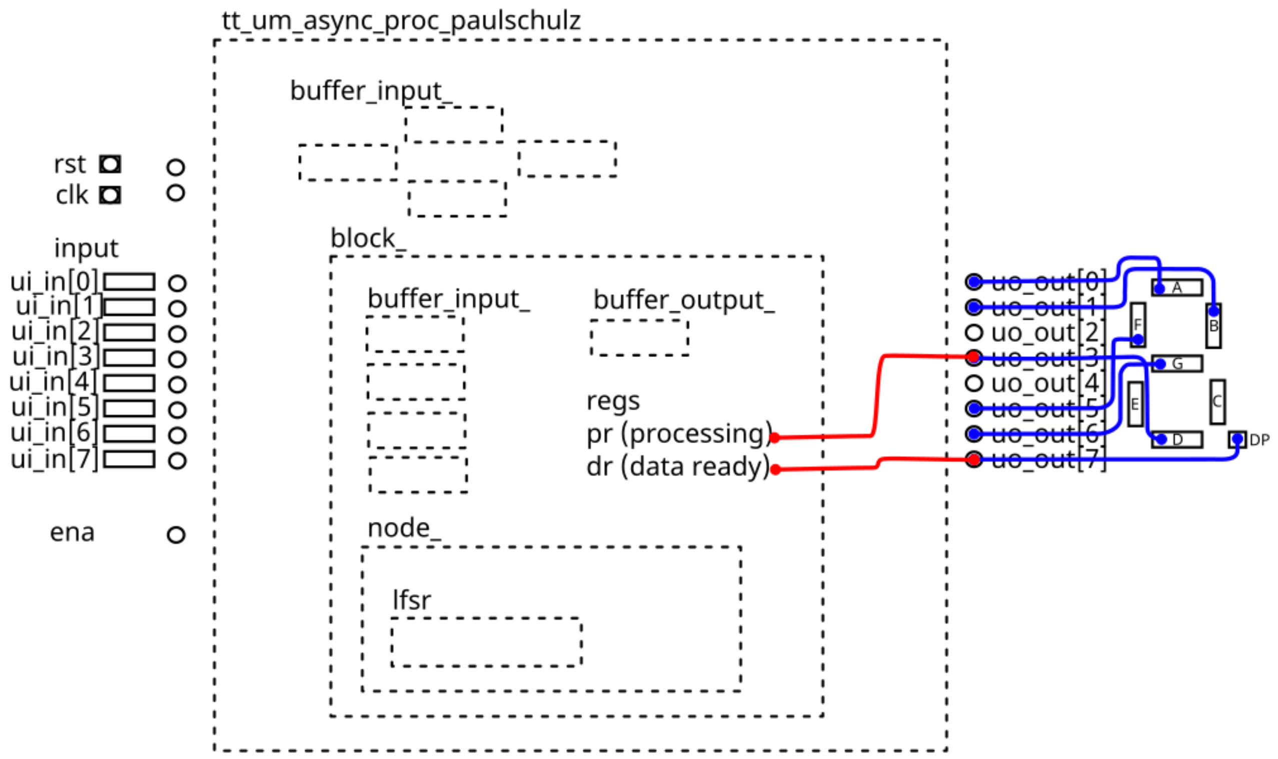
The clock input should be set to 25MHz (or 25.179MHz, either should be close enough). ui_in[0] should be connected to a debounced button that goes low when pressed. The VGA output is compatible with the Tiny VGA PMOD (https://tinytapeout.com/specs/pinouts/). Once everything is connected, a reset may need to be triggered before normal operation. An image with a yellow square, green pillars, and blue background should appear. Pressing the button should cause the bird to flap and start moving towards the pipes. The goal is to go through the gap in the pipes. Your score will count up in binary on the bidirectional pins each time you successfully make it through a pipe.

Pinout

#	Input	Output	Bidirectional
0	Button	R1	score[0]

#	Input	Output	Bidirectional
1	none	G1	score1
2	none	B1	score2
3	none	vsync	score[3]
4	none	R0	score[4]
5	none	G0	score[5]
6	none	B0	score[6]
7	none	hsync	score[7]

Asynchronous Parallel Processor Demonstrator [136]



- Author: Paul Schulz
- Description: Implementation for an Asynchronous Parallel Processor
- GitHub repository
- HDL project
- Mux address: 136
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

See Github: https://github.com/PaulSchulz/tt05-async-proc

This circuit is an investigation into an asynchronous parallel processor design. (Work in progress.)

Note: This is a very early design and doesn't do very much.

A processing node follows the following state flow:

- Wait for valid data;
- Process the data to produce an output value, and let neighboring nodes know that processing in being done;
- Make the result available; and wait for more data to process.

In this example, the processing node is doing a calculation on four(4) inputs of 4 bits. The calculation is based on a deconstruction of the the "Arctic Circle Theorem" model. (video)

In future designs: 1) allow the processing nodes to be programmable; 2) layout a multinode interconnected array (with global clocking); 3) investigate a clockless version; and 4)

How to test

Reset to clear internal buffers.

Set inputs and load them into the input buffers.

Set clock to calculate result and display on outputs.

Experiment by trying different input values.

Pinout

#	Input	Output	Bidirectional
0	input bit 0	segment a (up)	bit 0 (not used)
1	input bit 1	segment b (right)	bit 1 (not used)
2	input bit 2	segment c	bit 2 (not used)
3	input bit 3	segment d (processing)	bit 3 (not used)
4	load input 1 (right)	segment e	bit 4 (not used)
5	load input 2 (up)	segment f (left)	bit 5 (not used)
6	load input 3 (left)	segment g (down)	bit 6 (not used)
7	load input 4 (down)	dot (data ready)	bit 7 (not used)

Hex Countdown [138]

- Author: Jorge Gómez y Felipe Gómez
- Description: Hexadecimal countdown from F to 0
- GitHub repository
- Wokwi project
- Mux address: 138
- Extra docs
- Clock: 1 Hz
- External hardware:

How it works

Statemachine that on each clock pulse subttracts one on the 7 segment display. Starting in F and finishing on 0.

How to test

Connect a 1Hz square signal as clock and by turning input 1 to 1, the counter will start counting down stopping on 0. On reset the value will return to F.

Pinout

#	Input	Output	Bidirectional
0	Activation signal	segment a	none
1	none	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

Matrix multiply coprocessor (8x8 1bit) [140]

- Author: Nick Hay
- Description: Implements a 1bit 8x8 matrix multiple using a systolic array.
- GitHub repository
- HDL project
- Mux address: 140
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

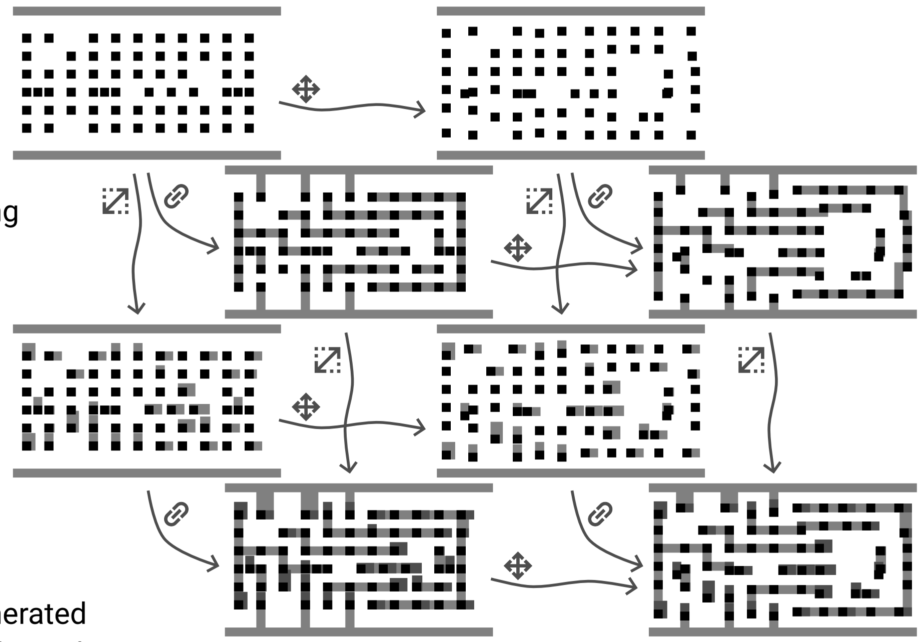
Explain how to test your project

Pinout

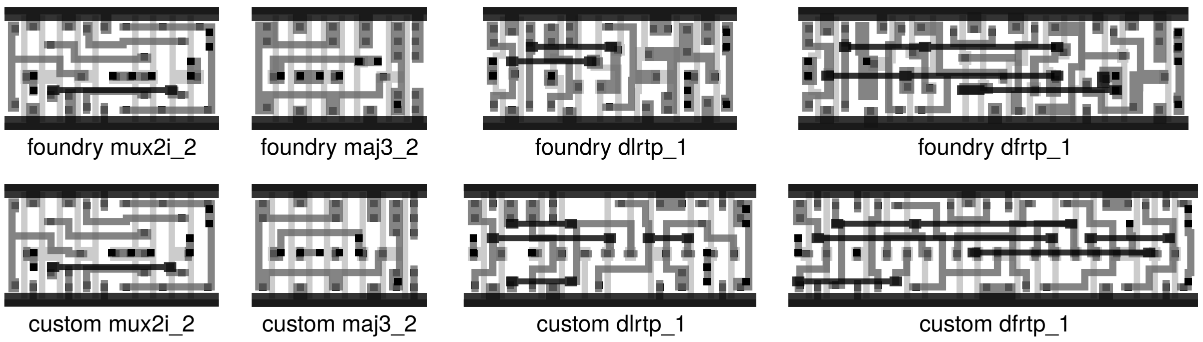
#	Input	Output	Bidirectional
0	data 0	output 0	sayhi: outputs greeting
1	data 1	output 1	read out multiplied matrix
2	data 2	output 2	use xor rather than or
3	data 3	output 3	none
4	data 4	output 4	none
5	data 5	output 5	none
6	data 6	output 6	none
7	data 7	output 7	none

Standard cell generator and tester [142]

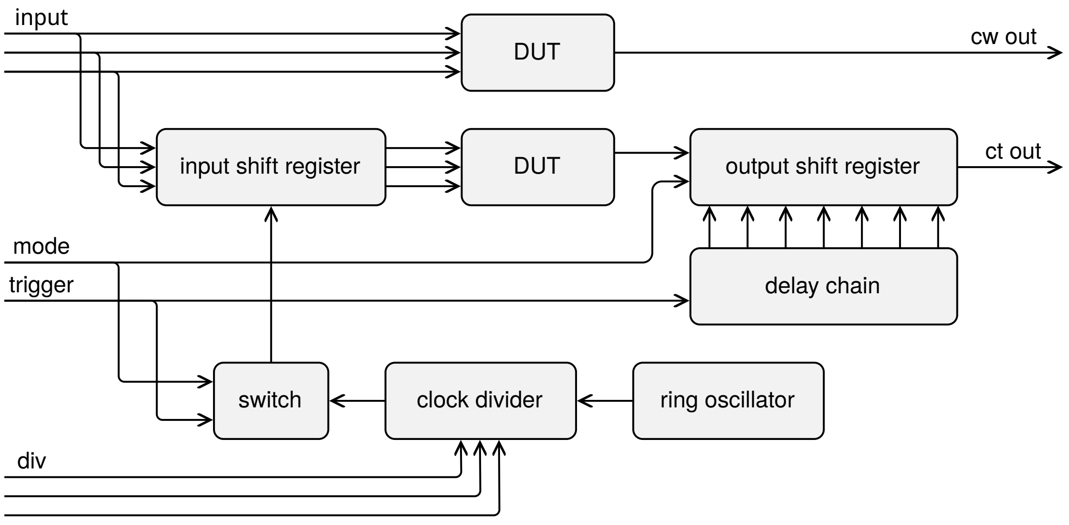
(a) steps of generating a standard cell layer



(b) example cells generated vs their foundry counterparts



(c) structure of digital design to test the custom cells



- Author: htfab

- Description: Contains a sky130 compatible standard cell generator, a few example cells generated, and a TinyTapeout design for testing them
- GitHub repository
- HDL project
- Mux address: 142
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

This project consists of three parts:

- a standard cell generator for sky130, written in python using the gdstk library
- four example cells ready to drop into the openlane flow
- a digital design wrapping the example cells in an instrumentation framework

Cell generator

Cells are built from a discrete representation. For each layer, blocks are placed in some tiles of a 6 $\times$ n grid. These blocks are then shifted and resized in fixed increments, and certain pairs of adjacent blocks are connected to each other as shown in figure *(a)*.

Generated cells are then written to gds, lef, mag & maglef files to allow using them in the openlane flow. Verilog models and liberty characterization data have to be created separately. Cells are designed to be mixed-and-matched with cells from the `sky130_fd_sc_hd` library.

The cell generator lives in the `pdk-gen` directory of the source tree. The generator itself is in `skygen.py` while inputs for the example cells are in `cells.py`.

Example cells

Four cells from the `sky130_fd_sc_hd` library were recreated using the generator. They are shown in figure *(b)*, with more detailed images in the `README.md`.

The `pdk` directory is structured in the same way as the sky130 pdk so that you can copy its contents into `$PDK_ROOT/sky130A/libs.ref/sky130_fd_sc_hd` to use the cells with openlane. Just don't use them for anything serious, they are not that thoroughly tested.

The subdirectories `gds`, `lef`, `mag` and `maglef` are outputs from the generator. Netlists in `spice` were extracted using magic while models in `verilog` and characterization data in `lib` were just copied from the corresponding foundry cells.

There are some quick analog tests using ngspice in the `pdk-test` directory.

TinyTapeout design

A digital design wrapping the example cells in an instrumentation framework is included in the TinyTapeout 5 shuttle.

It contains 8 copies of the structure in figure *(c)* with the 4 foundry cells and the 4 custom cells inserted as DUT. The ring oscillator, clock divider and switch are shared between the copies.

For simple tests, a copy of the cell is directly attached to the inputs and one of the outputs.

For advanced tests, a shift register is inserted in the input and output paths that can be driven much faster than the chip IO would allow.

When mode is 0, the switch relays the trigger signal and the output shift register performs regular rotations. This allows slow rotation from input to output through the DUT to check the pipeline as well as preloading inputs and reading outputs of the advanced tests.

When mode is 1, the switch gates the divided clock from the ring oscillator using the trigger signal, and the output shift register captures the DUT output into each of its bits according to the trigger running through a fast delay chain. So on a trigger signal the preloaded inputs are played at the pace of the divided clock and the DUT output is sampled into the output buffer at times indicated by the delay chain.

Verilog sources for the design are in the `src` directory, along with a cocotb testbench in `test.py`.

How to test

Note that the outputs are in pairs that should ideally behave in the same way during the tests below.

Test 1

- Adjust inputs 0, 1 & 2 manually and check the outputs.
- Outputs 0 & 1 (`mux2i`) should equal the negation of `A0` (input 0) if `S` (input 2) is low, and the negation of `A1` (input 1) if `S` is high.
- Outputs 2 & 3 (`maj3`) should be high if at least two of inputs 0, 1 & 2 is high.
- Outputs 4 & 5 (`dlrtp`) should behave as a latch. If `RESET_B` (input 2) is low, the output should be low as well, otherwise it should relay `D` (input 1) if `GATE` (input 0) is high and keep its output when `GATE` is low.

- Outputs 6 & 7 (`dfrtp`) should behave as a flop. If `RESET_B` (input 2) is low, it should reset into the low state. Otherwise it should save the `D` (input 1) state when `CLK` (input 0) is low and update the output it when `CLK` is high.

Test 2

- Make sure the `mode` bit (input 3) is low.
- Adjust inputs 0, 1 & 2, and keep toggling the `trigger` bit (input 4).
- On each positive edge of the trigger, a set of inputs is pushed into the pipeline and the corresponding outputs should emerge on the bidirectional pins 56 ticks later.

Test 3

- Set the `mode` bit (input 3) low.
- Preload a sequence of up to 12 inputs by adjusting pins 0, 1 & 2, then toggling the `trigger` bit (input 4) high and back low.
- Set the clock divider bits (inputs 5-7) as appropiate; zero should be fine for a first test.
- Set the `mode` bit (input 3) high.
- Toggle the `trigger` bit (input 4) high and back low.
- Set the `mode` bit (input 3) low.
- Read out the output sequence by toggling the `trigger` bit (input 4) up to 44 times.

Pinout

#	Input	Output	Bidirectional
0	A0/A/GATE/CLK	foundry mux2i direct	foundry mux2i instrumented
1	A1/B/D	custom mux2i direct	custom mux2i instrumented
2	S/C/RESET_B	foundry maj3 direct	foundry maj3 instrumented
3	mode bit	custom maj3 direct	custom maj3 instrumented
4	trigger bit	foundry dlrtp direct	foundry dlrtp instrumented
5	clock divider bit 0	custom dlrtp direct	custom dlrtp instrumented
6	clock divider bit 1	foundry dfrtp direct	foundry dfrtp instrumented
7	clock divider bit 2	custom dfrtp direct	custom dfrtp instrumented

Winner-Take-All Network (Verilog Demo) [160]

- Author: Nicholas Kuipers
- Description: Implement a WTA network
- GitHub repository
- HDL project
- Mux address: 160
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Take in an 8-bit current and parse two sets of 4 bits. Only output the 4 bits of highest value (if equal, preference to MSB)

How to test

After reset, result values and comparator are reset to 0

Pinout

#	Input	Output	Bidirectional
0	current bit 11	membrane potential bit a	spike bit 0
1	current bit 12	membrane potential bit b	unspecified
2	current bit 13	membrane potential bit c	unspecified
3	current bit 14	membrane potential bit d	unspecified
4	current bit 15	membrane potential bit e	unspecified
5	current bit 16	membrane potential bit f	unspecified
6	current bit 17	membrane potential bit g	unspecified
7	current bit 18	membrane potential bit h	unspecified

Lion cage [161]

- Author: Axel Andersson & Per Andersson
- Description: Count up to 15 lions, moving through a tunnel between a cage and an enclosure.
- GitHub repository
- HDL project
- Mux address: 161
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Two sensors are set in the tunnel. The sensor closest to the cage is G1, the other G2. Only one lion can walk through the tunnel at a time. A lion is allowed to reverse in the tunnel.

As the lion starts walking out, (G1, G2) = (1, 0), the counter increments by 1. One of two scenarios then occurs: either the sensors read (G1, G2) = (0, 0) before it reads (G1, G2) = (0, 1). In that case, the counter will decrement as the lion must have gone back into the cage. Otherwise, the sensors will read (G1, G2) = (0, 1) before it reads (G1, G2) = (0, 0), thereby letting us know that it has passed (G1, G2) = (1, 1) as well. The counter will not change in this case.

If we read (G1, G2) = (1, 1) before reading (G1, G2) = (1, 0), we know a lion is moving from the enclosure to the cage. We then repeat the above two cases. This allows us to create a two state graph of the problem, transitioning from S0 to S1 on either (G1, G2) = (1, 0) or (G1, G2) = (1, 1) and returning back on (G1, G2) = (0, 1) or (G1, G2) = (0, 0), incrementing the counter if S0->S1 on (G1, G2) = (1, 0) and decrementing the counter if S1->S0 on (G1, G2) = (0, 0).

How to test

After reset, the counter should increase by 1 if a lion moves from the cage to the enclosure, and the opposite if vice versa.

Pinout

#	Input	Output	Bidirectional
0	G1, first sensor in tunnel	segment a	They do nothing.
1	G2, second sensor in tunnnel	segment b	n/a
2	n/a	segment c	n/a
3	n/a	segment d	n/a
4	n/a	segment e	n/a
5	n/a	segment f	n/a
6	n/a	segment g	n/a
7	n/a	n/a	n/a

Brain Inspired Random Dropout Circuit [162]

- Author: Kevin Sandoval
- Description: This random dropout circuit simulates a dropout mechanism that is commonly used in neural networks for the ourpose of preventing overfitting.
- GitHub repository
- HDL project
- Mux address: 162
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

not there yet |

How to test

not there yet|

Pinout

#	Input	Output	Bidirectional
0	compare bit 11	segment a	second counter bit 0
1	compare bit 12	segment b	second counter bit 1
2	compare bit 13	segment c	second counter bit 2
3	compare bit 14	segment d	second counter bit 3
4	compare bit 15	segment e	second counter bit 4
5	compare bit 16	segment f	second counter bit 5
6	compare bit 17	segment g	second counter bit 6
7	compare bit 18	dot	second counter bit 7

Event-Based Denoising Circuit [163]

- Author: Sean Venadas
- Description: Takes an 8-bit signal with 4 parameters: x, y, p, t. When p is high, signal is outputted and filtered to reduce noise. Otherwise, output signal is zero.
- GitHub repository
- HDL project
- Mux address: 163
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Uses a set of registers to divide the clock, and then some combinational logic to convert from binary to decimal for the display.

Puts the bottom 8 bits of the counter on the bidirectional outputs.

With all the inputs set to 0, the internal 24 bit compare is set to 10,000,000. This means the counter will increment by one each second.

If any inputs are non zero, then the input will be used as an bits 11 to 18 of the 24 bit compare register. Example: setting the inputs to 00010000 will program 16384 into the compare register. With a 10MHz clock the counter will increment ~610 times per second.

How to test

After reset, the counter should increase by one every second with a 10MHz input clock. Experiment by changing the inputs to change the counting speed.

Pinout

#	Input	Output	Bidirectional
0	compare bit 11	segment a	second counter bit 0
1	compare bit 12	segment b	second counter bit 1
2	compare bit 13	segment c	second counter bit 2
3	compare bit 14	segment d	second counter bit 3
4	compare bit 15	segment e	second counter bit 4

#	Input	Output	Bidirectional
5	compare bit 16	segment f	second counter bit 5
6	compare bit 17	segment g	second counter bit 6
7	compare bit 18	dot	second counter bit 7

RAM cell test [164]

- Author: Rodolfo Sanchez
- Description: Simple test with of a memory cell
- GitHub repository
- Wokwi project
- Mux address: 164
- Extra docs
- Clock: 0 Hz
- External hardware: None

How it works

Simple test wiht

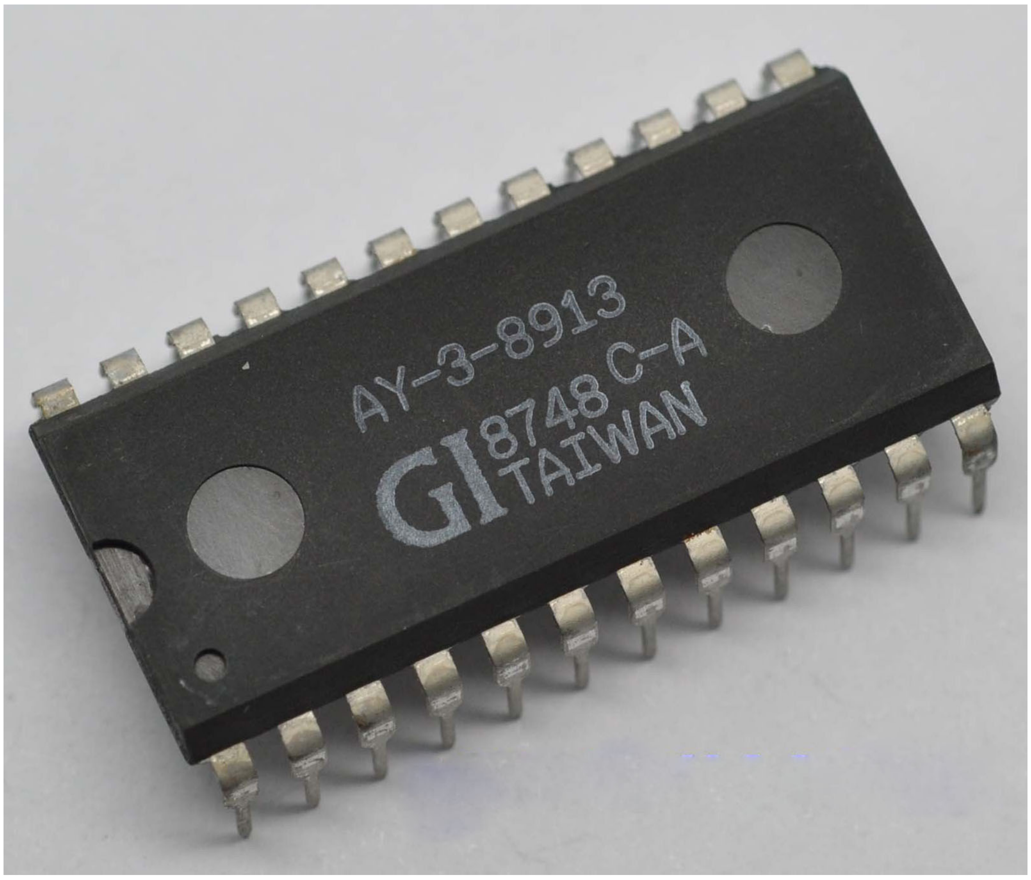
How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	IN0 to IN3 - input data	OUT0 - OUT3 - output data	not used
1	IN4, IN5 - address selection	n/a	n/a
2	IN6 - write	read	n/a
3	n/a	n/a	n/a
4	n/a	n/a	n/a
5	n/a	n/a	n/a
6	n/a	n/a	n/a
7	n/a	n/a	n/a

Classic 8-bit era Programmable Sound Generator AY-3-8913 [165]



- Author: ReJ aka Renaldas Zioma
- Description: The AY-3-8913 is a 3-voice programmable sound generator (PSG) chip from General Instruments. The AY-3-8913 is a smaller variant of AY-3-8910 or its analog YM2149.
- GitHub repository
- HDL project
- Mux address: 165
- Extra docs
- Clock: 2000000 Hz
- External hardware: DAC (for ex. Digilent R2R PMOD), RC filter, amplifier, speaker

How it works

This Verilog implementation is a replica of the classical **AY-3-8913** programmable sound generator. With roughly a 1500 logic gates this design fits on a **single tile** of the TinyTapeout.

The goals of this project

1. closely replicate the behavior and eventually the complete **design of the original** AY-3-891x with builtin DACs
2. provide a readable and well documented code for educational and hardware **preservation** purposes
3. leverage the **modern fabrication** process

A significant effort was put into a thorough **test suite** for regression testing and validation against the original chip behavior.

Chip technical capabilities

- **3 square wave** tone generators
- A single **white noise** generator
- A single **envelope** generator able to produce 10 different shapes
- Chip is capable to produce a range of waves from a **30 Hz** to **125 kHz**, defined by **12-bit** registers.
- **16** different volume levels

Registers The behavior of the AY-3-891x is defined by 14 registers.

Register	Bits used	Function	Description
0	`xxxxxxxx`	Channel A Tone	8-bit fine frequency
1	`....xxxx`	—//—	4-bit coarse frequency
2	`xxxxxxxx`	Channel B Tone	8-bit fine frequency
3	`....xxxx`	—//—	4-bit coarse frequency
4	`xxxxxxxx`	Channel C Tone	8-bit fine frequency
5	`....xxxx`	—//—	4-bit coarse frequency
6	`...xxxxx`	Noise	5-bit noise frequency
7	`..CBACBA`	Mixer	Tone and/or Noise per channel
8	`...xxxxx`	Channel A Volume	Envelope enable or 4-bit amplitude
9	`...xxxxx`	Channel B Volume	Envelope enable or 4-bit amplitude
10	`...xxxxx`	Channel C Volume	Envelope enable or 4-bit amplitude
11	`xxxxxxxx`	Envelope	8-bit fine frequency
12	`xxxxxxxx`	—//—	8-bit coarse frequency
13	`....xxxx`	Envelope Shape	4-bit shape control

Square wave tone generators Square waves are produced by counting down the 12-bit counters. Counter counts up from 0. Once the corresponsding register value is reached, counter is reset and the output bit of the channel is flipped producing square waves.

Noise generator Noise is produced with 17-bit Linear-feedback Shift Register (LFSR) that flips the output bit pseudo randomly. The shift rate of the LFSR register is controller by the 5-bit counter.

Envelope The envelope shape is controlled with 4-bit register, but can take only 10 distinct patterns. The speed of the envelope is controlled with 16-bit counter. Only a single envelope is produced that can be shared by any combination of the channels.

Volume Each of the three AY-3-891x channels have dedicated DAC that converts 16 levels of volume to analog output. Volume levels are 3 dB apart in AY-3-891x.

Historical use of the AY-3-891x

The AY-3-891x family of programmable sound generators was introduced by General Instrument in 1978. Soon Yamaha Corporation licensed and released a very similar chip under YM2149 name.

Both variants of the AY-3-891x and YM2149 were broadly used in home computers, game consoles and arcade machines in the early 80ies.

- home computers: Apple II Mockingboard sound card, Amstrad CPC, Atari ST, Oric-1, Sharp X1, MSX, ZX Spectrum 128/+2/+3
- game consoles: Intellivision, Vectrex, Amstrad GX4000
- arcade machines: Frogger, 1942, Spy Hunter and etc.

The AY-3-891x chip family competed with the similar Texas Instruments SN76489.

The original pinout of the AY-3-8913

The **AY-3-8913** was a 24-pin package release of the AY-3-8910 with a number of internal pins left simply unconnected. The goal of AY-3-8913 was to reduce complexity for the designer and reduce the foot print on the PCB. Otherwise the functionality of the chip is identical to AY-3-8910 and AY-3-8912.

```
              ,--._.--.
  GND  ---|1      24|<-- /cs*
 BDIR  -->|2      23|<--  a8*
  BC1  -->|3      22|<-- /a9*
  DA7  <->|4      21|<-- /RESET
  DA6  <->|5      20|<-- CLOCK
  DA5  <->|6      19|--- GND
  DA4  <->|7      18|--> CHANNEL C OUT
  DA3  <->|8      17|--> CHANNEL A OUT
  DA2  <->|9      16|    not connected
  DA1  <->|10     15|--> CHANNEL B OUT
  DA0  <->|11     14|<-- test*
```

```
    test* <--|12   13|<-- VCC
             `-------'
      * -- omitted from this Verilog implementation
```

Difference from the original hardware

This Verilog implementation is a completely digital and synchronous design that differs from the original AY-3-8913 design which incorporated internal DACs and analog outputs.

Audio signal output While the original chip had no summation The module provides two alternative outputs for the generated audio signal:

1. digital 8-bit audio output suitable for external Digital to Analog Converter (DAC)
2. pseudo analog output through Pulse Width Modulation (PWM)

Master output channel In contrast to the original chip which had only separate channel outputs, this implementation also provides an optional summation of the channels into a single master output.

No DC offset This implementation produces output 0/1 waveforms without DC offset.

No /A8, A9 and /CS pins The combination of **/A8**, **A9** and **/CS** pins orginially were intended to select a specific sound chip out the larger array of devices connected to the same bus. In this implementation this mechanism is omitted for simplicity, **/A8**, **A9** and **/CS** are considered to be tied **low** and chip behaves as always enabled.

Synchronous reset and single phase clock The original design employed 2 phases of the clock and asynchronous reset mechanism for operation of the registers.

To make it easier to synthesize and test on FPGAs this implementation uses single clock phase and synchronous reset for registers.

The reverse engineered AY-3-891x

This implementation would not be possible without the reverse engineered schematics and analysis based on decapped AY-3-8910 and AY-3-8914 chips.

How to test

The data bus of the AY-3-8913 chip has to be connected to microcontroller and receive a regular stream of commands. The AY-3-8913 produces audio output and has to be connected to a speaker. There are several ways how the overall schematics can be established.

8-bit parallel output via DAC One option is to connect off the shelf data parallel Digital to Analog Converter (DAC) for example Digilent R2R Pmod to the output pins and route the resulting analog audio to piezo speaker or amplifier.

```
uController             AY-3-8913
,---------.            ,---._.---.
|         |    2 Mhz ->|CLK  SEL0|<-- 0
|    GPIOx|----------->|BC1  SEL1|<-- 0
|    GPIOx|----------->|BDI      |         ,----------.
|    GPIOx|----------->|DA0  OUT0|-------->|LSB       |
|    GPIOx|----------->|DA1  OUT1|-------->|          |
|    GPIOx|----------->|DA2  OUT2|-------->|   pDAC   |  Headphones
|    GPIOx|----------->|DA3  OUT3|-------->|    or    |      or
|    GPIOx|----------->|DA4  OUT4|-------->| RESISTOR |    Buzzer
|    GPIOx|----------->|DA5  OUT5|-------->|  ladder  |          /|
|    GPIOx|----------->|DA6  OUT6|-------->|          |      .--/ |
|    GPIOx|----------->|DA7  OUT7|-------->|MSB       |-----|    |
`---------'            `---------'         `----------'     `--` |
                                                             |  `|
                                                             |
                                                        GND ---
```

AUDIO OUT through RC filter Another option is to use the Pulse Width Modulated (PWM) AUDIO OUT pin that combines 4 channels with the Resistor-Capacitor based low-pass filter or better the Operation Amplifier (Op-amp) & Capacitor based integrator:

```
uController             AY-3-8913
,---------.            ,---._.---.
|         |    2 Mhz ->|CLK  SEL0|<-- 0
|    GPIOx|----------->|BC1  SEL1|<-- 0
|    GPIOx|----------->|BDIR     |
|    GPIOx|----------->|DA0      |
|    GPIOx|----------->|DA1      |
|    GPIOx|----------->|DA2      |          C1
|    GPIOx|----------->|DA3      |     ,----||----.
|    GPIOx|----------->|DA4      |     |          |
|    GPIOx|----------->|DA5      |     |  Op-amp  |        Speaker
|    GPIOx|----------->|DA6 AUDIO|     |   |X     |            /|
|    GPIOx|----------->|DA7  OUT |-----+---|-X    |   C2   .--/ |
`---------'            `---------'         |  }---+---||---|    |
                                        ,--|+/             `--` |
                                        |  |/               |  `|
                                        |                   |
                                   GND ---             GND ---
```

Separate channels through the Op-amp The third option is to externally combine 4 channels with the Operational Amplifier and low-pass filter:

```
uController             AY-3-8913
,---------.            ,---._.---.
|         |    2 Mhz ->|CLK   SEL0|<-- 0
|    GPIOx|----------->|BC1   SEL1|<-- 0
|    GPIOx|----------->|BDIR      |
|    GPIOx|----------->|DA0       |
|    GPIOx|----------->|DA1       |
|    GPIOx|----------->|DA2       |           C1
|    GPIOx|----------->|DA3       |      ,----||----.
|    GPIOx|----------->|DA4       |      |          |
|    GPIOx|----------->|DA5      A|---.  |  Op-amp  |        Speaker
|    GPIOx|----------->|DA6      B|---+  |   |X     |            /|
|    GPIOx|----------->|DA7      C|---+--+---|-X    |   C2   .--/ |
`---------'            `---------'          |  }---+---||---|    |
                                         ,--|+/             `--` |
                                         |  |/               |  `|
                                         |                   |
                                    GND ---             GND ---
```

Summary of commands to communicate with the chip

The AY-3-8913 is programmed by updating its internal registers via the data bus. Below is a short summary of the communication protocol of AY-3-891x. Please consult AY-3-891x Technical Manual for more information.

BDIR	BC1	Bus state description
0	0	Bus is inactive
0	1	(Not implemented)
1	0	Write bus value to the previously latched register #
1	1	Latch bus value as the destination register #

Latch register address First, put the destination register adress on the bus of the chip and latch it by pulling both **BDIR** and **BC1** pins **high**.

