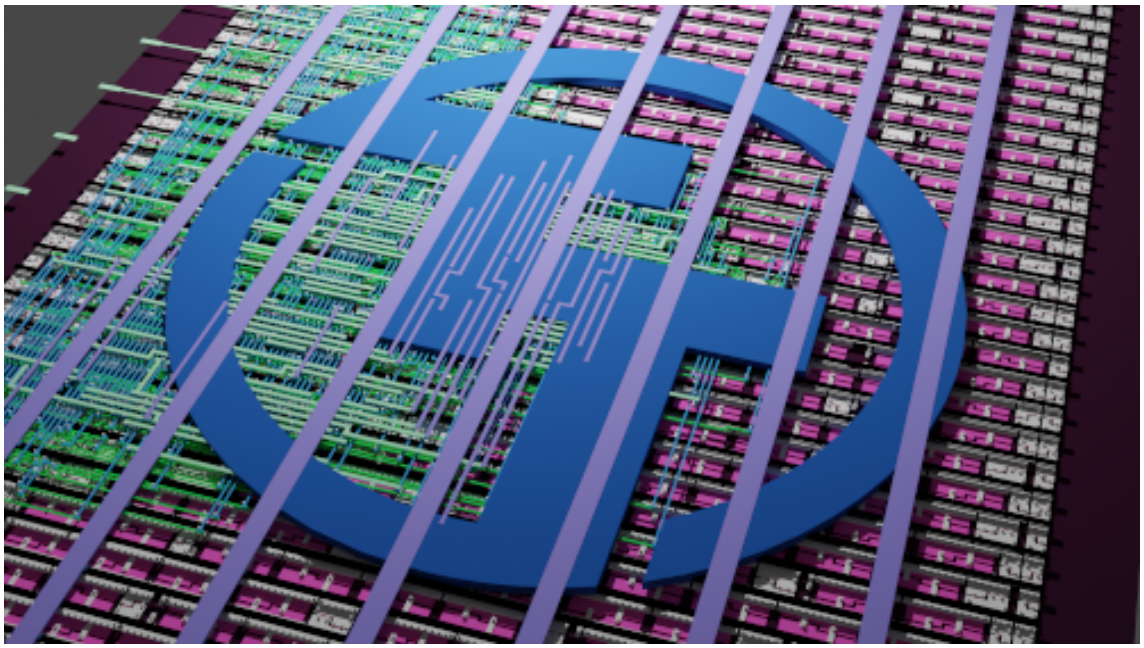

Tiny Tapeout 02 Datasheet

Project Repository
https://github.com/TinyTapeout/tinytapeout-02

November 22, 2022

Contents

Render of whole chip

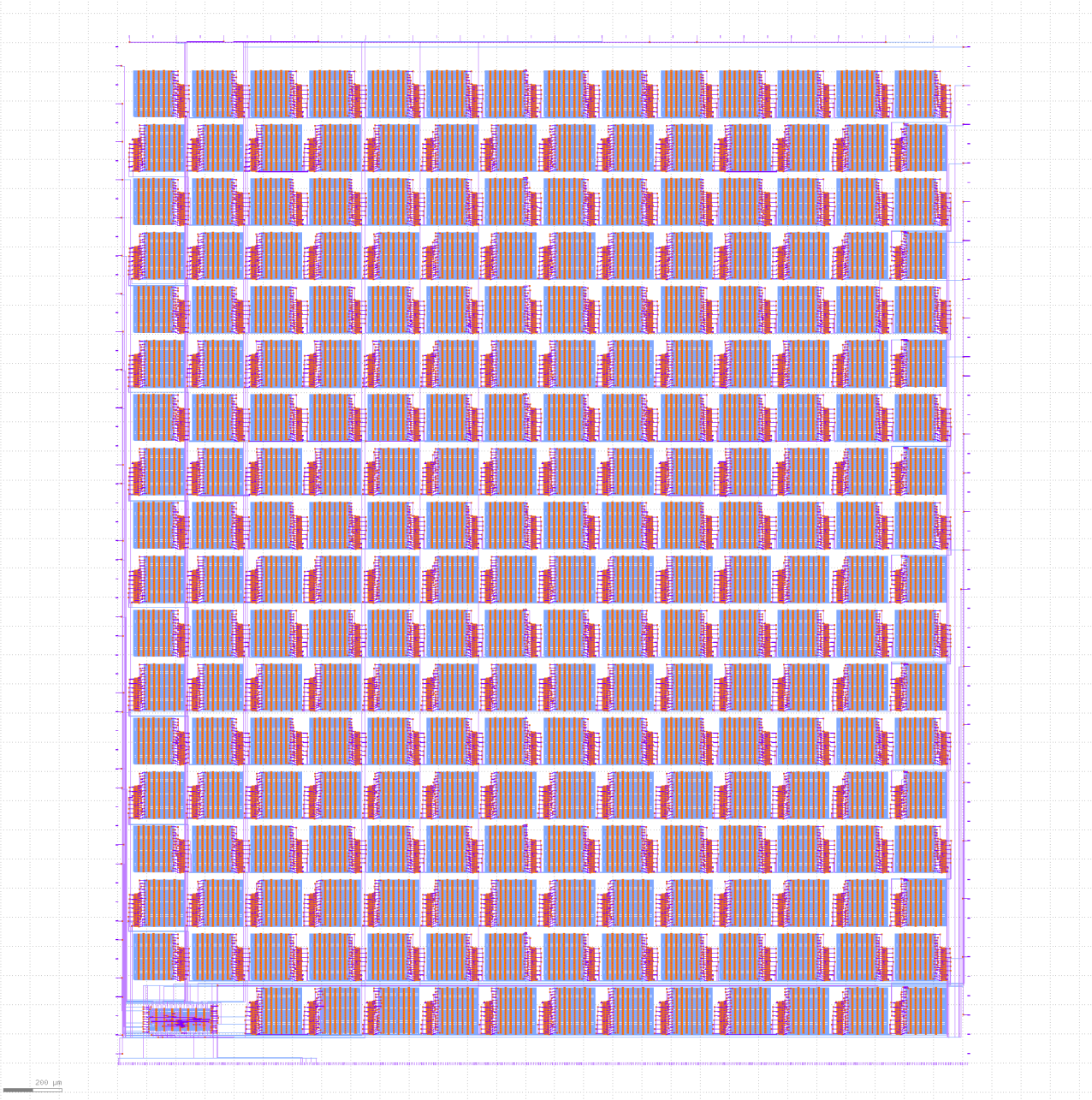

Figure 1: Full GDS

Projects

Test Straight Project

- Author Matt Venn
- Description connects the inputs direct to the outputs
- GitHub repository
- Wokwi
- Extra docs
- Clock 0 Hz
- External hardware

How it works

n/a

How to test

setting each switch to on should set the corresponding led on

IO

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

SIMON Cipher

- Author Fraser Price
- Description Simon32/64 Encryption
- GitHub repository
- HDL
- Extra docs
- Clock 1000 Hz
- External hardware

How it works

Encrypts data by sending it through a feistel network for 32 rounds where it is combined with the round subkey and the last round. Data is entered into the core via shift registers.

How to test

Set shift high and shift data in lsb first, 4 bits at a time. Shift in 96 bits, 32 being data and 64 being the key, with the plaintext being shifted in first. Eg if the plaintext was 32'h65656877 and key was 64'h1918111009080100, then 96'h191811100908010065656877 would be shifted in. Once bits have been shifted in, bring shift low, wait 32 clock cycles then set it high again. The ciphertext will be shifted out lsb first.

IO

#	Input	Output
0	clock	data_out[0]
1	shift	data_out[1]
2	data_in[0]	data_out[2]
3	data_in[1]	data_out[3]
4	data_in[2]	segment e
5	data_in[3]	segment f
6	none	segment g
7	none	none

HD74480 Clock

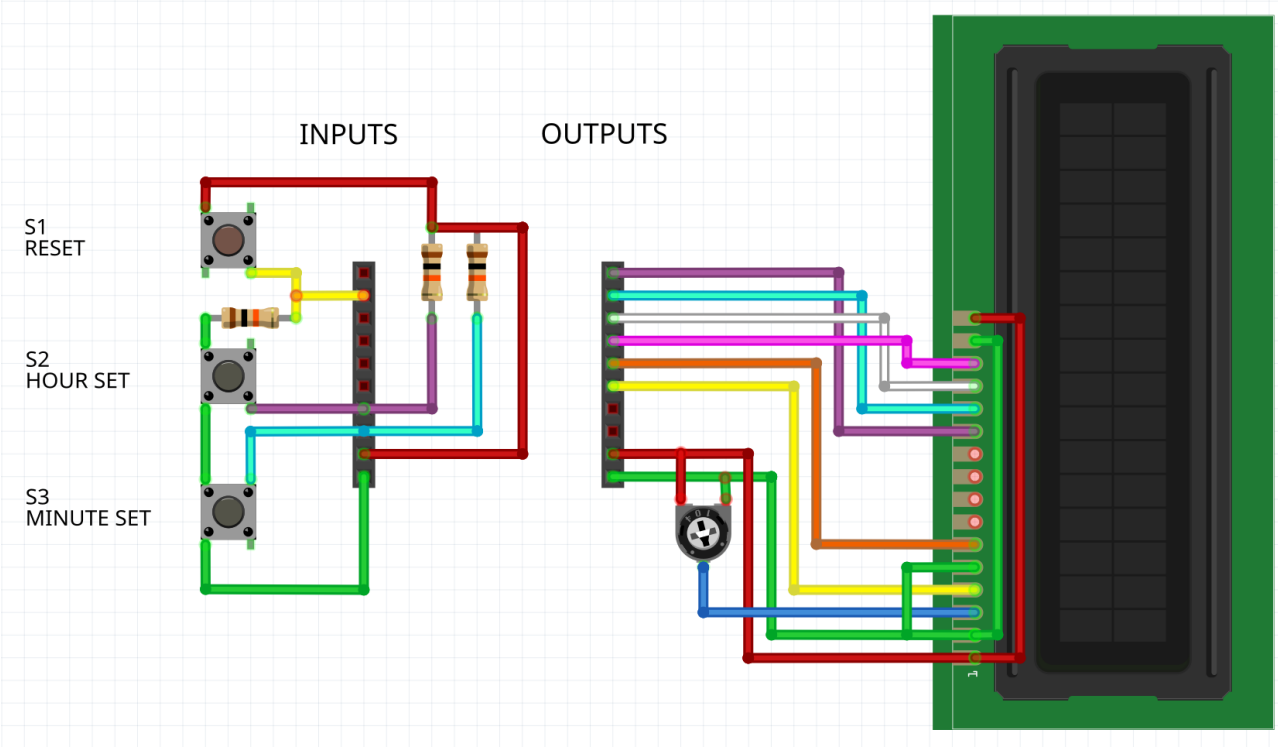

Figure 2: picture

- Author Tom Keddie
- Description Displays a clock on a attached HD74480
- GitHub repository
- HDL
- Extra docs
- Clock 1000 Hz
- External hardware HD74480

How it works

See https://github.com/TomKeddie/tinytapeout-2022-2/blob/main/doc/README.md

How to test

See https://github.com/TomKeddie/tinytapeout-2022-2/blob/main/doc/README.md

IO

#	Input	Output
0	clock	lcd D4

#	Input	Output
1	reset	lcd D5
2	none	lcd D6
3	none	lcd D7
4	none	lcd EN
5	none	lcd RS
6	hour set	none
7	minute set	none

Scrolling Binary Matrix display

- Author Chris
- Description Display scrolling binary data from input pin on 8x8 SK9822 matrix display
- GitHub repository
- HDL
- Extra docs
- Clock 6000 Hz
- External hardware Requires SK9822 display and 3.3V to 5V logic level shifter

How it works

Uses 8x8 matrix SK9822 display to scroll binary data from input pin

How to test

Need 8x8 SK9822 display and level shifter to convert output clock and data logic to 5V logic

IO

#	Input	Output
0	clock	LED Clock
1	reset	LED Data
2	digit	none
3	none	none
4	none	none
5	none	none
6	none	none
7	none	none

Power supply sequencer

- Author Jon Klein
- Description Sequentially enable and disable channels with configurable delay
- GitHub repository
- HDL
- Extra docs
- Clock 12500 Hz
- External hardware None, but could be useful for GaAs amplifiers or other circuits which need sequenced power supplies.

How it works

Counters and registers control and track the state of channel activations. The delay input sets the counter threshold.

How to test

After reset, bring enable high to enable channels sequentially, starting with channel 0. Bring enable low to switch off channels sequentially, starting with channel 7.

IO

#	Input	Output
0	clock	channel 0
1	reset	channel 1
2	enable	channel 2
3	delay0	channel 3
4	delay1	channel 4
5	delay2	channel 5
6	delay3	channel 6
7	delay4	channel 7

Duty Controller

- Author Marcelo Pouso / Miguel Correia
- Description Increase/Decrease a duty cycle of square signal.
- GitHub repository
- HDL
- Extra docs
- Clock 12500 Hz
- External hardware A 12.5Khz clock signal generator and 2 bottoms for incremental and decremental inputs. An oscilloscope to see the output PWM 1.2KHZ signal.

How it works

Enter a square clock of 12.5Khz, and change its duty cycle by pressing increase or decrease bottom. The change will be in steps of 10%. The increase and decrease inputs have an internal debouncer that could be disabled with the input disable_debouncer = 1.

How to test

Connect a signal clock (io_in[0]), reset active high signal (io_in[1]), a button to control the incremental input (io_in[2]) and another button to control the decremental input(io_in[3]), and finally forced to 0 the disable_debouncer input (io_in[4]). The output signal will be in the pwm (io_out[0]) port and the negate output in pwm_neg (io_out[1]). The signal output will have a frecuency of clk/10 = 1.2Khz. When you press the incremental input bottom then the signal will increment by 10% Its duty cycle and when you press the decremental input bottom you will see that the output signal decrement by 10%.

IO

#	Input	Output
0	clock	pwm
1	reset	pwm_neg
2	increase	increase
3	decrease	decrease
4	disable_debouncer	none
5	none	none
6	none	none
7	none	none

S4GA: Super Slow Serial SRAM FPGA

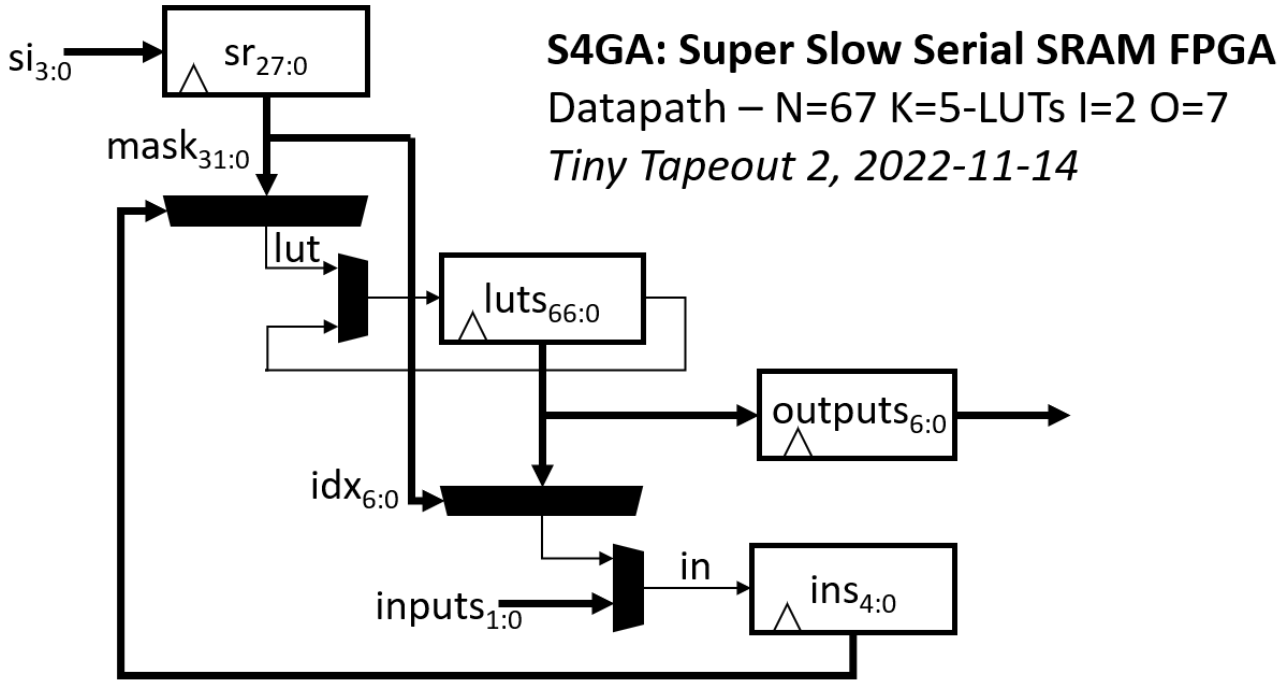

Figure 3: picture

- Author Jan Gray
- Description one fracturable 5-LUT that receives FPGA LUT configuration frames, serially evaluates LUT inputs and LUT outputs
- GitHub repository
- HDL
- Extra docs
- Clock Hz
- External hardware serial SRAM or FLASH

How it works

The design is a single physical LUT into which an external agent pours a series of 72b LUT configuration frames, four bits per cycle. Every 18 clock cycles it evaluates a 5-input LUT. The last N=67 LUT output values are kept on die to be used as LUT inputs of subsequent LUTs. The design also has 2 FPGA input pins and 7 FPGA output pins.

How to test

tricky

IO

#	Input	Output
0	clk	out[0]
1	rst	out[1]
2	si[0]	out[2]
3	si[1]	out[3]
4	si[2]	out[4]
5	si[3]	out[5]
6	in[0]	out[6]
7	in[1]	debug

ALU

- Author Ryan C
- Description 2bit ALU with a ripple carry adder that has the capability to perform 16 different calculations
- GitHub repository
- HDL
- Extra docs
- Clock 0 Hz
- External hardware todo

How it works

todo

How to test

todo

IO

#	Input	Output
0	A1	ALU_Out1
1	A2	ALU_Out2
2	B1	ALU_Out3
3	B2	ALU_Out4
4	ALU_Sel1	ALU_Out5
5	ALU_Sel2	ALU_Out6
6	ALU_Sel3	ALU_Out7
7	ALU_Sel4	CarryOut

The McCoy 6-bit Microprocessor

- Author Aidan Good
- Description Custom RISC-V inspired microprocessor capable of simple arithmatic, branching, and jumps through a custom ISA.
- GitHub repository
- HDL
- Extra docs
- Clock None Hz
- External hardware Any source that allows for 14 GPIO pins. 8 to set the input pins, 6 to read the output pins.

How it works

This chip contains an opcode decoder, 6-bit ALU, 7 general purpose and 3 special purpose 6-bit registers, branch target selector, and other supporting structures all connected together to make a 1-stage microprocessor

How to test

To put the processor in a valid state, hold the reset pin high for one clock cycle. Instructions can begin to be fed into the processor at the beginning of the next cycle when reset is set low. When the clock signal is high, the PC will be output. When the clock signal is low, the x8 register will be output. There are example programs in the testbench folder and a more thourough explaination in the project readme.

IO

#	Input	Output
0	clk	out0
1	reset	out1
2	in0	out2
3	in1	out3
4	in2	out4
5	in3	out5
6	in4	N/A
7	in5	N/A

binary clock

- Author Azdle
- Description A binary clock using multiplexed LEDs
- GitHub repository
- HDL
- Extra docs
- Clock 200 Hz
- External hardware This design expects a matrix of 12 LEDs wired to the outputs. The LEDs should be wired so that current can flow from column to row. Optionally, a real time clock or GPS device with PPS output may be connected to the pps pin for more accurate time keeping. If unused this pin must be pulled to ground.

How it works

Hours, minutes, and seconds are counted in registers with an overflow comparison. An overflow in one, triggers a rising edge on the input of the successive register. The values of each register are connected to the input to a multiplexer, which is able to control 12 LEDs using just 7 of the outputs. This design also allows use of the PPS input for more accurate time keeping. This input takes a 1 Hz clock with a rising edge on the start of each second. The hours[4:0] inputs allow setting of the hours value displayed on the clock when coming out of reset. This can be used for manually setting the time, so it can be done on the hour of any hour. It can also be used by an automatic time keeping controller to ensure the time is perfectly synced daily, for instance at 03:00 to be compatible with DST.

How to test

After reset, the output shows the current Hours:Minutes that have elapsed since coming out of reset, along wit the 1s bit of seconds, multiplexed across the rows of the LED array. The matrix is scanned for values: rows[2:0] = 4'b110; cols[3:0] = 4'bMMMS; rows[2:0] = 4'b101; cols[3:0] = 4'bHHMM; rows[2:0] = 4'b011; cols[3:0] = 4'bHHHH;

(M: Minutes, H: Hours, x: Unused) Directly out of reset, at 0:00, a scan would be: rows[2:0] = 4'b110; cols[3:0] = 4'b0000; rows[2:0] = 4'b101; cols[3:0] = 4'b0000; rows[2:0] = 4'b011; cols[3:0] = 4'b0000;

After one second, at 00:00:01, a scan would be: rows[2:0] = 4'b110; cols[3:0] = 4'b0001; rows[2:0] = 4'b101; cols[3:0] = 4'b0000; rows[2:0] = 4'b011; cols[3:0] = 4'b0000;

After one hour and two minutes, at 1:02, a scan would be: rows[2:0] = 4'b110; cols[3:0] = 4'b0110; rows[2:0] = 4'b101; cols[3:0] = 4'b0100; rows[2:0] = 4'b011; cols[3:0] =

4'b0000;

The above can be sped up using the PPS (Pulse Per Second) input, as long as the PPS pulses are kept to 1 pulse per 2 clock cycles or slower. The hours input can be tested by applying the binary value of the desired hour. Asserting reset for at least one clock cycle, and checking the value of hours displayed in the matrix.

IO

#	Input	Output
0	clock	col 0
1	reset	col 1
2	pps	col 2
3	hours_b1	col 3
4	hours_b2	row 0
5	hours_b4	row 2
6	hours_b8	row 3
7	hours_b16	none

TinySensor

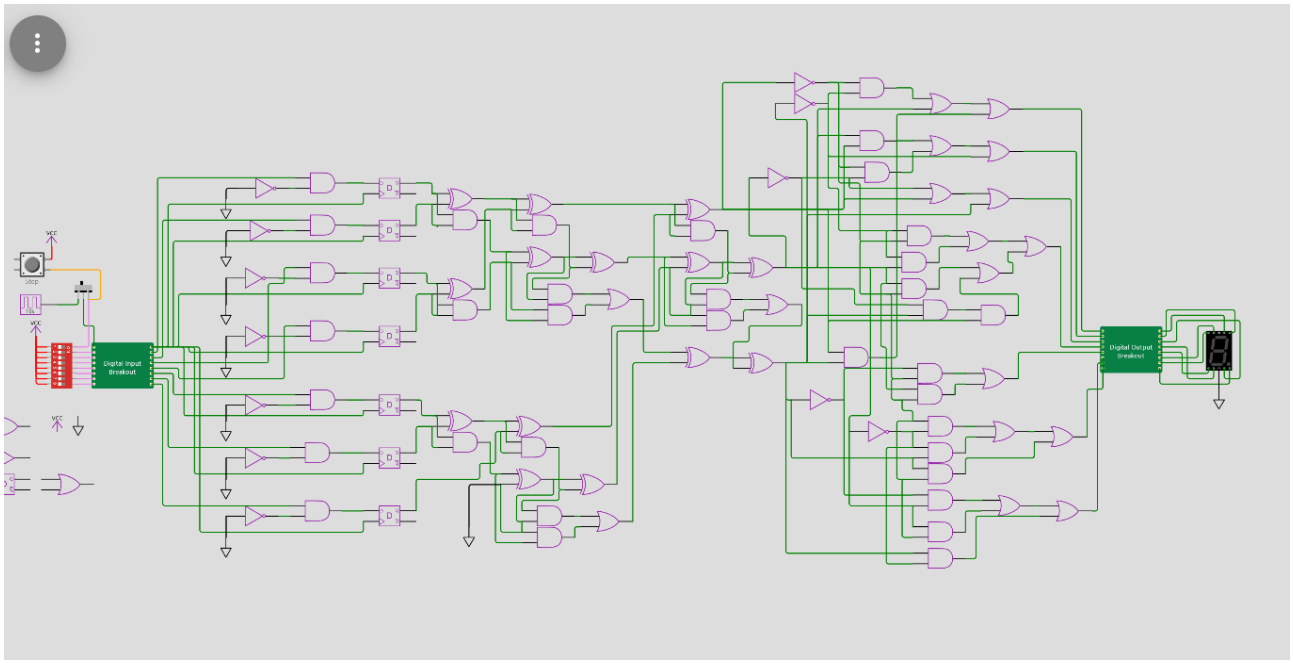

Figure 4: picture

- Author Justin Pelan
- Description Using external hardware photodiodes as inputs, display light intensity on the 7-segment display
- GitHub repository
- Wokwi
- Extra docs
- Clock 0 Hz
- External hardware Breadboard, resistors, photodiodes, specific part# TBD