Write data to register Put the desired value on the bus of the chip. Pull **BC1** pin **low** while keeping **BDIR** pin **high** to write the value of the bus to the latched register address.

Inactivate bus by pulling both **BDIR** and **BC1** pins **low**.

Register	Format	Description	Parameters
0,2,4	`ffffffff`	A/B/C tone period	`f` - low bits
1,3,5	`0000FFFF`	—//—	`F` - high bits
6	`000fffff`	Noise period	`f` - noise period
7	`00CBAcba`	Noise / tone per channel	`CBA` - noise off, `cba` - tone off
8,9,10	`000Evvvv`	A/B/C volume	`E` - envelope on, `v` - volume level
11	`ffffffff`	Envelope period	`f` - low bits
12	`FFFFFFFF`	—//—	`F` - high bits
13	`0000caAh`	Envelope Shape	`c` - continue, `a` - attack, `A` - alternate, `h` - hold

Note frequency

Use the following formula to calculate the 12-bit period value for a particular note:

$$toneperiod_{cycles} = clock_{frequency}/(16_{cycles} * note_{frequency})$$

For example 12-bit period that plays 440 Hz note on a chip clocked at 2 MHz would be:

$$toneperiod_{cycles} = 2000000Hz/(16_{cycles} * 440Hz) = 284 = 11C_{hex}$$

An example to play a note at a maximum volume

BDIR	BC1	DA7..DA0	Explanation
1	1	xxxx0000	**Latch** tone A coarse register address $0 = 0000_{bin}$
1	0	xxxx0001	**Write** high 4-bits of the 440 Hz note $1 = 0001_{bin}$
1	1	xxxx0001	**Latch** tone A fine register address $1_{dec} = 0001_{bin}$
1	0	00011100	**Write** low 8-bits of the note $1C_{hex} = 00011100_{bin}$
1	1	xxxx1000	**Latch** channel A volume register address $8 = 1000_{bin}$
1	0	xxx01111	**Write** maximum volume level $15_{dec} = 1111_{bin}$ with the envelope disabled

```
Timing diagram

CLK   ____      ____      ____      ____      ____      ____
   __/    `____/    `____/    `____/    `____/    `____/    `____  . . .
    |         |         |         |         |         |
    |         |         |         |         |         |


BDIR ______    ______    ______    ______    ______    ______
   _/      `__/      `__/      `__/      `__/      `__/      `__

BC1  _______             _______            ________
   _/       `___________/       `__________/        `___________

DA7..DA0_____  ________  ________  ________  ________  ________
   _/  0000  `/xxxx0001`/  0001  `/00011100`/  1000  `/xxx01111`
      latch     write     latch      write    latch
```

Externally configurable clock divider

SEL1	SEL0	Description	Clock frequency
0	0	Standard mode, clock divided by 8	1.7 .. 2.0 MHz
1	1	—//—	1.7 .. 2.0 MHz
0	1	New mode for TT05, no clock divider	250 .. 500 kHZ
1	0	New mode for TT05, clock div. 128	25 .. 50 MHz

SEL1	SEL0	Formula to calculate the 12-bit tone period value for a note
0	0	$clock_{frequency}/(16_{cycles} * note_{frequency})$
1	1	—//—
0	1	$clock_{frequency}/(2_{cycles} * note_{frequency})$

SEL1	SEL0	Formula to calculate the 12-bit tone period value for a note
1	0	$clock_{frequency}/(128_{cycles} * note_{frequency})$

Pinout

#	Input	Output	Bidirectional
0	DA0 - multiplexed data/address bus	audio out (pwm)	(in) **BC1** bus control
1	DA1 - multiplexed data/address bus	digita audio least significant bit	(in) **BDIR** bus direction
2	DA2 - multiplexed data/address bus	digita audio	(in) **SEL0** clock divider
3	DA3 - multiplexed data/address bus	digita audio	(in) **SEL1** clock divider
4	DA4 - multiplexed data/address bus	digita audio	(out) channel A (PWM)
5	DA5 - multiplexed data/address bus	digita audio	(out) channel B (PWM)
6	DA6 - multiplexed data/address bus	digita audio	(out) channel C (PWM)
7	DA7 - multiplexed data/address bus	digita audio most significant bit	(out) AUDIO OUT master (PWM)

RNN (Demo) [166]

- Author: Ridger Zhu
- Description: Count up to 10, one second at a time.
- GitHub repository
- HDL project
- Mux address: 166
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Apply an input current to an RNN hidden state, where the hidden state will multiply with a 8x8 matrix.

How to test

After reset, the hidden state will be set to 0.

Then change the inputs to change the current. You can read the output current to know the result.

Pinout

#	Input	Output	Bidirectional
0	current bit 11	out bit a	unspecified
1	current bit 12	out bit b	unspecified
2	current bit 13	out bit c	unspecified
3	current bit 14	out bit d	unspecified
4	current bit 15	out bit e	unspecified
5	current bit 16	out bit f	unspecified
6	current bit 17	out bit g	unspecified
7	current bit 18	out bit h	unspecified

STDP Neuron [167]

- Author: William Bodeau
- Description: A single LIF neuron with post-synaptic STDP learning.
- GitHub repository
- HDL project
- Mux address: 167
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Oh god I don't wanna

How to test

Oh god I don't wanna

Pinout

#	Input	Output	Bidirectional
0	compare bit 11	segment a	second counter bit 0
1	compare bit 12	segment b	second counter bit 1
2	compare bit 13	segment c	second counter bit 2
3	compare bit 14	segment d	second counter bit 3
4	compare bit 15	segment e	second counter bit 4
5	compare bit 16	segment f	second counter bit 5
6	compare bit 17	segment g	second counter bit 6
7	compare bit 18	dot	second counter bit 7

Basic Spiking Neural Network [168]

- Author: Abhinandan singh
- Description: Study of spike generation in a SNN.
- GitHub repository
- HDL project
- Mux address: 168
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Spiking pattern of the three input neurons will be used as an input. The Spikes of the two output neurons can be plotted on time charts. Example:

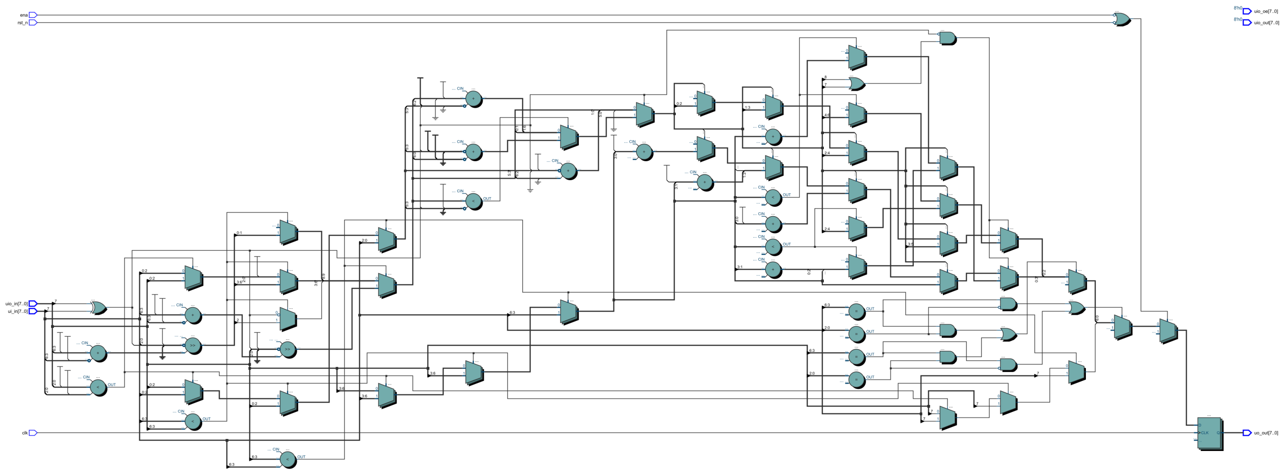
How to test

Feed in diferent spiking patterns of the input neurons (1, 2 and 3). The network will pass the spikes in the forward direction accoding to the defined weights.

Pinout

#	Input	Output	Bidirectional
0	spike bit 11	spike bit a	unspecified
1	spike bit 12	spike bit b	unspecified
2	spike bit 13	unspecified	unspecified
3	unspecified	unspecified	unspecified
4	unspecified	unspecified	unspecified
5	unspecified	unspecified	unspecified
6	unspecified	unspecified	unspecified
7	unspecified	unspecified	unspecified

8 bit floating point adder [169]



- Author: Philip Mohr
- Description: Adds two 8 Bit floating point numbers
- GitHub repository
- HDL project
- Mux address: 169
- Extra docs
- Clock: None Hz
- External hardware:

How it works

Adds two 8 bit floating point numbers under consideration of rounding and infinity cases. The two floats use the 8 bit input and the 8 bit bidirectional input. 1 bit sign, 4 bit exponent, 3 bit mantissa.

How to test

Every clock the output should give the addition of the two floats.

Pinout

#	Input	Output	Bidirectional
0	Float a Mant[0]	Float out Mant[0]	Float b Mant[0]
1	Float a Mant1	Float out Mant1	Float b Mant1

#	Input	Output	Bidirectional
2	Float a Mant2	Float out Mant2	Float b Mant2
3	Float a Exp[0]	Float out Exp[0]	Float b Exp[0]
4	Float a Exp1	Float out Exp1	Float b Exp1
5	Float a Exp2	Float out Exp2	Float b Exp2
6	Float a Exp[3]	Float out Exp[3]	Float b Exp[3]
7	Float a Sign	Float out Sign	Float b Sign

Perceptron Hardcoded [170]

- Author: Sathyaprakash Narayanan
- Description: Hardcoded Perceptron
- GitHub repository
- HDL project
- Mux address: 170
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Uses a set of registers to divide the clock, and then some combinational logic to convert from binary to decimal for the display.

Puts the bottom 8 bits of the counter on the bidirectional outputs.

With all the inputs set to 0, the internal 24 bit compare is set to 10,000,000. This means the counter will increment by one each second.

If any inputs are non zero, then the input will be used as an bits 11 to 18 of the 24 bit compare register. Example: setting the inputs to 00010000 will program 16384 into the compare register. With a 10MHz clock the counter will increment ~610 times per second.

How to test

After reset, the counter should increase by one every second with a 10MHz input clock. Experiment by changing the inputs to change the counting speed.

Pinout

#	Input	Output	Bidirectional
0	unspecified	perceptron_output a	unspecified
1	unspecified	perceptron_output b	unspecified
2	unspecified	perceptron_output c	unspecified
3	unspecified	perceptron_output d	unspecified
4	unspecified	perceptron_output e	unspecified
5	unspecified	perceptron_output f	unspecified

#	Input	Output	Bidirectional
6	unspecified	perceptron_output g	unspecified
7	unspecified	perceptron_output h	unspecified

Cheap and quick STDP [171]

- Author: J. Przepioski
- Description: Due to schedule: Implement most basic functional STDP
- GitHub repository
- HDL project
- Mux address: 171
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Apply an input current injection to the LIF neuron using switches.

This gets added to a membrane potential which is decayed over time. If the membrane potential exceeds the threshold then trigger a spike.

How to test

After reset, the membrane potential will be set to 0.

Then change the inputs to change the current. A higher current should trigger a higher firing rate.

Pinout

#	Input	Output	Bidirectional
0	current bit 11	membrane potential bit a	spike bit 0
1	current bit 12	membrane potential bit b	unspecified
2	current bit 13	membrane potential bit c	unspecified
3	current bit 14	membrane potential bit d	unspecified
4	current bit 15	membrane potential bit e	unspecified
5	current bit 16	membrane potential bit f	unspecified
6	current bit 17	membrane potential bit g	unspecified
7	current bit 18	membrane potential bit h	unspecified

Brain-Inspired Oscillatory Network [172]

- Author: Derek Abarca
- Description: Two neuron modules interact with a synapse module to produce rhythmic oscillations.
- GitHub repository
- HDL project
- Mux address: 172
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Fill in later

How to test

Fill in later

Pinout

#	Input	Output	Bidirectional
0	compare bit 11	segment a	second counter bit 0
1	compare bit 12	segment b	second counter bit 1
2	compare bit 13	segment c	second counter bit 2
3	compare bit 14	segment d	second counter bit 3
4	compare bit 15	segment e	second counter bit 4
5	compare bit 16	segment f	second counter bit 5
6	compare bit 17	segment g	second counter bit 6
7	compare bit 18	dot	second counter bit 7

UART uwuifier [173]

- Author: Anish Singhani
- Description: Operates on a UART signal and uwuifies it
- GitHub repository
- HDL project
- Mux address: 173
- Extra docs
- Clock: 6000000 Hz
- External hardware: UART transceiver

How it works

UART interface 115200 baud at 6MHz

How to test

Connect inline with a UART

Pinout

#	Input	Output	Bidirectional
0	none	none	none
1	none	none	none
2	none	none	none
3	uart rx	none	none
4	none	uart tx	none
5	none	none	none
6	none	none	none
7	none	none	none

Perceptron and basic binary neural network [174]

- Author: Connor Guzikowski
- Description: Taking in the number of curves and edges of a number, the output is the expected number.
- GitHub repository
- HDL project
- Mux address: 174
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Takes in a 7 bit input, with the 3 leftmost bits being the number of edges of a number, and the other bits are the numbers of curves in the number. The output has 8 bits, separated into two halves the first half is the output of the perceptron, and the second half is the output of the binary neural network.

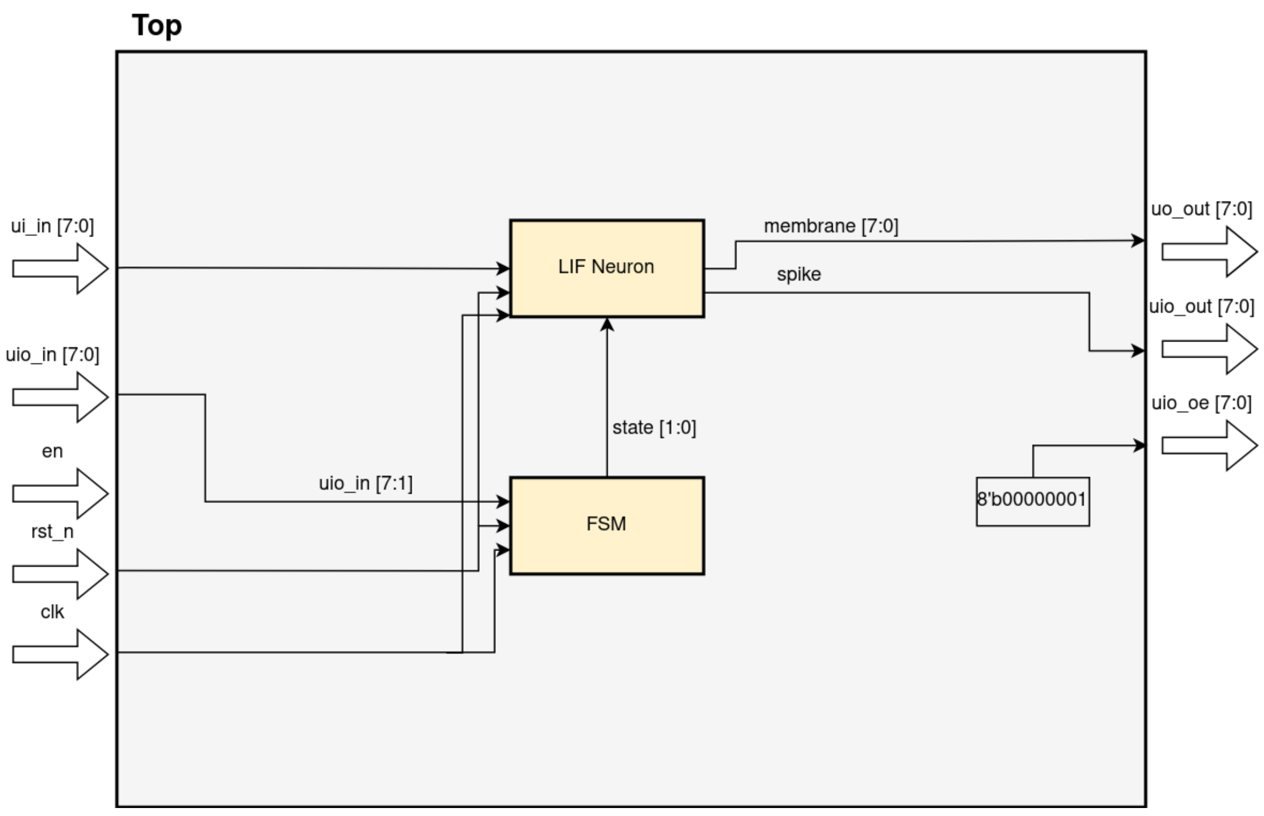
How to test

After reset, the counter should increase by one every second with a 10MHz input clock. Experiment by changing the inputs to change the counting speed.

Pinout

#	Input	Output	Bidirectional
0	compare bit 11	segment a	second counter bit 0
1	compare bit 12	segment b	second counter bit 1
2	compare bit 13	segment c	second counter bit 2
3	compare bit 14	segment d	second counter bit 3
4	compare bit 15	segment e	second counter bit 4
5	compare bit 16	segment f	second counter bit 5
6	compare bit 17	segment g	second counter bit 6
7	compare bit 18	dot	second counter bit 7

Leaky Integrate-and-Fire Neuron [175]



- Author: Muhammad Hadir Khan
- Description: A Leaky Integrate-and-Fire Neuron that mimics the biological neuron and is configurable from the outside world
- GitHub repository
- HDL project
- Mux address: 175
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

The state machine first configures the different parameters of the neuron which is then provided a synaptic current to read out the membrane potential and spiking of the neuron.

The configurable parameters of the neuron are: 1) beta (which controls the decay of the membrane potential) 2) threshold (which is used in comparison with the membrane potential to generate a spike)

Initially, the neuron is in an IDLE state where everything remains 0. Upon configuring the setting bits with `uio_in[7:1]` the state goes to BETA where the `beta` value of the neuron is configured using the `ui_in[7:0]` bits. After which the setting bits is again configured to make the state go to THRESH state which configures the threshold value of the neuron. Finally, the setting bits are used to go to the READ state and the current injection is provided with `ui_in` where the neuron starts to integrate the current onto the membrane potential.

At each timestep, the membrane potential is analyzed with `uo_out[7:0]` and the spike is outputted through `uio_out[0]`.

How to test

After reset, the neuron is in the idle state and remains there unless the setting bits are configured. Setting the `uio_in[7:1]` bits to 1 changes the state to BETA. After configuring the beta value changing the setting bits to 2 takes to the THRESH state where the threshold of the neuron is configured. Then changing the setting bits to 3 takes to the READ state where the current is integrated and the membrane potential changes as well as the spike is outputted if the membrane potential becomes equal or greater than the configured threshold.

Experiment by changing the beta, threshold and input current values to see how the neuron reacts.

Pinout

#	Input	Output	Bidirectional
0	configure beta, threshold and current	membrane potential	spike output
1	configure beta, threshold and current	membrane potential	configure the state
2	configure beta, threshold and current	membrane potential	configure the state
3	configure beta, threshold and current	membrane potential	configure the state
4	configure beta, threshold and current	membrane potential	configure the state
5	configure beta, threshold and current	membrane potential	configure the state
6	configure beta, threshold and current	membrane potential	configure the state

#	Input	Output	Bidirectional
7	configure beta, threshold and current	membrane potential	configure the state

7 segment seconds [192]

- Author: Matt Venn
- Description: counts up to 9 and wraps to 0. One step per clock cycle
- GitHub repository
- Wokwi project
- Mux address: 192
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Uses SR flops and avoid combinational logic in the clock path which caused clock glitches on the previous version.

How to test

Press the reset button, then press the clock button to advance the count.

Pinout

#	Input	Output	Bidirectional
0	none	segment a	none
1	none	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

UABC-ELECTRONICA [194]

- Author: UABC
- Description: Displays the word UABC-ELECTRONICA on a 7-segment display. Each letter is displayed in a time, one by one.
- GitHub repository
- HDL project
- Mux address: 194
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Uses a set of registers to divide the clock, and then some combinational logic to convert from binary to decimal for the display.

Puts the bottom 8 bits of the counter on the bidirectional outputs.

With all the inputs set to 0, the internal 24 bit compare is set to 10,000,000. This means the counter will increment by one each second.

If any inputs are non zero, then the input will be used as an bits 11 to 18 of the 24 bit compare register. Example: setting the inputs to 00010000 will program 16384 into the compare register. With a 10MHz clock the counter will increment ~610 times per second.

How to test

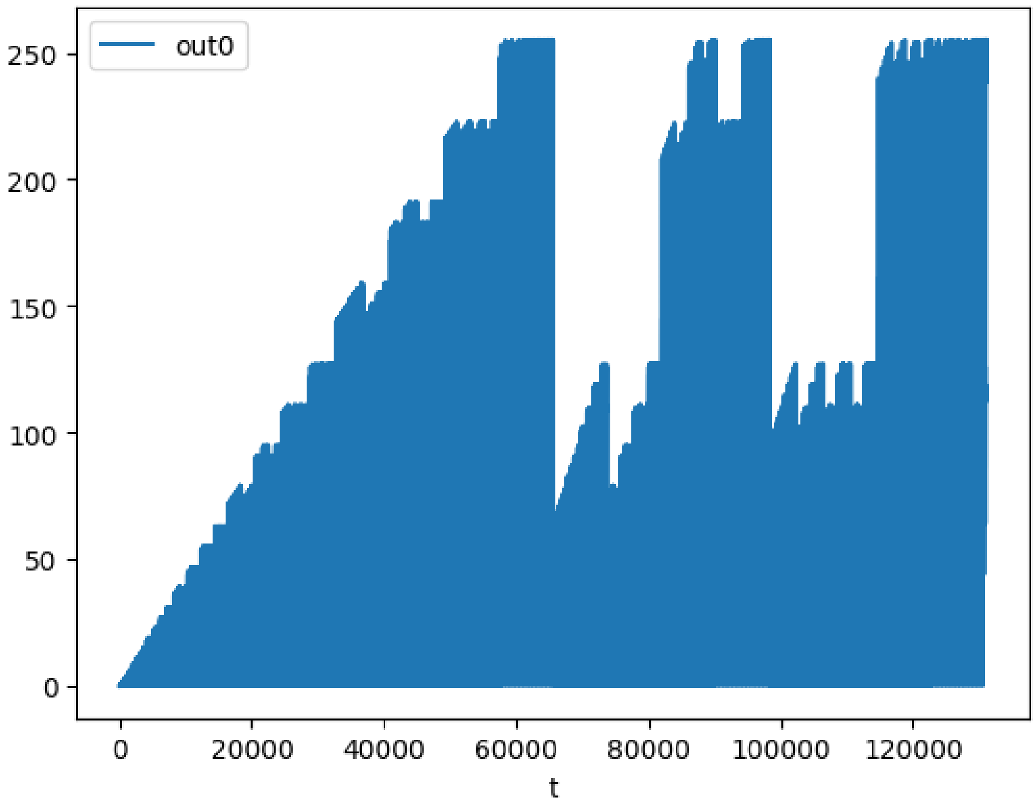
After reset, the counter should increase by one every second with a 10MHz input clock. Experiment by changing the inputs to change the counting speed.

Pinout

#	Input	Output	Bidirectional
0	compare bit 11	segment a	second counter bit 0
1	compare bit 12	segment b	second counter bit 1
2	compare bit 13	segment c	second counter bit 2
3	compare bit 14	segment d	second counter bit 3
4	compare bit 15	segment e	second counter bit 4

#	Input	Output	Bidirectional
5	compare bit 16	segment f	second counter bit 5
6	compare bit 17	segment g	second counter bit 6
7	compare bit 18	dot	second counter bit 7

bytebeat [196]



- Author: proppy
- Description: Attempt implement the formula from one of the original bytebeat video in hardware.
- GitHub repository
- HDL project
- Mux address: 196
- Extra docs
- Clock: 8000 Hz
- External hardware: 8bit pcm DAC, rotary encoder

How it works

The main module accept parameters from 4x 4-bit parameters buses and generate PCM samples according to the following formula: `((t*a)&amp;(t&gt;&gt;b))|((t*c)&amp;(t&g`
Derivative of this project can easily be created by editing the formula in `src/bytebeat.x` and using the XLS: Accelerated HW Synthesis toolkit to regenerate the Verilog code. See the following notebook for more information.

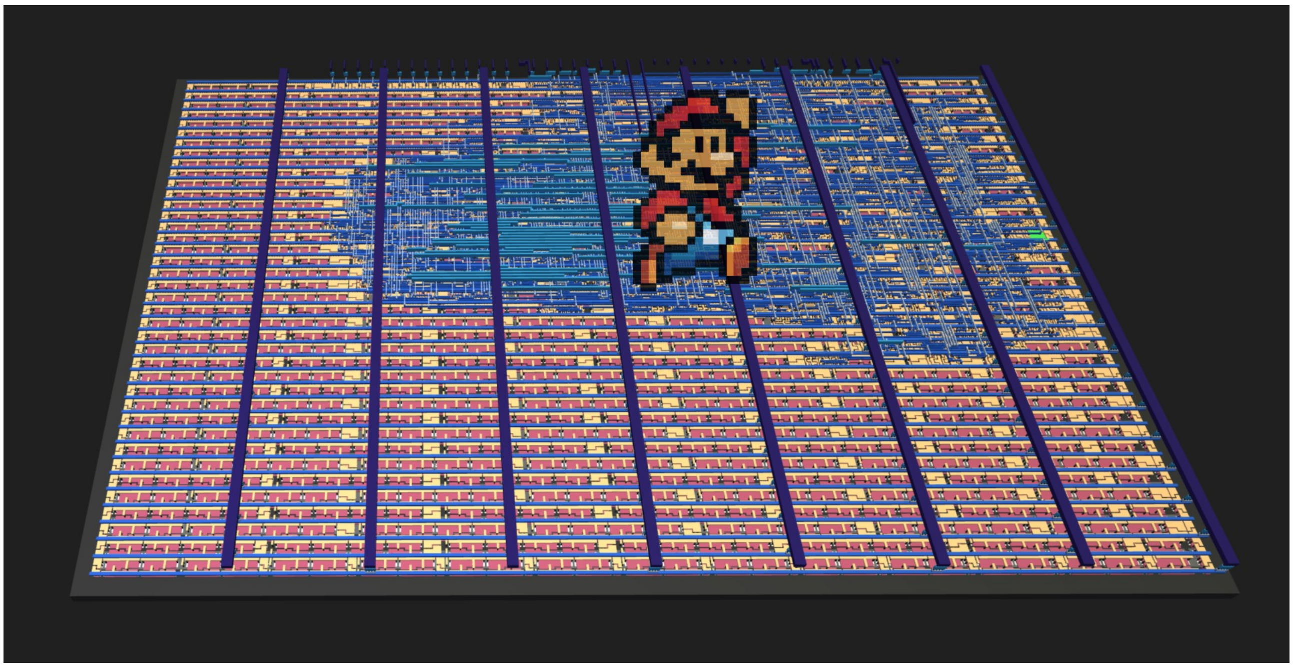
How to test

- Tweak parameters pins using a absolute encoders
- Feed the data coming from the sample bus to a DAC

Pinout

#	Input	Output	Bidirectional
0	param a bit 0/3	pcm sample bit 0/7	param c bit 0/3
1	param a bit 1/3	pcm sample bit 1/7	param c bit 1/3
2	param a bit 2/3	pcm sample bit 2/7	param c bit 2/3
3	param a bit 3/3	pcm sample bit 3/7	param c bit 3/3
4	param b bit 0/3	pcm sample bit 4/7	param d bit 0/3
5	param b bit 1/3	pcm sample bit 5/7	param d bit 1/3
6	param b bit 2/3	pcm sample bit 6/7	param d bit 2/3
7	param b bit 3/3	pcm sample bit 7/7	param d bit 3/3

Super Mario Tune on A Piezo Speaker [197]



- Author: Milosch Meriac
- Description: Plays Super Mario Tune over a Piezo Speaker connected across uio_out[1:0]
- GitHub repository
- HDL project
- Mux address: 197
- Extra docs
- Clock: 100000 Hz
- External hardware: Piezo speaker connected across io_out[1:0]

How it works

Converts an RTTL ringtone into verilog using Python - and plays it back using differential PWM modulation

How to test

Provide 100kHz clock on "clk" pin, briefly hit reset low ("rst_n") and uio_out[1:0] will play a differential sound wave over a connected piezo speaker (Super Mario)

Pinout

#	Input	Output	Bidirectional
0	ui_in[0]	ui_in[0]	piezo_speaker_p (uio_out[0])
1	ui_in1	ui_in1	piezo_speaker_n (uio_out1)
2	ui_in2	ui_in2	GND
3	ui_in[3]	ui_in[3]	GND
4	ui_in[4]	ui_in[4]	GND
5	ui_in[5]	ui_in[5]	GND
6	ui_in[6]	ui_in[6]	GND
7	ui_in[7]	ui_in[7]	GND

Byte Computer [198]

- Author: Rutuparn Pawar
- Description: An 8 bit turing complete computer
- GitHub repository
- HDL project
- Mux address: 198
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Byte computer is an in-order, no register file, no-cache, non-pipelined and no branch predictor implementation of an 8-bit Turing complete computer thus making it extremely simple and small enough for TinyTapeout. The computer fetches instruction at the address in the program counter which has an initial value of zero. The instruction is decoded and then executed followed by setting appropriate condition flags. The program counter is incremented and the fetch -> decode -> execute process repeats until a halt instruction is fetched and executed. See README in project repository for waveform illustrating the fetch -> decode -> execute process.

How to test

External memory and memory control logic is required to test the design which can be implemented using a microcontroller. The memory should be preloaded with a program created using the available instructions and the data processed by the program. The expected memory behaviour is to write data to memory at address indicated by the addr signal when we signal is high and vice versa. The halt signal indicates that the computer has encountered and ececuted a halt instruction.

Pinout

#	Input	Output	Bidirectional
0	data[7]	data[7]	we
1	data[6]	data[6]	halt
2	data[5]	data[5]	none
3	data[4]	data[4]	addr[4]
4	data[3]	data[3]	addr[3]

#	Input	Output	Bidirectional
5	data2	data2	addr2
6	data1	data1	addr1
7	data[0]	data[0]	addr[0]

7 segment seconds (VHDL Demo) [199]

- Author: Matt Venn
- Description: Count up to 10, one second at a time.
- GitHub repository
- HDL project
- Mux address: 199
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Uses a set of registers to divide the clock, and then some combinational logic to convert from binary to decimal for the display.

Puts the bottom 8 bits of the counter on the bidirectional outputs.

With all the inputs set to 0, the internal 24 bit compare is set to 10,000,000. This means the counter will increment by one each second.

If any inputs are non zero, then the input will be used as an bits 11 to 18 of the 24 bit compare register. Example: setting the inputs to 00010000 will program 16384 into the compare register. With a 10MHz clock the counter will increment ~610 times per second.

How to test

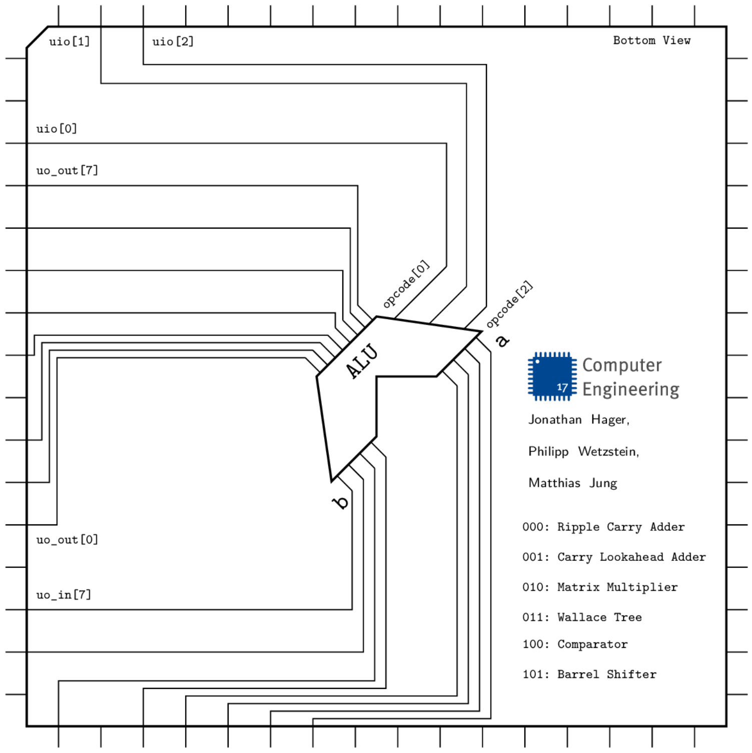
After reset, the counter should increase by one every second with a 10MHz input clock. Experiment by changing the inputs to change the counting speed.

Pinout

#	Input	Output	Bidirectional
0	compare bit 11	segment a	second counter bit 0
1	compare bit 12	segment b	second counter bit 1
2	compare bit 13	segment c	second counter bit 2
3	compare bit 14	segment d	second counter bit 3
4	compare bit 15	segment e	second counter bit 4
5	compare bit 16	segment f	second counter bit 5

#	Input	Output	Bidirectional
6	compare bit 17	segment g	second counter bit 6
7	compare bit 18	dot	second counter bit 7

4-Bit ALU [200]



- Author: CE JMU Wuerzburg
- Description: A simple 4-Bit ALU which contains two types of adders, multipliers, a comparator and a barrel-shifter
- GitHub repository
- HDL project
- Mux address: 200
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

The ALU contains a ripple carry adder, a carry lookahead adder, a matrix multiplier, a wallace-tree multiplier, a comparator and a barrel-shifter. Everything is implemented fully combinational. A 3-bit opcode is used to select the respective component.

How to test

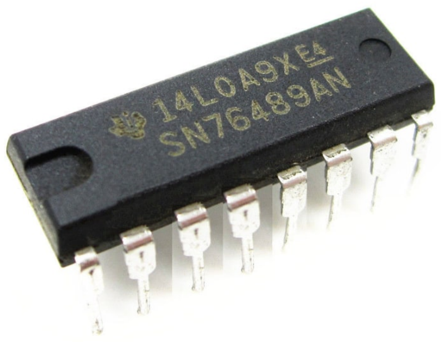
No clock is required. The first 4 input bits a[3...0] form the first operand, the last 4 input bits b[3...0] form the second operand. The outputs s[7...0] are used for the compuational results, the results for shifting a, or the results of comparing a with b.

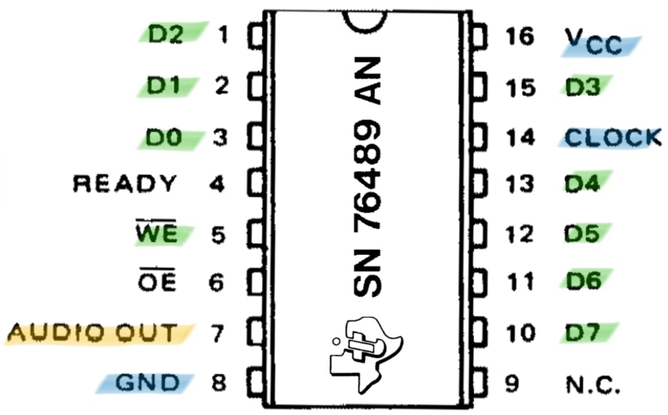
The bidirectional input bits 0, 1 and 2 are used as opcode to select the component, c.f. Figure above. If the barrel-shifter is used, a[3…0] will be shifted, b[1…0] is used to specify the shift width, whereas b2 selects the shift direction.