How it works

inputs 1 - 7 will be connected to external photodiodes to read either a '0' or '1', inputs will be added together and displayed on the 7-segment display

How to test

the dip switches can be used in place of external hw, simply throw the switches and the total number should show up on the 7-segment display

IO

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

8x8 SRAM & Streaming Signal Generator

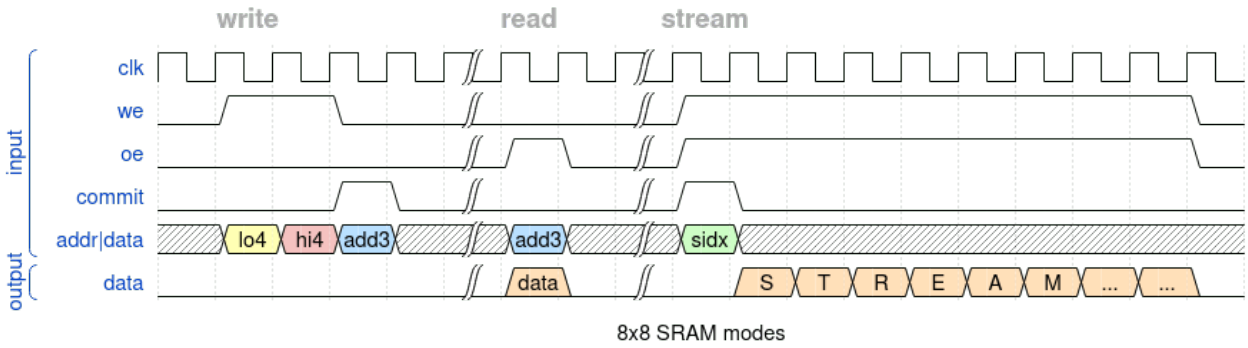

Figure 5: picture

- Author James Ross
- Description Write to, Read from, and Stream 8 addressable 8-bit words of memory
- GitHub repository
- HDL
- Extra docs
- Clock 0 Hz
- External hardware

How it works

WRITE MODE: Write Enable (WE) pin high while passing 4-bits low data, 4-bits high data into an 8-bit temporary shift register. After loading data into the temporary shift register, setting Commit high while passing a 3-bit address places the register value into memory. Fast memset, such as zeroing memory, can be performed with Commit high while passing a new address per clock cycle. *READ MODE*: While Output Enable (OE) high, a 3-bit address places the data from memory into the temporary register returns 8-bit register to output data interface. *STREAM MODE*: While WE, OE, and Commit high, pass the starting stream index address. Then, while WE and OE are both high, the output cycles through all values in memory. This may be used as a streaming signal generator. *ALL MODES*: The highest bit of the address or stream index is ignored. Regrettably, 16 bytes doesn't fit in Tiny Tapeout #2.

How to test

After reset, you can write values into memory and read back. See the verilator testbench.

IO

#	Input	Output
0	clk	data[0]
1	we	data[1]
2	oe	data[2]
3	commit	data[3]
4	addr[0]/high[0]/low[0]	data[4]
5	addr[1]/high[1]/low[1]	data[5]
6	addr[2]/high[2]/low[2]	data[6]
7	xxxxxxx/high[3]/low[3]	data[7]

German Traffic Light State Machine

- Author Jens Schleusner
- Description A state machine to control german traffic lights at an intersection.
- GitHub repository
- Wokwi
- Extra docs
- Clock 1 Hz
- External hardware An additional inverter is required to generate the pedestrian red signals from the green output. Hookup your own LEDs for the signals.

How it works

A state machine generates signals for vehicle and pedestrian traffic lights at an intersection of a main street and a side street. A blinking yellow light for the side street is generated in the reset state.

How to test

Provide a clock, hook up LEDs and generate a reset signal to reset the intersection to all-red. If your leave the reset signal enabled, a blinking yellow light is shown for the side street.

IO

#	Input	Output
0	clock	main street red
1	reset	main street yellow
2	none	main street green
3	none	main street pedestrian green
4	none	side street red
5	none	side street yellow
6	none	side street green
7	none	side street pedestrian green

4-spin Ising Chain Simulation

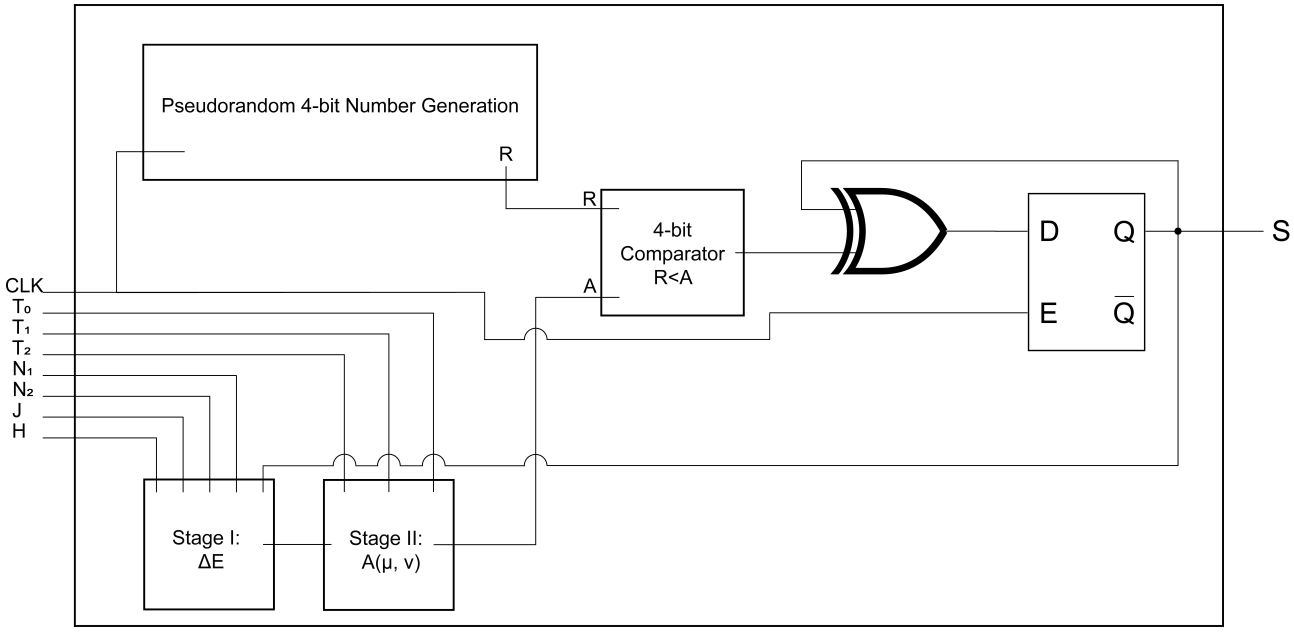

Figure 6: picture

- Author Seppe Van Dyck
- Description A self-contained physics simulation. This circuit simulates 4 spins of an Ising chain in an external field.
- GitHub repository
- Wokwi
- Extra docs
- Clock 20 Hz
- External hardware

How it works

It runs the Metropolis-Hastings monte-carlo algorithm to simulate 4 Ising spins in a linear chain with two external neighbours and an external field. Every monte-carlo step (10 clock cycles) a random number is created through a 32-bit LFSR and is compared to an 8-bit representations of the acceptance probability of a random spin flip.

How to test

When you only enable one of the neighbours and leave everything else 0, the system will evolve into a ground state with every other spin pointing up.

IO

#	Input	Output
0	clock, clock input.	segment a, Spin 0.
1	T0, LSB of the 3-bit temperature representation.	segment b, Spin 1.
2	T1, Middle bit of the 3-bit temperature.	segment c, Spin 2.
3	T2, MSB of the 3-bit temperature.	segment d, Spin 3.
4	N1, Value of neighbour 1 (up/1 or down/0).	segment e, Neighbour 2.
5	N2, Value of neighbour 2 (up/1 or down/0).	segment f, Neighbour 1.
6	J, The sign of the NN coupling constant J (+/1 or -/0).	none
7	H, Value of the coupling to the external field H (1 or 0).	segment h, MC Step Indicator.

Avalon Semiconductors '5401' 4-bit Microprocessor

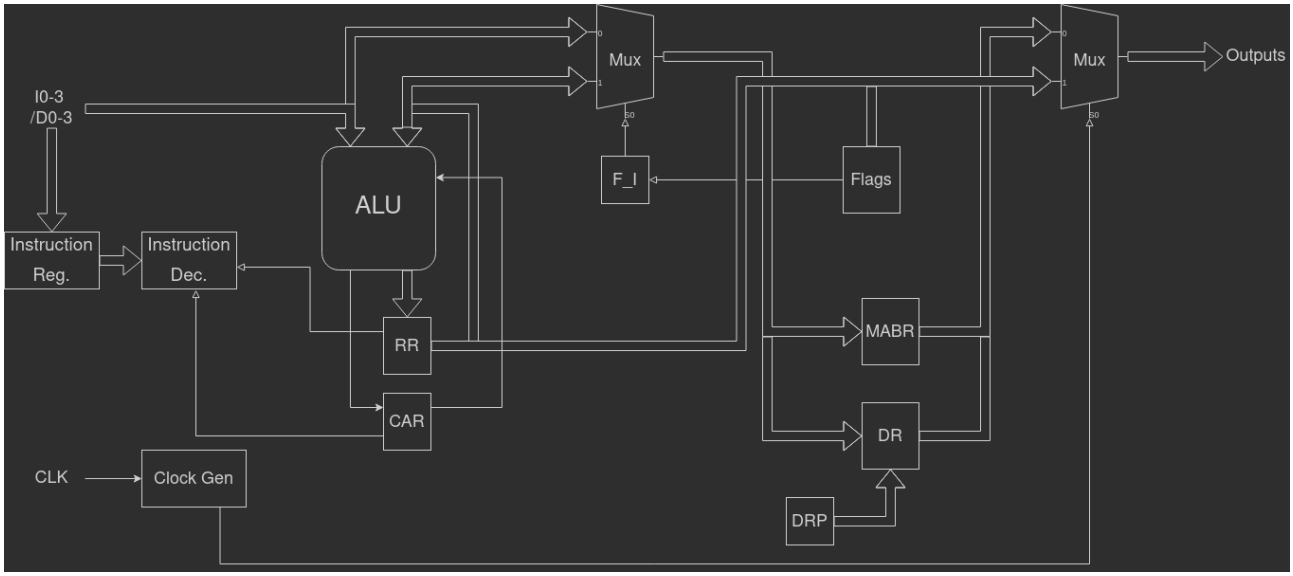

Figure 7: picture

- Author Tholin
- Description 4-bit CPU capable of addressing 4096 bytes program memory and 254 words data memory, with 6 words of on-chip RAM and two general-purpose input ports. Hopefully capable of more complex computation than previous CPU submissions.
- GitHub repository
- HDL
- Extra docs
- Clock 6000 Hz
- External hardware At the very minimum a program memory, and the required glue logic. See "Example system diagram" in the full documentation.

How it works

The chip contains a 4-bit ALU, a 4-bit Accumulator, 8-bit Memory Address Register and 12-bit "Destination Register", which is used to buffer branch target addresses. It also has two general-purpose input ports. The instruction set consists of 16 instructions, containing arihmatic, logical, load/store, branch and conditional branch instruction. See the full documentation for a complete architectural description.

How to test

It is possible to test the CPU using a debounced push button as the clock, and using the DIP switches on the PCB to key in instructions manually. Set the switches to 0100_0000 to assert RST, and pulse the clock a few times. Then, change the switches

to 0000_1000 (SEI instruction) and pulse the clock four times. After that, set the switches to all 0s (LD instruction). Pulse the clock once, then change the switches to 0001_0100, and pulse the clock three more times. Lastly, set the switches to 0011_1100 (LMH instruction). If done correctly, after two pulses of the clock, the outputs will read 0101_0000 and two more pulses after that, they will be xxxx_1000 ('x' means don't care). This sequence should repeat for as long as you keep pulsing the clock, without changing the inputs.

IO

#	Input	Output
0	CLK	MAR0 / DR0 / DR8 / RR0
1	RST	MAR1 / DR1 / DR9 / RR1
2	I0 / D0	MAR2 / DR2 / DR10 / RR2
3	I1 / D1	MAR3 / DR3 / DR11 / RR3
4	I2 / D2	MAR4 / DR4 / F_MAR
5	I3 / D3	MAR5 / DR5 / F_WRITE
6	EF0	MAR6 / DR6 / F_JMP
7	EF1	MAR7 / DR7 / F_I

small FFT

- Author Rice Shelley
- Description Computes a small fft
- GitHub repository
- HDL
- Extra docs
- Clock 1000 Hz
- External hardware

How it works

Takes 4 4-bit signed inputs (real integer numbers) and outputs 4 6-bit complex numbers

How to test

after reset, use the write enable signal to write 4 inputs. Read the output for the computer FFT.

IO

#	Input	Output
0	clock	rd_idx_zero
1	reset	none
2	wrEn	data_out_0
3	none	data_out_1
4	data_in_0	data_out_2
5	data_in_1	data_out_3
6	data_in_2	data_out_4
7	data_in_3	data_out_5

Stream Integrator

- Author William Moyes
- Description A silicon implementation of a simple optical computation
- GitHub repository
- Wokwi
- Extra docs
- Clock 0 Hz
- External hardware

How it works

It is possible to generate a pseudorandom bit sequence optomechanically using four loops of punched paper tape. Each of the four tape loops, labeled A, B, C, and D, encodes one bit of information per linear position using a tape specific hole pattern. The patterns are TapeA_0=XOOO, TapeA_1=OXOO, TapeB_0=OOXO, TapeB_1=OOOX, TapeC_0=OOXX, TapeC_1=XXOO, TapeD_0=OXOX, TapeD_1=XOXO, where O is a hole, and X is filled. The pseudorandom sequence is obtained by physically stacking the four tapes together at a single linear point, and observing if light can pass through any of the four hole positions. If all four hole positions are blocked, a 0 is generated. If any of the four holes allows light to pass, a 1 is generated. The next bit is obtained by advancing all four tapes by one linear position and repeating the observation. By using the specified bit encoding patterns, the expression (C ? A : B) ^ D is calculated. If all four tapes are punched with randomly chosen 1 and 0 patterns, and each tape's length is relatively prime to the other tape lengths, then a maximum generator period is obtained. This TinyTapeout-02 minimal project was inspired by the paper tape pseudorandom bit sequence generator. It implements the core (C ? A : B) ^ D operation electrically instead of optomechanically. An extra ^ E term is added for ease of use.

How to test

Run through the 32 possible input patterns, and verify the expected output value is observed. Counting from 00000 to 111111, where IN0 is the LSB (i.e. Tape A), and IN4 (i.e. Extra E) is the MSB should yield the pattern: 01010011101011001010110001010011.

IO

#	Input	Output
0	Value from Tape A	Output
1	Value from Tape B	none

#	Input	Output
2	Value from Tape C	none
3	Value from Tape D	none
4	Extra term XORed with generator output	none
5	none	none
6	none	none
7	none	none

tiny-fir

- Author Tom Schucker
- Description 4bit 2-stage FIR filter
- GitHub repository
- Wokwi
- Extra docs
- Clock 0 Hz
- External hardware Arduino or FPGA

How it works

Multiplies the input by the tap coefficient for each stage and outputs the sum of all stages

How to test

Load tap coefficients by setting the value and pulsing 2 times, then repeat for second tap. Change input value each clock to run filter. Select signals change output to debug 00(normal) 01(output of mult 2) 10(tap values in mem) 11(output of mult 1). FIR output discards least significant bit due to output limitations

IO

#	Input	Output
0	clock	fir1/mult0/tap10
1	data0/tap0	fir2/mult1/tap11
2	data1/tap1	fir3/mult2/tap12
3	data2/tap2	fir4/mult3/tap13
4	data3/tap3	fir5/mult4/tap20
5	select0	fir6/mult5/tap21
6	select1	fir7/mult6/tap22
7	loadpulse	fir8/mult7/tap23

Configurable SR

- Author Greg Steiert
- Description Two configurable gates driving an SR flip-flop
- GitHub repository
- Wokwi
- Extra docs
- Clock 0 Hz
- External hardware none

How it works

Connect to each of the configurable gates to control the SR FF

How to test

set up inputs like 1G99 configurable gate to drive SR FF

IO

#	Input	Output
0	A0	Amux
1	A1	Axor
2	Asel	SR_Q
3	Ainv	D_Q
4	B0	Bmux
5	B1	Bxor
6	Bsel	SR_Qbar
7	Binv	D_Qbar

LUTRAM

- Author Luis Ardila
- Description LUTRAM with 4 bit address and 8 bit output preloaded with a binary to 7 segments decoder, sadly it was too big for 0-F, so now it is 0-9?
- GitHub repository
- Wokwi
- Extra docs
- Clock 0 Hz
- External hardware

How it works

uses the address bits to pull from memory the value to be displayed on the 7 segments, content of the memory can be updated via a clock and data pins, reset button will revert to default info, you would need to issue one clock cycle to load the default info

How to test

clk, data, rst, nc, address [4:0]

IO

#	Input	Output
0	clock	segment a
1	data	segment b
2	reset	segment c
3	nc	segment d
4	address bit 3	segment e
5	address bit 2	segment f
6	address bit 1	segment g
7	address bit 0	segment pd

chase the beat

- Author Emil J Tywoniak
- Description Tap twice to the beat, the outputs will chase the beat. Or generate some audio noise!
- GitHub repository
- HDL
- Extra docs
- Clock 1000 Hz
- External hardware A button on the tap input, a switch on the mode input, LEDs on the 8 outputs, and audio output on the first output. Don't just connect headphones or speakers directly! It could fry the circuit, you need some sort of amplifier.

How it works

The second button press sets a ceiling value for the 1kHz counter. When the counter hits that value, it barrel-shifts the outputs by one bit. When the mode pin isn't asserted, the first output pin emits digital noise generated by a LFSR

How to test

Set 1kHz clock on first input. After reset, set mode to 1, tap the tap button twice within one second. The outputs should set to the beat

IO

#	Input	Output
0	clk	o_0 - LED or noise output
1	rst	o_1 - LED
2	tap	o_2 - LED
3	mode	o_3 - LED
4	none	o_4 - LED
5	none	o_5 - LED
6	none	o_6 - LED
7	none	o_7 - LED

BCD to 7-segment encoder

- Author maehw
- Description Encode binary coded decimals (BCD) in the range 0..9 to 7-segment display control signals
- GitHub repository
- Wokwi
- Extra docs
- Clock 0 Hz
- External hardware Input switches and 7-segment display (should be on the PCB)

How it works

The design has been fully generated using https://github.com/maehw/wokwi-lookup-table-generator using a truth table (https://github.com/maehw/wokwi-lookup-table-generator/blob/main/demos/bcd_7segment_lut.logic.json). The truth table describes the translation of binary coded decimal (BCD) numbers to wokwi 7-segment display (https://docs.wokwi.com/parts/wokwi-7segment). Valid BCD input values are in the range 0..9, other values will show a blank display.

How to test

Control the input switches on the PCB and check the digit displayed on the 7-segment display.

IO

#	Input	Output
0	w	segment a
1	x	segment b
2	y	segment c
3	z	segment d
4	none	segment e
5	none	segment f
6	none	segment g
7	none	none

4-bit Multiplier

- Author Fernando Dominguez Pousa
- Description 4-bit Multiplier based on single bit full adders
- GitHub repository
- HDL
- Extra docs
- Clock 2500 Hz
- External hardware Clock divider to 2500 Hz. Seven segment display with dot led. 8-bit DIP Switch

How it works

Inputs to the multiplier are provided with the switch. As only eight inputs are available including clock and reset, only three bits remain available for each multiplication factor. Thus, a bit zero is set as the fourth bit. The output product is showed in the 7 segment display. Inputs are registered and a product is calculated. As output is 8-bit number, every 500ms a number appears. First the less significant 4 bits, after 500ms the most significant. When less significant 4-bits are displayed, the led dot including in the display is powered on.

How to test

HDL code is tested using Makefile and cocotb. 4 set of tests are included: the single bit adder, the 4-bit adder, the 4-bit multiplier and the top design. In real hardware, the three less significant bits can create a number times the number created with the next three bits. Reset is asserted with the seventh bit of the switch.

IO

#	Input	Output
0	clock	segment_1 (o_segments[0])
1	reset	segment_2 (o_segments[1])
2	i_factor_a[0]	segment_3 (o_segments[2])
3	i_factor_a[1]	segment_4 (o_segments[3])
4	i_factor_a[2]	segment_5 (o_segments[4])
5	i_factor_b[3]	segment_6 (o_segments[5])
6	i_factor_b[4]	segment_7 (o_segments[6])
7	i_factor_b[5]	segment_dot (o_lsb_digit)

Avalon Semiconductors 'TBB1143' Programmable Sound Generator

- Author Tholin
- Description Sound generator with two basic, square-wave voices. Can also be used as a general-purpose frequency generator.
- GitHub repository
- HDL
- Extra docs
- Clock 6000 Hz
- External hardware Two speakers, or a suitable way to combine the audio signals, as well as a microprocessor or microcontroller to program the 1143.

How it works

Either tone generator simply takes the input clock frequency, multiplied by 128 and divides it by 8 times the generator's divisor setting. It does this by using a ring oscillator to generate a faster internal clock to be able to generate a wider range of tones. Of course, the outputs are stil only updated as fast as the scan chain allows.

How to test

It is possible to use the DIP switches to program the generator according to the documentation. Writing 1101 into address 1, 1010 into address 2, 0000 into address 3 and finally 0001 into address 7 will cause a ~440Hz tone to appear on SOUT0.

IO

#	Input	Output
0	CLK	SOUT0
1	RST	SOUT1
2	D0	NC
3	D1	NC
4	D2	NC
5	D3	NC
6	A0	NC
7	WRT	NC

Transmit UART

- Author Tom Keddie
- Description Transmits a async serial string on a pin
- GitHub repository
- HDL
- Extra docs
- Clock 1200 Hz
- External hardware Serial cable

How it works

See https://github.com/TomKeddie/tinytapeout-2022-2a/blob/main/doc/README.md

How to test

See https://github.com/TomKeddie/tinytapeout-2022-2a/blob/main/doc/README.md

IO

#	Input	Output
0	clock	uart_tx_1
1	reset	none
2	none	none
3	none	none
4	none	none
5	none	none
6	none	none
7	none	none

RGB LED Matrix Driver

- Author Matt M
- Description Drives a simple animation on SparkFun's RGB LED 8x8 matrix backpack
- GitHub repository
- HDL
- Extra docs
- Clock 6250 Hz
- External hardware RGB LED matrix backpack from SparkFun: https://www.sparkfun.com/

How it works

Implements an SPI master to drive an animation with overlapping green/blue waves and a moving white diagonal. Some 7-segment wires are used for a 'sanity check' animation.

How to test

Wire accordingly and use a clock up to 12.5 KHz. Asynchronous reset is synchronized to the clock.

IO

#	Input	Output
0	clock	SCLK
1	reset	MOSI
2	none	segment c
3	none	segment d
4	none	segment e
5	none	nCS
6	none	segment g
7	none	none (always high)

Tiny Phase/Frequency Detector

- Author argunda
- Description Detect phase shifts between 2 square waves.
- GitHub repository
- Wokwi
- Extra docs
- Clock 0 Hz
- External hardware Signal generators for square wave inputs.

How it works

This is one of the blocks of a phased locked loop. The inputs are a reference clock and feedback clock and the outputs are the phase difference on either up or /down pin.

How to test

If the phase of the feedback clock is leading the reference clock, the up signal should show the phase difference. If it's lagging, the down signal will show the difference.