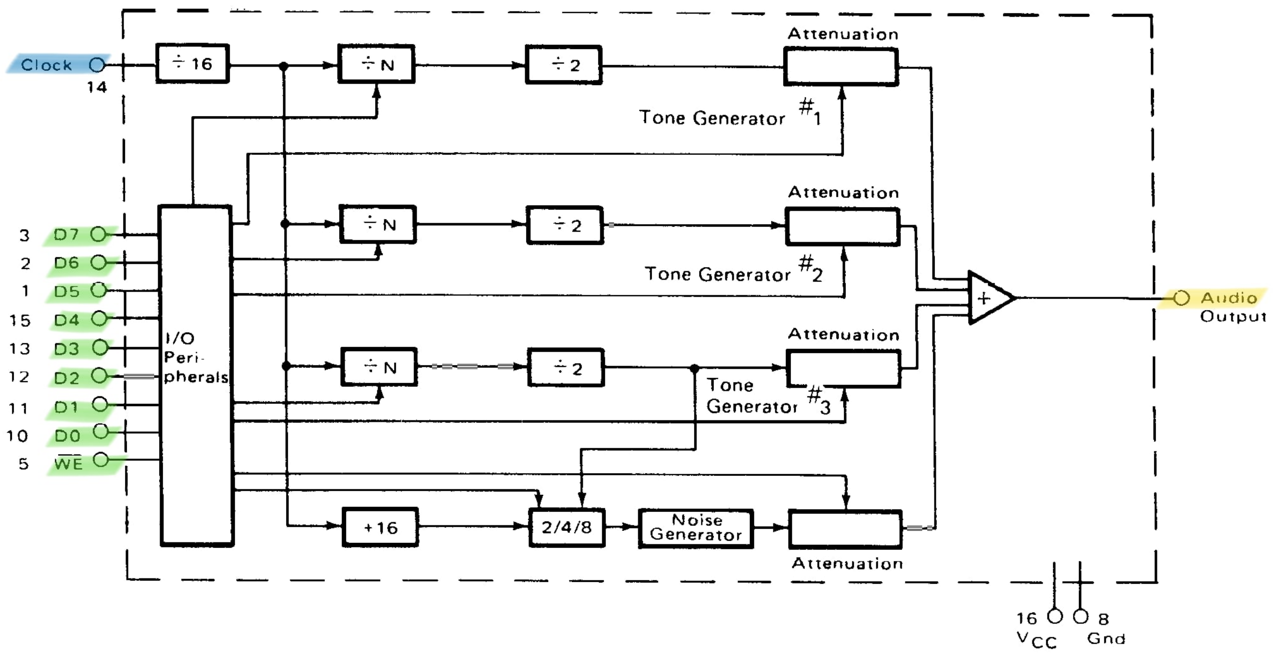
Pinout

#	Input	Output	Bidirectional
0	a[0]	s[0] (1, if $a > b$)	opcode[0]
1	a1	s1 (1, if $a < b$)	opcode1
2	a2	s2 (1, if $a == b$)	opcode2
3	a[3]	s[3]	none
4	b[0]	s[4]	none
5	b1	s[5]	none
6	b2 (0 = shift right, 1 = shift left)	s[6]	none
7	b[3]	s[7]	none

Classic 8-bit era Programmable Sound Generator SN76489 [201]







- Author: ReJ aka Renaldas Zioma
- Description: The SN76489 Digital Complex Sound Generator (DCSG) is a programmable sound generator chip from Texas Instruments.
- GitHub repository
- HDL project
- Mux address: 201
- Extra docs
- Clock: 4000000 Hz

- External hardware: DAC (for ex. Digilent R2R PMOD), RC filter, amplifier, speaker

How it works

This Verilog implementation is a replica of the classical **SN76489** programmable sound generator. With roughly a 1400 logic gates this design fits on a **single tile** of the TinyTapeout.

The goals of this project

1. closely replicate the behavior and eventually the complete **design of the original** SN76489
2. provide a readable and well documented code for educational and hardware **preservation** purposes
3. leverage the **modern fabrication** process

A significant effort was put into a thorough **test suite** for regression testing and validation against the original chip behavior.

The future work

The next step is to incorporate analog elements into the design to match the original SN76489 - DAC for each channel and an analog OpAmp for channel summation.

Chip technical capabilities

- **3 square wave** tone generators
- **1 noise** generator
- 2 types of noise: *white* and *periodic*
- Capable to produce a range of waves typically from **122 Hz** to **125 kHz**, defined by **10-bit** registers.
- **16** different volume levels

Registers The behavior of the SN76489 is defined by 8 "registers" - 4 x 4 bit volume registers, 3 x 10 bit tone registers and 1 x 3 bit noise configuration register.

Channel	Volume registers	Tone & noise registers
0	Channel #0 attenuation	Tone #0 frequency
1	Channel #1 attenuation	Tone #1 frequency
2	Channel #2 attenuation	Tone #2 frequency
3	Channel #3 attenuation	Noise type and frequency

Square wave tone generators Square waves are produced by counting down the 10-bit counters. Each time the counter reaches the 0 it is reloaded with the corresponding value from the configuration register and the output bit of the channel is flipped producing square waves.

Noise generator Noise is produced with 15-bit Linear-feedback Shift Register (LFSR) that flips the output bit pseudo randomly. The shift rate of the LFSR register is controller either by one of the 3 hardcoded power-of-two dividers or output from the channel #2 tone generator is used.

Attenuation Each of the four SN76489 channels have dedicated attenuation modules. The SN76489 has 16 steps of attenuation, each step is 2 dB and maximum possible attenuation is 28 dB. Note that the attenuation definition is the opposite of volume / loudness. Attenuation of 0 means maximum volume.

Finally, all the 4 attenuated signals are summed up and are sent to the output pin of the chip.

Historical use of the SN76489

The SN76489 family of programmable sound generators was introduced by Texas Instruments in 1980. Variants of the SN76489 were used in a number of home computers, game consoles and arcade boards:

- home computers: TI-99/4, BBC Micro, IBM PCjr, Sega SC-3000, Tandy 1000
- game consoles: ColecoVision, Sega SG-1000, Sega Master System, Game Gear, Neo Geo Pocket and Sega Genesis
- arcade machines by Sega & Konami and would usually include 2 or 4 SN76489 chips

The SN76489 chip family competed with the similar General Instrument AY-3-8910.

The original pinout of the SN76489AN

```
             ,--._.--.
      D5   -->|1    16|<-- VCC
      D6   -->|2    15|<-- D4
      D7   -->|3    14|<-- CLOCK
   ready* <--|4    13|<-- D3
     /WE   -->|5    12|<-- D2
     /ce* -->|6    11|<-- D1
AUDIO OUT <--|7    10|<-- D0
      GND ---|8     9|    not connected*
             `--------'
      * -- omitted from this Verilog implementation
```

Difference from the original hardware

This Verilog implementation is a completely digital and synchronous design that differs from the original SN76489 design which incorporated analog parts.

Audio signal output While the original chip had integrated OpAmp to sum generated channels in analog fashion, this implementation does digital signal summation and digital output. The module provides two alternative outputs for the generated audio signal:

1. digital 8-bit audio output suitable for external Digital to Analog Converter (DAC)
2. pseudo analog output through Pulse Width Modulation (PWM)

Separate 4 channel output Outputs of all 4 channels are exposed along with the master output. This allows to validate and mix signals externally. In contrast the original chip was limited to a single audio output pin due to the PDIP-16 package.

No DC offset This implementation produces output 0/1 waveforms without DC offset.

No /CE and READY pins Chip enable control pin **/CE** is omitted in this design for simplicity. The behavior is the same as if **/CE** is tied *low* and the chip is considered always enabled.

Unlike the original SN76489 which took 32 cycles to update registers, this implementation handles register writes in a single cycle and chip behaves as always **READY**.

Synchronous reset and single phase clock The original design employed 2 phases of the clock for the operation of the registers. The original chip had no reset pin and would wake up to a random state.

To make it easier to synthesize and test on FPGAs this implementation uses single clock phase and synchronous reset for registers.

A configurable clock divider was introduced in this implementation.

1. the original SN76489 with the master clock internally divided by 16. This classical chip was intended for PAL and NTSC frequencies. However in BBC Micro 4 MHz clock was employed.
2. SN94624/SN76494 variants without internal clock divider. These chips were intended for use with 250 to 500 KHz clocks.
3. high frequency clock configuration for TinyTapeout, suitable for a range between 25 MHz and 50 Mhz. In this configuration the master clock is internally divided by 128.

The reverse engineered SN76489

This implementation is based on the results from these reverse engineering efforts:

1. Annotations and analysis of a decapped SN76489A chip.
2. Reverse engineered schematics based on a decapped VDP chip from Sega Mega Drive which included a SN76496 variant.

How to test

The data bus of the SN76489 chip has to be connected to microcontroller and receive a regular stream of commands. The SN76489 produces audio output and has to be connected to a speaker. There are several ways how the overall schematics can be established.

8-bit parallel output via DAC One option is to connect off the shelf data parallel Digital to Analog Converter (DAC) for example Digilent R2R Pmod to the output pins and route the resulting analog audio to piezo speaker or amplifier.

```
uController              SN76489
,---------.            ,---._.---.
|         |     4 Mhz ->|CLK   SEL0|<-- 0
|    GPIOx|------------>|D0    SEL1|<-- 0
|    GPIOx|------------>|D1        |         ,----------.
|    GPIOx|------------>|D2    OUT0|-------->|LSB       |
|    GPIOx|------------>|D3    OUT1|-------->|          |
|    GPIOx|------------>|D4    OUT2|-------->|   pDAC   |  Headphones
|    GPIOx|------------>|D5    OUT3|-------->|    or    |      or
|    GPIOx|------------>|D6    OUT4|-------->| RESISTOR |    Buzzer
|    GPIOx|------------>|D7    OUT5|-------->|  ladder  |         /|
|    GPIOx|------------>|/WE   OUT6|-------->|          |     .--/ |
`---------'             |      OUT7|-------->|MSB       |-----|    |
                        `---------'          `----------'     `--` |
                                                               |  `|
                                                               |
                                                        GND ---
```

AUDIO OUT through RC filter Another option is to use the Pulse Width Modulated (PWM) AUDIO OUT pin that combines 4 channels with the Resistor-Capacitor based low-pass filter or better the Operation Amplifier (Op-amp) & Capacitor based integrator:

```
uController              SN76489
,---------.            ,---._.---.
|         |     4 Mhz ->|CLK   SEL0|<-- 0
|    GPIOx|----------->|D0   SEL1|<-- 0
|    GPIOx|----------->|D1       |
|    GPIOx|----------->|D2       |
|    GPIOx|----------->|D3       |          C1
|    GPIOx|----------->|D4       |     ,----||----.
|    GPIOx|----------->|D5       |     |          |
|    GPIOx|----------->|D6       |     |  Op-amp  |        Speaker
|    GPIOx|----------->|D7  AUDIO|     |   |X     |            /|
|    GPIOx|----------->|/WE  OUT |-----+---|-X    |   C2   .--/ |
`---------'            `---------'         |  }---+---||---|    |
                                        ,--|+/             `--` |
                                        |  |/               |  `|
                                        |                   |
                                   GND ---             GND ---
```

Separate channels through the Op-amp The third option is to externally combine 4 channels with the Operational Amplifier and low-pass filter:

```
uController              SN76489
,---------.            ,---._.---.
|         |    4 Mhz ->|CLK  SEL0|<-- 0
|    GPIOx|----------->|D0   SEL1|<-- 0
|    GPIOx|----------->|D1       |
|    GPIOx|----------->|D2       |
|    GPIOx|----------->|D3       |           C1
|    GPIOx|----------->|D4       |      ,----||----.
|    GPIOx|----------->|D5  chan0|---.  |          |
|    GPIOx|----------->|D6  chan1|---+  |  Op-amp  |        Speaker
|    GPIOx|----------->|D7  chan2|---+  |   |X     |            /|
|    GPIOx|----------->|/WE chan3|---+--+---|-X    |   C2   .--/ |
`---------'            `---------'          |  }---+---||---|    |
                                         ,--|+/             `--` |
                                         |  |/               |  `|
                                         |                   |
                                    GND ---             GND ---
```

Summary of commands to communicate with the chip

The SN76489 is programmed by updating its internal registers via the data bus. Below is a short summary of the communication protocol of SN76489. Please consult SN76489 Technical Manual for more information.

Command	Description	Parameters
`1cc0ffff`	Set tone fine frequency	`f` - 4 low bits, `c` - channel #
`00ffffff`	Follow up with coarse frequency	`f` - 6 high bits
`11100bff`	Set noise type and frequency	`b` - white/periodic, `f` - frequency control
`1cc1aaaa`	Set channel attenuation	`a` - 4 bit attenuation, `c` - channel #

NF1	NF0	Noise frequency control
0	0	Clock divided by 512
0	1	Clock divided by 1024
1	0	Clock divided by 2048
1	1	Use channel #2 tone frequency

Write to SN76489 Hold **/WE** low once data bus pins are set to the desired values. Pull **/WE** high before setting different value on the data bus.

Note frequency

Use the following formula to calculate the 10-bit period value for a particular note :

$$toneperiod_{cycles} = clock_{frequency}/(32_{cycles} * note_{frequency})$$

For example 10-bit value that plays 440 Hz note on a chip clocked at 4 MHz would be:

$$toneperiod_{cycles} = 4000000Hz/(32_{cycles} * 440Hz) = 284 = 11C_{hex}$$

An example to play a note accompanied with a lower volume noise

/WE	D7	D6/5	D4..D0	Explanation
0	1	00	01100	Set **channel #0** tone low 4-bits to $C_{hex} = 1100_{bin}$
0	0	00	10001	Set **channel #0** tone high 6-bits to $11_{hex} = 010001_{bin}$
0	1	00	10000	Set **channel #0** volume to **100%**, attenuation 4-bits are $0_{dec} = 0000_{bin}$
0	1	11	00100	Set **channel #3** noise type to **white** and divider to **512**

/WE	D7	D6/5	D4..D0	Explanation
0	1	11	11000	Set **channel #3** noise volume to **50%**, attenuation 4-bits are $8_{dec} = 1000_{bin}$

```
Timing diagram


CLK   ----      ----      ----      ----      ----      ----
   __/    `____/    `____/    `____/    `____/    `____/    `___ ...
     |        |         |         |         |         |
     |        |         |         |         |         |


/WE _       --        --        --        --        -------
     `_____/  `______/  `______/  `______/  `______/   *
                                                       ^
D7..D0_______  ________  ________  ________  ________  |
   _/10001100  00010001  10010000  11100100  11111000`_|______
      chan#0    chan#0    chan#0    chan#3    chan#3   |
    tone=h??C    =h11C   atten=0    div=16    atten=8  |
      h011C = 440 Hz              /16 = ~1 Khz         |
                                 white noise           |
                                                       |
                                                 noise restarts
                                            after /WE goes high and
                                     there was a write to noise register
```

Configurable clock divider

Clock divider can be controlled through **SEL0** and **SEL1** control pins and allows to select between 3 chip variants.

SEL1	SEL0	Description	Clock frequency
0	0	SN76489 mode, clock divided by 16	3.5 .. 4.2 MHz
1	1	—–//—–	3.5 .. 4.2 MHz
0	1	SN76494 mode, no clock divider	250 .. 500 kHZ
1	0	New mode for TT05, clock div. 128	25 .. 50 MHz

SEL1	SEL0	Formula to calculate the 10-bit tone period value for a note
0	0	$clock_{frequency}/(32_{cycles} * note_{frequency})$
1	1	—//—
0	1	$clock_{frequency}/(2_{cycles} * note_{frequency})$
1	0	$clock_{frequency}/(256_{cycles} * note_{frequency})$

Some examples of music recorded from the chip simulation

- https://www.youtube.com/watch?v=ghBGasckpSY
- https://www.youtube.com/watch?v=HXLAdA02I-w

Pinout

#	Input	Output	Bidirectional
0	D0 data bus	digital audio LSB	(in) **/WE** write enable
1	D1 data bus	digital audio	(in) **SEL0** clock divider
2	D2 data bus	digital audio	(in) **SEL1** clock divider
3	D3 data bus	digital audio	(out) channel 0 (PWM)
4	D4 data bus	digital audio	(out) channel 1 (PWM)
5	D5 data bus	digital audio	(out) channel 2 (PWM)
6	D6 data bus	digital audio	(out) channel 3 (PWM)
7	D7 data bus	digital audio MSB	(out) AUDIO OUT master (PWM)

Miniature Programmable Interrupt Timer [202]

- Author: Steve Jenson
- Description: When the given 16-bit counter reaches 0 an interrupt pin is asserted for one clock cycle.
- GitHub repository
- HDL project
- Mux address: 202
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

A minimal clone of a programmable interrupt timer. Inspried by the Intel 8253 but without most of the features or headaches. See the `README.md` for detailed documentation.

How to test

set input pins to 0x00. pull write enable high, address line 0 low, address line 0 low. set input pins to 0x10, pull write enable high, address line 0 low, address line 1 high. pull bidi pin 3 (timer_start) high, count 10 clock cycles and see if the interrupt pin has pulled high for 1 cycle

Pinout

#	Input	Output	Bidirectional
0	config[0] - use a clock divider	divider on?	/we write enable for config
1	config1 - repeat the interrupt?	counter set?	set config address 0
2	config2	pit active?	set config address 1
3	config[3]	pit in reset?	start the timer
4	config[4]	pit currently interrupting?	none
5	config[5]	f	none
6	config[6]	g	none
7	config[7]	h	none

7-segment Name Display [203]

- Author: Gerry Chen
- Description: Displays names on the 7-segment display one at a time.
- GitHub repository
- HDL project
- Mux address: 203
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Uses a counter to display names on a 7-segment display.

A clock divider slows down the segments to 1 per second (default) as in the 7-segment counter template project. The bottom 8-bits of the counter are output on the bidirectional outputs. The bottom-5 bits of the dedicated inputs define how fast the clock divider is: if non-zero, this formula is used for the wraparound value of the divider: {ui_in[4:2], 18'b0, ui_in[1:0]}. Setting the input to 0bxxx00001 will therefore have clock divider of 1 (i.e. match the clock) so that a manual debounced push-button can be used in place of the clock.

Each second, one letter of a name is displayed. The top 3-bits of the dedicated inputs define which name is displayed.

How to test

After reset, a new letter should displayed each second with a 10MHz input clock. Changing the 3 MSB of the input should change which name is displayed. Changing the 5 LSB of the input should change how quickly the letters are updated.

Pinout

#	Input	Output	Bidirectional
0	name bit 2	segment a	second counter bit 0
1	name bit 1	segment b	second counter bit 1
2	name bit 0	segment c	second counter bit 2
3	clock divider bit 23	segment d	second counter bit 3
4	clock divider bit 22	segment e	second counter bit 4

#	Input	Output	Bidirectional
5	clock divider bit 21	segment f	second counter bit 5
6	clock divider bit 1	segment g	second counter bit 6
7	clock divider bit 0	dot	second counter bit 7

Tetris [204]

- Author: Carson Swoveland
- Description: Implements the second-most-popular game of all time in hardware
- GitHub repository
- HDL project
- Mux address: 204
- Extra docs
- Clock: 6250000 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	Move Left	VGA HSync	A0/D0
1	Move Down	VGA VSync	A1/D1
2	Move Left	VGA Red	A2/D2
3	Spin Counterclockwise	VGA Green	A3
4	Spin Clockwise	VGA Blue	A4
5	none	Memory Start	A5
6	none	Memory Continue	none
7	none	Memory Write Enable	none

Simple_Timer-MBA [205]

- Author: Morteza Biglari-Abhari
- Description: Count up to the specified value (between 01 to 99), one second at a time. Time_Out will be '1' when reaches the expected value
- GitHub repository
- HDL project
- Mux address: 205
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

This is a two-digits timer, which can count from 00 to 99 in seconds. The time to stop counting is given through 8 input switches (ui_in) as two BCD numbers (which can be from 00 to 99). This number is loaded into an internal register when input Load is '1'. Then when input Start is '1' the counting begins. The timer stops when it reaches the specified count number and then output Time_Out will become '1'. Seconds (either Ones or Tens) is displayed on 7-Seg display depending on uio_in[3].

How to test

After reset, when Start and Load inputs are activated the counter should increase by one every second with a 10MHz input clock. Experiment by changing the inputs to count different number of seconds

Pinout

#	Input	Output	Bidirectional
0	Timeout (in Seconds) bits 7 to 0	segment a	Load bit 0 (uio_in[0])
1	n/a	segment b	Start bit 1 (uio_in1)
2	n/a	segment c	Tens or Ones select bit 3 (uio_in[3])
3	n/a	segment d	Time_Out uio_out[7]
4	n/a	segment e	n/a
5	n/a	segment f	n/a
6	n/a	segment g	n/a
7	n/a	dot	n/a

UART Transceiver [206]

- Author: Nathan Zhu
- Description: UART Transceiver with tx and rx functions at 9600 baud rate
- GitHub repository
- HDL project
- Mux address: 206
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Given paramaters of the clock frequency and the desired baud rate, we can calculate the number of ticks of the clock to correspond to a tick at the desired baud rate. Then we can send the start bit, 8 data bits, and a stop bit. Our design uses oversampling to get the value at the middle of the pulse, and then returns our data bit with a read_done signal. For the transmitter, we take a data byte of input and, using the pulse width calculated earlier, send a proper UART sequence with the correct timing.

How to test

After reset, the receiver will wait for the start bit, and then 8 data bits, and then a stop bit. After reset, we can set the 8 data bits and a data_ready bit and the resulting uart transmission sequence will appear on the tx output signal.

Pinout

#	Input	Output	Bidirectional
0	rx bit to signal the bits we receive, dataReady highlighting data is ready for tx	segment a / dataOut[0] / tx for uart packet bits	finished_read - finished reception / dataIn[0]
1	none	segment b / dataOut1	dataIn1
2	none	segment c / dataOut2	dataIn2

#	Input	Output	Bidirectional
3	none	segment d / dataOut[3]	dataIn[3]
4	none	segment e / dataOut[4]	dataIn[4]
5	none	segment f / dataOut[5]	dataIn[5]
6	none	segment g / dataOut[6]	dataIn[6]
7	bit to test if we want tx or rx	segment h / dataOut[7]	dataIn[7]

AGL CorticoNeuro-1 [207]

- Author: Arfan Ghani
- Description: Information is encoded as a sequence of events or spikes in neuro-inspired computing. Investigating how information is represented and processed as spike trains is of particular interest. This chip implements several test clusters featuring various spike trains.
- GitHub repository
- Wokwi project
- Mux address: 207
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

The input clock is connected with the inputs of the neuron clusters. The bi-directional pins are provided where external input stimulus could be provided. The raster spiking plots are generated to observe the variability of different spiking neuron clusters.

How to test

Provide input clock frequencies to the neuron clusters and observe the output through the oscilloscope.

Pinout

#	Input	Output	Bidirectional
0	CLOCK Attached to the on-board clock	OUT0 on-board CLOCK	D0 OUTPUT from a 1-bit FF
1	INO Connected with a 1-bit FF	OUT1 Output from the LFSR	D7 OUTPUT from (1x3x3) cluster
2	IN1 external input to the MUX	OUT2 Output from 2-bit FF	D1 OUTPUT from the (5X5) cluster
3	IN2 Enable signal to the MUX	OUT3 Output from 3-bit FF	D2 OUTPUT from (6x6x6) cluster

#	Input	Output	Bidirectional
4	IN3 Connected as a SELECT pin for the MUX (connected with (1x3x2x1) and (6x6x6) cluster.	OUT4 Output from 4-bit FF	D3 OUTPUT from (6x6x6) cluster
5	IN4 Input to the (6x6x6) cluster	OUT5 Output from 5-bit FF	D4 connected as an OUTPUT pin from either the (6x6x6) cluster or the (1x3x2x1) cluster (where IN3 is the input select pin)
6	IN5 Input to the (6x6x6) cluster	OUT6 Output from 6-bit FF	D5 INPUT to the (6x6x6) cluster
7	IN6 Input to the (6x6x6) cluster	OUT7 MUX output)	D6 INPUT to the (6x6x6) cluster

Leaky-Integrated Fire Neuron [224]

- Author: Ruhai Lin
- Description: Adaptive LIF Neuron
- GitHub repository
- HDL project
- Mux address: 224
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

When a LIF (Leaky - Integrate and Fire) neuron integrates enough current stimulation, it will be activated and Fire once spike. This current is introduced by the 8-bit chip input pin, but while integrating, the LIF neuron gradually loses the previously accumulated current like an hourglass, so it is called Leaky. this module implements this biological behavior with a mathematical equation. The state of the neuron can be monitored externally through the 8-bit chip output pins.

The LIF neuron module also includes adaptive threshold and adaptive decay rate to dynamically adjust its own fire threshold or decay rate. The adaptive threshold can be enabled by setting bit 0 of the bidirectional IO, and the adaptive decay rate can be enabled by setting bit 1 of the bidirectional IO. simulations show that this allows the LIF neuron to enhance the sparsity of spikes while preserving the input features, which improves the efficiency of the chip.

How to test

LIF neurons will receive current inputs in three different gears (strong, medium, and weak). The spike rate should be higher when the current is stronger and lower when the current is weaker. After turning on adaptive threshold and adaptive decay rate the chip needs to retain this feature while trying to enhance sparsity to avoid neurons that fire frequently or not at all, to make it consistent with realistic biological characteristics.

Pinout

#	Input	Output	Bidirectional
0	current bit 11	membrane a	adaptive_threshold_enable bit 0

#	Input	Output	Bidirectional
1	current bit 12	membrane b	adaptive_beta_enable bit 1
2	current bit 13	membrane c	second counter bit 2
3	current bit 14	membrane d	second counter bit 3
4	current bit 15	membrane e	second counter bit 4
5	current bit 16	membrane f	second counter bit 5
6	current bit 17	membrane g	second counter bit 6
7	current bit 18	membrane h	spike bit 7

MyUART [225]

- Author: LogicComputing
- Description: A small UART that outputs my name
- GitHub repository
- HDL project
- Mux address: 225
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

This design contains a small UART that will output a string every ~1s. No input is required. It expect a 10 MHz clock.

How to test

You simply need to connect an UART RX on uo_out[0] and you will see my name ! UART is 115200 baud, one start bit, eight bit of data, one parity bit and one stop bit. I generate a sinus signal on uo_out[7:1].

Pinout

#	Input	Output	Bidirectional
0	none	{'uo_out[0]': 'UART with my name every ~1s.'}	none
1	none	{'uo_out[7:1]': 'A sinus is generated.'}	none
2	none	n/a	none
3	none	n/a	none
4	none	n/a	none
5	none	n/a	none
6	none	n/a	none
7	none	n/a	none

UART test [226]

- Author: Rodolfo Sanchez Fraga
- Description: UART test
- GitHub repository
- Wokwi project
- Mux address: 226
- Extra docs
- Clock: 0 Hz
- External hardware: UART receiver

How it works

This project is an edited version of the example CUSTOMISABLE DESIGN - UART from digital design guide. Implements a a UART transmitter using registers made from D-flip flops and multiplexers. The characters QSM are sent continuously.

How to test

To begin transmission:

1. Connect CLK signal
2. Set IN6 ("Load") to OFF
3. Set IN7 ("Output Enable") to ON
4. Set IN6 ("Load") to ON

Pinout

#	Input	Output	Bidirectional
0	IN6 - Load	TX	OUT0 - Output enable indicator
1	IN7 - Output enable	OUT1 - Load	TX indicator
2	n/a	n/a	n/a
3	n/a	n/a	n/a
4	n/a	n/a	n/a
5	n/a	n/a	n/a
6	n/a	n/a	n/a
7	n/a	n/a	n/a

Heart Rhythm Analyzer [227]

- Author: Nissan Kunju
- Description: The design integrates a threshold-based filtering mechanism followed by peak detection on the filtered data.
- GitHub repository
- HDL project
- Mux address: 227
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

The peak detection module implements a peak detection circuit that checks for the occurrence of a peak in the input data stream over three consecutive clock cycles. The threshold filtering module is a threshold filter that processes the input data based on the threshold and higher flag, and then passes it to the peak detection module. The clock divider module divides the input clock signal by 2 to generate a new clock signal clk2. Example: setting the inputs to 00010000 will program 16384 into the compare register. With a 10MHz clock the counter will increment ~610 times per second.

How to test

Keep the reset at 0 for two clock pulses. Change the reset to 1. Set the threshold pin to 1 and send the lower four bits first. Set the higher pin to 1 and send the higher four bits. Switch the threshold to 0. Alternate between sending the lower and higher four bits as inputs.

Pinout

#	Input	Output	Bidirectional
0	compare bit 11	segment a	second counter bit 0
1	compare bit 12	segment b	second counter bit 1
2	compare bit 13	segment c	second counter bit 2
3	compare bit 14	segment d	second counter bit 3
4	compare bit 15	segment e	second counter bit 4
5	compare bit 16	segment f	second counter bit 5
6	compare bit 17	segment g	second counter bit 6

#	Input	Output	Bidirectional
7	compare bit 18	dot	second counter bit 7

Spike-timing dependent plasticity (Verilog Demo) [228]

- Author: Binh Nguyen
- Description: Update neuron weight using spike-timing dependent plasticity
- GitHub repository
- HDL project
- Mux address: 228
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Uses a set of registers to implement a leaky integrate-and-fire (LIF) neuron for spike-timing dependent plasticity learning (STDP) rule. Two LIF neurons are instantiated and a stdp module handles the logic for the timing and weight update.

How to test

After reset, a current is applied at different amplitudes and the input to the neuron is integrated at every clock cycle If a pre-synaptic spike and post-synaptic spike occurs, time difference is measured and applied to the synatic weight update for this connection.

Pinout

#	Input	Output	Bidirectional
0	drive current	LIF spike	second counter bit 0
1	n/a	LIF state	second counter bit 1
2	n/a	synaptic weight	second counter bit 2
3	n/a	n/a	second counter bit 3
4	n/a	n/a	second counter bit 4
5	n/a	n/a	second counter bit 5
6	n/a	n/a	second counter bit 6
7	n/a	n/a	second counter bit 7

Tiny Tapeout 5 TM project1 [229]

- Author: Miho Yamada
- Description: counter
- GitHub repository
- Wokwi project
- Mux address: 229
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	none	segment a	none
1	none	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

Thermocouple-to-temperature converter (digital backend) [230]

- Author: Aidan Medcalf
- Description: Converts digitized thermocouple voltage into temperature.
- GitHub repository
- HDL project
- Mux address: 230
- Extra docs
- Clock: 10000000 Hz
- External hardware: Thermocouple AFE with compatible ranging, for chosen thermocouple type

How it works

Converts 10-bit thermocouple ADC counts into temperature by approximating the transfer function with piecewise linear segments and interpolating.

- Interface: SPI (16-bit word)
- ADC interface: SPI (16-bit word, 10 bits used)
- Output: Temperature in "centi-celsius", predivided by 4; 16-bit over full positive range of thermocouple type
- ADC passthrough: When enabled, directly connects SPI master to ADC for configuration
- Type-J and type-K thermocouples supported

ADC range: 0 counts = 0 mV = 0 C, max counts (1023) = max mV = max C. Example: For type-K thermocouple, 1023 counts = 54.886 mV = 1372 C

Temperature output: Output is in "centi-Celsius", or hundredths of degrees C, pre-divided by 4, with a granularity of 0.4C. `T = A / 25.0` Where T is in degrees C, and A is the value read from SPI. For example, for a type-K thermocouple at 415.06C, A = 10376 (0x2888), and T = 415.04. Note the error of 0.02C.

Configuration: There are two configuration bits. Write to cfg[1:0] by issuing a SPI transaction with the high bit set (i.e. write 0x800X).

- cfg1: Thermocouple type: 0 = J, 1 = K
- cfg[0]: ADC passthrough enable

How to test

Requires a J or K thermocouple analog front-end with compatible ranging. Wait 20 clocks after reset, then read 16-bit temperaute from device.

Pinout

#	Input	Output	Bidirectional
0	unused	unused	SCE
1	unused	unused	SIN
2	unused	unused	SOUT
3	unused	unused	SCK
4	unused	unused	ADC_SCE
5	unused	unused	ADC_SOUT
6	unused	unused	ADC_SIN
7	unused	unused	ADC_SCK

Naive 8-bit Binary Counter [231]

- Author: Sean Bruton
- Description: A simple 8-bit binary counter
- GitHub repository
- Wokwi project
- Mux address: 231
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Eight D flip flops chained together count the clock input and use the 8 outputs to represent the binary value. The counter lacks useful features like a deterministic initial state or a reset function. This was constructed during the Hackaday Supercon 2023 ASIC workshop as a rapid learning exercise.

How to test

Pulse the clock and monitor the outputs for the binary value.

Pinout

#	Input	Output	Bidirectional
0	none	segment a	none
1	none	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

tinyscanchain Test Design [232]

- Author: Anish Singhani
- Description: Test design for tinyscanchain, based on seven segment seconds design
- GitHub repository
- HDL project
- Mux address: 232
- Extra docs
- Clock: 1000 Hz
- External hardware:

How it works

tinyscanchain is a scan-chain implementation in less than 80 lines of Python. This is a test design based on the use of seven segment seconds.

How to test

After reset, the counter should increase by one every second with a 1kHz input clock. Experiment by changing the inputs to change the counting speed. Use the scan chain to test the internal state of the design.

Pinout

#	Input	Output	Bidirectional
0	enable	segment a	unused
1	scan chain input	segment b	unused
2	scan chain enable	segment c	unused
3	unused	segment d	unused
4	unused	segment e	unused
5	unused	segment f	unused
6	unused	segment g	unused
7	unused	scan chain output	unused

6 digit chronometer. [233]

- Author: Carlos Guerra & Marco Gurrola
- Description: 6 digit chronometer. Displays 2 digits for minutes, 2 digits for seconds and 2 digits for hundredths of a second.
- GitHub repository
- HDL project
- Mux address: 233
- Extra docs
- Clock: Hz
- External hardware: You need six 7 segment common cathode displays, push buttons.

How it works

The project consists of a 50 MHz chronometer in which minutes, seconds and hundredths of a second are shown through six 7 segment displays. It can be initialized or paused pressing the start button, pressing the reset button will cause it to restart the counter.

How to test

For testing the chronometer project connect push buttons to the reset and bt_ent (start button) inputs. It is designed to work with six 7 segment common cathode displays. Unidirectional output pins must be connected to displays cathodes. Bidirectioanl output pin must be connected to displays anodes.

Pinout

#	Input	Output	Bidirectional
0	bt_ent (start button)	Display cathode 1	Segment a
1	reset	Display cathode 2	Segment b
2	clk	Display cathode 3	Segment c
3	n/a	Display cathode 4	Segment d
4	n/a	Display cathode 5	Segment e
5	n/a	Display cathode 6	Segment f
6	n/a	n/a	Segment g
7	n/a	n/a	dot

Convolutional Network Circuit Chip Design [234]

- Author: Rogelio Franco
- Description: Silicon Chip design of a CNN
- GitHub repository
- HDL project
- Mux address: 234
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Explain how the project works later...

How to test

After reset, the counter should increase by one every second with a 10MHz input clock. Experiment by changing the inputs to change the counting speed.

Pinout

#	Input	Output	Bidirectional
0	compare bit 11	segment a	second counter bit 0
1	compare bit 12	segment b	second counter bit 1
2	compare bit 13	segment c	second counter bit 2
3	compare bit 14	segment d	second counter bit 3
4	compare bit 15	segment e	second counter bit 4
5	compare bit 16	segment f	second counter bit 5
6	compare bit 17	segment g	second counter bit 6
7	compare bit 18	dot	second counter bit 7

Matrix Vector Multiplication Accelerator [235]

- Author: Mathias Eriksen
- Description: This project takes in a 3x3 weight matrix in Compressed Sparse Row format, value is quantized and 8 bits long. It also takes in the corresponding 3 bit spike train. It then computes the matrix vector multiplication product and outputs the resulting vector on the output line
- GitHub repository
- HDL project
- Mux address: 235
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Uses a set of registers and flags from the CPU to fetch the weight matrix in CSR format as well as the spike train. The values are passed in one at a time, and the entire matrix is loaded into registers that are internal to the IC

Once the full sparse matrix and spike train are loaded in, an algorithm is used to compute the resultant vector of the matrix vector multiplication of the weight matrix and the spike train

Finally, the output vector is transmitted on the output line, along with a flag bit which flips each time a new value is sent out.

How to test

After reset, send values in CSR format using the input bits described below. Send a value by toggling the sending CPU flag for one clock cycle while the values are in their respective registers. Repeat for the entire matrix, toggling the sending CPU flag low between each value. Then, check the return values by waiting for the sending out flag from the IC to flip. After the first flip, the other two values will be sent on each clock edge.

Pinout

#	Input	Output	Bidirectional
0	Input Value bit 0	Output Value bit 0	FETCH Ready flag (out)

#	Input	Output	Bidirectional
1	Input Value bit 1	Output Value bit 1	Sending out flag (out)
2	Input Value bit 2	Output Value bit 2	Done sending flag (in)
3	Input Value bit 3	Output Value bit 3	Sending CPU flag (in)
4	Input Value bit 4	Output Value bit 4	Column Value bit 0
5	Input Value bit 5	Output Value bit 5	Column Value bit 1
6	Input Value bit 6	Output Value bit 6	Row Value bit 0
7	Input Value bit 7	Output Value bit 7	Row Value bit 1

Perceptron (Neuromeme) [236]

- Author: Dylan Louie
- Description: A perceptron or a 9 + 10 adder
- GitHub repository
- HDL project
- Mux address: 236
- Extra docs
- Clock: Hz
- External hardware:

How it works

Reads from two 8-bit input and creates a weighted sum
of the 16 bits.

The 8-bit wieights are default 10000000 and are unuptatable. (10000000 represents 0.5 if the you conceptualize a . on the far left or represents 128 if you conceptualize a . on the far right)

If the weighted sum is greater than the threshold, 11111110, than it will classify the input as 1 otherwise it will classify it as 0.

9 + 10 = 21

Credit/Thanks to my Professor: UCSC's Neuromorphic Lab's Jason K Eshraghian Ph.D.

How to test

Any input with all 0's should be classified as 0.

Math:

Note: The threshold is 11111110 which can be thought of as 0.99993896484

w0*i0 + w1*i1 + ... + w15*i15

0.5*0 + 0.5*0 + ... + 0.5*0 = 0

Any input with fifteen 0's and one 1's should be classified as 0.

Math:

w0*i0 + w1*i1 + ... + w15*i15

0.5_1 + 0.5_0 + … + 0.5*0 = 0.5

Any input with two or more 1's should be classified as 1.

Math:

w0_i0 + w1_i1 + w2_i2 + … + w15_i15

0.5_1 + 0.5_1 + 0.5_0 + … + 0.5_0 = 1

0.5_1 + 0.5_1 + 0.5_1 + 0.5_0 + … + 0.5*0 > 1

Pinout

#	Input	Output	Bidirectional
0	input 0 asssociated with weight 0	Read as an 8 bit output along with other outputs	input 9 asssociated with weight 9
1	input 1 asssociated with weight 1	Read as an 8 bit output along with other outputs	input 10 asssociated with weight 10
2	input 2 asssociated with weight 2	Read as an 8 bit output along with other outputs	input 11 asssociated with weight 11
3	input 3 asssociated with weight 3	Read as an 8 bit output along with other outputs	input 12 asssociated with weight 12
4	input 4 asssociated with weight 4	Read as an 8 bit output along with other outputs	input 13 asssociated with weight 13
5	input 5 asssociated with weight 5	Read as an 8 bit output along with other outputs	input 14 asssociated with weight 14
6	input 6 asssociated with weight 6	Read as an 8 bit output along with other outputs	input 15 asssociated with weight 15
7	input 7 asssociated with weight 7	None	input 16 asssociated with weight 16

4 Bit ALU [237]

- Author: Lucius Chee
- Description: A simple 4-bit, 13 instruction, arithmetic logic unit.
- GitHub repository
- HDL project
- Mux address: 237
- Extra docs
- Clock: 0 Hz
- External hardware: digital logic (e.g. buttons/sensors)

How it works

The input 8 bits are split into the upper 4 bits (value y), and lower 4 bits (value x). Depending on the instruction given after the select pin, operations will be performed on the values to give an 8 bit output. The select pins use the bi-directional I/O.

|Select (bidi 3 - 0)|Operation| |0|+| |1|-| |2|*| |3|/| |4|bitwise AND| |5|bitwise OR| |6|bitwise XOR| |7|bitwise NAND| |8|bitwise NOR| |9|~ (negation of 8 bits)| |10|% (modulo)| |11|« (left shift)| |12|» (right shift)| |other|input bits as is|

How to test

After reset, the counter should increase by one every second with a 10MHz input clock. Experiment by changing the inputs to change the counting speed.

Pinout

#	Input	Output	Bidirectional
0	x3	output 7	instruction select bit 0
1	x2	output 6	instruction select bit 1
2	x1	output 5	instruction select bit 2
3	x0	output 4	none
4	y3	output 3	none
5	y2	output 2	none
6	y1	output 1	none
7	y0	output 0	none

Binary Neural Network (Verilog Demo) [238]

- Author: Aravind Ramamoorthy
- Description: a single neuron in a Binarized Neural Network (BNN), performing binary multiplication with XNOR, accumulation, and sign activation.
- GitHub repository
- HDL project
- Mux address: 238
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

This model simulates the behavior of a single neuron within a Binarized Neural Network (BNN)

The XNOR operation is used to perform binary multiplication. A 32-bit signal used for accumulating the results of multiple XNOR operations, simulating the weighted sum of inputs.

"Sign activation function" applies to the accumulated result. It maps the accumulated value to either +1 or -1 based on the sign.

How to test

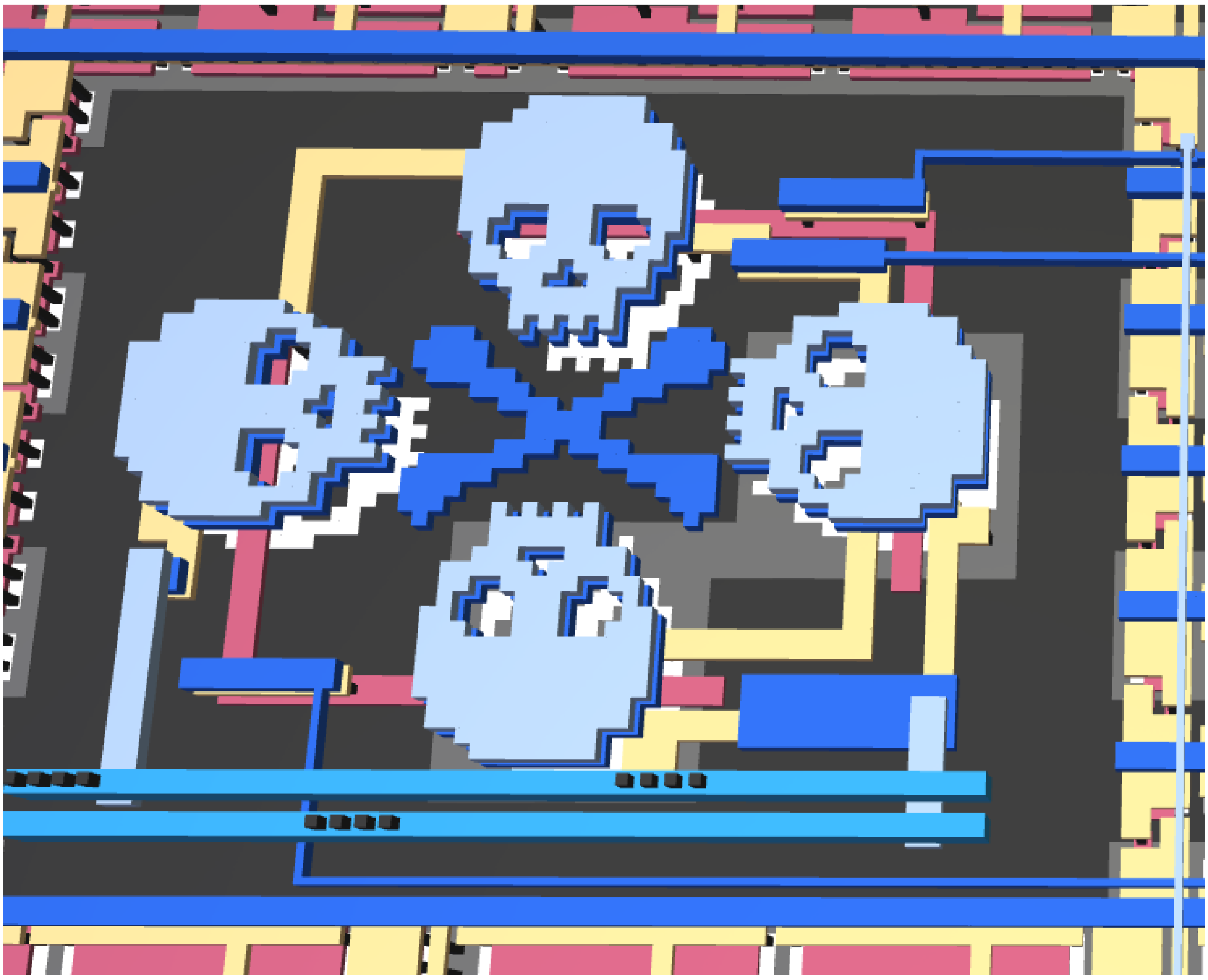
Reset the circuit to set to 0. The constant Input and weight is provided with enable signal to begin XNOR multiplication

Pinout

#	Input	Output	Bidirectional
0	compare bit 11	segment a	second counter bit 0
1	compare bit 12	segment b	second counter bit 1
2	compare bit 13	segment c	second counter bit 2
3	compare bit 14	segment d	second counter bit 3
4	compare bit 15	segment e	second counter bit 4
5	compare bit 16	segment f	second counter bit 5
6	compare bit 17	segment g	second counter bit 6

#	Input	Output	Bidirectional
7	compare bit 18	dot	second counter bit 7

SkullFET [239]



- Author: Uri Shaked
- Description: Bare-bone transistors
- GitHub repository
- HDL project
- Mux address: 239
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Hand-crafted, skull-shaped MOSFET transistors.

The project contains three SkullFET devices: a NOT gate, a NAND gate, and a SR flip-flop.

How to test

Input some values into A and B, and observe the outputs. The first output is connected to the SkullFET inverter, and the second output is connected to the SkullFET NAND gate.

Pulse ~S to set the SkullFlop (Q), and pulse ~R to reset it.

Pinout

#	Input	Output	Bidirectional
0	A	~A	none
1	B	~(A&B)	none
2	~S	~Q	none
3	~R	Q	none
4	none	none	none
5	none	none	none
6	none	none	none
7	none	none	none

Wavetable Sound Generator [256]

- Author: Ryota Suzuki
- Description: Small wavetable/PSG type sound generator with I2S output
- GitHub repository
- HDL project
- Mux address: 256
- Extra docs
- Clock: 50000000 Hz
- External hardware: I2S DAC is required (I tested this design with FPGA and PCM5102A DAC)

How it works

This project is Small wave table/PSG type sound generator with I2S output. Major features are:

- 4 channel sound generator
- 4-bit x 32depth wave table (can be uses as 2 of 16depth wave table)
- 8 selectable waveform (3x pulse,1x noise, 4x wave table)
- 8-bit volume(only for PSG mode, wave table mode is 4-step volume)
- 16-bit frequency
- Sampling Frequency is 48828.125Hz (at 50MHz clock)
- I2S output (16-bit mono)
- SPI control interface

You can control this sound generator by SPI interface. SPI mode is mode 0 (CPOL=0, CPHA=0), and transaction length is 24-bit. first 8-bit is register address, and next 16-bit is data. Data is MSB first.

Addr	Description	Width
0x00-0x03	Frequency[0]-[3]	16bit
0x04-0x07	Volume[0]-[3]	8bit (lower 8bits are valid)
0x08-0x0b	Waveform Select[0]-[3]	3bit (lower 3bits are valid)
0x20-0x3f	WaveTable[0]-[31]	4bit (lower 4bits are valid)

How to test

Connect I2S output to I2S DAC, and control this sound generator by SPI interface. SPI input is connected to RP2040's SPI1 on TT05 breakout board.

Pinout

#	Input	Output	Bidirectional
0	SPI CS Input	I2S Bit Clock	none
1	SPI CLK Input	I2S Word Select	none
2	SPI MOSI Input	I2S Data	none
3	none	none	none
4	none	none	none
5	none	none	none
6	none	none	none
7	none	none	none

PWM signal generation with Winner-Take-All selection [258]

- Author: Ruibin Mao
- Description: 8-channel 12-bit PWM signal generation. Time-domain Winner-Take-All (WTA) able to find smallest PWM signal and k-smallest signal
- GitHub repository
- HDL project
- Mux address: 258
- Extra docs
- Clock: 20000000 Hz
- External hardware:

How it works

- General Description This design aims to build a PWM generation and a Winner-Take-All selection circuit for smallest PWM duration detection. The circuit has 8 built-in 12-bit PWM signal generation with a common trigger. The circuit can also accept external 8-bit PWM signals with internal 8-bit switch. User can choose for each channel whether to use internal PWM signals or external signals. The 8-channel PWM signal will go through a synchronization stage to make sure it synchronizes with the internal clock. The winner-take-all is done by sensing the falling edge of the PWM signal. The falling edge detection pulse will be stored in the falling edge register once it's been triggered. The nearest neighbor (NN) signal or smallest duration signal will be detected once a first falling edge is triggered. An internal counter will count how many falling edges are triggered and once it reaches threshold K, the falling edge register will latch the address. So that the K nearest results are stored.
- Detail of the internal modules `SPI 1`: It's for the pulse-width configuration of 8 PWM signals. Users should latch 96-bit signals through the SPI 1 to configure all PWM signals. This channel can also shift out the results of 8-channel time-to-digital converter (TDC) which is used to convert the PWM duration into digital signals. `SPI 2`: It's for the configuration of internal switch of 8-channel. Each switch will select either internal PWM or external signal is used. Another 3-bit signal is used to set the number K which is K-smallest duration of input PWM signal. The MISO will shift out the 8-bit smallest PWM address and 8-bit K-smallest PWM address. `PWM_sync`: It synchronize the PWM signals comming in and convert it to digital signal with TDC and detect the falling edge. `k_nn`: It senses the 8 falling edges and store them once it's been triggered. An internal counter will count the number of falling edges at each clock cycle and latch the address of existing falling edges.

How to test

The testing can refer to the testbench in `src/test.py` After resetting, the user should do

1. Config the 8 channel PWM pulse width and 3-bit threshold K using SPI 1 and SPI 2. For SPI 1, user should send 12*8=96 bit signals using FPGA, the order is "Channel 0-1-2-3-4-5-6-7".
2. Config the switch and 3-bit threshold using SPI 2, the order is "8-bit switch - 3-bit threshold". For each channel, external signal will be used if switch bit is '1' or the internal PWM is used if switch bit is '0'.
3. Activate the PWM trigger which is ui_in[4].
4. Wait for at least 2**12 clock cycles.
5. Readout the 12-bit TDC result with SPI 1, the order is "Channel 0-1-2-3-4-5-6-7".
6. Readout the 8-bit smallest address and K-smallest address, the order is "nn - k_nn"

Pinout

#	Input	Output	Bidirectional
0	SPI 1 SS	SPI 1 MISO	External PWM signal 0
1	SPI 2 SS	SPI 2 MISO	External PWM signal 1
2	SPI 1 MOSI	PWM[0] signal	External PWM signal 2
3	SPI 2 MOSI	PWM[0] after cross-domain synchronization	External PWM signal 3
4	PWM Trigger	PWM[0] falling edge detection	External PWM signal 4
5	None	PWM[7] signal	External PWM signal 5
6	None	PWM[7] after cross-domain synchronization	External PWM signal 6
7	None	PWM[7] falling edge detection	External PWM signal 7

Multimode Modem [260]

- Author: Joerdson Silva
- Description: Performs digital modulation and demodulation in amplitude, frequency and phase schemes.
- GitHub repository
- HDL project
- Mux address: 260
- Extra docs
- Clock: 50000000 Hz
- External hardware: oscilloscope or signal analyzer

How it works

The multimode modem uses a clock signal to generate digitized signals over time, in sinusoidal format (carrier wave). From this digitized sinusoid, the modulation process is applied using different methods for each scheme, implemented through specific internal blocks to perform modulations ASK (switching the amplitude of the sine wave), FSK (switching the frequency of the sine wave through a digital signal modulator) and PSK (phase coding). In the demodulation stage, these three modulation schemes are analyzed to recover the original information, manifesting as '0' or '1' values that reflect the data signal already restored after the process.

How to test

The multimode modem has the following inputs and outputs:

- Input - clock (1 bit)
- Input - reset (1 bit)
- Input - sel (2 bits)
- Output - mod_out (7 bits)
- Output - demod_out (1 bit)

Apply a "clock" of 40~50 MHz. Then, apply a "reset" signal of logic level "1" to synchronize the modem system and then make the "reset" signal a logic level "0". After that, select the type of modulation to be used, as per the sequence below:

- Sel = "01" <= ASK modulation and demodulation
- Sel = "10" <= FSK modulation and demodulation
- Sel = "11" <= PSK modulation and demodulation

After selecting the modulation type, the modulated signal is expressed at the "mod_out" output, and the demodulated signal at the "demod_out" output.

Pinout

#	Input	Output	Bidirectional
0	clock	mod_out_0	none
1	reset	mod_out_1	none
2	sel_0	mod_out_2	none
3	sel_1	mod_out_3	none
4	none	mod_out_4	none
5	none	mod_out_5	none
6	none	mod_out_6	none
7	none	demod_out	none

Analog emulation monosynth [262]

- Author: Toivo Henningsson
- Description: One synth voice with two oscillators and a 2nd order filter
- GitHub repository
- HDL project
- Mux address: 262
- Extra docs
- Clock: 50000000 Hz
- External hardware: audio plug to connect to audio input, voltage divider to protect it!

How it works

The synth contains two oscillators with controllable frequency and waveform and a second order low pass filter with controllable cutoff frequency, resonance, and input amplification, similar to a simple analog synth. (Though the analog synth usually has the variable amplification after the filter.) The created audio samples are passed through a pulse width modulator (PWM) to create an audio signal on an output pin.

Sweep rates can be programmed for each parameter, to create simple envelopes. All parameters can be set through a register interface. By changing the sweep rates at specific points in time, more complex envelopes can be created.

The sample rate is 50 MHz/32, or 1.5625 MHz, far above the audible range, to avoid aliasing issues while allowing a fine enough spacing of oscillator frequencies: the oscillator period is always a whole number of samples, which avoid inharmonic aliasing effects.

The oscillators use counters that count down by a `2^n` each sample. If the counter would become negative, the period is added. A second (`n` bit) sawtooth counter counts how many times the period has been added. After one period, the sawtooth counter has incremented `2^n` times, and wraps around.

To reach lower octaves, an octave divider is used. `oct_enables[i]` is high once every `2^i` cycles. An octave `oct` is specified for the oscillator frequency, and the counter is only update when `oct_enables[oct]` is high. This keeps the same relative frequency accuracy for each octave.

The octave + period arrangement means that the full period is specified in a simple floating point format. This serves as a quasi exponential conversion, which emulates the V/octave, V/dB etc scales typically used in analog synths, and causes a quasi exponential response when sweeping the frequencies.

The filter is two pole filter with two states. Small update steps are taken every sample. Instead of a multiplier, a barrel shifter (variable right shift) is used to calculate the state change. The barrel shifter and associated adder is shared between all filter update steps (and the dither step for the PWM). The synth cycles through the 6 steps for each sample, and adds steps up to 32 to come up to 5 bits of PWM resolution. The PWM resolution is increased through dithering; at 48 kHz sample rate, it can be considered to be 10 bits. The resolution should further increase for lower frequencies.

The octave of the cutoff frequency is used to determine the shift amount. Depending on the position within the octave, the shift amount is decreased by one more or less often, to average the right amplification.

The volume is adjusted by tying the filter update that feeds the input signal into the filter to its own frequency. In the same way, the damping as adjusted by having a separate frequency for the filter's damping step.

The a dither signal is formed by bit reversing the `oct_counter` counter (which is used for the octave divider), and added to the output signal before rounding off to 5 bits for the PWM output.

For more details, see `README.md` in the project repository.

How to test

The synth is controlled by writing to its configuration registers:

- Keep the write strobe low when not writing.
- Set the 4 bit write address, and an 8 bit data value.
- While keeping the address and data stable, bring the write strobe high and then low again.
  - The write address and data are sampled at 2-10 cycles after the rising edge of the write strobe.

The output comes in two forms:

- As a Pulse Width Modulated (PWM) signal.
- As an 8 bit value on the 8 output pins, that can be reconstructed using a resistor ladder.

The PWM signal should be simpler to use, but be sure to reduce the voltage with a resistive divider or similar before connecting it to an audio device. **Note:Make sure that you know what you are doing when connecting an audio device to the output. Don't apply more than 1 V between the terminals of an audio**

plug that is connected to line in or similar. 3.3 V direct from the chip might damage your audio device.

Most control registers consume 16 bits of address space each. The memory map is laid out as follows: (one 16 bit word per line)

```
offset |  high byte |     low byte |
-------|------------|--------------|
 0     |        osc1_period        |
 2     |        osc2_period        |
 4     |       cutoff_period       |
 6     |        damp_period        |
 8     |         vol_period        |
10     | osc2_sweep |   osc1_sweep |
12     | damp_sweep | cutoff_sweep |
14     |        cfg |    vol_sweep |
```

The registers are initialized to all ones at reset, which turns off all oscillators. The frequency registers are in a kind of floating point format:

- Oscillator periods are 13 bits: 4 bits exponent + 9 bits mantissa
- Cutoff, damping, and volume periods are 9 bits: 4 bits octave + 5 bits period
- Sweep periods are 8 bits signed: 1 bit sign + 4 bits octave + 3 bits mantissa

Increasing the exponent by one doubles the period, and goes down one octave. An exponent of 15 turns off the oscillator. The volume depends on the ratio between the cutoff and volume periods (not their float representations). The damping depends on the ratio between the cutoff and damping periods (not their float representations). As the damping period gets longer than the cutoff period, resonance increases around the cutoff frequency. If damping is low and/or volume is high, the filter will begin to saturate (which is sometimes a desirable effect).

Each sweep will increase or decrease the corresponding period.

The `cfg` register contains additional settings:

- Bits 0-1: Waveform for oscillator 0: 0 = pulse, 1 = square, 2 = noise, 3 = saw
- Bits 2-3: Waveform for oscillator 1
- Bits 4-5: Unused
- Bits 6-7: Filter mode for oscillator 1 and 2 respectively, 0 = 1st order falloff, 1 = 2nd order falloff

For more details, see `README.md` in the project repository.

Pinout

#	Input	Output	Bidirectional
0	write data bit 0	sample bit 0	write address bit 0
1	write data bit 1	sample bit 1	write address bit 1
2	write data bit 2	sample bit 2	write address bit 2
3	write data bit 3	sample bit 3	write address bit 3
4	write data bit 4	sample bit 4	unused
5	write data bit 5	sample bit 5	unused
6	write data bit 6	sample bit 6	PWM output
7	write data bit 7	sample bit 7	write strobe

Tiny Game of Life [264]

- Author: Petros Emmanouilidis
- Description: Simulates cellular automaton Conway's Game of Life on an 8x8 grid using shift registers.
- GitHub repository
- HDL project
- Mux address: 264
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

The circuit employs 2 distinct shift registers to run the simulation: A Load Shift Register stores the initial state of the grid based on the user's input. Once the simulation commences, all cells in the Load Shift Register are updated and copied in parallel inside the Update Shift Register. This update step occurs within a single clock cycle. After updating the grid, the circuit outputs each new value sequentially before proceeding to the next state of the game. The output stage of the game lasts 64 clock cycles (one clock cycle for each cell in the grid) and involves pushing the updated cells from the Update Shift Register back to the Load Shift Register. Once all updated values have trickled into the Load Shift Register, the circuit returns to its update phase, restarting the cycle of update and output. After the simulation commences, the circuit will oscillate between updating and outputting indefinitely (unless reset) without any further user input.

Inputting Values:

Before starting the game, the user can sequentially load the grid's values into the circuit, one cell at a time. Cells are organized in row major order and the circuit can, at any time, hold 64 cells. If the user attempts to load more than 64 values, the oldest ones are pushed off the grid. The value of any inputted cell is specified using the 0th input line ui_in[0]. Loading a single cell into the circuit takes 1 clock cycle, meaning that inputting the entire table into the register takes 64 clock cycles.

Starting the Game:

To commence the simulation, the user must assert the 1st input line ui_in1. Upon doing so, the circuit stops receiving further user inputs and starts playing the game. The value present in ui_in[0] while ui_in1 is asserted is not loaded into the table.

Output Encoding:

During the output phase, the circuit drives all 8 output lines. The 0th bit uo_out[0] encodes the value of the currently displayed cell. Bits 1 to 8, uo_out[7:1] encode the location of the cell in the table. The location can take values 1 to 64 inclusive and is in row major order (meaning that 1 corresponds to the cell in the top left corner and 64 corresponds to the cell in the bottom right corner). During update, output bits uo_out[7:1] are set to 0 and the data output at uo_out[0] is invalid. During input, all output lines are invalid.

How to test

Load values through ui_in[0] (one cell value per clock cycle) and start the game by asserting ui_in1. Make sure that ui_in1 starts out disasserted; otherwise, the game will commence without any values loaded into the table, and the circuit will be inaccessible unless reset.

In general, pray it works.

Pinout

#	Input	Output	Bidirectional
0	Data Input Line	Data Output Line	none
1	Start Game	0th bit of cell location	none
2	none	1st bit of cell location	none
3	none	2nd bit of cell location	none
4	none	3rd bit of cell location	none
5	none	4th bit of cell location	none
6	none	5th bit of cell location	none
7	none	6th bit of cell location	none

Stack Machine [266]

- Author: Mingkai Chen
- Description: 8-bit stack machine
- GitHub repository
- HDL project
- Mux address: 266
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Simple 8-bit stack machine

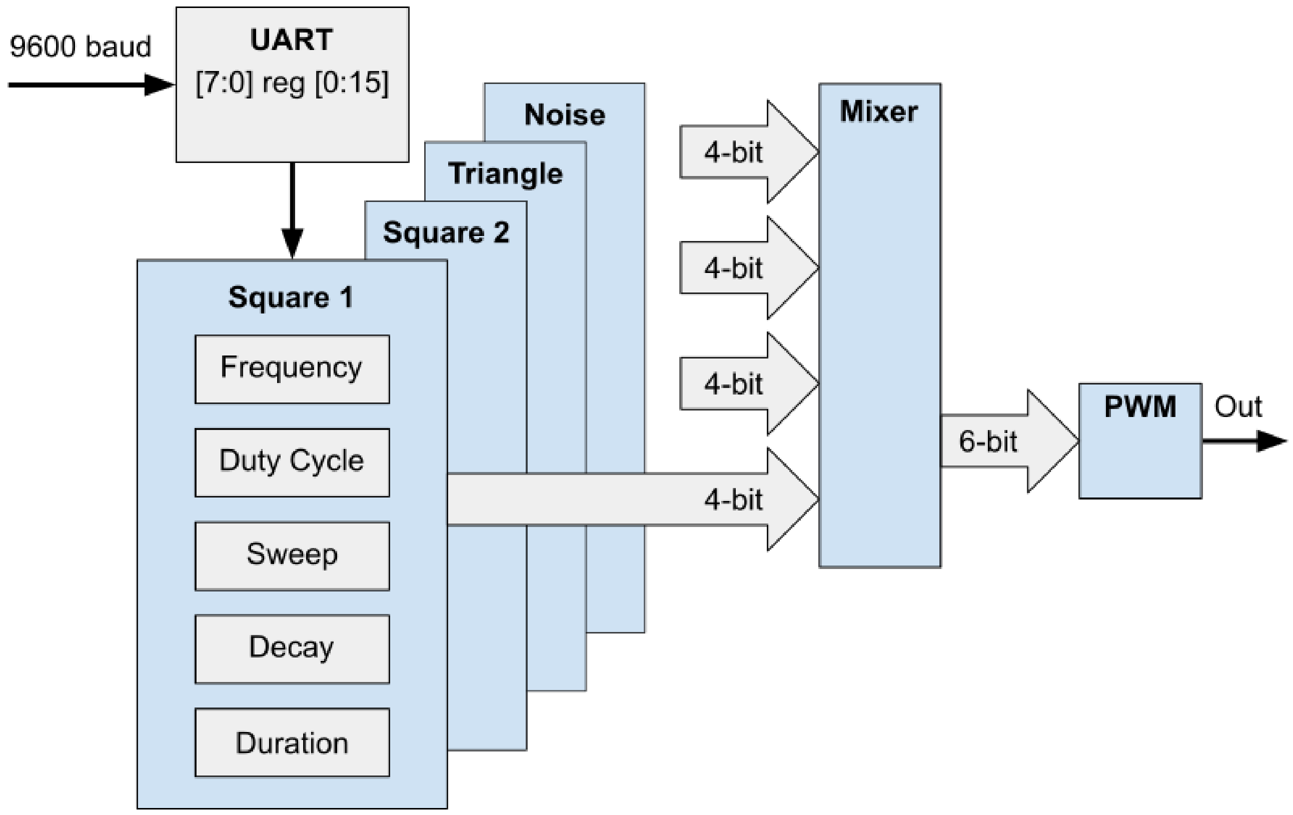
How to test

Test in hardware or with simulation

Pinout

#	Input	Output	Bidirectional
0	Data	Data	Data
1	n/a	n/a	n/a
2	n/a	n/a	n/a
3	n/a	n/a	n/a
4	n/a	n/a	n/a
5	n/a	n/a	n/a
6	n/a	n/a	n/a
7	n/a	n/a	n/a

ChipTune [268]



- Author: Wallace Everest
- Description: Vintage 8-bit sound generator
- GitHub repository
- HDL project
- Mux address: 268
- Extra docs
- Clock: 1789773 Hz
- External hardware: Computer COM port

How it works

ChipTune implements an 8-bit Programmable Sound Generator (PSG). Input is from a serial UART interface. Output is PWM audio.

Overview This project replicates the Audio Processing Unit (APU) of vintage video games.

Statistics

- Tiles: 1x2
- DFF: 458
- Total Cells: 2760
- Utilization: 72%

TinyTapeout 5 Configuration TT04 devices from the eFabless Multi-Project Wafer (MPW) shuttle are delivered in QFN-64 packages, mounted on a daughterboard for breakout.

Based on data from:

- https://github.com/WallieEverest/tt04

Changes: 1.) Static registers addressed by the serial UART have been connected to the external reset, providing a known startup. 2.) Default values for REG signals have been removed, allowing 'X' propagation during simulation until the design reaches steady state.

How to test

The ChipTune project can be interfaced to a computer COM port (9600,n,8,1). An analog PWM filter and audio driver are needed for the test rig.

Pinout

#	Input	Output	Bidirectional
0	None	Blink	None
1	None	Link	None
2	RX	TX	None
3	None	PWM	None
4	None	Square1	None
5	None	Square2	None
6	None	Triangle	None
7	None	Noise	None

Game of Life 8x8 (siLife) [270]

- Author: Uri Shaked
- Description: Silicon implementation of Conway's Game of Life
- GitHub repository
- HDL project
- Mux address: 270
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

It is a silicon implementation of Conway's Game of Life. The game is played on a 8x8 grid, and the rules are as follows:

- Any live cell with fewer than two live neighbours dies, as if by underpopulation.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overpopulation.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

How to test

Load initial grid row by row. Each row is loaded by selecting the row number (using the row_sel[2:0] inputs), setting the cell_in[7:0] inputs to the desired state, and pulsing the wr_en input.

Once the grid is loaded, set the en input to 1 to start the game. The game will advance one step in each clock cycle. To pause the game, set the en input to 0.

To view the current state of the grid, set the row_sel[2:0] inputs to the desired row number, and read the cell_out[7:0] outputs.

Pinout

#	Input	Output	Bidirectional
0	row_sel[0]	cell_out[0]	cell_in[0]
1	row_sel1	cell_out1	cell_in1
2	rol_sel2	cell_out2	cell_in2

#	Input	Output	Bidirectional
3	none	cell_out[3]	cell_in[3]
4	none	cell_out[4]	cell_in[4]
5	none	cell_out[5]	cell_in[5]
6	en	cell_out[6]	cell_in[6]
7	wr_en	cell_out[7]	cell_in[7]

TT05 Analog Testmacro (Ringo, DAC) [271]

- Author: Harald Pretl and Jakob Ratschenberger
- Description: For future analog enablement of TinyTapeout we designed a few simple analog blocks for testing the flow. The first block is a ca. 500kHz ring oscillator outputting a square-wave signal. The second block is a 3bit R-2R DAC outputting a programmable dc voltage. Both analog output signals can be gated or shorted using integrated transmission gates. To add a further level of madness, we have placed and routed this analog macro using an experimental automatic analog PnR tool, currently under development by the authors.
- GitHub repository
- HDL project
- Mux address: 271
- Extra docs
- Clock: 0 Hz
- External hardware: scope, multimeter

How it works

A ring oscillator (ca. 500kHz) produces a square-wave signal available at UA[0]. A 3-bit R-2R DAC produces a dc voltage availabel at UA1.

How to test

Enable the respective blocks, and enable the transmission gates to connect the block outputs to UA[0] and UA1, respectively. The DAC voltage can be changed by setting the digital inputs accordingly.

Pinout

#	Input	Output	Bidirectio
0	dac_in[0]	UA[0]: Ringo output (when TG enabled)	none
1	dac_in1	UA1: DAC output (when TG enabled)	none
2	dac_in2	none	none
3	Enable TG for DAC output to UA1	none	none
4	none	none	none
5	Enable TG for ringo output to UA[0]	none	none
6	Enable ringo	none	none

#	Input	Output	Bidirectio
7	Short UA[0] and UA1 for testing	none	none

RBUART [290]

- Author: Brian 'redbeard' Harrington
- Description: A simple UART device
- GitHub repository
- Wokwi project
- Mux address: 290
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

This implements a low baud rate UART which should output ASCII characters "Red."

How to test

To test the project, connect the TX and RX pins to the TX and RX pins on your computer. You should see the characters being printed on your computer.