IO

#	Input	Output
0	reference clock	up
1	feedback clock	(inverted) down
2	active-low reset	none
3	none	none
4	none	none
5	none	none
6	none	none
7	none	none

Loading Animation

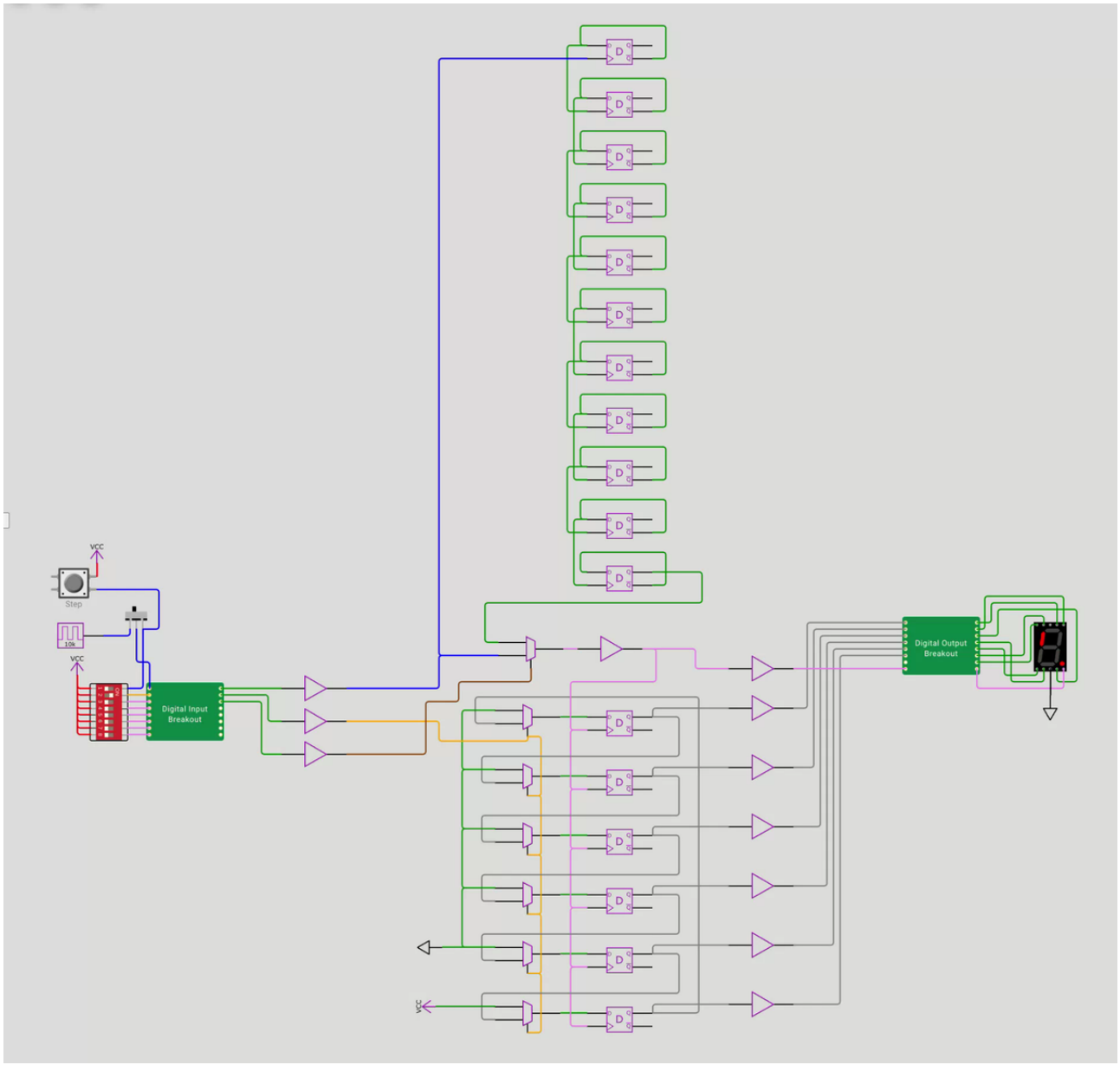

Figure 8: picture

- Author Andre & Milosch Meriac
- Description Submission for tt02 - Rotating Dash
- GitHub repository
- Wokwi
- Extra docs
- Clock 10000 Hz
- External hardware Default PCB

How it works

Slide switch to external clock. All DIP switches to off. DIP2 (Reset) on to run (Reset is low-active). By switching DIP3 (Mode) on and setting the sliding switch to Step-Button, the Step-Button can be now used to animate step by step.

How to test

Slide switch to external clock. All DIP switches to off. DIP2 (Reset) on to run (Reset is low-active).

IO

#	Input	Output
0	clock	segment a
1	reset	segment b
2	mode	segment c
3	none	segment d
4	none	segment e
5	none	segment f
6	none	none
7	none	none

tiny egg timer

- Author yubex
- Description tiny egg timer is a configurable small timer
- GitHub repository
- HDL
- Extra docs
- Clock 10000 Hz
- External hardware no external hw requrired

How it works

Its a simple FSM with 3 states (Idle, Waiting and Alarm) and counters regarding clk_cycles, seconds and minutes...

How to test

Set the clock to 10kHz, set the wait time you want (in minutes) by setting io_in[7:3], set the start switch to 1, the timer should be running, the dot of the 7segment display should toggle each second. If the time is expired, an A for alarm should be displayed. You can stop the alarm by setting the start switch to 0 again.

IO

#	Input	Output
0	clock	segment a
1	reset	segment b
2	start	segment c
3	wait time in minutes [0]	segment d
4	wait time in minutes [1]	segment e
5	wait time in minutes [2]	segment f
6	wait time in minutes [3]	segment g
7	wait time in minutes [4]	dot

Potato-1 (Brainfuck CPU)

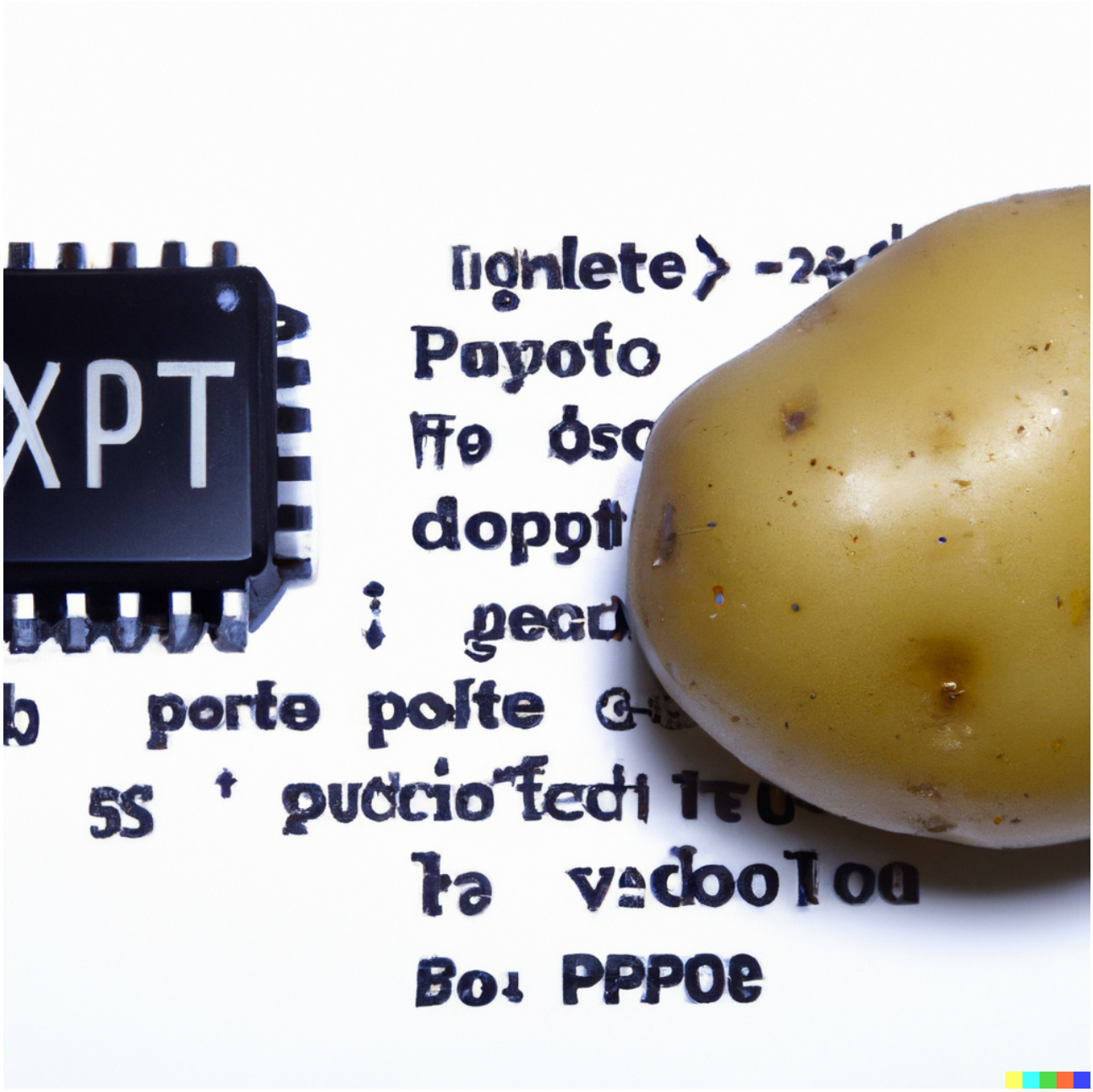

Figure 9: picture

- Author Pepper Gray (they/them)
- Description Potato-1 is part of a Brainfuck CPU. It is only the control logic, i.e. you have to bring your own registers, memory controller and io logic. It is very simple, hence likely very slow: You could probably run your brainfuck code on a potato and it would be equally fast, hence the name. The project picture was generated using DALL · E.
- GitHub repository
- HDL

- Extra docs
- Clock 12500 Hz
- External hardware Bidirectional Counter (3x)
  - program counter
  - data pointer
  - value ROM (addressed via programm counter) RAM (addressed via data pointer, all bytes must be zero after reset)

some TTL gates, e.g. to configure that the value is written to RAM every time it is changed or the data pointer is changed

How it works

Each rising edge the CU will read in the instruction, zero flag and IO Wait flag and process it. Each falling edge the output pins will be updated. The output pins indiciate which action to take, i.e. which registers to increment/decrement. If Put or Get pin is set, the CU will pause execution until IO Wait is unset. If IO Wait is already unset, the CU will immidiatly execute the next command without waiting.

Additionaly to the 8 original brainfuck instructions there is a HALT instruction to stop execution and a NOP instructions to do nothing, also there are unused instruction (some of them may be used to extend the instruction set in a later itteration).

Instructions: 0000 > Increment the data pointer 0001 < Decrement data pointer 0010 + Increment value 0011 - Decrement value 0100 . Write value 0101 , Read value 0110 [Start Loop (enter if value is non-zero, else jump to matchin ']') 0111] End Loop (leave if value is zero, , else jump to matchin'[') 1000 NOP No Operation 1111 HALT Halt Execution

How to test

Reset: Set Reset_n=0 and wait one clockycycle

Run: Set Reset_n=1

Simple Test: - all input pins zero - clock cycle - Reset_n high - clock cylce –> PC++ high, all outer outputs are low

Check test/test.py for small scripts to verify the CU logic

IO

#	Input	Output
0	Clock	PC++
1	Reset_n	PC–
2	IO Wait	X++
3	Zero Flag	X–
4	Instruction[0]	A++
5	Instruction[1]	A–
6	Instruction[2]	Put
7	Instruction[3]	Get

heart zoe mom dad

- Author zoe nguyen. taylor
- Description outputs my name and my age (zoe 4)
- GitHub repository
- HDL
- Extra docs
- Clock 1000 Hz
- External hardware

How it works

spells leters

How to test

shift 1 hot value

IO

#	Input	Output
0	Z	segment a
1	O	segment b
2	E	segment c
3	F	segment d
4	none	segment e
5	none	segment f
6	none	segment g
7	none	none

Tiny Synth

picture

- Author Nanik Adnani
- Description A tiny synthesizer! Modulates the frequency of the clock based on inputs, plays a C scale (hopefully).
- GitHub repository
- Wokwi
- Extra docs
- Clock 12500 Hz
- External hardware Not entirely sure yet, it outputs a square wave, I still need to figure out what to do with it to make it make sound.

How it works

Will come back and write more after my exams!

How to test

Make sure the clock is tied to input 0, whatever frequency you want, play with it! Then you can play different notes by toggling the other inputs.

IO

#	Input	Output
0	clock	square wave out
1	C	none
2	D	none
3	E	none
4	F	none
5	G	none
6	A	none
7	B	none

5-bit Galois LFSR

- Author Michael Bikovitsky
- Description 5-bit Galois LFSR with configurable taps and initial state. Outputs a value every second.
- GitHub repository
- HDL
- Extra docs
- Clock 1000 Hz
- External hardware

How it works

https://en.wikipedia.org/wiki/Linear-feedback_shift_register#Galois_LFSRs

How to test

1. Set the desired taps using the switches
2. Assert the reset_taps pin
3. Deassert reset_taps
4. Set the desired initial state
5. Assert reset_lfsr
6. Deassert reset_lfsr
7. Look at it go!
    - Values between 0x00-0x0F are output as hex digits.
    - Values between 0x10-0x1F are output as hex digits with a dot.

IO

#	Input	Output
0	clock	data_out1
1	reset_lfsr	data_out2
2	reset_taps	data_out3
3	data_in1	data_out4
4	data_in2	data_out5
5	data_in3	data_out6
6	data_in4	data_out7
7	data_in5	data_out8

prbs15

- Author Tom Schucker
- Description generates and checks prbs15 sequences
- GitHub repository
- Wokwi
- Extra docs
- Clock 0 Hz
- External hardware logic analyzer and jumper leads

How it works

uses lfsr to generate and check prbs15 sequence

How to test

running clk, gnd pin1, set enable high. feedback prbs15 output to check, monitor error for pulses

IO

#	Input	Output
0	clock	clk
1	gnd	prbs15
2	enable	error
3	check	checked
4	none	none
5	none	none
6	none	none
7	none	none

4-bit badge ALU

- Author Rolf Widenfelt
- Description A 4-bit ALU inspired by Supercon.6 badge
- GitHub repository
- HDL
- Extra docs
- Clock 0 Hz
- External hardware

How it works

finite state machine with combinational logic (in verilog)

How to test

cocotb

IO

#	Input	Output
0	clk	none
1	rst	none
2	ctl	none
3	none	cout
4	datain3	alu3
5	datain2	alu2
6	datain1	alu1
7	datain0	alu0

Illegal Logic

- Author James Ross
- Description This circuit logic emits a sequences of 32 hexadecimal digits to the 7 segment display which may be considered illegal in certain jurisdictions. The sequence is considered to be an illegal number.
- GitHub repository
- HDL
- Extra docs
- Clock 0 Hz
- External hardware The 7-segment display is used directly.

How it works

There is some combinatorial logic which generates 32 numbers and then decodes those digits to the 7 segment display.

How to test

First, raise the reset I/O (io_in[1]) high. Then Output Enable (io_in[2]) high for 32 cycles.

IO

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

Siren

- Author Alan Green
- Description Pretty patterns and a siren straight from the 1970s
- GitHub repository
- Wokwi
- Extra docs
- Clock 12500 Hz
- External hardware For the audio output on pin 7, either use an audio amplifier or, if bravely connecting a speaker directly, place a resistor in series to limit the current.

How it works

A long chain of D flip flops divides down the clock to produce a range of frequencies that are used for various purposes. Some of the higher frequencies are used to produce the tones. Lower frequencies are used to control the patterns of lights and to change the tones being sent to the speaker. An interesting part of the project is a counter that counts to 5 and resets to zero. This is used for one of the two patterns of lights, where the period of pattern is six.

How to test

Connect a speaker to the last digital output pin, the one which is also connected to the decimal point on the seven segment display. Switch 8 is used to select between two groups of patterns.

IO

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

YaFPGA

- Author Frans Skarman
- Description Yet another FPGA
- GitHub repository
- HDL
- Extra docs
- Clock 1000 Hz
- External hardware

How it works

TODO

How to test

TODO

IO

#	Input	Output
0	clock	output0
1	input1	output1
2	input2	output2
3	input3	output3
4	input4	none
5	config data	none
6	config clock	none
7	none	none

M0

- Author William Moyes
- Description todo
- GitHub repository
- HDL
- Extra docs
- Clock 12000 Hz
- External hardware

How it works

todo

How to test

todo

IO

#	Input	Output
0	clock	spi_cs0
1	reset	spi_cs1
2	spi_miso	spi_clk
3	none	spi_mosi
4	none	none
5	none	none
6	none	none
7	none	none

bitslam

- Author Jake "ferris" Taylor
- Description bitslam is a programmable sound chip with 2 LFSR voices.
- GitHub repository
- HDL
- Extra docs
- Clock 6000 Hz
- External hardware A 4-bit DAC connected to the four digital output pins.

How it works

bitslam is programmed via its register write interface. A register write is performed by first writing an internal address register, which selects which register will be written to, and then writing a value. Bit 1 distinguishes address writes (0) or data writes (1). Address bits 1-2 address different internal modules: 00 addresses voice 0, 01 addresses voice 1, and 10 addresses the mixer. Address bit 0 addresses a register in the internal module. Each voice is controlled by a clock divider and a tap mask for the LFSR state. The clock divider is at address 0 and controls the rate at which the LFSR is updated, effectively controlling the pitch. Since bitslam is (expected to be) clocked at 6khz, the pitch will be determined by 3khz / x where x is the 6-bit clock divider value. Each voice also contains a 4-bit LFSR tap mask (address 1) which determines which of 4 LFSR bits are XOR'd together to determine the new LFSR LSB. The LFSR is 10 bits wide and the tap mask bits correspond to positions 1, 4, 6, and 9, respectively. The mixer has a single register to control the volume of each voice. Bits 0-2 determine voice 0 volume, and bits 3-5 determine voice 1 volume. A value of 0 means a voice is silent, and a value of 7 is full volume. Special thanks to Daniel "trilader" Schulte for pointing out a crucial interconnect bug.

How to test

bitslam is meant to be driven and clocked by an external host, eg. a microcontroller. The microcontroller should use the register write interface described above to program the desired audio output (eg. to play a song or sound effects) and 4-bit digital audio should be generated on the 4 digital out pins.

IO

#	Input	Output
0	clock	digital out 0
1	address/data selector	digital out 1
2	address/data 0	digital out 2

#	Input	Output
3	address/data 1	digital out 3
4	address/data 2	none
5	address/data 3	none
6	address/data 4	none
7	address/data 5	none

8x8 Bit Pattern Player

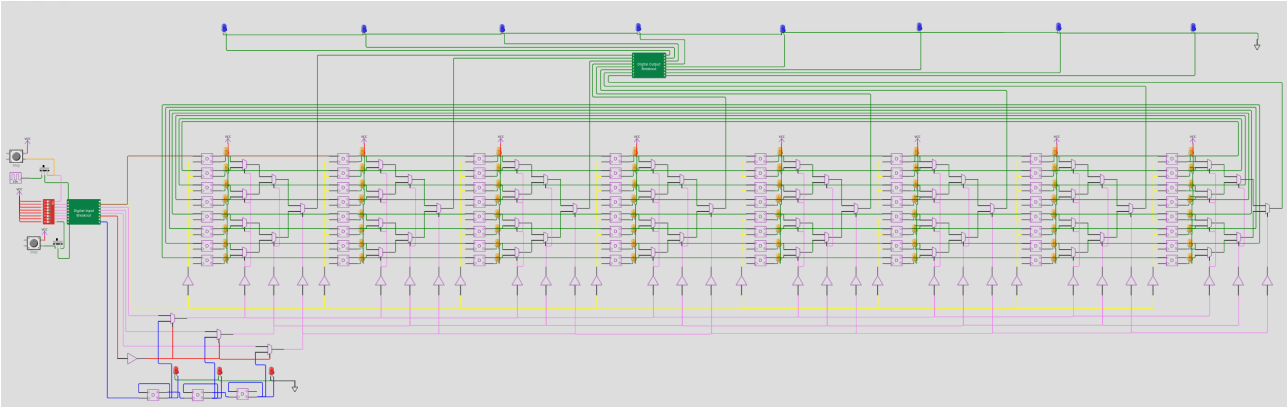

Figure 10: picture

- Author Thorsten Knoll
- Description 8x8 bit serial programmable, addressable and playable memory.
- GitHub repository
- Wokwi
- Extra docs
- Clock 0 Hz
- External hardware You could programm, address and play the 8x8 Bit Pattern Player with a breadboard, two clock buttons and some dipswitches on the input side. Add some LED to the output side. Just like the WOKWI simulation.

How it works

The 8x8 memory is a 64-bit shiftregister, consisting of 64 serial chained D-FlipFlops (data: IN0, clk_sr: IN1). 8 memoryslots of each 8 bit can be directly addressed via addresslines (3 bit: IN2, IN3, IN4) or from a clockdriven player (3 bit counter, clk_pl: IN7). A mode selector line (mode: IN5) sets the operation mode to addressing or to player. The 8 outputs are driven by the 8 bit of the addressed memoryslot.

How to test

Programm the memory: Start by filling the 64 bit shiftregister via data and clk_sr, each rising edge on clk_sr shifts a new data bit into the register. Select mode: Set mode input for direct addressing or clockdriven player. Address mode: Address a memoryslot via the three addresslines and watch the memoryslot at the outputs. Player mode: Each rising edge at clk_pl enables the next memoryslot to the outputs.

IO

#	Input	Output
0	data	bit 0
1	clk_sr	bit 1
2	address_0	bit 2
3	address_1	bit 3
4	address_2	bit 4
5	mode	bit 5
6	none	bit 6
7	clk_pl	bit 7

XLS: bit population count

- Author proppy
- Description Count bits set in the input.
- GitHub repository
- HDL
- Extra docs
- Clock 0 Hz
- External hardware LEDs and resistors

How it works

https://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetParallel

How to test

Pull up input bits, check that output bits represent the count.

IO

#	Input	Output
0	bit0	count0
1	bit1	count1
2	bit2	count2
3	bit3	count3
4	bit4	count4
5	bit5	count5
6	bit6	count6
7	bit7	count7

RC5 decoder

- Author Jean THOMAS
- Description Increment/decrement a counter with the press of a IR remote button!
- GitHub repository
- HDL
- Extra docs
- Clock 562 Hz
- External hardware Connect an IR demodulator (ie. TSOP1738) to the input pin

How it works

Decodes an RC5 remote signal, increments the counter if the volume up button is pressed, decrements the counter if the volume down button is pressed

How to test

after reset, point a remote to the IR received. Press the volume up button and the display count should increase.

IO

#	Input	Output
0	clock	segment a
1	reset	segment b
2	IR demodulator output	segment c
3	none	segment d
4	none	segment e
5	none	segment f
6	none	segment g
7	none	none

chiDOM

- Author Maria CHiara Molteni
- Description Chi function of Xoodoo protected at the first-order by DOM
- GitHub repository
- Wokwi
- Extra docs
- Clock 0 Hz
- External hardware

How it works

Chi function of Xoodoo protected at the first-order by DOM

How to test

Set on all the inputs

IO

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

Super Mario Tune on A Piezo Speaker

picture

- Author Milosch Meriac
- Description Plays Super Mario Tune over a Piezo Speaker connected across io_out[1:0]
- GitHub repository
- HDL
- Extra docs
- Clock 3000 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 3kHz clock on io_in[0], briefly hit reset io_in[1] (L->H->L) and io_out[1:0] will play a differential sound wave over piezo speaker (Super Mario)

IO

#	Input	Output
0	clock	piezo_speaker_p
1	reset	piezo_speaker_n
2	none	none
3	none	none
4	none	none
5	none	none
6	none	none
7	none	none

Tiny rot13

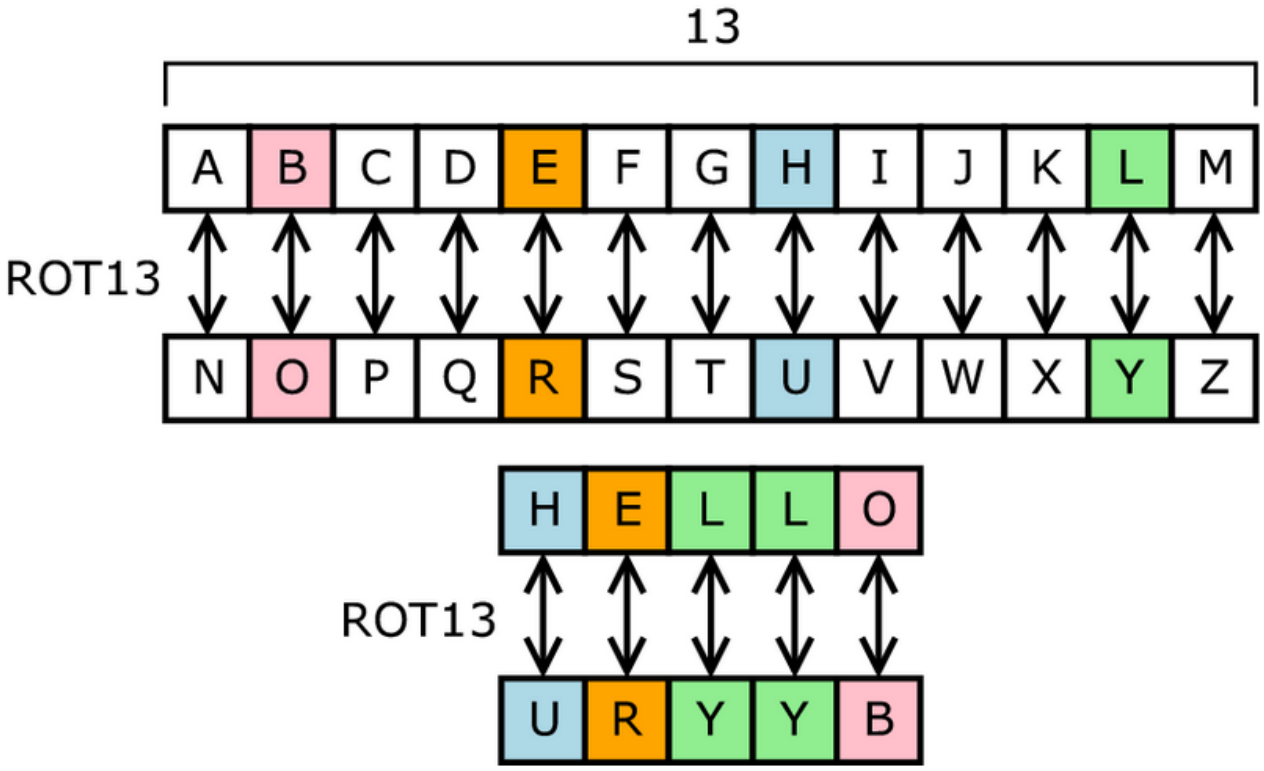

Figure 11: picture

- Author Phase Noise
- Description implements rot13 in the constraints of TT02
- GitHub repository
- HDL
- Extra docs
- Clock 1000 Hz
- External hardware For basic usage, the carrier board should suffice. An MCU or FPGA would be required to use this to the fullest extent, and an FPGA with PCIe would let you build the world's worst ROT13 Accelerator!