Pinout

#	Input	Output	Bidirectional
0	N/A	segment a	none
1	Bit 0	segment b	none
2	Bit 1	segment c	none
3	Bit 2	segment d	none
4	Bit 3	segment e	none
5	Bit 4	segment f	none
6	Bit 5	segment g	none
7	Bit 6	dot	none

8-bit Floating-Point Adder [292]

- Author: Matt Ngaw
- Description: A floating-point adder following the FP8 E5M2 standard.
- GitHub repository
- HDL project
- Mux address: 292
- Extra docs
- Clock: Hz
- External hardware:

How it works

The circuit combinationally computes the floating-point sum.

How to test

Hold two 8-bit inputs on the input and bi-directional pins, and the floating-point sum comes out of the output pins.

Pinout

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

6 bit Counter and Piano Music created by Chip Inventor [294]

- Author: Matheus
- Description: Chip Inventor
- GitHub repository
- HDL project
- Mux address: 294
- Extra docs
- Clock: 27000000 Hz
- External hardware:

How it works

There are two diagrams created by the Chip Inventor platform, whereas by using blocks, you can create your own semiconductor design. The piano tune is one diagram. A song-throwing buzzer can be configured with one button and a buzzer. A 6-bit counter diagram is the other. Chip Inventor website: https://chipinventor.com

How to test

Using a buzzer, it's possible to listen to a music note. Connecting 4 LEDs in pull_up makes it possible to see the binary value.

Pinout

#	Input	Output	Bidirectional
0	btn1	buzzer	none
1	none	l1	none
2	none	l2	none
3	none	l3	none
4	none	l4	none
5	none	led0	none
6	none	led1	none
7	none	none	none

4 Bit Pipelined Multiplier [296]

- Author: Aldo
- Description: A Pipelined Booth Multiplier
- GitHub repository
- HDL project
- Mux address: 296
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Not operation inteded, just for learning purposes

How to test

Not operation inteded, just for learning purposes

Pinout

#	Input	Output	Bidirectional
0	none	none	none
1	none	none	none
2	none	none	none
3	none	none	none
4	none	none	none
5	none	none	none
6	none	none	none
7	none	none	none

2-Bit ALU + Dice [298]

- Author: Andrew Nam
- Description: This is an extremely professional design that Steve Jobs approves. It consists of a 2-bit ALU and an impressive dice. Can translate binary code into single digit display.
- GitHub repository
- Wokwi project
- Mux address: 298
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

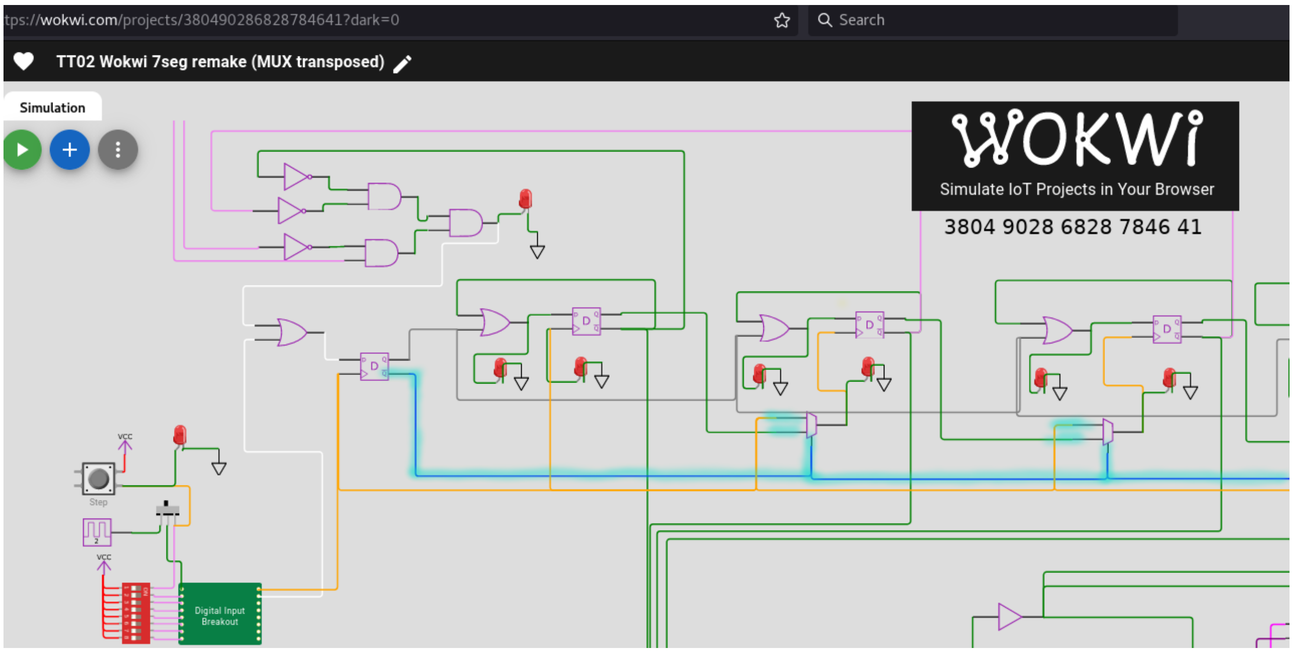
Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	{'IN0': 'Input A0'}	{'segment a': 'Normal digit display'}	Not used
1	{'IN1': 'Input A1'}	{'segment b': 'Normal digit display'}	Not used
2	{'IN2': 'Not used'}	{'segment c': 'Normal digit display'}	Not used
3	{'IN3': 'Selection bit (0,0)–> Addition, (0,1)–> Subtraction, (1,0)–> Logic AND, (1,1)–> Logic OR'}	{'segment d': 'Normal digit display'}	Not used

#	Input	Output	Bidirectional
4	{‘IN4’: ‘Selection bit’}	{‘segment e’: ‘Normal digit display’}	Not used
5	{‘IN5’: ‘Selection bit (0)–> ALU, (1)–> Dice’}	{‘segment f’: ‘Normal digit display’}	Not used
6	{‘IN6’: ‘Input B0’}	{‘segment g’: ‘Normal digit display’}	Not used
7	{‘IN7’: ‘Input B1’}	dot	Not used

TT02 Wokwi 7seg remake [300]


tps://wokwi.com/projects/380490286828784641?dark=0
Search
TT02 Wokwi 7seg remake (MUX transposed)
Simulation
WOKWi
Simulate IoT Projects in Your Browser
3804 9028 6828 7846 41
Step
Digital Input Breakout

- Author: Darryl Miles
- Description: TT02 Wokwi 7seg remake (MUX transposed)
- GitHub repository
- Wokwi project
- Mux address: 300
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

This is a remake of the Matt Venn's original TT02 7seg wokwi project.

This version inverted the MUX SEL lines at the reset, so the transition is on the opposite edge.

This project wokwi link:, https://wokwi.com/projects/380490286828784641

The original project wokwi link: https://wokwi.com/projects/380490286828784641

How to test

Select project and manually clock to see incrementing 7SEG output.

Pinout

#	Input	Output	Bidirectional
0	none	segment a	none
1	none	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

ping pong asic [302]

- Author: Timonas Juonys
- Description: Hardware implemented ping pong for two players on a 16x24 led matrix as a display
- GitHub repository
- Wokwi project
- Mux address: 302
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works The game uses a up/down counter for x/y for each player plus the ball. Button inputs are stored in input register, so bouncing should not be an issue. Rest pin (active low) should be pulsed low at every start up to clear all the clock registers. If this is not done, the multiple internal clock divider flip flops might start out wrong, and that could mess up other functions. The ball gets updated with every ball_en pulse, while the padles are updated with the padles_en pulse. These canot happen at the same time because then the could jump past the padles. The comparator logic is asyncronous, and it will reverse the balls direction if it registers a collision.

Since the led matrix can only light 1 collum or 1 row at any one time, the 3 objects to be lit(padle1, padle2, and the ball) have to be lit for a period of time before the next object is lit. this is achieved by the inner multiplexer which is driven byt a mod 3 counter which is driven byt the multiplexer clock. The chip outputs are the outputs of this multiplexer (some logic is done after the multiplexer but it is irrelevant). The x pixels (horizontal axis) is not decoded internally, and thus have to be decoded externally. They are connected as horz0 to horz4 pins. Even though there are 5 bits, the led display is only 24leds wide, so only a 5 to 24 decodes is necesary. The y pixels are decoded internaly since they need some processing done on them because they have to light multiple leds if a padle is to be lit contra one led for the ball. Thats why the y pixels get decoded and placed in a piso shift register which is controlled by D0 and D1. Shifting out these bits needs to be carefully timed with the multiplexer clock since the mux_clk is the one who decites which outputs (padle1, padle2, or ball) are in the shift register in the first place. Muxes for horixzontal pins and vertical pins are driven byt he same celect lines, so both x and y bits represent the same object at anny given time.

player1 and player 2 points pins are meant to go into a decade cointer driveing a 7 segment display. If a pause is wanted after a point os scored, these two can be monitored and the clk_in can be stoped to pause the game. The position registers are reset internally, so its not necessary to reset the whole chip at every point score.

maybe important: pixel[0,0] is in the bootm left corner

How to test

Explain how to test your project easiest way to test some functionality would be to hook up left right buttons for player 1, pull padels_en high, set the mux_clk low, and a clk on the clock line. As long as the board has been reset and the mux clk has not been active after that, the outputs will be of padle1. Then the horz0 to horz4 bits can be monitored. They should be still if no button has been presed, count up when the right button is pressed, and count down when teh left button is pressed. The ball_en should be puuled low to freze the ball. If the ball goes of the screen, padles1_x will reset to 0.

for full functionality the chip will need 16 bit sipo shift register, 5 to 24 decoder, 16*24 led matrix, 8 input buttons, bcd counters + 7 segment displays to display points and a timing unit to generate the necesary timing signals, padles_en, ball_en, mux_clk, shift_reg_en, shift_reg_clk and inp_reg_en

Pinout

#	Input	Output	Bidirectional
0	player1_up button	horz0 (lsd)	shift_reg_en (when D0 low, shift register ff mirror vertical pixels,when D0 high, then the shifting can start)
1	player1_down button	horz1 binary encoded position in the x direction	shift_reg_clk for vertical pixels
2	player1_left button	horz2	not used, pulled low internally
3	player1_right button	horz3	not used, pulled low internally
4	player2_up button	horz4	mux_clk - multiplexes between padle1, padle2, and ball, as the led matrix can only display one at a time

#	Input	Output	Bidirectional
5	player2_down button	points player1 pulses when player 1 gets a point	padles_en enable padle counters
6	player2_left button	points player2	ball_en enable ball counters
7	player2_right button	vertical pixels shift register out	inp_reg_en enables input register. this should happen when clk=1, ball_en=0, padles_en=0

A Boolean function based pseudo random number generator (PRNG) [320]

- Author: SEAL, CSE Department, IIT Kharagpur
- Description: Boolean function based pseudo random number generator implemented using finite field
- GitHub repository
- HDL project
- Mux address: 320
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Principle of operation of Boolean function based pseudo random number generator (PRNG)

This implementation of a PRNG contains linear mappings to and from the following blocks:

- one $GF(2^4)$ normal base,
- three instances of $GF(2^4)$ multipliers,
- one $GF(2^4)$ inverter, and
- one square scaler.
  The input and output strings of the PRNG are split into five and three shares, respectively. Our PRNG generates random values based on the five input bytes or variables. Instead of relying solely on a single seed or input, it takes several inputs thereby introducing more control over the randomness of the generated values. Thus, the multiple input bytes are used as seeds. The seeds are generated from external factors like time, user-provided data, and environmental conditions. Additionally, previous random values produced by our PRNG design can also be considered as a valid seed. This results in a more tailored or context-aware randomness, which finds its application in simulations, games, cryptography, or data generation. The operation of the Boolean function based PRNG can be classified into three phases, namely, Affine transformation (1^{st} phase), Finite field inversion (2^{nd} phase) and the combination of Finite field multiplication and inverse linear mapping (3^{rd} phase) as evident from the block diagram in Figure 1. The working procedure of these phases are discussed as follows:

First phase- Affine transformation

In the first phase, three shares are processed by the linear input mapping and afterwards fed into a multiplier. Similarly, a uniform reduction to two shares is fed into the square scaler.

$$(a, b, c)\backslash mapsto(a, b\backslash oplusc)\backslash\backslash\backslash\backslash(1)$$

The output of the multiplier is partially re-masked by 8 bits of randomness while the square scaler output is left as it is. We use fresh randomness at the end of the first phase to satisfy uniformity during the combination of the square scaler's and the multiplier's outputs. The result is saved in a register, P_1 as illustrated in the block diagram.

Second phase- Finite field inversion
In the second phase, the overall five shares are combined into four shares. Due to the previous remasking, this can be done uniformly as such:

$$(x, y, a, b, c)\backslash mapsto(x, y\backslash oplus(r_1\backslash oplusr_2), a\backslash oplus(b\backslash oplusr_1), c\backslash oplusr_2)\backslash\backslash\backslash\backslash(2)$$

In the above equation, x, y denote the square scaler output, while a, b, c denote the multiplier output. Note that a register needs to hold all five shares before recombination to prevent leakage. After recombination, the four shares are fed into the inverter and re-masked with 8 bits of randomness. A register stage named P_2, preventing glitches, follows this inverter.

Third phase- Finite field multiplication and inverse linear mapping
In the final stage, the re-masked outputs are reduced to three shares uniformly by the following function.

$$(a, b, c, d)\backslash mapsto(a\backslash oplus(b\backslash oplusr_3), c\backslash oplusr_4, d\backslash oplusr_3\backslash oplusr_4)\backslash\backslash\backslash\backslash(3)$$

Subsequently, these shares are fed into two multipliers. Finally, the inverse linear mapping follows. With this construction, it is enough to have three input shares to the generator since the multiplier block requires only three shares. At this stage, we again add a randomness after the inverter to break the dependency between the inputs of the multipliers in the third phase.
In general, we need to reduce the number of shares from five to four at the end of the first phase as the inverter in the second phase can process four input strings. Moreover, the multipliers in the final stage is capable of processing three shares of input thus enforcing the reduction of shares from four to three at the end of the second phase.
A working example is presented below for a better understanding:

In this example, the five input bytes are assigned values of $0x62, 0x04, 0x05, 0xf8$ and $0x95$, respectively. Preliminarily, the 'ena', an active high input signal is assigned a logic '0'. After the power on reset, the 'ena' is pulled up to logic '1', thus enabling the input data loading. The five input bytes are loaded sequentially into an input buffer which is 40 bits wide. As soon as the buffer is populated, the 'ena' signal is set to active low. This marks the end of the data loading procedure. After the data loading stage, the input values are then processed by linear mapping and three shares of data are produced which are $IN1 = 0xa8$, $IN2 = 0x81$ and $IN3 = 0x7e$. In the first and second phase, the remaining two input values of $R_0 = 0xf8$ and $R_1 = 0x95$ are utilized for introducing randomness.

The two inputs to the square scaler are $SQ_{IN1} = 0x2$ and $SQ_{IN2} = 0x0$. Our design acquires SQ_{IN1} by XOR-ing the first and last 4 bits of $IN1$, whereas SQ_{IN2} is acquired by XOR-ing the first and last 4 bits of $IN2\backslash oplusIN3$. The strings $IN1[7 : 4], IN2[7 : 4], IN3[7 : 4], IN1[3 : 0], IN2[3 : 0]$, and $IN3[3 : 0]$ are given as inputs to the multiplier and represented by $MUL_{IN1}, MUL_{IN2}, MUL_{IN3}, MUL_{IN4}, MUL_{IN5}$ and MUL_{IN6}, respectively. The signals, r_1 and r_2 are 4 bits wide, the values of which are obtained by slicing R_0. At the end of the first phase, five shares of data are produced along with the randomness, namely, $SQ_{OUT1}, SQ_{OUT2}, MUL_{OUT1}, MUL_{OUT2}, MUL_{OUT3}$ and r, respectively. The values of the individual signals are summarized below:
Inputs:
$r_1\backslash gets0xf$, $r_2\backslash gets0x8$,
$MUL_{IN1}\backslash gets0xa$, $MUL_{IN2}\backslash gets0x8$, $MUL_{IN3}\backslash gets0x7$, $MUL_{IN4}\backslash gets0x8$, $MUL_{IN5}\backslash gets0x1$, $MUL_{IN6}\backslash gets0xe$
Outputs:
$r\backslash gets0x7$,
$SQ_{OUT1}\backslash gets0x0$, $SQ_{OUT2}\backslash gets0x6$,
$MUL_{OUT1}\backslash gets0xf$, $MUL_{OUT2}\backslash gets0xe$, $MUL_{OUT3}\backslash gets0x8$
In the second phase, the corresponding input values, $INV_{IN1}, INV_{IN2}, INV_{IN3}$ and INV_{IN4} to the inverter are $0x0, 0x1, 0xe$ and $0x0$. The subsequent outputs, INV_{OUT2} and INV_{OUT3} are again combined with the random values r_3 and r_4, whereas the outputs, INV_{OUT1} and INV_{OUT4} are left as is. The values of r_3 and r_4 are acquired by slicing R_1. At the end of this phase, there are four shares of data along with the randomness bits, r. The remaining input and output values of this stage are summarized below:
Inputs:
$r_3\backslash gets0x9$, $r_4\backslash gets0x5$,
Outputs:
$r\backslash gets0xc$,
$INV_{OUT1}\backslash gets0x6$, $INV_{OUT2}\backslash gets0xb$, $INV_{OUT3}\backslash gets0x2$, $INV_{OUT4}\backslash gets0x0$

In the final stage, MUL_{IN1}, MUL_{IN2} and MUL_{IN3} are given as inputs to each of the multipliers (see Equation 3). The corresponding outputs of the two multipliers, $MUL_{OUT1}, MUL_{OUT2}, MUL_{OUT3}, MUL_{OUT4}, MUL_{OUT5}$ and MUL_{OUT6} are concatenated to form three strings of eight bits each and fed to the inverse linear mapping module. Thus, we acquire the final output bytes, $OUT1, OUT2$ and $OUT3$. These values are outlined below:
Inputs:

$MUL_{IN1}\backslash gets0x4$, $MUL_{IN2}\backslash gets0x7$, $MUL_{IN3}\backslash gets0xc$,
Outputs:
$MUL_{OUT1}, MUL_{OUT4}\backslash gets0xbb$, $MUL_{OUT2}, MUL_{OUT5}\backslash gets0xa6$, MUL_{OUT3}, MUL
$OUT1\backslash gets0x55$, $OUT2\backslash gets0xa2$, $OUT3\backslash gets0x0c$

How to test

After reset, the ena signal is set to logic '1'. This enables the device to load input values in multiple shares. After loading all the input shares, the ena signal is reset. After two clock cycles, the output ready (uio_out) is set to logic '1' and the multiple output shares are generated.

Pinout

#	Input	Output	Bidirectional
0	input bit	output bit	output ready
1	input bit	output bit	none
2	input bit	output bit	none
3	input bit	output bit	none
4	input bit	output bit	none
5	input bit	output bit	none
6	input bit	output bit	none
7	input bit	output bit	none

Digital Desk Clock [322]

- Author: Samuel Ellicott
- Description: Simple Digital Clock Project.
- GitHub repository
- HDL project
- Mux address: 322
- Extra docs
- Clock: 5000000 Hz
- External hardware: shift registers, 7-segment displays

How it works

Simple digital clock, displays hours, minutes, and seconds in either a 24h format. Since there are not enough output pins to directly drive a 6x 7-segment displays, the data is shifted out serially using an internal 8-bit shift register. The shift register drives 6-external 74xx596 shift registers to the displays. Clock and control signals (`serial_clk`, `serial_latch`) are also used to shift and latch the data into the external shift registers respectively. The time can be set using the `hours_set` and `minutes_set` inputs. If `set_fast` is high, then the the hours or minutes will be incremented at a rate of 5Hz, otherwise it will be set at a rate of 2Hz. Note that when setting either the minutes, rolling-over will not affect the hours setting. If both `hours_set` and `minutes_set` are presssed at the same time the seconds will be cleared to zero.

How to test

Connect serial output to a 6x 8-bit shift registers to display the output on 6x 7-segment displays

Pinout

#	Input	Output	Bidirectional
0	refclk	serial_data	none
1	use_refclk	serial_latch	none
2	fast_set	serial_clk	none
3	hours_set	none	none
4	minutes_set	none	none
5	none	none	none
6	none	none	none

#	Input	Output	Bidirectional
7	none	none	none

4-bit FIFO/LIFO [324]

- Author: Haozhe Zhu
- Description: A FIFO/LIFO memory
- GitHub repository
- HDL project
- Mux address: 324
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

This memory module can operate as both a First-In-First-Out (FIFO) and a Last-In-First-Out (LIFO) memory, which can be selected using the mode pin. It can store a maximum of 30 4-bit numbers, which is preserved upon mode switch. In addition, it is equipped with a 7-segment display that displays the current number of stored data entries (not more than nine). Should the stored entries surpass nine, the display will be deactivated, and an overflow flag will be triggered. If the memory is full, further write attempts has no effect on stored data. If the memory is empty, further read attempts will invalidate the output data and clear output valid flag. If no read operation has been performed after the most recent reset, the output is also invalid.

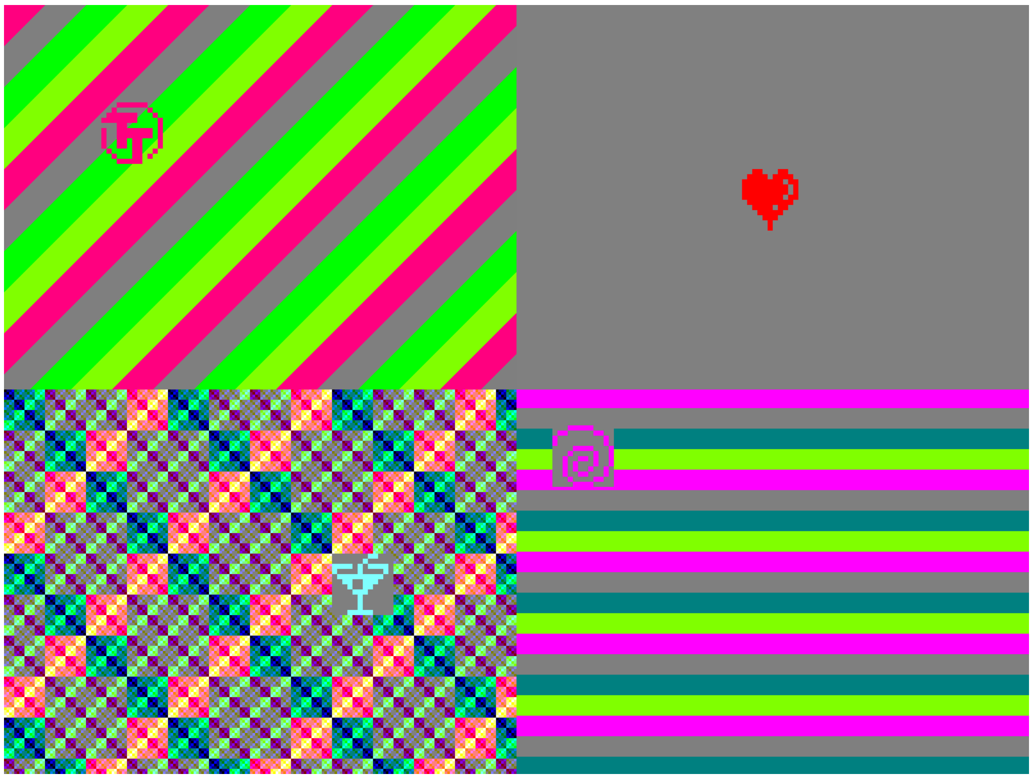
How to test

Load some data into the memory and then read them out.

Pinout

#	Input	Output	Bidirectional
0	Data In [0]	segment a	Data Out [0]
1	Data In 1	segment b	Data Out 1
2	Data In 2	segment c	Data Out 2
3	Data In [3]	segment d	Data Out [3]
4	Write Enable	segment e	Full Flag Out
5	Read Enable	segment f	Empty Flag Out
6	Mode (FIFO=0)	segment g	Output Valid Flag Out
7	Manual Clock	dot	Display Overflow Flag Out

One Sprite Pony [326]



- Author: Leo Moser
- Description: This SVGA design has exactly one trick up its sleeve: it displays a sprite!
- GitHub repository
- HDL project
- Mux address: 326
- Extra docs
- Clock: 40 MHz or 10 MHz Hz
- External hardware: Tiny VGA PMOD

How it works

A one-trick pony is someone or something that is good at doing only one thing. Accordingly, a one-sprite pony can display only one sprite, and that's exactly what this design does:

This Verilog design produces SVGA 800x600 60Hz output with a background and one sprite. Internally, the resolution is reduced to 100x75, thus one pixel of the sprite is actually 8x8 pixels. The design can operate at either a 40 MHz pixel clock or a 10 MHz pixel clock by setting a configuration bit.

The sprite is 12x12 pixel in size and is initialized at startup with a pixelated version of the Tiny Tapeout logo.

An SPI receiver accepts various commands, e.g. to replace the sprite data, change the colors or set the background.

How to test

Connect a Tiny VGA to the output Pmod connector. By default, you should see the TinyTapeout logo moving around the screen. By sending commands over SPI via the bidirectional Pmod you can change the sprite and the background, set the sprite position and much more. See the longer documentation for all commands.

Pinout

#	Input	Output	Bidirectional
0	none	R1	CS
1	none	G1	MOSI
2	none	B1	MISO
3	none	VS	SCK
4	none	R0	Vertical Pulse
5	none	G0	Horizontal Pulse
6	none	B0	none
7	none	HS	none

4 bit Sync Gray Code Counter [328]

- Author: EconomIC Engineers
- Description: Using a clock, a counter will rise using gray code binary values
- GitHub repository
- HDL project
- Mux address: 328
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

4 bit Syncronous Gray Code Counter

How to test

Connect the Input to a clock and Output to LEDs to demonstrate binary values changing

Pinout

#	Input	Output	Bidirectional
0	CLK	LED a	none
1	none	LED b	none
2	none	LED c	none
3	none	LED d	none
4	none	none	none
5	none	none	none
6	none	none	none
7	none	none	none

Clock and Random Number Gen [330]

- Author: Austin Lo
- Description: Divider up it 16bit input and a simple random number
- GitHub repository
- HDL project
- Mux address: 330
- Extra docs
- Clock: 200000000 Hz
- External hardware:

How it works

Uses a set of registers to divide the clock.

Both 8 bit input are used for the divider

Can divide up to 65565, target uses 20 Mhz as example.

How to test

After reset, the clock should be divided by the input a 20MHz input clock. Experiment by changing the inputs

Pinout

#	Input	Output	Bidirectional
0	division input bit 0	uo_out[0] clock output	division input bit 8
1	division input bit 1	Random Number Output	division input bit 9
2	division input bit 2	Random Number Output	division input bit 10
3	division input bit 3	Random Number Output	division input bit 11
4	division input bit 4	Random Number Output	division input bit 12
5	division input bit 5	Random Number Output	division input bit 13
6	division input bit 6	Random Number Output	division input bit 14
7	division input bit 7	Random Number Output	division input bit 15

TT05 Analog Test [332]

- Author: Matt Venn
- Description: Test voltage divider
- GitHub repository
- HDL project
- Mux address: 332
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

3 resistors are used to build a voltage divider connected between power & ground. Output taps are uo_out[2:0]

How to test

Enable the block and check the uo_outputs. Expected that only uo_out2 is high.

Pinout

#	Input	Output	Bidirectional
0	none	{'UA[0]': 'divider out 0'}	not used
1	n/a	{'UA1': 'divider out 1'}	n/a
2	n/a	{'UA2': 'divider out 2'}	n/a
3	n/a	n/a	n/a
4	n/a	n/a	n/a
5	n/a	n/a	n/a
6	n/a	n/a	n/a
7	n/a	n/a	n/a

VGA Experiments [334]

- Author: Tom Keddie
- Description: Simple Game
- GitHub repository
- HDL project
- Mux address: 334
- Extra docs
- Clock: 25175000 Hz
- External hardware: Digilent VGA PMOD or mole99 vga pmod

How it works

VGA game using paddles attached to input. No scoring, no diagonal ball movement

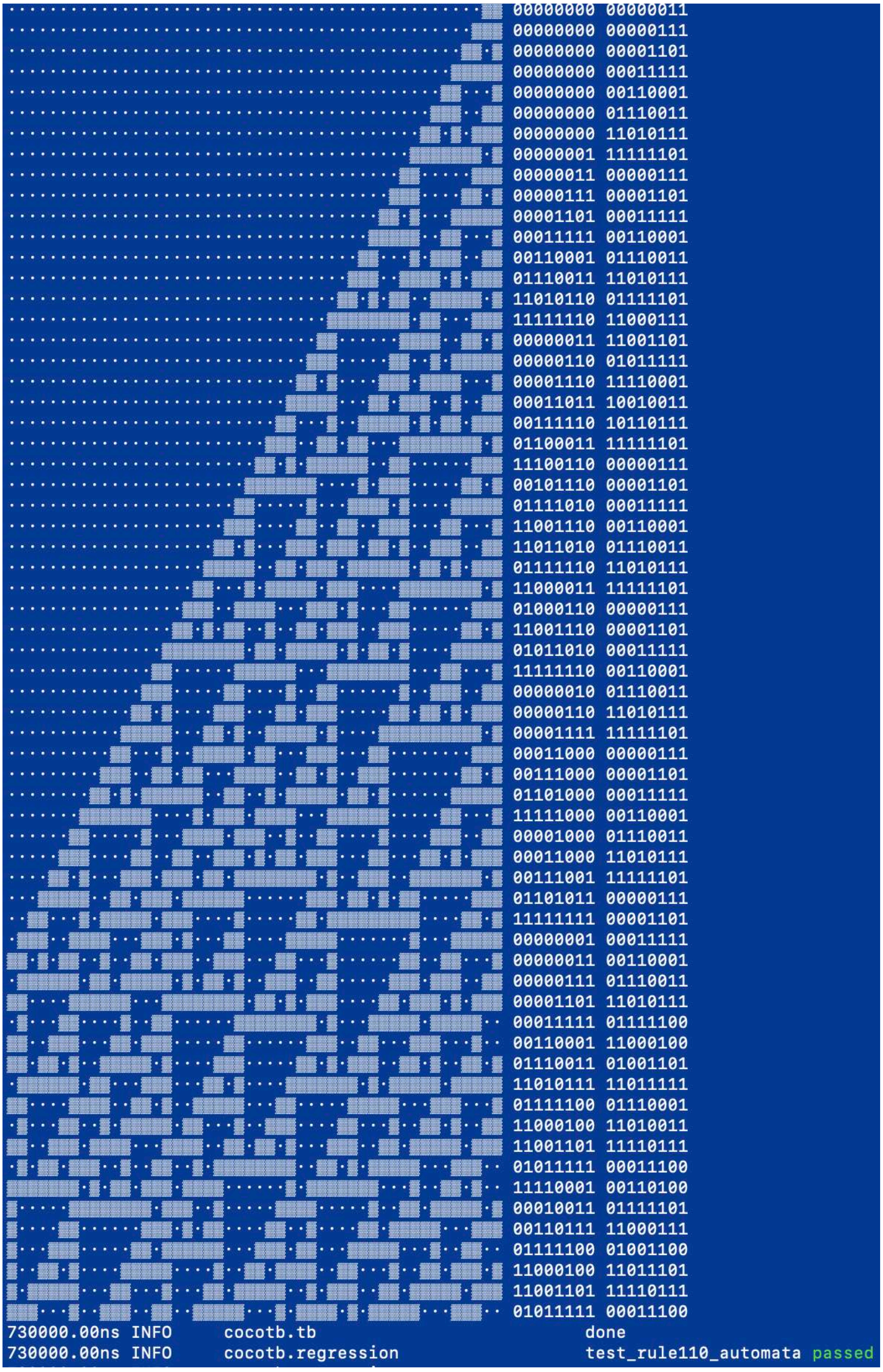
How to test

Attach VGA pmod and connect to monitor. Use the inputs to move the paddles

Pinout

#	Input	Output	Bidirectional
0	left paddle up	r1/r0 (mole99/digilent)	g0
1	left paddle down	g1/r1 (mole99/digilent)	g1
2	right paddle up	b1/r2 (mole99/digilent)	g2
3	right paddle down	vsync/r3 (mole99/digilent)	g3
4	none	r0/b0 (mole99/digilent)	hsync
5	none	g0/b1 (mole99/digilent)	vsync
6	none	b0/b2 (mole99/digilent)	tied low
7	pmod sel (high=mole99, low=digilent)	hsync/b3 (mole99/digilent)	tied low

Rule110 cell automata [384]



```
00000000 00000011
00000000 00000111
00000000 00001101
00000000 00011111
00000000 00110001
00000000 01110011
00000000 11010111
00000001 11111101
00000011 00000111
00000111 00001101
00001101 00011111
00011111 00110001
00110001 01110011
01110011 11010111
11010110 01111101
11111110 11000111
00000011 11001101
00000110 01011111
00001110 11110001
00011011 10010011
00111110 10110111
01100011 11111101
11100110 00000111
00101110 00001101
01111010 00011111
11001110 00110001
11011010 01110011
01111110 11010111
11000011 11111101
01000110 00000111
11001110 00001101
01011010 00011111
11111110 00110001
00000010 01110011
00000110 11010111
00001111 11111101
00011000 00000111
00111000 00001101
01101000 00011111
11111000 00110001
00001000 01110011
00011000 11010111
00111001 11111101
01101011 00000111
11111111 00001101
00000001 00011111
00000011 00110001
00000111 01110011
00001101 11010111
00011111 01111100
00110001 11000100
01110011 01001101
11010111 11011111
01111100 01110001
11000100 11010011
11001101 11110111
01011111 00011100
11110001 00110100
00010011 01111101
00110111 11000111
01111100 01001100
11000100 11011101
11001101 11110111
01011111 00011100
730000.00ns INFO     cocotb.tb                         done
730000.00ns INFO     cocotb.regression                 test_rule110_automata passed
```

- Author: ReJ aka Renaldas Zioma
- Description: Cellular automaton based on the Rule 110
- GitHub repository
- HDL project
- Mux address: 384
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

This design executes **over 200 cells** of an elementary cellular automaton **every cycle** applying Rule 110 to all of them **in parallel**. Roughly 115 cells with parallel read/write bus can be placed on 1x1 TinyTapeout tile. Without read/write bus, up to 240 cells fit on a 1x1 tile!

The edge of chaos - Rule 110 exhibits complex behavior on the boundary **between stability and chaos**. It could be explored for pseudo random number generator and data compression.

Gliders - periodic structures with complex behaviour, universal computation and self-reproduction can be implemented with Rule 110.

Turing complete - with a particular repeating background pattern Rule 110 is known to be Turing complete. This implies that, in principle, **any** calculation or computer program can be simulated using such automaton!

Definition of Rule 110

The following rule is applied to each triplet of the neighboring cells. Binary representation 01101110 of 110 defines the transformation pattern.

```
1. Current iteration of the automaton
            111  110  101  100  011  010  001  000
             |    |    |    |    |    |    |    |
             v    v    v    v    v    v    v    v
2. The next iteration of the automaton
            .0.  .1.  .1.  .0.  .1.  .1.  .1.  .0.
```

Interesting links for further reading

- Elemental Cellular Automaton Rule 110
- Gliders in Rule 110

- Compression-based investigation of the dynamical properties of cellular automata and other systems

How to test

After **RESET** all cells will be set to 0 except the rightmost that is going to be 1. Automaton will immediately start running. Automaton produce new state every cycle for all the cells in parallel. One hardware cycle is one iteration of the automaton. Automaton will run until **/HALT** pin is pulled low.