How it works

shifts in low and high nibble to construct an ASCII character, performs rot13, then outputs. The design uses some registers to store the low and high nibbles before constructing them into the ASCII character. ROT13 is implemented with a LUT genetated from Python. Controlled using control lines instead of specific clock cycles. Any non-alphabetic characters are passed through

How to test

CTL0 and CTL1 are control lines. Let CTL[1:0], 2b00 -> Shift in low nibble on D[3:0] and set output[7:0]=0x0f, 2b01 -> Shift in high nibble on D[3:0] and set output[7:0]=0xf0, 2b1X -> Shift out S on output[7:0]. Shift in the low and high nibbles of rot13, then read the result on the next cycle. Internal registers are init to 0, so by default after a RST, the output will be 0x00 for a single cycle(if CTL=2'b10), otherwise it will 2'b00 before updating to whatever the control lines set it to. Every operation effectively sets the output of the next clock cycle. Every complete operation effectively takes 4 cycles. To test, Set RST, then write 0x1 as the low nibble (clock 0), 0x4 as the high nibble (clock 1), then set the control lines to output (clock 1). 0x4e should be read at clock 4, with the output sequence being C=0 out=0x00, C=1 out=0x01, C=2 out=0x10, C=3 out=0x4e. 0x00 should produce 0x00 while 0x7f should produce 0x7f.

IO

#	Input	Output
0	clock	DO0 - LSB of output
1	reset - Resets the system to a clean state	DO1
2	CTL0 - LSB of control	DO2
3	CTL1 - MSB of control	DO3
4	D0 - LSB of input nibble	DO4
5	D1	DO5
6	D2	DO6
7	D3 - MSB of input nibble	DO7 - MSB of output

4 bit counter on steamdeck

- Author 13arn
- Description copy of my tt01 submission, enable first input and press button to use the counter
- GitHub repository
- Wokwi
- Extra docs
- Clock 0 Hz
- External hardware

How it works

fsm that uses 1 input to progress abd count from 0 to 3. Other inputs have various logic to play with

How to test

enable first input so it is on and connected to the button. All other inputs are off. Press button to progress the fsm.

IO

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

Shiftregister Challenge 40 Bit

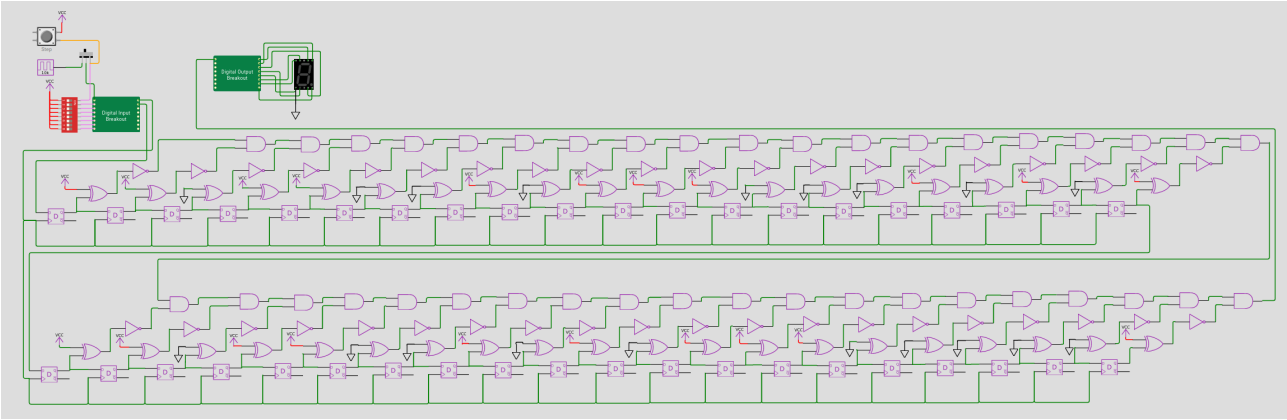

Figure 12: picture

- Author Thorsten Knoll
- Description The design is a 40 bit shiftregister with a hardcoded 40 bit number. The challenge is to find the correct 40 bit to enable the output to high. With all other numbers the output will be low.
- GitHub repository
- Wokwi
- Extra docs
- Clock 0 Hz
- External hardware To test when knowing the correct 40 bit, only a dipswitch (data), a button (clk) and a LED (output) is needed. Without knowing the number it becomes the challenge and more hardware might be required.

How it works

Shift a 40 bit number into the chip with the two inputs data (IN0) and clk (IN1). If the shifted 40 bit match the hardcoded internal 40 bit, then and only then the output will become high. Having only the mikrochip without the design files, one might need reverse engineering and/or side channel attacks to fing the correct 40 bit.

How to test

Get the correct 40 bit from the design and shift them into the shiftregister. Each rising edge at the clk will push the next bit into the register. At the correct 40 bit, the output will enable high.

IO

#	Input	Output
0	data	output
1	clk	none
2	none	none
3	none	none
4	none	none
5	none	none
6	none	none
7	none	none

TinyTapeout2 4-bit multiplier.

- Author Tholin
- Description Multiplies two 4-bit numbers presented on the input pins and outputs an 8-bit result.
- GitHub repository
- HDL
- Extra docs
- Clock 6000 Hz
- External hardware DIP switches for the inputs, and LEDs on the outputs, to be able to read the binary result.

How it works

The multiplier is implemented using purely combinatorial logic. One 6-bit adder and two 8-bit adders as well as a heap of AND gates are the only used components.

How to test

Input any two numbers on the input ports, and check if the 8-bit result is correct.

IO

#	Input	Output
0	A0	R0
1	A1	R1
2	A2	R2
3	A3	R3
4	B0	R4
5	B1	R5
6	B2	R6
7	B3	R7

TinyTapeout2 multiplexed segment display timer.

- Author Tholin
- Description Measures time up to 99 minutes and 59 seconds by multiplexing 4 seven-segment displays.
- GitHub repository
- HDL
- Extra docs
- Clock 1024 Hz
- External hardware 4 sevent segment displays, plus some 74-series chips to build the select logic.

How it works

TODO

How to test

TODO

IO

#	Input	Output
0	CLK	A
1	RST	B
2	NC	C
3	NC	D
4	NC	E
5	NC	F
6	NC	G
7	NC	SEL

XorShift32

- Author Ethan Mahintorabi
- Description XorShift32 random number generator
- GitHub repository
- HDL
- Extra docs
- Clock 1000 Hz
- External hardware

How it works

Uses the Xorshift32 algorithm to generate a random 32 bit number. Number is truncated to 3 bits and displayed

How to test

While reset is set, hardware reads in seed value from input bits 2:7 and sets the inital seed as that binary number. After reset is deasserted, the hardware will generate a new number every 1000 clock cycles.

IO

#	Input	Output
0	clock	segment a
1	reset	segment b
2	seed_bit0	segment c
3	seed_bit1	segment d
4	seed_bit2	segment e
5	seed_bit3	segment f
6	seed_bit4	segment g
7	seed_bit5	none

XorShift32

- Author Ethan Mahintorabi
- Description XorShift32 random number generator
- GitHub repository
- HDL
- Extra docs
- Clock 1000 Hz
- External hardware

How it works

Uses the Xorshift32 algorithm to generate a random 32 bit number. Number is truncated to 3 bits and displayed

How to test

While reset is set, hardware reads in seed value from input bits 2:7 and sets the inital seed as that binary number. After reset is deasserted, the hardware will generate a new number every 1000 clock cycles.

IO

#	Input	Output
0	clock	segment a
1	reset	segment b
2	seed_bit0	segment c
3	seed_bit1	segment d
4	seed_bit2	segment e
5	seed_bit3	segment f
6	seed_bit4	segment g
7	seed_bit5	none

Multiple Tunes on A Piezo Speaker

- Author Jiaxun Yang
- Description Plays multiple Tunes over a Piezo Speaker connected across io_out[1:0]
- GitHub repository
- HDL
- Extra docs
- Clock 10000 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 10kHz clock on io_in[0], briefly hit reset io_in[1] (L->H->L) and io_out[1:0] will play a differential sound wave over piezo speaker, different tunes can be selected by different tune_sel inputs

IO

#	Input	Output
0	clock	piezo_speaker_p
1	reset	piezo_speaker_n
2	tune_sel0	ledout_0
3	tune_sel1	ledout_1
4	none	ledout_2
5	none	ledout_3
6	none	none
7	none	none

clash cpu

- Author Jack Leightcap
- Description tiny register machine written in clash
- GitHub repository
- HDL
- Extra docs
- Clock 1000 Hz
- External hardware

How it works

poorly

How to test

encode instructions. tick clock.

IO

#	Input	Output
0	instr 5	jump
1	instr 4	unused
2	instr 3	unused
3	instr 2	register 4
4	instr 1	register 3
5	instr 0	register 2
6	rst	register 1
7	clock	register 0

TinyTapeout 2 LCD Nametag

- Author Tholin
- Description Echoes out a predefined text onto a 20x4 character LCD.
- GitHub repository
- HDL
- Extra docs
- Clock 100 Hz
- External hardware A 20x4 character LCD.

How it works

Mostly just contains a ROM holding the text to be printed, and some logic to print the reset sequence and cursor position changes.

How to test

Connect up a character LCD according to the pinout, set the clock and hit reset. Run using an extra slow clock, as there is no internal clock divider. It'll send data to the display as fast as it's able to. After that, it should initialize the display and start printing stuff.

IO

#	Input	Output
0	CLK	RS
1	RST	E
2	NC	D4
3	NC	D5
4	NC	D6
5	NC	D7
6	NC	LED
7	NC	NC

UART-CC

- Author Christina Cyr
- Description UART Template
- GitHub repository
- Wokwi
- Extra docs
- Clock 0 Hz
- External hardware Arduino

How it works

You can hook this up to an arduino

How to test

Use an arduino

IO

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

Technical info

Scan chain

All 498 designs are joined together in a long chain similiar to JTAG. We provide the inputs and outputs of that chain (see pinout below) externally, to the Caravel logic analyser, and to an internal scan chain driver.

The default is to use an external driver, this is in case anything goes wrong with the Caravel logic analyser or the internal driver.

The scan chain is identical for each little project, and you can read it here.

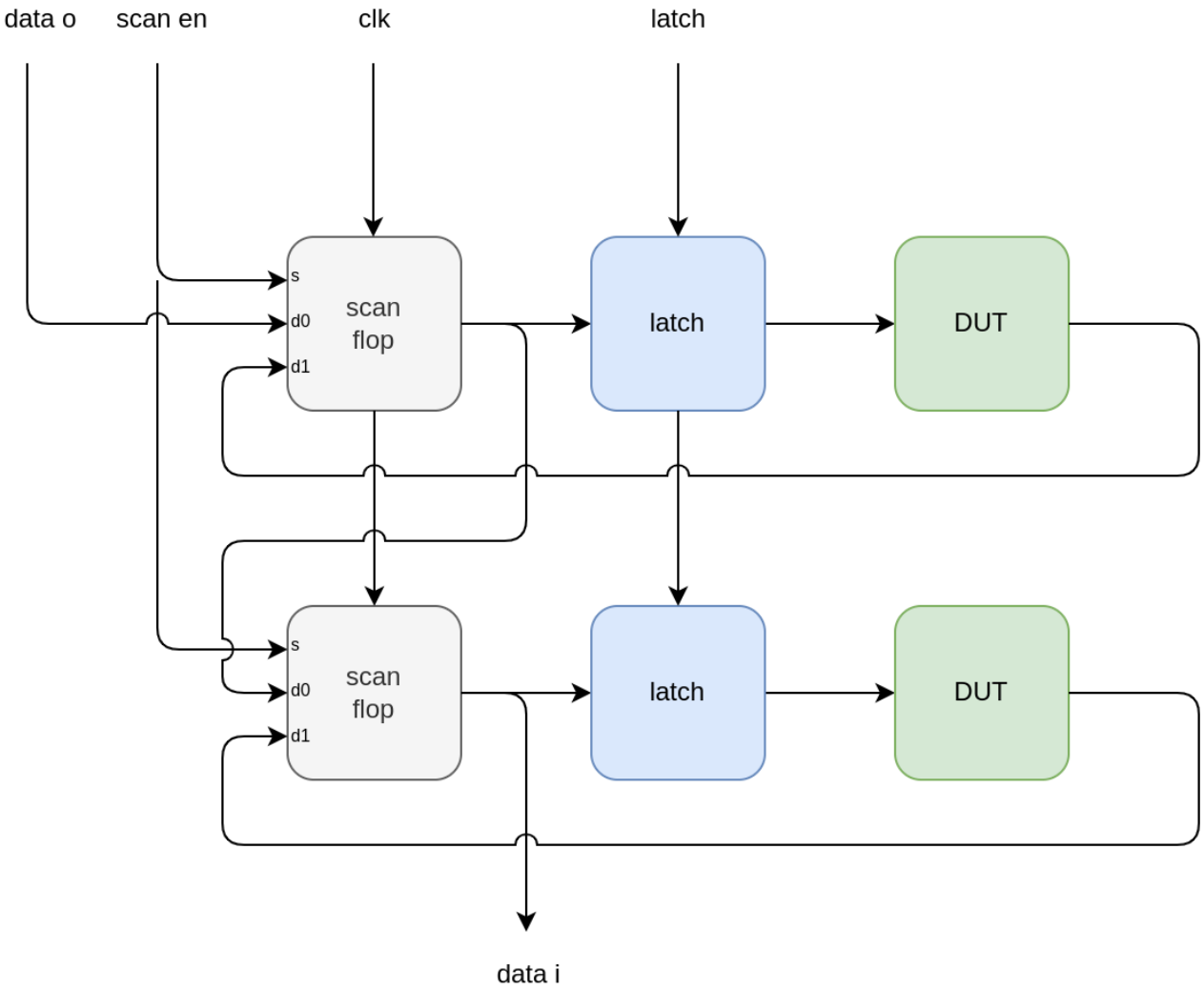

Figure 13: block diagram

Updating inputs and outputs of a specified design

A good way to see how this works is to read the FSM in the scan controller. You can also run one of the simple tests and check the waveforms. See how in the scan chain verification doc.

- Signal names are from the perspective of the scan chain driver.
- The desired project shall be called DUT (design under test)

Assuming you want to update DUT at position 2 (0 indexed) with inputs = 0x02 and then fetch the output. This design connects an inverter between each input and output.

- Set scan_select low so that the data is clocked into the scan flops (rather than from the design)
- For the next 8 clocks, set scan_data_out to 0, 0, 0, 0, 0, 0, 1, 0
- Toggle scan_clk_out 16 times to deliver the data to the DUT
- Toggle scan_latch_en to deliver the data from the scan chain to the DUT
- Set scan_select high to set the scan flop's input to be from the DUT
- Toggle the scan_clk_out to capture the DUT's data into the scan chain
- Toggle the scan_clk_out another 8 x number of remaining designs to receive the data at scan_data_in

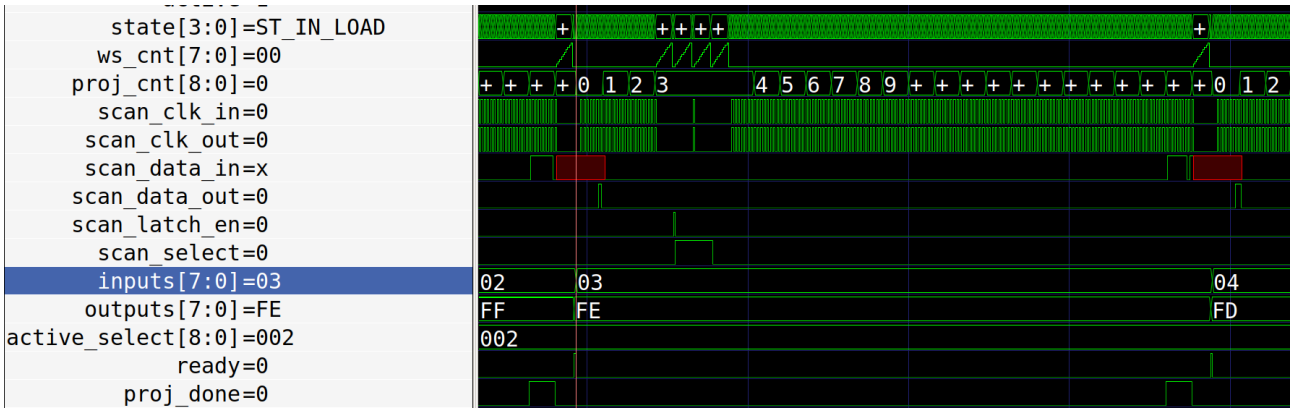

Figure 14: update cycle

Notes on understanding the trace

- There are large wait times between the latch and scan signals to ensure no hold violations across the whole chain. For the internal scan controller, these can be configured (see section on wait states below).
- The input looks wrong (0x03) because the input is incremented by the test bench as soon as the scan controller captures the data. The input is actually 0x02.
- The output in the trace looks wrong (0xFE) because it's updated after a full refresh, the output is 0xFD.

Clocking

Assuming:

- 100MHz input clock
- 8 ins & 8 outs

- 2 clock cycles to push one bit through the scan chain (scan clock is half input clock rate)
- 500 designs
- scan controller can do a read/write cycle in one refresh

So the max refresh rate is 100MHz / (8 * 2 * 500) = 12500Hz.

Clock divider

A rising edge on the set_clk_div input will capture what is set on the input pins and use this as a divider for an internal slow clock that can be provided to the first input bit.

The slow clock is only enabled if the set_clk_div is set, and the resulting clock is connected to input0 and also output on the slow_clk pin.

The slow clock is synced with the scan rate. A divider of 0 mean it toggles the input0 every scan. Divider of 1 toggles it every 2 cycles. So the resultant slow clock frequency is scan_rate / (2 * (N+1)).

See the test_clock_div test in the scan chain verification.

Wait states

This dictates how many wait cycle we insert in various state of the load process. We have a sane default, but also allow override externally.

To override, set the wait amount on the inputs, set the driver_sel inputs both high, and then reset the chip.

See the test_wait_state test in the scan chain verification.

Pinout

```
PIN     NAME                DESCRIPTION
20:12   active_select       9 bit input to set which design is active
28:21   inputs              8 inputs
36:29   outputs             8 outputs
37      ready               goes high for one cycle everytime the scancha
10      slow_clk            slow clock from internal clock divider
11      set_clk_div         enable clock divider
9:8     driver_sel          which scan chain driver: 00 = external, 01 =

21      ext_scan_clk_out    for external driver, clk input
```

```
22      ext_scan_data_out   data input
23      ext_scan_select     scan select
24      ext_scan_latch_en   latch
29      ext_scan_clk_in     clk output from end of chain
30      ext_scan_data_in    data output from end of chain
```

Instructions to build GDS

To run the tool locally or have a fork's GitHub action work, you need the GH_USERNAME and GH_TOKEN set in your environment.

GH_USERNAME should be set to your GitHub username.

To generate your GH_TOKEN go to https://github.com/settings/tokens/new . Set the checkboxes for repo and workflow.

To run locally, make a file like this:

```
export GH_USERNAME=<username>
export GH_TOKEN=<token>
```

And then source it before running the tool.

Fetch all the projects

This goes through all the projects in project_urls.py, and fetches the latest artifact zip from GitHub. It takes the verilog, the GL verilog, and the GDS and copies them to the correct place.

```
./configure.py --clone-all
```

Configure Caravel

Caravel needs the list of macros, how power is connected, instantiation of all the projects etc. This command builds these configs and also makes the README.md index.

```
./configure.py --update-caravel
```

Build the GDS

To build the GDS and run the simulations, you will need to install the Sky130 PDK and OpenLane tool. It takes about 5 minutes and needs about 3GB of disk space.

```
export PDK_ROOT=<some dir>/pdk
export OPENLANE_ROOT=<some dir>/openlane
```

```
cd <the root of this repo>
make setup
```

Then to create the GDS:

```
make user_project_wrapper
```

Changing macro block size

After working out what size you want:

- adjust configure.py in `CaravelConfig.create_macro_config()`.
- adjust the PDN spacing to match in openlane/user_project_wrapper/config.tcl:
  - `set ::env(FP_PDN_HPITCH)`
  - `set ::env(FP_PDN_HOFFSET)`

Verification

We are not trying to verify every single design. That is up to the person who makes it. What we want is to ensure that every design is accessible, even if some designs are broken.

We can split the verification effort into functional testing (simulation), static tests (formal verification), timing tests (STA) and physical tests (LVS & DRC).

See the sections below for details on each type of verification.

Setup

You will need the GitHub tokens setup as described in INFO.

The default of 498 projects takes a very long time to simulate, so I advise overriding the configuration:

```
# fetch the test projects
./configure.py --test --clone-all
# rebuild config with only 20 projects
./configure.py --test --update-caravel --limit 20
```

You will also need iVerilog & cocotb. The easist way to install these are to download and install the oss-cad-suite.

Simulations

- Simulation of some test projects at RTL and GL level.
- Simulation of the whole chip with scan controller, external controller, logic analyser.
- Check wait state setting.
- Check clock divider setting.

Scan controller

This test only instantiates user_project_wrapper (which contains all the small projects). It doesn't simulate the rest of the ASIC.

```
cd verilog/dv/scan_controller
make test_scan_controller
```

The Gate Level simulation requires scan_controller and user_project_wrapper to be re-hardened to get the correct gate level netlists:

- Edit openlane/scan_controller/config.tcl and change NUM_DESIGNS=498 to NUM_DESIGNS=20.
- Then from the top level directory:

  make scan_controller make user_project_wrapper
- Then run the GL test

  cd verilog/dv/scan_controller make test_scan_controller_gl

single

Just check one inverter module. Mainly for easy understanding of the traces.

```
make test_single
```

custom wait state

Just check one inverter module. Set a custom wait state value.

```
make test_wait_state
```

clock divider

Test one inverter module with an automatically generated clock on input 0. Sets the clock rate to 1/2 of the scan refresh rate.

```
make test_clock_div
```

Top level tests setup

For all the top level tests, you will also need a RISCV compiler to build the firmware.

You will also need to install the 'management core' for the Caravel ASIC submission wrapper. This is done automatically by following the PDK install instructions.

Top level test: internal control

Uses the scan controller, instantiated inside the whole chip.

```
cd verilog/dv/scan_controller_int
make coco_test
```

Top level test: external control

Uses external signals to control the scan chain. Simulates the whole chip.

```
cd verilog/dv/scan_controller_ext
make coco_test
```

Top level test: logic analyser control

Uses the RISCV co-processor to drive the scanchain with firmware. Simulates the whole chip.

```
cd verilog/dv/scan_controller_la
make coco_test
```

Formal Verification

- Formal verification that each small project's scan chain is correct.
- Formal verification that the correct signals are passed through for the 3 different scan chain control modes.

Scan chain

Each GL netlist for each small project is proven to be equivalent to the reference scan chain implementation. The verification is done on the GL netlist, so an RTL version of the cells used needed to be created. See here for more info.

Scan controller MUX

In case the internal scan controller doesn't work, we also have ability to control the chain from external pins or the Caravel Logic Analyser. We implement a simple MUX to achieve this and formally prove it is correct.

Timing constraints

Due to limitations in OpenLane - a top level timing analyis is not possible. This would allow us to detect setup and hold violations in the scan chain.

Instead, we design the chain and the timing constraints for each project and the scan controller with this in mind.

- Each small project has a negedge flop flop at the end of the shift register to reclock the data. This gives more hold margin.

- Each small project has SDC timing constraints
- Scan controller uses a shift register clocked with the end of the chain to ensure correct data is captured.
- Scan controller has its own SDC timing constraints
- Scan controller can be configured to wait for a programmable time at latching data into the design and capturing it from the design.
- External pins (by default) control the scan chain.

Physical tests

- LVS
- DRC
- CVC

LVS

Each project is built with OpenLane, which will check LVS for each small project. Then when we combine all the projects together we run a top level LVS & DRC for routing, power supply and macro placement.

The extracted netlist from the GDS is what is used in the formal scan chain proof.

DRC

DRC is checked by OpenLane for each small project, and then again at the top level when we combine all the projects.

CVC

Mitch Bailey' CVC checker is a device level static verification system for quickly and easily detecting common circuit errors in CDL (Circuit Definition Language) netlists. We ran the test on the final design and found no errors.

- See the paper here.
- Github repo for the tool: https://github.com/d-m-bailey/cvc

Sponsored by

Team

Tiny Tapeout would not be possible without a lot of people helping. We would especially like to thank:

- Uri Shaked for wokwi development and lots more
- Sylvain Munaut for help with scan chain improvements
- Mike Thompson 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