The following diagram shows 10 first iteration of the automaton after **RESET**.

```
                                                    X
                                                   XX
                                                  XXX
                                                 XX X
                                                XXXXX
                                               XX   X
                                              XXX  XX
                                             XX X XXX
                                            XXXXXXX X
  automaton state on the                   XX     XXX
10th iteration after RESET  ---->         XXX    XX X
```

To read automaton state

1) pull **/HALT** pin low and 2) set the cell block address pins.

Cells are read in 8 cell blocks and are addressed sequentially from right to left. Adress #0 represents the rightmost 8 cells. Adress #1 represents the cells from 16 to 9 on the rights and so forth.

```
  automaton state on the
10th iteration after RESET  ---->  XXX    XX X
00000000  ...  0000000000000000000011100001101
|      |              |       |       |       |
[adr#14]  ...  [addr#3][addr#2][addr#1][addr#0]
    cells are addressed in blocks of 8 bits
```

The state of the 8 cells in the block will appear on the **Output** pins once the cell block address is set.

```
Timing diagram

CLK   ___     ___     ___     ___     ___     ___           ___
   __/   `___/   `___/   `___/   `___/   `___/   `___ ... _/   `___
     |       |       |       |       |       |             |
     |       |       |       |       |       |             |


WRITE  ____                                                 _______
    X__HALT__________________________________________ ... _/

WRITE_______________  ______________  _______________
   _/ ADDR#0        `/ ADDR#1       `/ ADDR#2

READ OUTPUT_______         ________        ________
   ______/00001101`_______/00000111`______/00000000`_
              ^                ^
              |                |
       these are the expected values on
          the 10th cycle after RESET


      ____
      HALT  - /HALT, inverted halt automata
      ADDR# - cell block address bits 0..4
```

(Over)write automaton state

To write state of the cells, 1) pull **/HALT** pin low, 2) set the cell block address pins, 3) set the new desired cell state on the **Input** pins and 4) finally pull **/WE** pin low.

Cells are updated in 8 cell blocks and are addressed sequentially from right to left. Adress #0 represents the rightmost 8 cells. Adress #1 represents the cells from 16 to 9 on the rights and so forth.

```
Timing diagram

CLK   ___     ___     ___     ___     ___     ___           ___
   __/   `___/   `___/   `___/   `___/   `___/   `___ ... _/   `___
     |       |       |       |       |       |             |
     |       |       |       |       |       |             |
WRITE  ____                                                 _______
    X__HALT__________________________________________ ... _/
```

```
WRITE_______________  ______________  _______________
   _/ ADDR#0        `/ ADDR#1       `/ ADDR#2

WRITE INPUT_________  ______________  _____________
   __/ 00000111     `/ 11100110     `/ 11010111    `_

WRITE______  __    ________  __    ________  __    __ ... _________
           `_WE___/        `_WE___/        `_WE___/
                 wait 1 cycle    wait 1 cycle
```

```
         --
    ____ WE   - /WE, inverted write enable
    HALT      - /HALT, inverted halt automata
        ADDR# - cell block address bits 0..4
```

The following diagram shows 10 cycles of automaton after **/HALT** pulled back to high.

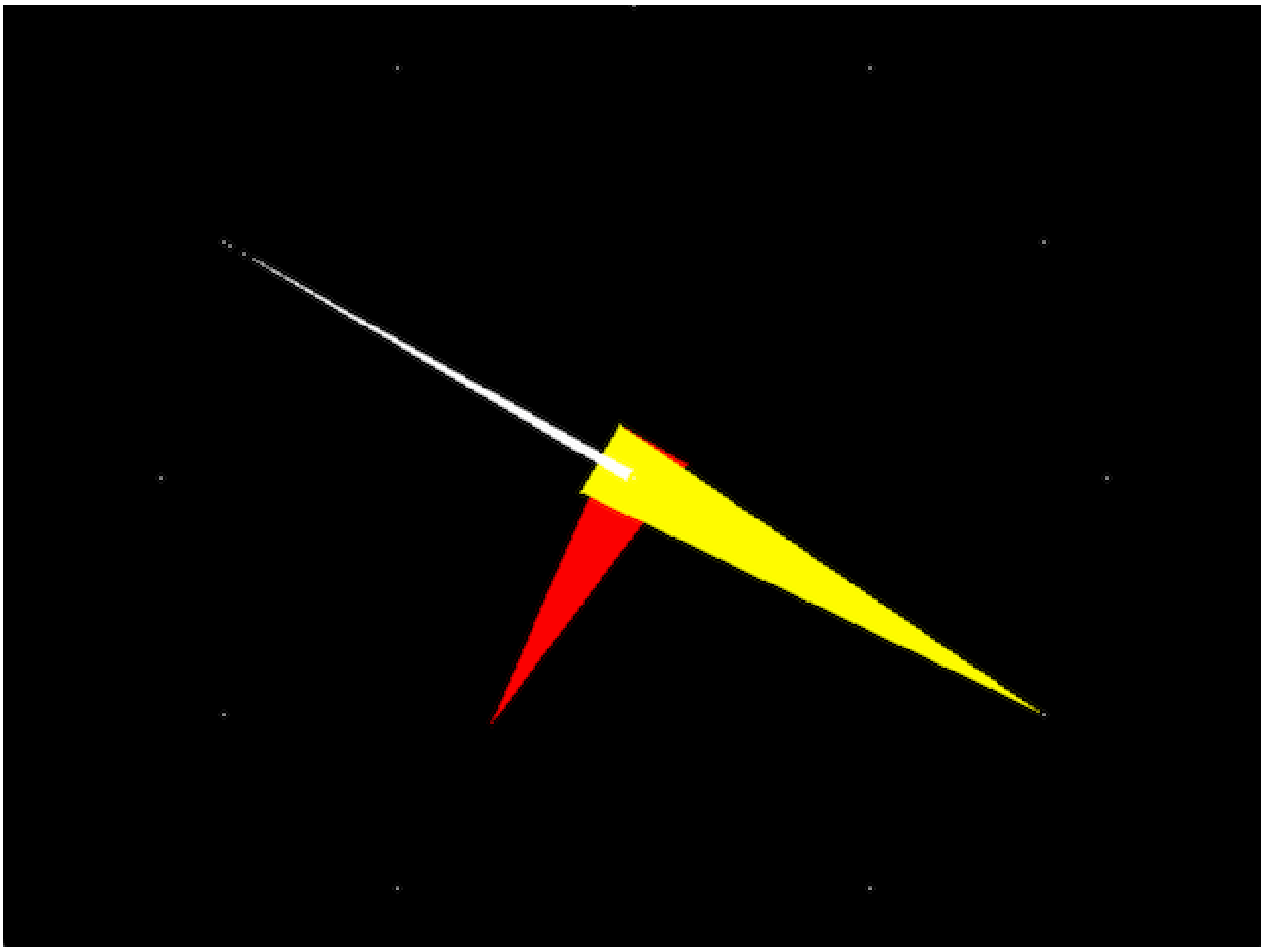
```
      [adr#14]  ...  [addr#3][addr#2][addr#1][addr#0]
      |      |              |       |       |       |
      00000000  ...  00000000110101111110011000000111
                             XX X XXXXXX  XX      XXX
                            XXXXXXX    X XXX     XX X
                           XX     X   XXXX X    XXXXX
                          XXX    XX  XX  XXX   XX   X
                         XX X   XXX XXX XX X  XXX  XX
                        XXXXX  XX XXX XXXXXX XX X XXX
                       XX   X XXXXX XXX    XXXXXXXX X
                      XXX  XXXX   XXX X   XX      XXX
                     XX X XX  X  XX XXX  XXX     XX X
10 cyles later ->   XXXXXXXX XX XXXXX X XX X    XXXXX
```

Pinout

#	Input	Output	Bidirectional
0	write cell 0 state	read cell 0 state	/WE, inverted write enable
1	write cell 1 state	read cell 1 state	/HALT, inverted halt automata
2	write cell 2 state	read cell 2 state	ADDR#, cell block address bit 0
3	write cell 3 state	read cell 3 state	ADDR#, cell block address bit 1

#	Input	Output	Bidirectional
4	write cell 4 state	read cell 4 state	ADDR#, cell block address bit 2
5	write cell 5 state	read cell 5 state	ADDR#, cell block address bit 3
6	write cell 6 state	read cell 6 state	ADDR#, cell block address bit 4
7	write cell 7 state	read cell 7 state	none

No Time for Squares [390]



- Author: Tommy Thorn
- Description: It's a 12-hour clock, drawn with triangles rendered by a race-the-beam triangle render
- GitHub repository
- HDL project
- Mux address: 390
- Extra docs
- Clock: 31500000 Hz
- External hardware: TinyVGA, 31.5 MHz clock, reset, hour & min buttons

How it works

Every frame the 640x480 VGA matrix is scanned, advancing the state of the intersecting lines of the three triangles. If the (x,y) coordinate of the "beam" lines on the positive side of each line, the beam is inside the triangle. Among the visible triangles, the highest priority triangle sets the color, else we default to a grey color. Twelve dots are also marked, to make it easier to read the clock.

The algorithm might be easily understood by examining the software model in Rust, in the `sw` directory.

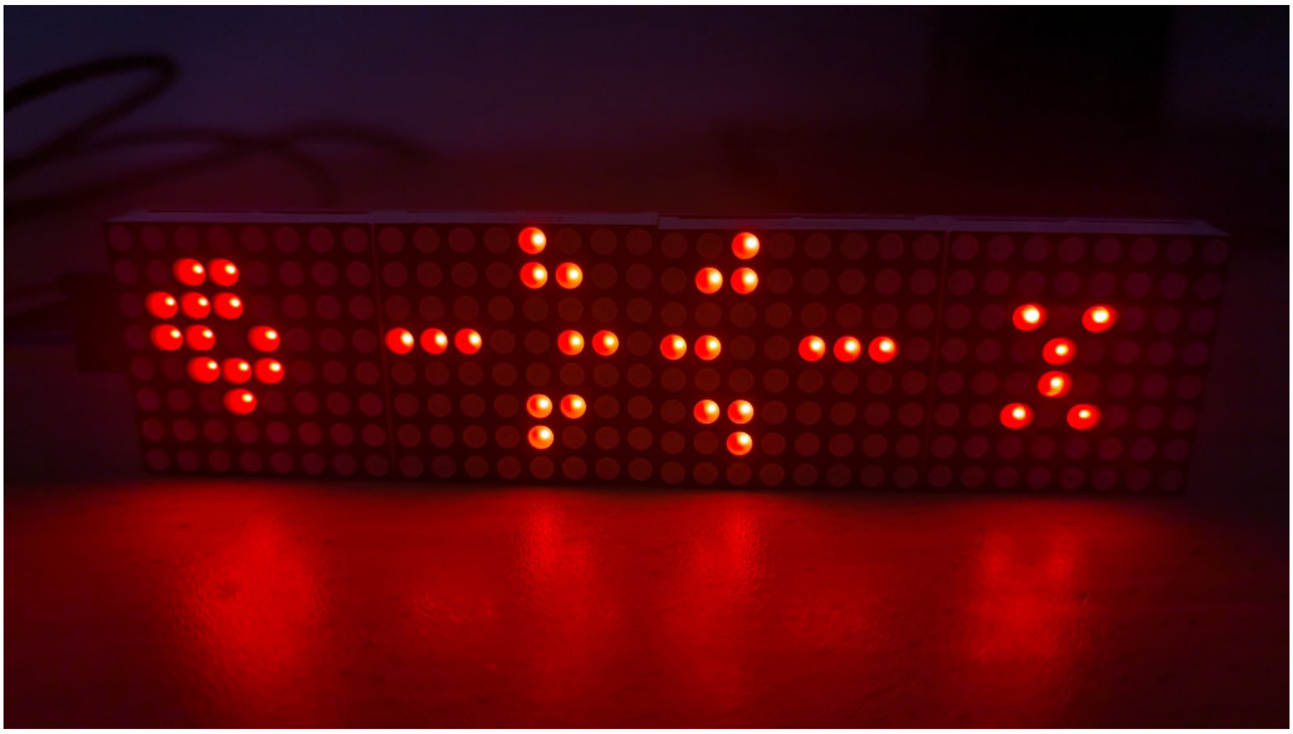
How to test

Hook up the Tiny VGA interface and connect a VGA monitor. Hope it works.

Pinout

#	Input	Output	Bidirectional
0	clock	R1	debug[7]
1	reset	G1	debug[6]
2	hour, advance hour	B1	debug[5]
3	minute, advance minute	vsync	debug[4]
4	unused	R0	debug[3]
5	unused	G0	debug[2]
6	debugsel[1]	B0	debug[1]
7	debugsel[0]	hsync	debug[0]

Game of Life 8x32 (siLife) [396]



- Author: Uri Shaked
- Description: Silicon implementation of Conway's Game of Life with LED Dot Matrix Output
- GitHub repository
- HDL project
- Mux address: 396
- Extra docs
- Clock: 10000000 Hz
- External hardware: MAX7219 LED Matrix (FC-16 module)

How it works

It is a silicon implementation of Conway's Game of Life. The game is played on a 8x32 grid, and the rules are as follows:

- Any live cell with fewer than two live neighbours dies, as if by underpopulation.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overpopulation.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

How to test

Demo mode: The demo mode loads a pre-defined game into the grid and advances it automatically. To enter the demo mode, `wr_en` high while reseting the design (`rst_n` low). Use the `pattern_sel` inputs to select the desired demo pattern. Set `en` to 1 to automatically advance one generation every 0.4 seconds (assuming a 10MHz clock). To pause the game, set `en` to 0.

Manual mode: Load the initial grid row by row. Each row is loaded by selecting the row number (using the `row_sel[4:0]` inputs), setting the `cell_in[7:0]` inputs to the desired state, and pulsing the `wr_en` input.

Once the grid is loaded, set the `en` input to 1 to start the game. The game will advance one step in each clock cycle. To pause the game, set the `en` input to 0.

To view the current state of the grid, set the `row_sel[4:0]` inputs to the desired row number, `max7219_en` to 0, and read the `cell_out[7:0]` outputs.

Alternatively, set `max7129_en` to 1 to display the grid on a MAX7219 LED Matrix (FC-16 module).

Pinout

#	Input	Output	Bidirectional
0	row_sel[0] / pattern_sel	cell_out[0] / max7129_cs	cell_in[0]
1	row_sel1	cell_out1 / max7129_clk	cell_in1
2	rol_sel2	cell_out2 / max7129_din	cell_in2
3	rol_sel[3]	cell_out[3]	cell_in[3]
4	rol_sel[4]	cell_out[4]	cell_in[4]
5	max7129_en	cell_out[5]	cell_in[5]
6	en	cell_out[6]	cell_in[6]
7	wr_en	cell_out[7]	cell_in[7]

TROS [398]

- Author: Gerrit Grutzeck
- Description: Three different ring oscillator, with different temperature dependence
- GitHub repository
- HDL project
- Mux address: 398
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

This design implements three different ring oscillators. The first one is a basic NAND based oscillator. The second one adds additional NAND gates to the outputs of the stages of the oscillator to increase the capacitve loading. The last one uses the tri-state inverts with a sub-threshold tri-state enable.

For measuring the frequencies each oscillator is driving a counter. This counters are latched with the latch counter input. With the input transfer counter the currently selected counter (counter select bits) is transfered via the serial data stream. The transfer is driven by the clock of the design. As encoding a manchester encoding is used.

Furthermore, a divided version of the clock of each oscillator is outputted. The division can be configured with the frequency selection bits.

How to test

TODO

Pinout

#	Input	Output	Bidirectional
0	latch counter	not used	not used
1	counter reset	not used	not used
2	transfer counter	not used	not used
3	counter select bit 0	not used	not used
4	counter select bit 1	serial data stream	not used

#	Input	Output	Bidirectional
5	select latch counter (sync/async)	divided clock of oscillator 0	not used
6	frequency divider select bit 0	divided clock of oscillator 1	not used
7	frequency divider select bit 1	divided clock of oscillator 2	not used

ChatGPT designed Spiking Neural Network [450]

- Author: Michael Tomlinson, Joe Lie, ChatGPT-4, Andreas Andreou - mtomlin5@jh.edu
- Description: SPI Programmable spiking neural network with 6 LIF neurons (3 input - 3 output) with fully programmable weights (8-bit)
- GitHub repository
- HDL project
- Mux address: 450
- Extra docs
- Clock: 50000000 Hz
- External hardware: fpga

How it works

This project implements 6 programmable digital LIF neurons. The neurons are arranged in 2 layers (3 in each). Spikes_in directly maps to the inputs of the first layer neurons. When an input spike is received, it is first multiplied by an 8 bit weight, programmable from the spi interface, 1 per input neuron. This 8 bit value is then added to the membrane potential of the respective neuron.

When the first layer neurons activate, its pulse is routed to each of the 3 neurons in the next layer. There are 9 programmable weights describing the connectivity between the first and second layers. Output spikes from the 2nd layer drive spikes_out.

How to test

After reset, program the neuron threshold, leak rate, and refractory period. Additionally program the first and 2nd layer weights (all programming is done over spi). Once programmed activate spikes_in to represent input data, track spikes_out synchronously (1 clock cycle pulses).

Pinout

#	Input	Output	Bidirectional
0	ui_in[7] - unused	uo_out[7] - unused	GPIO pins are wired to outputs and driven high (unused by the design).
1	ui_in[6] - unused	uo_out[6] - unused	unused

#	Input	Output	Bidirectional
2	ui_in[5] - spikes_in2	uo_out[5] - unused	unused
3	ui_in[4] - spikes_in1	uo_out[4] - unused	unused
4	ui_in[3] - spikes_in[0]	uo_out[3] - cipo	unused
5	ui_in2 - copi	uo_out2 - spikes_out2	unused
6	ui_in1 - cs_n	uo_out1 - spikes_out1	unused
7	ui_in[0] - sclk	uo_out[0] - spikes_out[0]	unused

Karplus-Strong String Synthesis [454]

- Author: Chinmay Patil
- Description: Plucked string sound synthesizer
- GitHub repository
- HDL project
- Mux address: 454
- Extra docs
- Clock: 256000 Hz
- External hardware:

How it works

This is simplified implementation of Karplus-Strong (KS) string synthesis based on papers, Digital Synthesis of Plucked-String and Drum Timbres and Extensions of the Karplus-Strong Plucked-String Algorithm.

A register map controls and configures the KS synthesis module. This register map is accessed through a SPI interface. Synthesized sound samples can be accessed through the I2S transmitter interface.

SPI Frame

SPI Mode: CPOL = 0, CPHA = 1

The 16-bit SPI frame is defined as,

Read=1/Write=0	Address[6:0]	Data[7:0]

Register Map

The Register Map has 16 Registers of 8-bits each.

Complete register map is described in the repository at https://github.com/pyamnihc/tt04-um-ks-pyamnihc.

I2S Transmitter

The 8-bit signed sound samples can be read out at `f_sck = 256 kHz` through this interface.

How to test

Connect a clock with frequency `f_clk = 256 kHz` and apply a reset cycle to initialize the design, this sets the audio sample rate at `fs = 16 kHz`. Use the spi register map or the ui_in to futher configure the design. The synthesized samples are sent continuously on the I2S transmitter interface.

Pinout

#	Input	Output	Bidirectional
0	~rst_n_prbs_15, ~rst_n_prbs_7	segment a	sck_i
1	load_prbs_15, load_prbs_7	segment b	sdi_i
2	freeze_prbs_15	segment c	sdo_o
3	freeze_prbs_7	segment d	cs_ni
4	i2s_noise_sel	segment e	i2s_sck_o
5	~rst_n_ks_string	segment f	i2s_ws_o
6	pluck	segment g	i2s_sd_o
7	NOT CONNECTED	dot	prbs_15

VGA Dino Game [458]

- Author: Anish Singhani
- Description: An endless-runner game implemented on a VGA monitor
- GitHub repository
- HDL project
- Mux address: 458
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Connect to VGA and some buttons and play!

How to test

Connect to a VGA monitor

Pinout

#	Input	Output	Bidirectional
0	jump btn	VGA R0	none
1	halt btn	VGA G0	none
2	debug btn	VGA B0	none
3	mode btn	VGA vsync	none
4	none	VGA R1	none
5	none	VGA G1	none
6	none	VGA B1	none
7	none	VGA hsync	none

Dual Compute Unit [460]

- Author: Himanshu Yadav
- Description: ComputeUnit implementation
- GitHub repository
- HDL project
- Mux address: 460
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

The project has two compute unit which perform some ALU operations based on input instructions and the final output is xor of compute unit output.

How to test

Reset needs to be 0 to make design go to reset mode and then set reset to 1 and ena to 1 to shift the design to functional mode. I tested my design on EDA playground by creating testbench there. Testbench and design files are there in test/ directory.

Pinout

#	Input	Output	Bidirectional
0	none	segment a	none
1	none	segment b	none
2	none	segment c	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

Collatz conjecture brute-forcer [462]

- Author: Vytautas Šaltenis
- Description: Runs a Collatz sequence calculation for a given number, outputs the number of steps it took to reach 1 (a.k.a. orbit length) and the upper 16 bits of the highest number of the sequence.
- GitHub repository
- HDL project
- Mux address: 462
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

The module takes a (large) integer number `N` as an input and computes the Collatz sequence until it reaches 1. When it does, it allows reading back two numbers:

1) The orbit length (i.e. the number of steps it took to reach 1)
2) The highest recorded value of the upper 16 bits of the 144-bit internal iterator

The latter number is an indicator for good candidates for computing path records. The non-zero upper bits indicate that the highest iterator value `Mx(N)` is in the range of the previous path records and should be recomputed in the full offline. (Holding on to the entire 144 bits of `Mx(N)` number would be more obvious, but this almost doubles the footprint of the design, hence, this optimisation).

How to test

The module can be in 2 states: IO and COMPUTE. After reset, the chip will be in IO mode. Since the input is intended to be much larger that the available pins, the input number is uploaded one byte at a time, increasing the address of where in the internal 144-bit-wide register that byte should be stored.

Same for reading the output, except that the output numbers are limited to 16-bits each, so it takes much fewer operations to read them.

The full loop of computations works like this:

1) Set input (see below)
2) Pull `start compute` pin to high. The chip will start computations and will pull `compute busy indicator` pin to high
3) Keep reading `compute busy indicator` pin until it gets low again
4) Read the output (see below)

Writing input:

1) Set `write enable` pin to low
2) Wait at least one cycle
3) Expose your input byte to `input0-7`
4) Expose the target address for that byte to `address0-4`
5) Wait at least one cycle
6) Set `write enable` pin to high

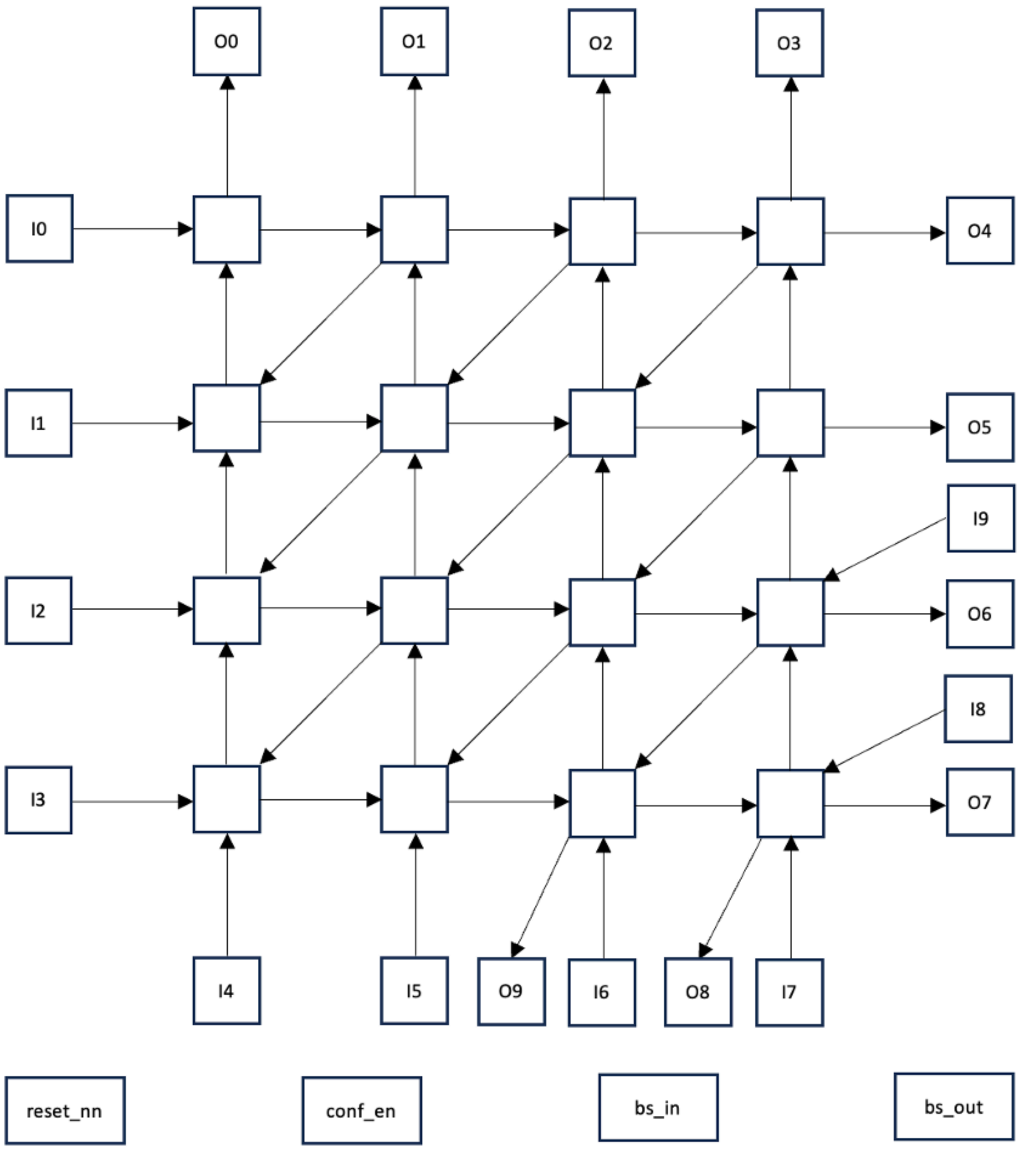
Reading output:

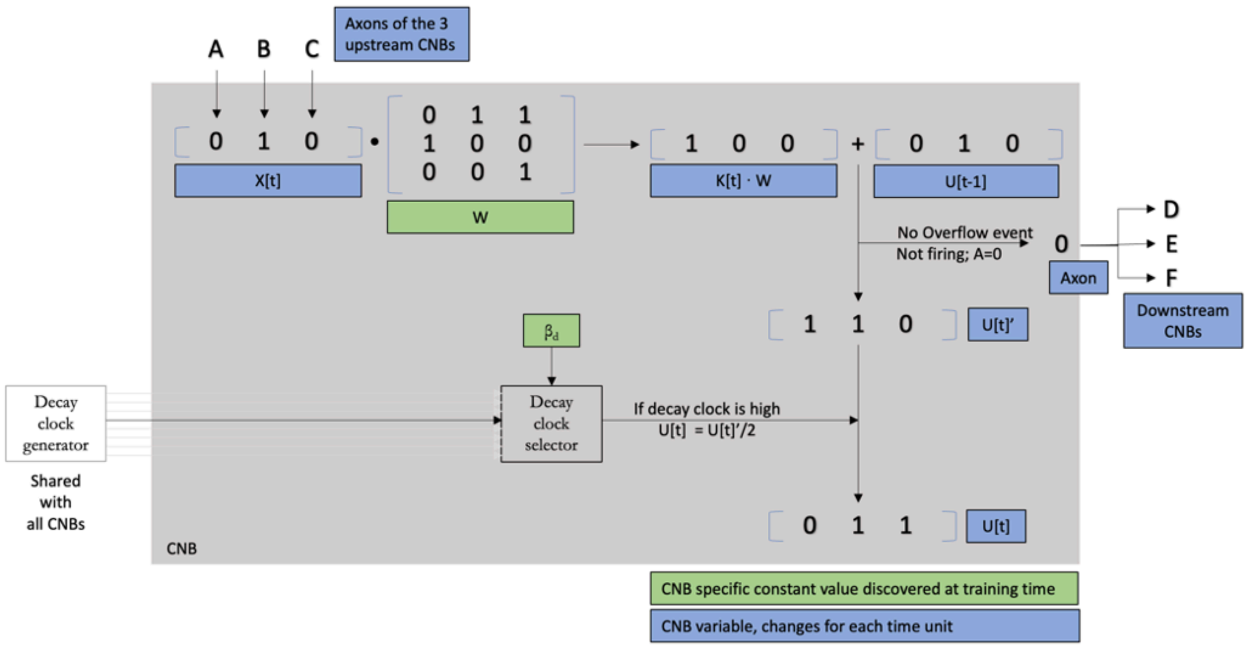
1) Set `orbit/max select` pin to low
2) Set `address0-4` to 0
3) Read low byte of orbit length from `output0-7`
4) Set `address0-4` to 1
5) Read high byte of orbit length from `output0-7`
6) Set `orbit/max select` pin to high
7) Repeat steps 2-5 to read the upper `Mx(N)` bits

Pinout

#	Input	Output	Bidirectional
0	input0	output0	address0
1	input1	output1	address1
2	input2	output2	address2
3	input3	output3	address3
4	input4	output4	address4
5	input5	output5	orbit/max select
6	input6	output6	start compute
7	input7	output7	write enable or compute busy indicator

Field Programmable Neural Array [518]





- Author: Reto Stamm
- Description: A collection of 50 interconnected simulated leaky neurons that can be programmed to do cognitive tasks.
- GitHub repository
- HDL project
- Mux address: 518
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Neuromorphic neural nets are more power efficient than traditional machine learning. It replicates an array of leaky neurons, a simple structure that exists in the brain. This design defines a Field Programmable Neural Array (FPNA). (1)

A mental model for a leaky neuron is a capacitor that drains at some rate. It gets charged up by some amount (its weight) whenever an input (a dendrite) receives a pulse from somewhere else. It sends a pulse (fire) its output (axon) when it reaches a specified level.

This circuit contains an array of 5*10 interconnected, heavily simplified configurable neuron blocks (CNBs). Instead of continuous weights, we have three bits per weight. Instead of a continuous decay of the charge in the capacitor, it halves at a somewhat configurable interval. Each CNB has its own set of weights, and a somewhat configurable rate of decay. In this design, each CNB had 4 inputs (dendrites), each with its own weight, one output (axon), and a choice of 8 different time decays.

An array of neuromorphic CNBs (Configurable Neuron Blocks). Each CNB has a 4 inputs, and each input has an associated weight that gets added to the CNBs membrane potential whenever the relevant input fires. When a CNB reaches a treshhold (rolls over, in this case), it fires and sends a pulse to 3 of its neighbours. Each CNB is subscribed to one of 8 decay clock tools.

The configuration data (Bitstream, or BS), including all the weights, the desired timing divisions, and the weights for each CNB are shifted in through the bs_in pin when the config_en pin is high. The BS can be read back from the bs_out pin.

The naxon tool is an example that shows how to train a neural network, generate all the relevant data and the BS that is needed to configure that model into this design https://github.com/retospect/naxon. More up-to-date design documents may also be found there.

References (1) Eshraghian, Jason K., Max Ward, Emre Neftci, Xinxin Wang, Gregor Lenz, Girish Dwivedi, Mohammed Bennamoun, Doo Seok Jeong, and Wei D. Lu. 2023. "Training Spiking Neural Networks Using Lessons From Deep Learning."

How to test

After reset, clock in the bitstream to configure all the weights and stuff. Then clock in the test data from the generated test bench from naxon, and see the appropriate answer come out!

Pinout

#	Input	Output	Bidirectional
0	dendritic input 0	output axon 0	reset_nn reset neural network (active high)
1	dendritic input 1	output axon 1	bs_in bitstream readout
2	dendritic input 2	output axon 2	bs_out bitstream input
3	dendritic input 3	output axon 3	config_en - shift bitstream
4	dendritic input 4	output axon 4	output axon 8
5	dendritic input 5	output axon 5	output axon 9
6	dendritic input 6	output axon 6	dendritic input 9
7	dendritic input 7	output axon 7	dendritic input 8

DFFRAM Example (128 bytes) [526]

- Author: Uri Shaked
- Description: 128 bytes DFFRAM module
- GitHub repository
- HDL project
- Mux address: 526
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

It uses a 32x32 1RW DFFRAM macro to implement a 128 bytes (1 kilobit) RAM module.

Reseting the project does not reset the RAM contents.

How to test

Set the `addr` pins to the desired address, and set the `in` pins to the desired value. Then, set the `wen` pin to `1` to write the value to the RAM, or set it to `0` to read the value from the RAM, and pulse `clk`.

The `out` pins will contain the value read from the RAM.

Pinout

#	Input	Output	Bidirectional
0	addr[0]	out[0]	in[0]
1	addr1	out1	in1
2	addr2	out2	in2
3	addr[3]	out[3]	in[3]
4	addr[4]	out[4]	in[4]
5	addr[5]	out[5]	in[5]
6	addr[6]	out[6]	in[6]
7	wen	out[7]	in[7]

Chonky Spiking Neural Net [582]

- Author: ReJ aka Renaldas Zioma, Paola Vitolo, Andrew Wabnitz. Big thanks to Jason Eshraghian!
- Description: 3 layer Spiking Neural Net with on-chip weights
- GitHub repository
- HDL project
- Mux address: 582
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

3 layer Spiking Neural Net with on-chip weights

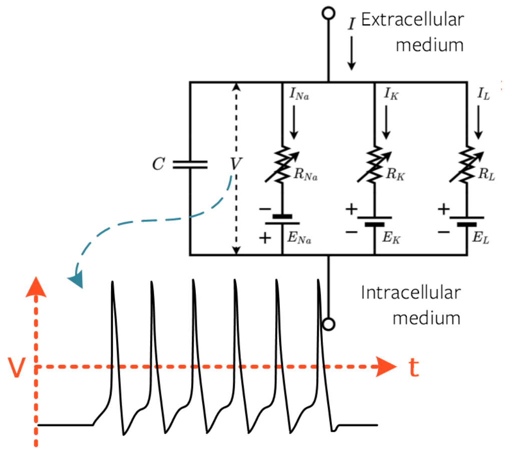
How to test

After reset...

Pinout

#	Input	Output	Bidirectional
0	input bus LSB	last layer neuron 0	(in) EXEC
1	input bus	last layer neuron 1	(in) SYNC
2	input bus	last layer neuron 2	(in) SETUP_CONTROL 0 bit
3	input bus	last layer neuron 3	(in) SETUP_CONTROL 1 bit
4	input bus	last layer neuron 4	(in) SETUP_CONTROL 2 bit
5	input bus	last layer neuron 5	(out) debug neuron layer 1
6	input bus	last layer neuron 6	(out) debug neuron layer 2
7	input bus MSB	last layer neuron 7	(out) debug neuron layer 2

Hodgkin-Huxley Neuron [590]



- Author: Jason Eshraghian
- Description: Implement a Hodgkin-Huxley neuron in silicon.
- GitHub repository
- HDL project
- Mux address: 590
- Extra docs
- Clock: 20000000 Hz
- External hardware:

How it works

Apply an input current injection to the LIF neuron. This will modify a neuron membrane potential, and with sufficient current injection, will cause periodic action potentials.

How to test

After reset, all state variables will be initialized. A minimum of 2 clock cycles of reset is needed.

An 8-bit input current is then applied to uio_in. The current is treated as the LSB of a 16-bit signal by concatenating 8x0's to the front. The first bit is a sign bit, the following 8-bits are treated as the whole number while the final 7-bits are the fraction. The current is interpreted in dimensions of uA/cm^2. This means the maximum value that can be represented is 1.992 uA/cm^2.

The membrane potential of the neuron will respond accordingly. Larger currents will elicit more firing. Simulations show that the neuron will start firing, reach a steady state where it stops firing in absence of input stimulus, and start firing again if the current exceeds approximately 8'b00001100 = 8'd12.

Pinout

#	Input	Output	Bidirectional
0	current bit 11	membrane potential bit a	membrane potential output bit i
1	current bit 12	membrane potential bit b	membrane potential fractional output bit j
2	current bit 13	membrane potential bit c	membrane potential fractional output bit k
3	current bit 14	membrane potential bit d	membrane potential fractional output bit l
4	current bit 15	membrane potential bit e	membrane potential fractional output bit m
5	current bit 16	membrane potential bit f	membrane potential fractional output bit n
6	current bit 17	membrane potential bit g	membrane potential fractional output bit o
7	current bit 18	membrane potential bit h	membrane potential fractional output bit p

PRBS Generator [641]

- Author: Ivan M Bow
- Description: Generates a PRBS that is configureable up to 8-bits.
- GitHub repository
- Wokwi project
- Mux address: 641
- Extra docs
- Clock: The input drives the output frequency. Hz
- External hardware: Connection to SPI port, clock input, and analyzer to observe.

How it works

Pseudo Random Binary Sequence (PRBS) Generator

Author: Ivan M Bow

This project was created using Wokwi and submitted to Tiny Tapeout for fabrication. The goal is to create a fully configurable, burst PRBS output. See Wiki for implementation details of PRBS and details on the operations of and polynomials for Linear-Feedback-Shift-Registers (LFSR).

Features

- Implements a Galois LFSR with XOR taps for PRN generation.
- Estimated 500kHz Max output PRBS rate, at PRBS2.
  - With 8-bit polynomial, 30 MHz should be achievable.
  - Max frequency reduces as PRBS size is reduced.
    * Estimated Max = (30 MHz / 2 ^ (8 - Nbits))
- Fail safe all 0's check to ensure no lock up.
- Clock Divider
- SPI Interface
  - CLK, MOSI, CS
  - SPI Mode 0, CS Active Low, MSB First
- Register access for configuration
- Differential Output
- Look-ahead Outputs
  - For each of the differential outputs, the next bit coming is output.
  - Useful for waveshaping or other information.
- Logic added in so a bit cannot be XOR'ed if the previous bit is disabled.

    - The highest order bit is not XOR'ed with the output bit, despite being in the poly.

- Enable pin for starting and resetting the output.
- Data pin for inverting the output.

Description The 8-bit PRBS generator has several 8-bit registers that are used to configure the output. Using the Tiny Tapeout board that is supplied with each project, the PRBS generator will take in a clock of any frequency output by the RP2040. The input clock is divided by the configured factor of 2, then this frequency is used to run the generator. The bit length and the polynomial of the output are configured in the registers. The output of the PRBS generator starts when the enable pin is set high.

There are 2 counters that control the output of the PRBS generator. The binary sequence will run for a configured number of times, with an output "clock" indicating this "rate". For Example, if the register is set to 20, the PRBS will be repeated 10 times, the output clock goes low, then another 10 times, and the output clock goes high. The idea behind this clock output is to signal to an external device for sending data. When the output clock goes low, the data needs to be set. When the output clock goes high, the data on the input pin is clocked in for the remainder of the output clock period.

The data bit is XOR'ed with the PRBS output to create a non-inverted or inverted sequence. Another register is configured to have the number of data bits that will be clocked into the PRBS generator. This number of data bits is the number of clock periods that are given from the output clock. Once the number of data bits has been completed, the PRBS generator automatically stops running. The generator remains off until the enable pin goes low, which resets the generator, and then high again to start another "data bits" cycles of the PRBS.

Registers are configured using SPI. For setting up each 8-bit register, the first byte sent is the command byte and must be hexadecimal 0x80 plus the address of the register to be configured. The second byte sent is the data that will be placed in the register and stored until changed or reset. The address field is the last 3-bits of the command byte and valid range is 1-5. Chip select high resets the command byte, and only 1 register may be written to per cycle of chip select.

A debug setup has been included for easy setup and testing. The debug mode sets the generator to divide the input clock by 16, the sequences per data bit to 7, the data bits count to 7, enables bits 0x0F (4 bits), and the polynomial to 0x0C (x^4 + x^3 + 1). To use the debug feature, start by placing all inputs low (including RST_N) to reset all registers and counters. Then:

1) Set the RST_N line high.

2) Set DEBUG high.
3) Set ENABLE high.

The PRBS generator is now running, and the data line can be toggled to invert the output. Once the PRBS has repeated 49 times, the generator will stop. To start the sequence again, toggle the enable line.

Note: While the DEBUG line is high, all registers will be non-configureable. To use the SPI and configure the PRBS generator, set the DEBUG line low.

Registers

- 5 registers control the PRBS generator
  - Register 0: Command and Address of register to configure *
  - Register 1: Clock Divider **
  - Register 2: PRBS count per data bit ***
  - Register 3: Count of data bits ***
  - Register 4: Bits to enable ****
  - Register 5: Polynomial XOR taps to enable *****
- Addressing and commands happen in a single CS session.
  - CS low -> 0x80 + 3-bit address -> 8-bit data -> CS high
- Reset_N clears all registers

Inputs

- **CLK** (RP2040 Clock)
- **RST_N** (Reset Low)
- **IN0**: SPI CS (Active Low)
- **IN1**: SPI CLK (Active High)
- **IN2**: SPI MOSI
- **IN3**: ENABLE (PRBS Generator Enable - Active High)
- **IN4**: DATA Bit Input
- **IN5**: No Connect
- **IN6**: No Connect
- **IN7**: DEBUG (Debug mode - Active high)

Outputs

- **OUT0**: PRBS_OUT_1 (PRBS Positive Look-ahead)
- **OUT1**: PRBS_OUT (PRBS Positive)
- **OUT2**: PRBS_OUT_N (PRBS Negative)
- **OUT3**: PRBS_OUT_N_1 (PRBS Negative Look-ahead)
- **OUT4**: DATA_CLK (Data Clock Output)
- **OUT5**: BUSY (PRBS Running)
- **OUT6**: CLK_OUT (RP2040 Clock)
- **OUT7**: CLK_PRBS_OUT (PRBS Generator Clock)

Bidirectional (All DIO are set to output and used for debug purposes.)

- **D0**: REG_SEL_0
- **D1**: REG_SEL_1
- **D2**: REG_SEL_2
- **D3**: PRBS_CLK_BYPASS
- **D4**: DATA_COUNT_CLK
- **D5**: DATA_COUNT_COMB_OUT
- **D6**: SEQ_COUNT_COMB_OUT
- **D7**: No Connect

Register Contents Register 0: Command & Address

7	6	5	4	3	2	1	0
C0	X	X	X	X	A2	A1	A0

- bits [7] - 0: Nothing occurs. 1: Writes the following word into the register
- bits [6:3] - Do Not Care
- bits [1:2] - 3-bit address of register to place the following data in.
  - (Address 0 is this register.)

Register 1: Clock Divider

7	6	5	4	3	2	1	0
X	X	X	X	X	D2	D1	D0

- bits [7:3] - Do Not Care
- bits [2:0] - Clock Divider
  - 0: /1

  - 1: /2
  - 2: /4
  - 3: /8
  - 4: /16
  - 5: /32
  - 6: /64
  - 7: /128

Register 2: Sequence Repeat Count

7	6	5	4	3	2	1	0
C7	C6	C5	C4	C3	C2	C1	C0

- bits [7:0] - Count of times PRBS sequence is repeated per bit.

Register 3: Data Bit Count

7	6	5	4	3	2	1	0
E8	E7	E6	E5	E4	E3	E2	E1

- bits [7:0] - Count of bits of data for which the generator runs.

Register 4: Polynomial Enable Bits

7	6	5	4	3	2	1	0
E8	E7	E6	E5	E4	E3	E2	E1

- bits [7:0] - E(n+1) is the enable bit for the polynomial size.
  - E() is 1 indexed to match the polynomial exponents.
    - 3-bit polynomial is b'111 or h'7.
    - 8-bit polynomial is b'11111111 or h'FF.
  - Bits must be sequential from bit 0. Other values are undefined.

Register 5: Polynomial Tap Bits

7	6	5	4	3	2	1	0
x^8	x^7	x^6	x^5	x^4	x^3	x^2	x^1

- bits [7:0] - E(n+1) is the enable bit for the polynomial taps.
  - E() is 1 indexed to match the polynomial exponents.

    * x^4 + x^2 + 1 is b’1010 or h’A.
    * x^5 + x^4 + x^3 + 1 is b’11100 or h’1C.

```
*     Do not address the command byte register, address 0. If the command
        data, then the data could trigger the command byte to transfer to
        whose address is based on the contents of bits 0-2 when bit 7 is
**    Clock divider bits 3-7 are unused and have no effect.
***   How the counters operate, a count of "0" is considered to be 65,536
        of "1" does not work as expected, and is equivalent to a count of
****  Bits must be enabled sequentially, starting with bit 0. Any bit ena
        sequential is an undefined state. I do not believe it will break
        looked into what this will do to the output.
***** Enabling an XOR tap bypasses the bit enable register setting. For e
        enabled but bit 6 has the XOR tap set, then the output polynomial
        of the polynomial settings.
```

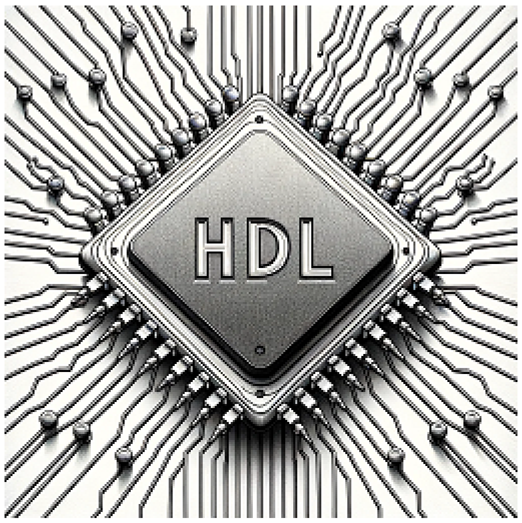
How to test

1) Clear inputs and reset to ensure known states.
2) Configure the registers using SPI or using the debug_setup pin.
3) Set “output_en” high and observe “prbs_out”.
4) Toggle “data_in” to invert “prbs_out” on next rising edge of “data_clk”.
5) To restart PRBS after “busy” goes low, clear “output_en” and set “output_en” again.

Pinout

#	Input	Output	Bidirectional
0	spi_cs	prbs_out_1	debug_out_reg_sel_0
1	spi_clk	prbs_out	debug_out_reg_sel_1
2	spi_mosi	prbs_out_n	debug_out_reg_sel_2
3	output_en	prbs_out_n_1	debug_out_prbs_clk_bypass
4	data_in	data_clk	debug_out_data_count_clk
5	none	busy	debug_out_data_count_comb_out
6	none	debug_out_system_clk	debug_out_seq_count_comb_out
7	debug_setup	debug_out_prbs_clk	none

Stop Watch [643]



- Author: Devin Atkin
- Description: Stop Watch System
- GitHub repository
- HDL project
- Mux address: 643
- Extra docs
- Clock: 25 000 000 Hz
- External hardware: 7 Segement Display (Common Anode Segments), Active Low top and Bottom

How it works

This creates a stop watch type of behavior. It was originally written and verified on the Basys 3 board. The Inc Switch controls whether the timer increments or decrements when the increment buttons are pressed. When the timer is running it can be paused by pressing the stop button. The timer can be reset by pressing the soft reset button.

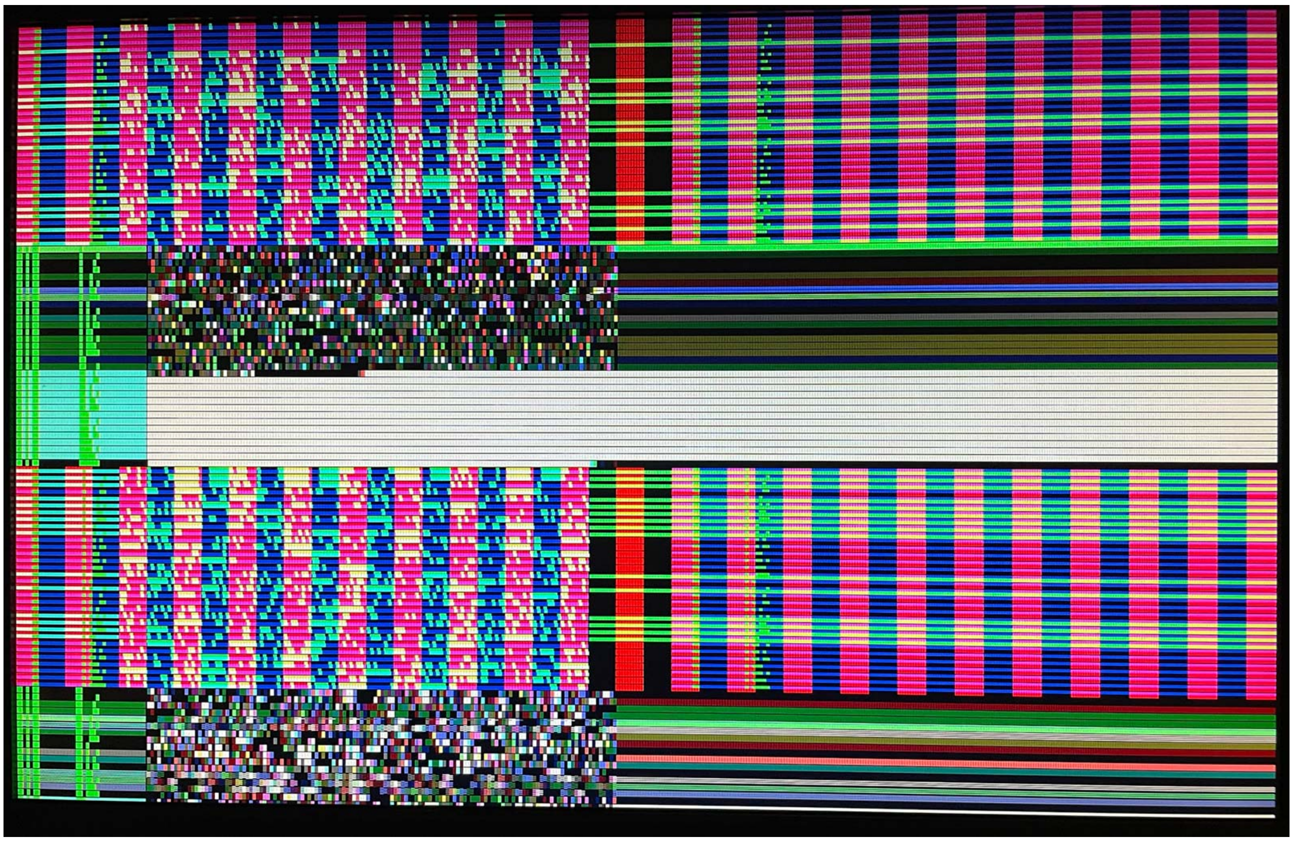
How to test

Provide a 7 segment display, some buttons, and a switch to control the behavior and see the output

Pinout

#	Input	Output	Bidirectional
0	start button	segment a	anode 1
1	stop button	segment b	anode 2
2	soft reset button	segment c	anode 3
3	inc minute button	segment d	anode 4
4	inc second button	segment e	none
5	inc switch	segment f	none
6	mode switch	segment g	none
7	none	none	none

vga_spi_rom [645]



- Author: algofoogle (Anton Maurovic)
- Description: Test reading/buffering/displaying SPI flash ROM data on VGA display
- GitHub repository
- HDL project
- Mux address: 645
- Extra docs
- Clock: 25.0MHz, 25.175MHz, or 26.6175MHz Hz
- External hardware: VGA DAC (RGB111 or 222 depth) and SPI flash memory

How it works

TBC.

Reads data from an SPI flash ROM (or any memory compatible with, say, W25Q10 or above) and displays it on a VGA display.

Drives a display at one of two resolutions (selectable by `vga_mode`):

- 0: 640x480 60Hz, from 25.1750MHz clock (though 25.0000MHz should do OK).
- 1: 1440x900 60Hz, from 26.6175MHz (or as close as you can get to it).

NOTE: Some monitors will also sync 640x480 using the 26.6175MHz clock that's otherwise used for vga_mode 1... they'll get ~63Hz instead of 60Hz. That means if you can get near this frequency, you might find it to be a good middle ground that allows you to switch live between 640x480 and 1440x900. My HP L1908wm monitor works fine this way for clocks in the range of 26.3MHz to 27.0MHz, and might even work *slightly* beyond that.

The flash memory contents are displayed using two alternating line modes (4 lines each):

1. As VGA scans near the middle of each line, read up to 136 bits (17 bytes) and store in local register memory, then display on the next scanline. NOTE: Because it buffers to memory on one line and displays the buffer on the next line, the *first* line is blanked out automatically by the design to avoid confusion. Hence you only actually see 3 lines for this line mode, before it switches to the next line mode.
2. Just send unregistered SPI data output (MISO) directly to the VGA display. In this line mode, you should see 4 identical lines before it switches back to line mode above.

These two line modes are timed to line up, so we can check for consistency between them, but note that there is actually expected to be a slight delay in the 'MISO direct' mode because the MISO output data appears on the *falling* SCLK edge. At 1440x900, this should be apparent.

The `reg_outs` signal, if HIGH, specifies that the VGA output signals should be registered. If LOW, the raw outputs go directly to the VGA display (inc. directly from the SPI memory when not in a buffered line).

NOTE: This tries to use `!CLK` (main clock, inverted) to drive the SPI SCLK directly, so that we don't need a *faster* system CLK in order to manage extra states. It keeps this `SCLK` output running constantly, relying on `/CS`.

NOTE: Besides the main design, I've got simple loopback test:

- `Test_in[2:0]` feeds a 3-input logical `AND` which outputs to `Test_out`

It could be interesting, when we get the actual chip back, to compare inputs to outputs on an oscilloscope.

NOTE: The sync polarity of `hsync` and `vsync` will be determined by vga_mode.

How to test

TBC.

Attach an SPI memory chip with some data in it, e.g. SPI flash ROM like W25Q10. Anything that accepts a 24-bit address and supports at least 27MHz reads should be fine. I used a generic 25Q80 (8Mbit) that I pulled off an ESP-01.

Attach a VGA display:

- For simplicity, I suggest starting with 640x480@60Hz mode. To do that, strap `vga_mode` to GND, and use a 25.000MHz (or ideally 25.175MHz) clock source. If you want to try 1440x900@60Hz mode instead, strap `vga_mode` high and use ~26.6175MHz as your clock source.
- NOTE: VGA modes 0 and 1 output different VSYNC polarity, as recommended by the spec.
- NOTE: I don't yet know what current a VGA display will sink per each input, nor what current the TT05 chip can safely supply per pin (or in total), so for now I recommend using some bigger resistors in series with each signal, even if it means an impedance mismatch. Either that, or just properly buffer each TT05 chip output with something like a 74ALVC245 (https://www.digikey.com.au/en/products/detail/texas-instruments/SN74ALVC245DWR/374035). Supposedly the Caravel IOs we're using might sink/source a max of 4mA, so assuming they're at 3.3V we could use 1kΩ resistors (which would hit 3.3mA per pin)…?
- `hsync`, `vsync` must be connected to the display; I recommend buffering them (as above) and then a 100Ω resistor then in series with their respective VGA pins (for safe current limiting) even though 100Ω is more often done.
- For a minimum display up and running quickly, attach the high bit (bit 2) of each of `red`, `green`, and `blue`, each via a 1kΩ resistor, to their respective VGA colour channel input pins. It might be pretty dark, but should be safe on Caravel IO pins if you don't otherwise buffer them.
- For a much better display, use an RGB222 (upper 2 bits per channel), ensuring it buffers each of those digital outputs. A good option is Tiny VGA (see: https://tinytapeout.com/specs/pinouts/)

Pinout

#	Input	Output	Bidirectional
0	In: vga_mode	Out: red1	Out: SPI /CS
1	In: rst_mode	Out: green1	Out: SPI io[0] / MOSI
2	In: reg_outs	Out: blue1	Out: SPI io1 / MISO

#	Input	Output	Bidirectional
3	In: N/C	Out: vsync	Out: SPI SCLK
4	In: N/C	Out: red[0]	Out: Test_out
5	In: Test_in[0]	Out: green[0]	Out: SPI /RST
6	In: Test_in1	Out: blue[0]	Out: SPI io2 (/WP)
7	In: Test_in2	Out: hsync	Out: SPI io[3] (/HLD)

RO and counter [647]

- Author: akita11
- Description: 8 inv and counter.
- GitHub repository
- HDL project
- Mux address: 647
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

This is a blink.

How to test

Check reset, ena, counter function.

Pinout

#	Input	Output	Bidirectional
0	inv1	counter[15]	output of inv1
1	inv2	counter[14]	output of inv2
2	inv3	counter[13]	output of inv3
3	inv4	counter[12]	output of inv4
4	inv5	counter[11]	output of inv5
5	inv6	counter[10]	output of inv6
6	inv7	counter[9]	output of inv7
7	inv8	counter[8]	output of inv8

8-Bit Shift Register with Output Latches 74HC595 [649]

- Author: Hirosh Dabui
- Description: The 74HC595 shift register
- GitHub repository
- HDL project
- Mux address: 649
- Extra docs
- Clock: Hz
- External hardware: You should connect 8 LEDs; perhaps a Pmod might also work.

How it works

https://www.onsemi.com/pdf/datasheet/mm74hc595-d.pdf

How to test

https://www.onsemi.com/pdf/datasheet/mm74hc595-d.pdf

Pinout

#	Input	Output	Bidirectional
0	sclrn	none	q[0]
1	ser	none	q1
2	rck	none	q2
3	srck	none	q[3]
4	G	none	q[4]
5	n/a	none	q[5]
6	n/a	none	q[6]
7	n/a	none	q[7]

Neptune guitar tuner (proportional window, version b, debug output on bidir pins, larger set of frequencies) [651]

- Author: Pat Deegan
- Description: It's a guitar tuner! and so much more…
- GitHub repository
- HDL project
- Mux address: 651
- Extra docs
- Clock: 1000 Hz
- External hardware: Digital input required: may need massaging if looking at actual guitar signals… see documentation

How it works

The rising edge of the input is counted over a set period of time and attempt is made to tell if this count is at, or near, a frequency of interest–namely the E,A,D,G,B notes of guitar standard tuning. In this version, the system should be capable of detecting: E2,A2,A3,B3,D3,E3,G3,D4,E4 and G4 so from about ~80 to 400 Hz. Clock config settings (using 3 input bits) 0: 1kHz 1: 2kHz 2: 4kHz 3: 3.277kHz 4: 10 kHz 5: 32.768kHz 6: 40kHz 7: 60kHz

How to test

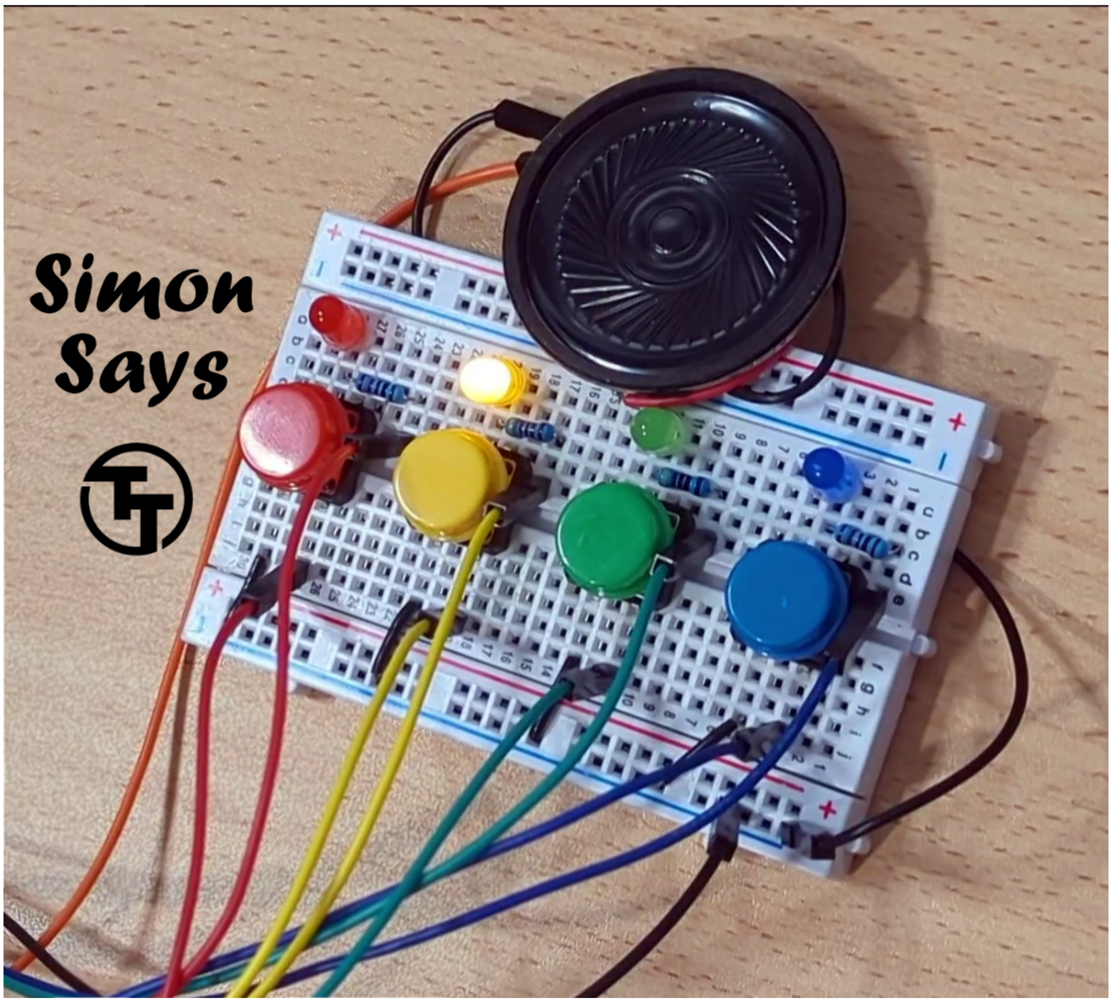
Set the clocking bits to 0b000 for a 1kHz clock input (input bits 2,3 and 4). Input pulses are fed to input bit 5. The raw count of pulses over the sampling period (hard-coded here to 0.5 seconds) is output on the bidir pins. The output is setup to drive a dual 7-segment display, or a single 7-segment (by using the output_display bits on the input). In single mode, the 7-seg dot marks exact match.

Pinout

#	Input	Output	Bidirectional
0	n/a	segment a	raw input pulse count bit 0
1	n/a	segment b	raw input pulse count bit 1
2	clk config 0	segment c	raw input pulse count bit 2
3	clk config 1	segment d	raw input pulse count bit 3
4	clk config 2	segment e	raw input pulse count bit 4
5	input pulse	segment f	raw input pulse count bit 5

#	Input	Output	Bidirectional
6	output display single enable (LOW == dual)	segment g	raw input pulse count bit 6
7	output display select	dot or select (for dual)	raw input pulse count bit 7

Simon Says game [653]



- Author: Uri Shaked
- Description: A simple memory game
- GitHub repository
- HDL project
- Mux address: 653
- Extra docs
- Clock: 50000 Hz
- External hardware: Four push buttons (with pull-down resistors), four LEDs, and optionally a speaker/buzzer and two digit 7-segment display

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/371755521090136065 (including wiring diagram).

How to test

You need four buttons, four LEDs, resistors, and optionally a speaker/buzzer and a two digit 7-segment display for the score.

Ideally, you want to use 4 different colors for the buttons/LEDs (red, green, blue, yellow).

1. Connect the buttons to pins `btn1`, `btn2`, `btn3`, and `btn4`, and 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.)
3. Connect the speaker to the `speaker` pin.
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!

Note: the game requires 50KHz clock input.

Pinout

#	Input	Output	Bidirectional
0	btn1	led1	seg_a
1	btn2	led2	seg_b
2	btn3	led3	seg_c
3	btn4	led4	seg_d
4	seginv	speaker	seg_e
5	none	dig1	seg_f

#	Input	Output	Bidirectional
6	none	dig2	seg_g
7	none	none	none

KianV uLinux SoC [654]

- Author: Hirosh Dabui
- Description: A RISC-V ASIC that can boot Linux.
- GitHub repository
- HDL project
- Mux address: 654
- Extra docs
- Clock: 50MHz Hz
- External hardware:

How it works

Here is an RV32IMA RISC-V processor that can boot and run uLinux.

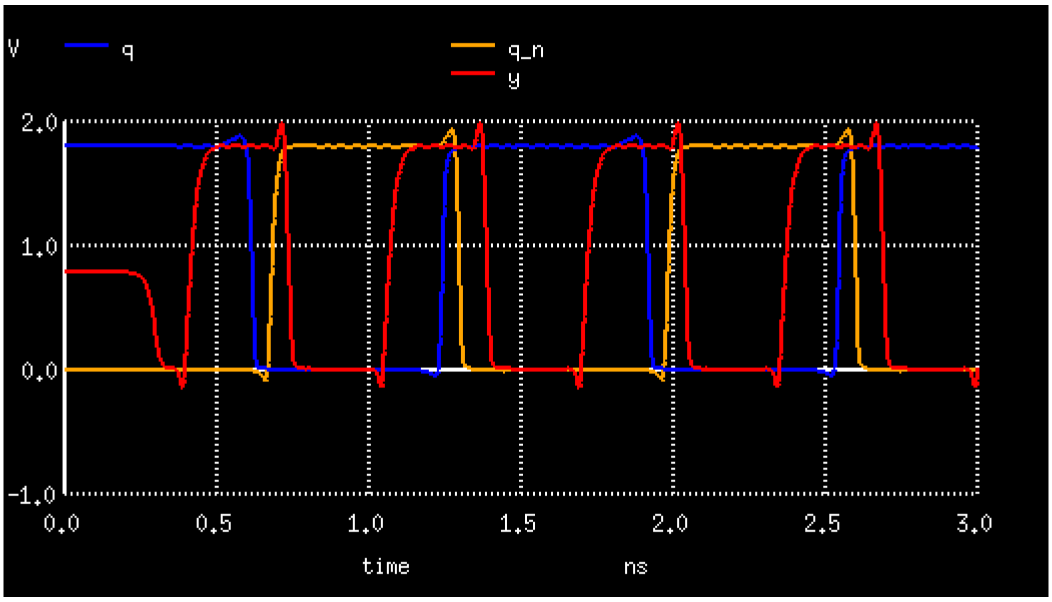
How to test

You need to flash the bootloader, dtb, and the Linux image onto the NOR flash. It was tested on the ICE40 with the same design at 35MHz. There is a divider register located at 0x10_000_010. With the upper 16 bits, the CLINT can be configured, and with the lower 16 bits, the UART can be configured.

Pinout

#	Input	Output	Bidirectional
0	none	led[0]	ce0
1	none	led1	sio0_si_mosi_i
2	none	led2	sio1_so_miso_o
3	uart_rx	led[3]	sio2_o
4	none	uart_tx	sio3_o
5	none	led[4]	sclk_ram
6	none	led[5]	ce1
7	none	led[6]	sclk_nor

Ring oscillator with counter [655]



- Author: Uri Shaked
- Description: Test module for the TT05 power switching FET
- GitHub repository
- HDL project
- Mux address: 655
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

A ring oscillator with a 64-bit counter that counts the number of oscillations. The counter is connected to pins ou_out, and is shifted by the cnt_shift input. The counter is reset when cnt_reset is high, and stops when cnt_stop is high.

How to test

Set inputs 0 to 5 to the desired counter shift value, and observe the counter on outputs 0 to 7.

Pinout

#	Input	Output	Bidirectional
0	cnt_shift[0]	cnt[0]	none
1	cnt_shift1	cnt1	none
2	cnt_shift2	cnt2	none
3	cnt_shift[3]	cnt[3]	none
4	cnt_shift[4]	cnt[4]	none
5	cnt_shift[5]	cnt[5]	none
6	cnt_stop	cnt[6]	none
7	cnt_reset	cnt[7]	none

cpu_8bit [705]

- Author: Sunao Furukawa
- Description: This Verilog code is geenrated by Bing AI
- GitHub repository
- HDL project
- Mux address: 705
- Extra docs
- Clock: 50000000 Hz
- External hardware:

How it works

Explain how your project works

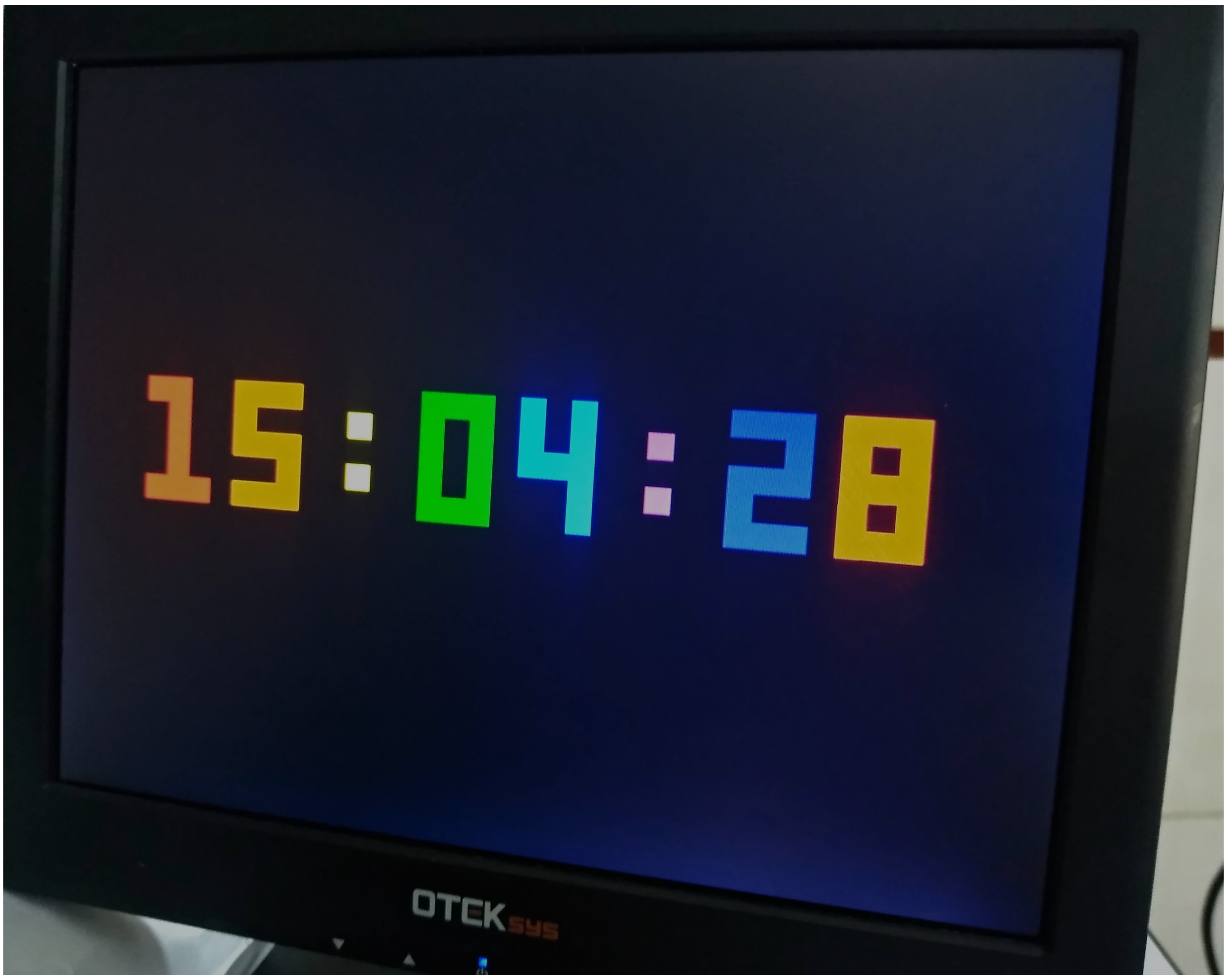
How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	instruction register [0]	segment a	none
1	instruction register 1	segment b	none
2	instruction register 2	segment c	none
3	instruction register [3]	segment d	none
4	formula right side [4]	segment e	none
5	formula right side [5]	segment f	none
6	formula right side [6]	segment g	none
7	formula right side [7]	dot	none

VGA clock [707]



- Author: Matt Venn
- Description: Shows the time on a VGA screen
- GitHub repository
- HDL project
- Mux address: 707
- Extra docs
- Clock: 31500000 Hz
- External hardware: R2R dac for the VGA signals

How it works

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

How to test

Hook up a VGA monitor to the outputs and provide a clock at 31.5 MHz. Connect the 6 bit colour output up with resistors to make a R2R DAC. See the circuit here: https://github.com/mattvenn/6bit-pmod-vga

Pinout

#	Input	Output	Bidirectional
0	clock	hsync	none
1	reset	vsync	none
2	adjust hours	r0	none
3	adjust minutes	r1	none
4	adjust seconds	g0	none
5	none	g1	none
6	none	b0	none
7	none	b1	none

7 segment seconds (Verilog Demo) [709]

- Author: Matt Venn
- Description: Count up to 10, one second at a time.
- GitHub repository
- HDL project
- Mux address: 709
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Uses a set of registers to divide the clock, and then some combinational logic to convert from binary to decimal for the display.

Puts the bottom 8 bits of the counter on the bidirectional outputs.

With all the inputs set to 0, the internal 24 bit compare is set to 10,000,000. This means the counter will increment by one each second.

If any inputs are non zero, then the input will be used as an bits 11 to 18 of the 24 bit compare register. Example: setting the inputs to 00010000 will program 16384 into the compare register. With a 10MHz clock the counter will increment ~610 times per second.

How to test

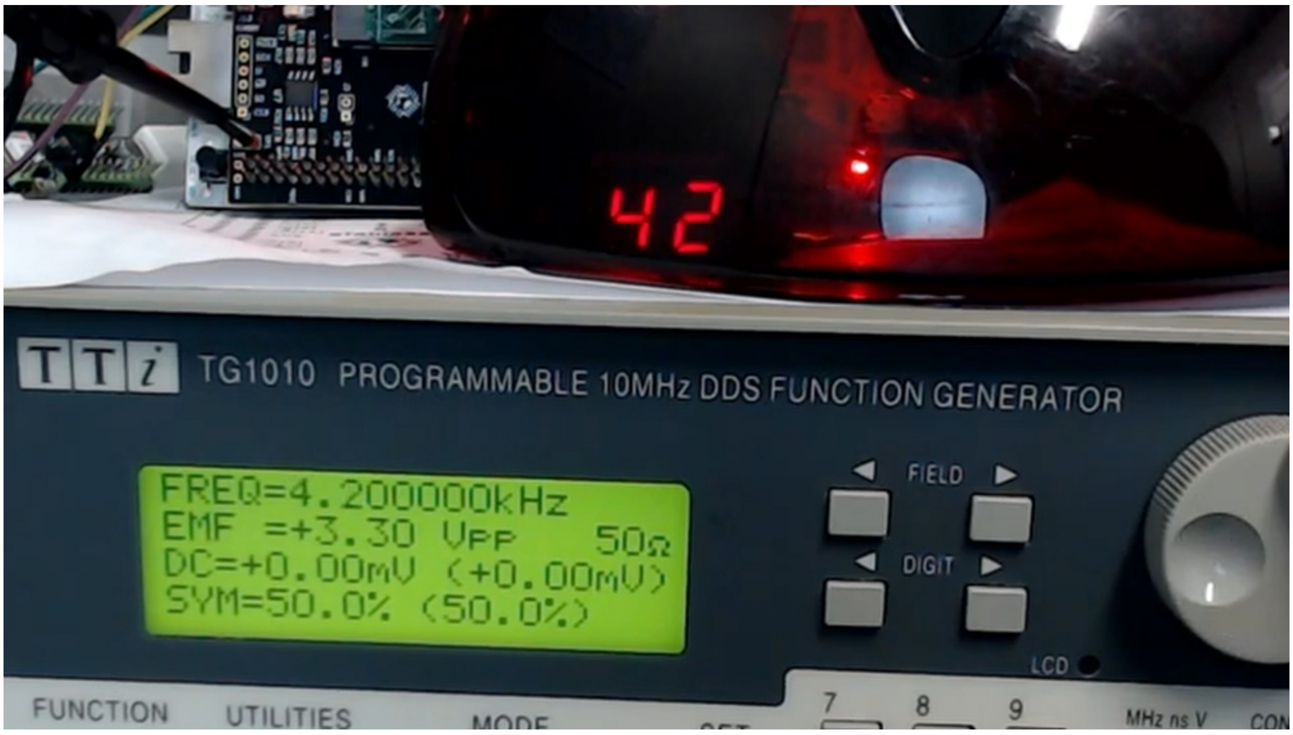
After reset, the counter should increase by one every second with a 10MHz input clock. Experiment by changing the inputs to change the counting speed.

Pinout

#	Input	Output	Bidirectional
0	compare bit 11	segment a	second counter bit 0
1	compare bit 12	segment b	second counter bit 1
2	compare bit 13	segment c	second counter bit 2
3	compare bit 14	segment d	second counter bit 3
4	compare bit 15	segment e	second counter bit 4
5	compare bit 16	segment f	second counter bit 5

#	Input	Output	Bidirectional
6	compare bit 17	segment g	second counter bit 6
7	compare bit 18	dot	second counter bit 7

Frequency counter [711]



- Author: Matt Venn
- Description: measured frequency of a signal on pin 0 and displays on the 7 segment display
- GitHub repository
- HDL project
- Mux address: 711
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Debounces the input signal and counts how many transistions occur in a given period. A state machine then divides the count by ten by repeatedly subtracting ten and then displays the tens and units on the seven segment display.

How to test

Apply a signal to the signal input and the frequency will be measured and displayed on the seven segment. The dot is used to select between display tens and units.

To change the count period (to get accurate counts), set it to match the clock frequency: clock_mhz * 100 - 1. So for a 10MHz clock, set to 999. Set the desired period (top 4 bits ui_in and all of uio_in) on the bidirectional inputs and toggle load input.

To debug, enable debug mode and check the bidirectional outputs for state machine, clock count and edge count information.

Pinout

#	Input	Output	Bidirectional
0	signal	segment a	count period bit 07 or debug edge bit 2
1	debug mode (on will put debug signals on bidirectional outputs)	segment b	count period bit 06 or debug edge bit 1
2	load new period. toggle this to register the value in the bidirectional inputs	segment c	count period bit 05 or debug edge bit 0
3	none	segment d	count period bit 04 or debug clock bit 2
4	count period bit 11	segment e	count period bit 03 or debug clock bit 1
5	count period bit 10	segment f	count period bit 02 or debug clock bit 0
6	count period bit 09	segment g	count period bit 01 or debug state bit 1
7	count period bit 08	digit select	count period bit 00 or debug state bit 0

RGB Mixer [713]

- Author: Matt Venn
- Description: Use 3 rotary encoder to control 3 PWM generators
- GitHub repository
- HDL project
- Mux address: 713
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

3 PWM generators are fed by 3 debounced encoder peripherals.

How to test

Connect 3 digital rotary encoders to the first 6 inputs. Changing the encoders will change the PWM outputs on the first 3 outputs.

Select a channel with the debug enc sel bits and that channel's encoder internal value will be output to the bidirectional outputs. The output of the 2 debouncers will also be output.

Pinout

#	Input	Output	Bidirectional
0	encoder 0 pin a	pwm 0	debug encoder bit 0
1	encoder 0 pin b	pwm 1	debug encoder bit 1
2	encoder 1 pin a	pwm 2	debug encoder bit 2
3	encoder 1 pin b	debug debounce a	debug encoder bit 3
4	encoder 2 pin a	debug debounce b	debug encoder bit 4
5	encoder 2 pin b	n/a	debug encoder bit 5
6	debug encoder select bit 0	n/a	debug encoder bit 6
7	debug encoder select bit 1	n/a	debug encoder bit 7

SPI Peripheral [715]

- Author: Mike Bell
- Description: SPI RAM/ROM/Random source
- GitHub repository
- HDL project
- Mux address: 715
- Extra docs
- Clock: 10000000 Hz
- External hardware: A custom RP2040 board to make full use of the ROM

How it works

The project implements an 8 byte RAM, 324 byte ROM and a random source supporting standard SPI read/write (03h/02h) and SPI quad read/write (6Bh/32h) commands. The quad read commands have 2 delay cycles.

The address map is:

Address	Item
0x000	RP2040 boot stage 2 ROM
0x100	8 byte RAM, wrapped 32 times
0x200	RP2040 program ROM
0x300	Mirror of the RAM
0x400	Random source

See the README for more details.

Note the default project clk is a debug clock only, the project is internally clocked off SPI CLK, input 0.

How to test

You will need to use an SPI/QSPI master, unfortunately it was not possible to set up the pinout to support both QSPI and match the native RP2040 SPI block, so you'll need a PIO (Q)SPI implementation. I'll make that available before the chips are available.

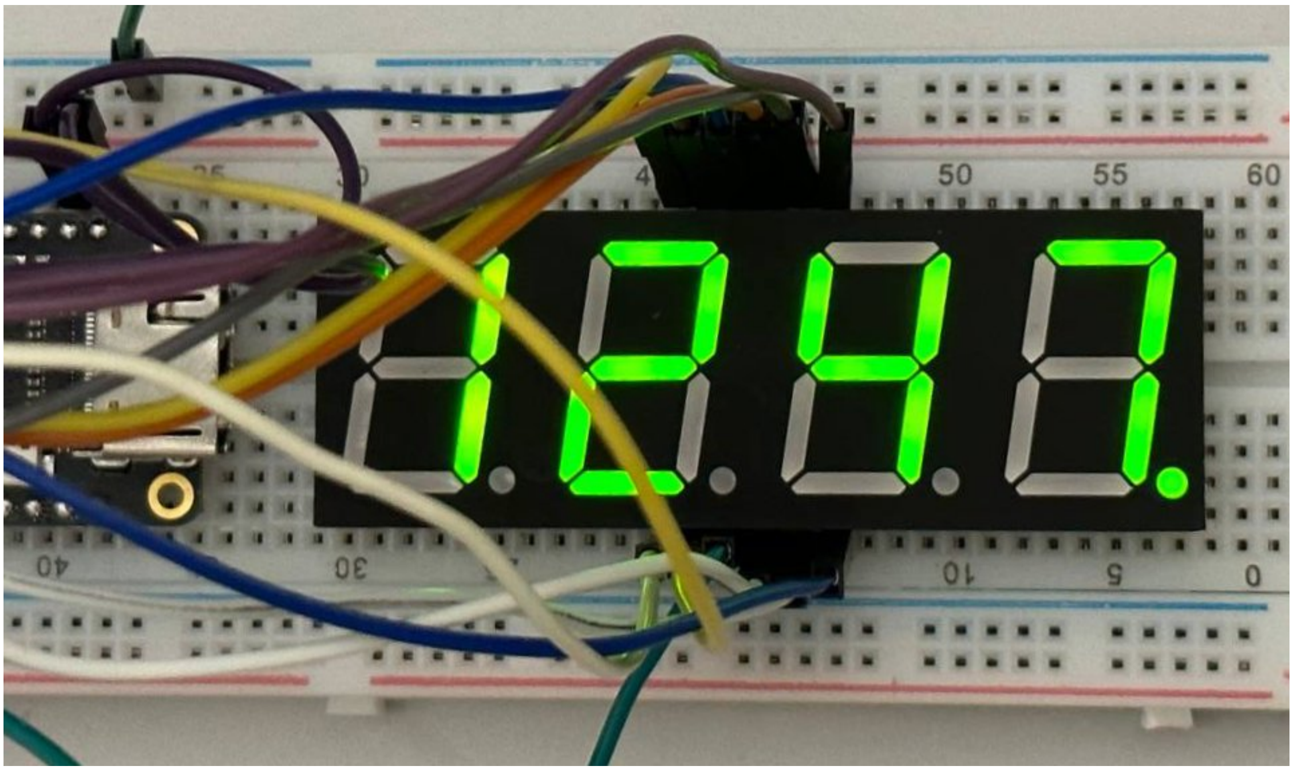
The values in the RAM may be inspected by setting the address on the input toggle switches 2-5 and pressing the single clock button to latch the data, which is displayed on the 7 segment display and presented on uio pins 4-7. The default project clock should otherwise not be used - the project is clocked from the SPI clock.

The project is also designed to be used as a ROM connected to an RP2040's QSPI pins (instead of the more normal flash). You'll need a custom board for this as the QSPI pins are generally connected directly to a flash chip, I have a couple which I could potentially send to interested people in the UK - contact me on Discord.

Pinout

#	Input	Output	Bidirectional
0	SPI CLK	segment a	SPI MOSI / D0
1	SPI CSn	segment b	SPI MISO / D1
2	Debug nibble select	segment c	D2
3	Debug addr 0	segment d	D3
4	Debug addr 1	segment e	Debug bit 0
5	Debug addr 2	segment f	Debug bit 1
6	Unused	segment g	Debug bit 2
7	Unused	dot (Set to SPI CSn)	Debug bit 3

Multiplexed clock [717]



- Author: Gustavo Gomez
- Description: Multiplexed clock with buttons
- GitHub repository
- HDL project
- Mux address: 717
- Extra docs
- Clock: 32728 Hz
- External hardware: 7 segment 4 digits multiplexed

How it works

Basically this is a clock that counts minutes shows the hours in the 24-hour format, it uses the dot in the 7 segments to indicate 15s 30s 45s and 60s respectibly.

```
   -- 0 --           -- 0 --           -- 0 --           -- 0 --
  |       |         |       |         |       |         |       |
  5       1         5       1         5       1         5       1
  |       |         |       |         |       |         |       |
   -- 6 --           -- 6 --           -- 6 --           -- 6 --
  |       |         |       |         |       |         |       |
  4       2         4       2         4       2         4       2
```

```
 |       |       |       |       |       |       |       |
  -- 3 --   :60s   -- 3 --   :45s   -- 3 --   :30s   -- 3 --   :15s
     ↑                ↑                ↑                ↑
   Hours H          Hours L        Minutes H        Minutes L
 uio_out[3]       uio_out[2]       uio_out[1]       uio_out[0]
```

[6:0] of the seven segments are connected to the **uo_out** output and the 7 bit is for the dot of the seven sevements. The digist are multiplexed, each digit is shown 1ms, those pins are **uio_out[3:0]** and **uio_out[5:4]** are used for debuging showing the clock of the seconds and minutes.

For the test i have used this 7 segment with common Cathode. But you can use which ever 7 segmnet display of 4 digits common or anode thats to the pins **ui_in[3:2]** with are use to negate the 7 segmetents or the multixplexing. Finally, **ui_in[1:0]** are used with a button to increase the hours or minutes.

How to test

I have selected a clock 32,768khz because i thought it will be easy to buy a ready commponent that generates a squera wave with that frecuency, we will see about that :stuck_out_tongue_closed_eyes:. Just connect the 7 segments to the **uo_out** pins and select your configuration anode or catothe with the **ui_in[3]** pin. For the multiplexing connected uio_out[3:0] to the digits as show in how to use drawing.

And finally if you want to increase the numbers connect a button pull up to the pins **ui_in[1:0]** to increase hours or minutes.

Generate a reset to start to init all the registers.

Pinout

#	Input	Output	Bidirectional
0	pull up button that increases minutes	segment a	output multiplex first digit
1	pull up button that increases hours	segment b	output multiplex second digit
2	pin used to negate 7 segment ouputs if necesary / for catode or anode configurations	segment c	output multiplex third digit

#	Input	Output	Bidirectional
3	pin used to negate 4 pins to multiplex if necesary / for catode or anode configurations	segment d	output multiplex forth digit
4	not used	segment e	output clock of seconds / testing purposes
5	not used	segment f	output clock of minutes / testing purposes
6	not used	segment g	output not used
7	not used	dot	output not used

Shaman: SHA-256 hasher [718]

- Author: Pat Deegan, psychogenic.com
- Description: Generate a SHA256 digest for data of arbitrary length
- GitHub repository
- HDL project
- Mux address: 718
- Extra docs
- Clock: 10000000 Hz
- External hardware: An MCU or something to feed in the bytes and receive the results

How it works

This implements the SHA-256 digest to create hashes of the data you feed in. It is a naive, mostly unoptimized, implementation of the algorithm (though you can interleave data input while it's processing, using parallel mode, if you respect busy).

Data is fed into the system in 64 byte blocks. The hash is available after each 64 byte block has been input (allowing for some cycles to finish processing).

The process is to:

- toggle start, to reset the digest
- put data byte on the databyte input (the "in" port)
- wait until busy is de-asserted (if required)
- clock the clockin_data pin

After each complete block, the digest will become available after some clocks. In short if

- busy is not asserted; and
- result_ready goes high

The first hash byte will be available on the out port. To get the next bytes, clock result_next and read the port.

Parallel mode allows you to start feeding in more input data while the system is still processing the previous block. You need to pay attention to and respect "busy", here, or things will get badly munged.
Also, in parallel mode, you need to hold the clockin_data for an extra cycle when you bring it high.

Pinout looks a little weird but it is hoped this will be a nice match for the PMOD arrangement on the demo boards.

NOTES

It does NOT massage the input data into suitable blocks. Messages need to be appended with an 0x80 byte, padded such that the entire thing, along with an 8 byte suffix containing the length (big end), is a multiple of 512 bits (64 bytes). You can see this in action in the message_to_blocks() function, in test.py.

I don't think it's super fast but, in parallel mode, I *think* simulation indicates it takes on the order of 8.3 microseconds per byte using a 1MHz system clock. So, if we could feed this say a 50MHz clock, we'd get down to 166 ns/byte.
That's only on the order of 6 megabytes per second, I dunno maybe 100x slower than my laptop, but my laptop doesn't run on a 50MHz clock and whatevs: should do the

job if it holds in realy life. All this is when processing longer messages, to swamp out the minor overhead of setup etc.

When loading input data, if using parallel mode, hold clockin_data for an extra system clock. So

- data byte on inputs
- clockin_data HIGH
- hold one system clock
- clockin_data LOW
- … loop for next byte

How to test

Might be good to run the cocotb test to get VCDs if you really want to see it in action. But we want to play with hardware! So… There will be a python script in the repository to convert any content into the expected 512 bit blocks of bytes padded and everything to make the system happy.

With that list of bytes in hand, this should work nicely:

1) hold n_reset low for a few clock cycles
2) bring n_reset high, and give it a few cycles
3) start a new message digest my clocking start (bring high for one cycle, then low)
4) for each block in your message

    - while "busy" is HIGH, wait a bit and check again
    - for each byte in that block
        - put the byte on in port (dedicated input pins)
        - while "busy" is HIGH, wait a bit and check again
        - bring clockin_data HIGH
        - if using parallel mode, hold for an extra clock cycle
        - bring clockin_data LOW

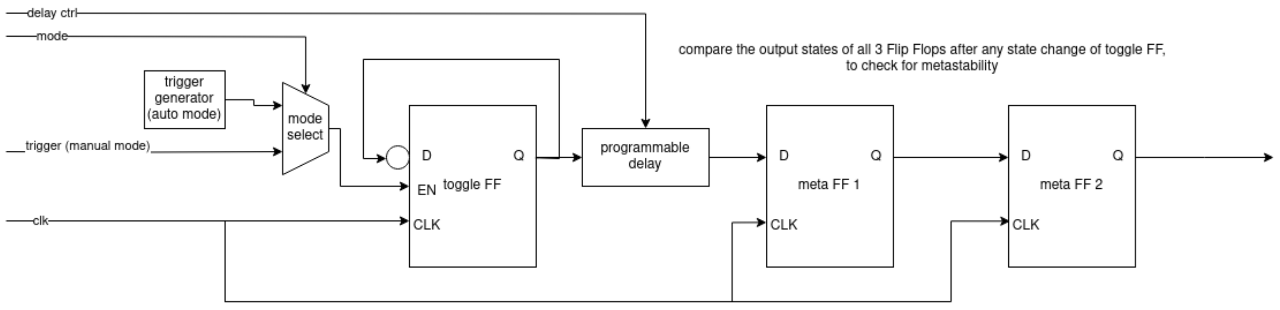
Check and wait until "busy" is LOW and "result_ready" goes HIGH. Your first result byte will already be present on the output port. Grab it and stash it. Then, for the next 31 bytes: bring result_next HIGH hold it there for one clock cycle bring result_next LOW grab and stash the byte on output pins

If the hash is going to be, say "90fc0a268f8b81b…", they'll be present in that order 0x90, then 0xfc, then 0x0a etc

Pinout

#	Input	Output	Bidirectional
0	data_input bit 0	result_byte bit 0	OUTPUT, result_ready
1	data_input bit 1	result_byte bit 1	OUTPUT, begin processing data debug
2	data_input bit 2	result_byte bit 2	INPUT, parallel loading enable
3	data_input bit 3	result_byte bit 3	INPUT, result_next
4	data_input bit 4	result_byte bit 4	OUTPUT, busy
5	data_input bit 5	result_byte bit 5	OUTPUT, processing data block debug
6	data_input bit 6	result_byte bit 6	INPUT, start new digest
7	data_input bit 7	result_byte bit 7	INPUT, clockin_data

metastability experiment [719]


delay ctrl
mode
trigger generator (auto mode)
mode select
trigger (manual mode)
clk
D
Q
EN
toggle FF
CLK
programmable delay
compare the output states of all 3 Flip Flops after any state change of toggle FF, to check for metastability
meta FF 1
meta FF 2

- Author: yubex
- Description: The design purpose is to evaluate, if metastability can be used as a true random number generator source in an ASIC design.
- GitHub repository
- HDL project
- Mux address: 719
- Extra docs
- Clock: try various Hz
- External hardware: none

How it works

Generally a Flip Flop can enter a metastable state if the setup or hold time is violated. In this design I try to reach this usually unwanted behaviour.
A toggle Flip Flop is used to create edges on the data inputs of the other 2 Flip Flops. There are 2 modes: Manual and Auto. Manual mode uses the switch on the PCB as trigger for one edge. Auto mode generates a cyclic trigger within the ASIC. The data output of the toggle Flip Flop is connected to the next Flip Flop by a programmable delay line. The delay line is created by pairs of inverters. The verilog keep attribute is necessary here, to avoid optimization during synthesis. The delay_ctrl input selects the number of inverters which are used as delay. The number of inverters used is the delay_ctrl input value times 2. If you set delay_ctrl to 1, 2 inverters are used. The maximum number of inverters is 128. After each state change of the toggle Flip Flop (exactly 3 clock cycles after that) the output of all Flip Flops are compared. In case the states are different, metastability has occured.

How to test

For testing select the mode you want to try out. Set the mode to 0 for auto mode and to 1 for manual mode. You can experiment with delay_ctrl input an try to produce metastability. Also try different clock frequency's. In case of metastability the dot of the 7 segment display should change its state.

Pinout

#	Input	Output	Bidirectional
0	trigger	segment a, mode	none
1	mode	segment b, toggle_dff_en	none
2	delay_ctrl[0]	segment c, toggle_dff	none
3	delay_ctrl1	segment d, delayed_toggle_dff	none
4	delay_ctrl2	segment e, meta_dff_0	none
5	delay_ctrl[3]	segment f, meta_dff_1	none
6	delay_ctrl[4]	segment g, toggle_dff_en_3t	none
7	delay_ctrl[5]	dot, meta_err_detected	none

Pinout

The chip is packaged in a 64-pin QFN package. The pinout is shown below.

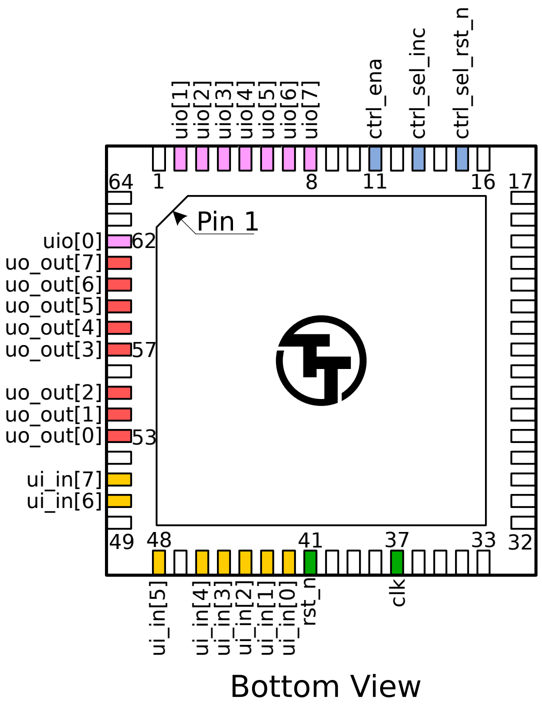


Figure 4: Pinout

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 384 user designs (24 mux units, each with up to 16 designs)
- Designs can have different sizes. The basic unit is a called a tile, and designs can occupy 1, 2, 4, 6, 8, or 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:

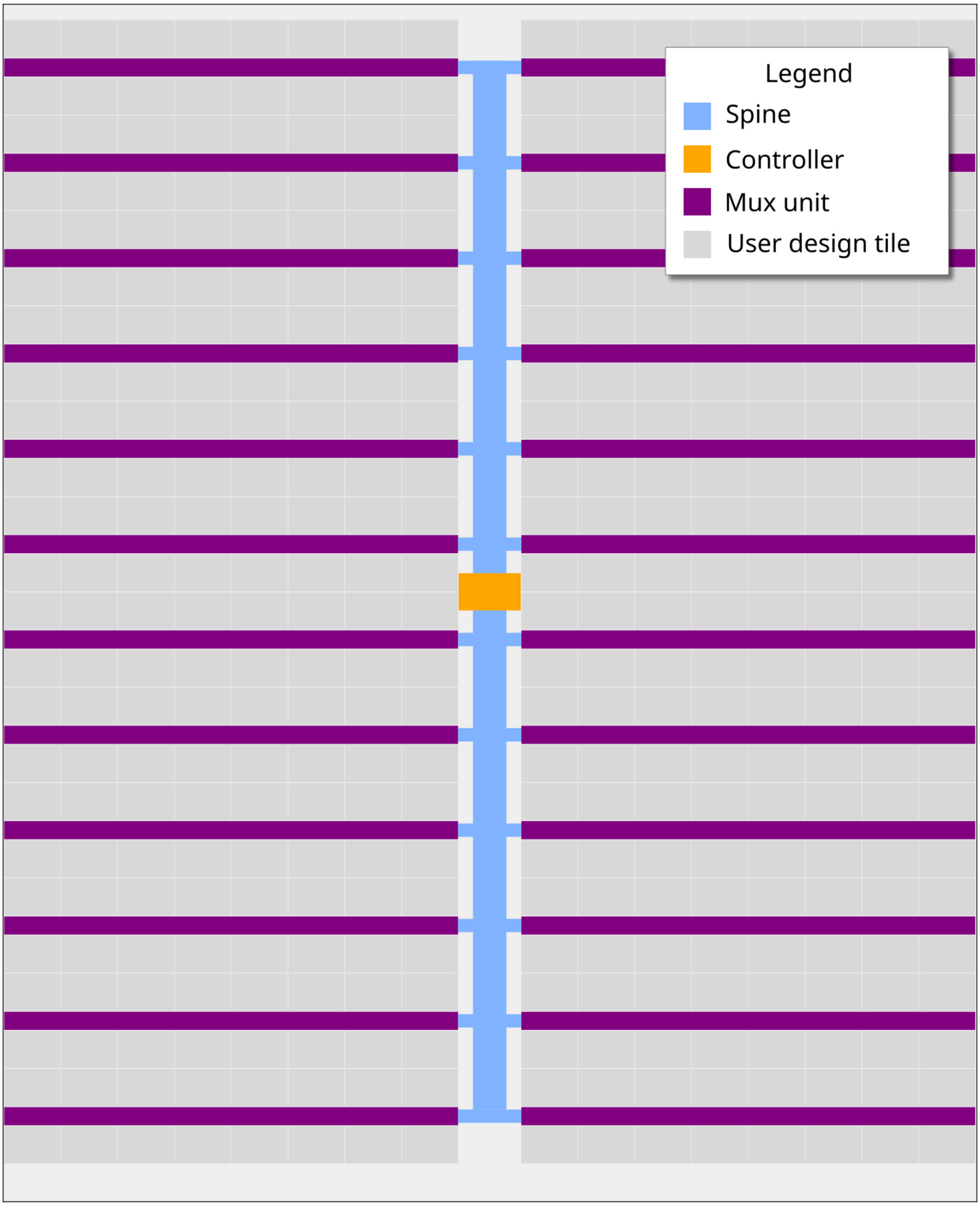


Figure 5: Mux Diagram

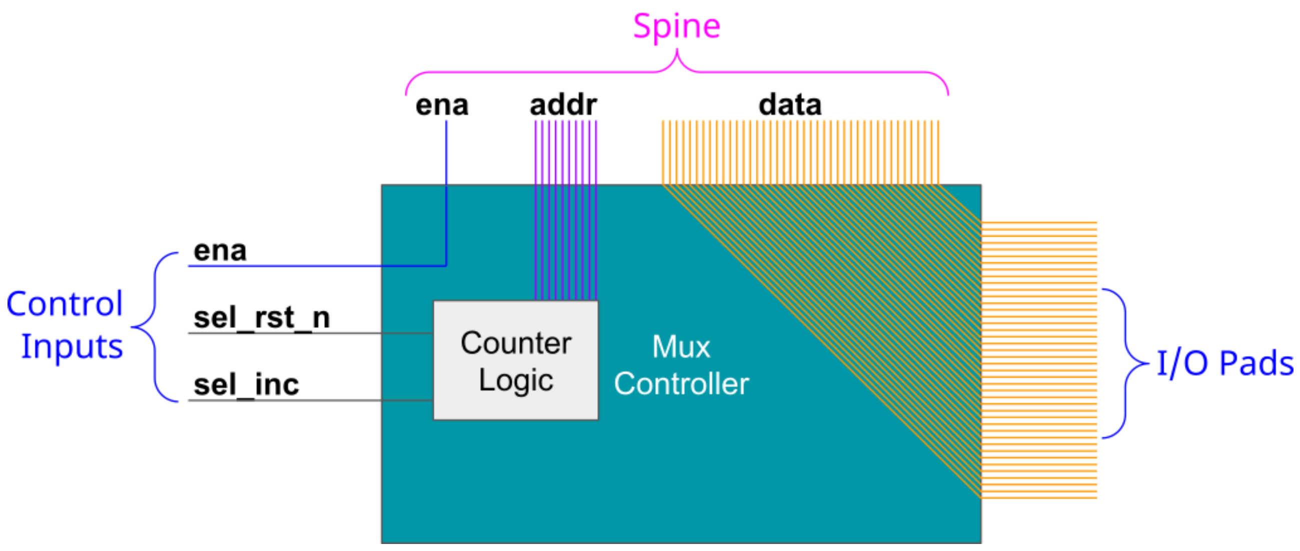


Figure 6: Mux Controller Diagram

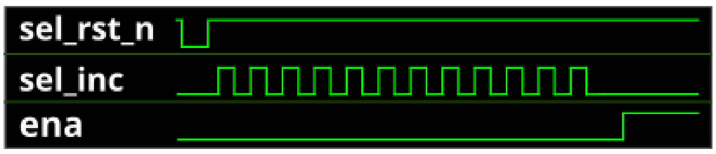


Figure 7: Mux signals for activating the design at address 12

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/36434780766 It contains an Arduino Nano that decodes the currently selected mux address and displays it on a 7-segment display. Click on the button labeled `RST_N` to reset the counter, and click on the button labeled `INC` to increment the counter.

The Spine

The controller and all 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		(none)	31
1	Housekeeping SPI *	SDO	32
2	Housekeeping SPI	SDI	33
3	Housekeeping SPI	CSB	34
4	Housekeeping SPI	SCK	35

mprj_io pin	Function	Signal	QFN64 pin
5	Clock output	user_clock2 †	36
6	Input	clk	37
7	Input	rst_n	41
8	Input	ui_in[0] ‡	42
9	Input	ui_in1	43
10	Input	ui_in2	44
11	Input	ui_in[3]	45
12	Input	ui_in[4]	46
13	Input	ui_in[5]	48
14	Input	ui_in[6]	50
15	Input	ui_in[7]	51
16	Output	uo_out[0]	53
17	Output	uo_out1	54
18	Output	uo_out2	55
19	Output	uo_out[3]	57
20	Output	uo_out[4]	58
21	Output	uo_out[5]	59
22	Output	uo_out[6]	60
23	Output	uo_out[7]	61
24	Bidirectional	uio[0]	62
25	Bidirectional	uio1	2
26	Bidirectional	uio2	3
27	Bidirectional	uio[3]	4
28	Bidirectional	uio[4]	5
29	Bidirectional	uio[5]	6
30	Bidirectional	uio[6]	7
31	Bidirectional	uio[7]	8
32	Mux Control	ctrl_ena	11
33		(none)	12
34	Mux Control	ctrl_sel_inc	13
35		(none)	14
36	Mux Control	ctrl_sel_rst_n	15
37		(none)	16

- The Housekeeping SPI is an SPI interfaces provided by the Caravel harness. You can use it to change the configuration of the GPIO pins and control the clock for the internal Caravel RISC-V core. We do not plan to use it in the Tiny Tapeout Demo board.
  † The `user_clock2` signal outputs the internal clock signal of caravel. You could use it to provide a clock to your design by connecting it to the `clk` input

(mprj_io pin 6). We do not plan to use it in the Tiny Tapeout Demo board.
‡ 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 bit.

Sponsored by

efabless.com

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
- Sylvain Munaut for help with scan chain improvements
- Mike Thompson and Mitch Bailey for verification 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
- Efabless 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
- Aisler for sponsoring PCB development