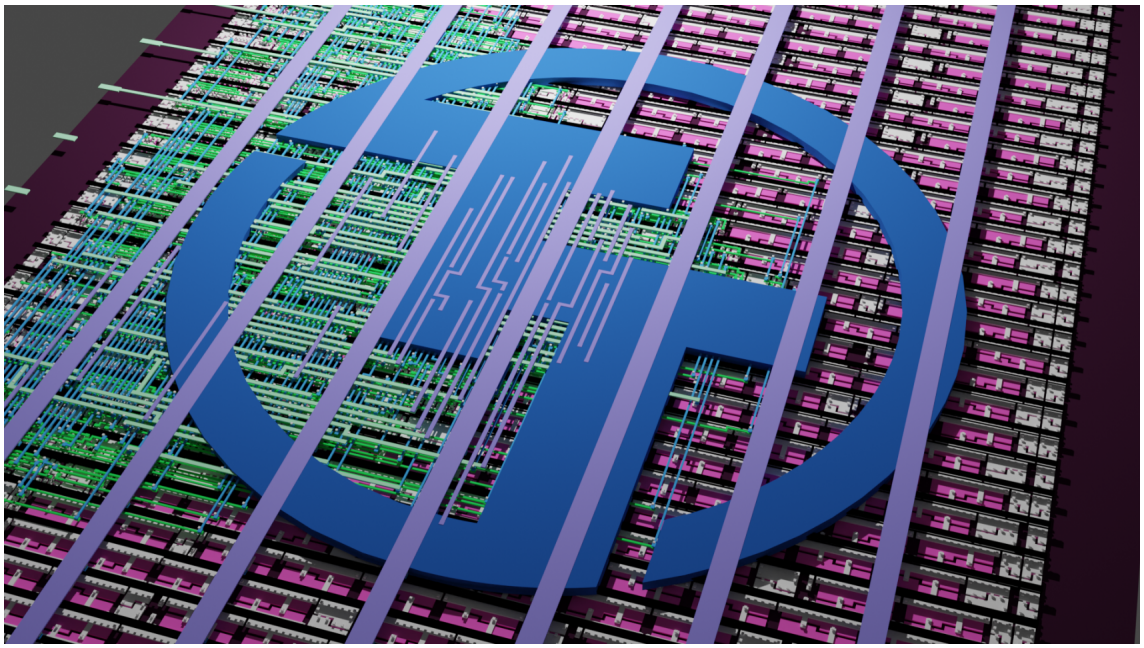

Tiny Tapeout 02 Datasheet

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

December 2, 2022

Contents

Render of whole chip

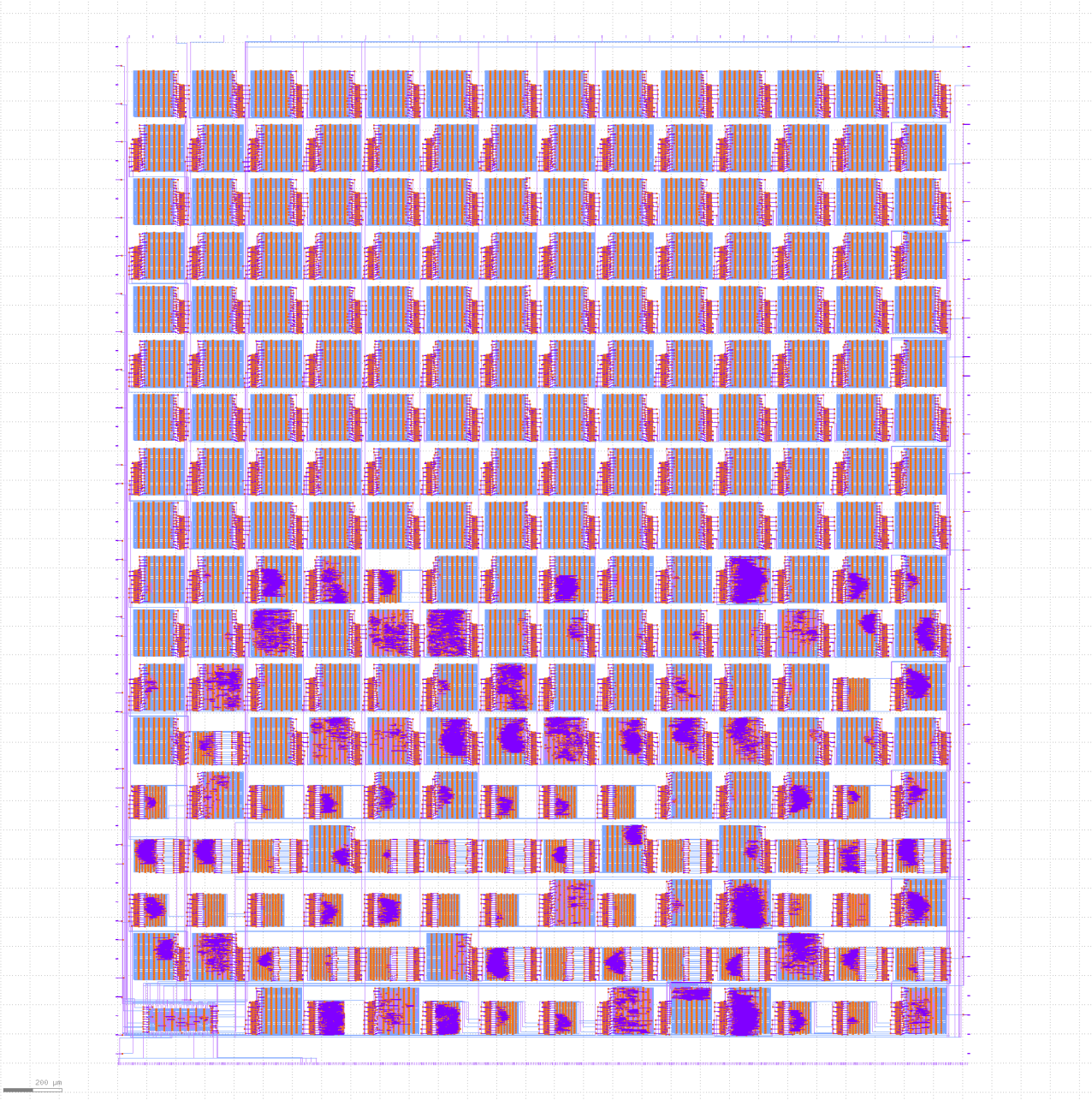

Figure 1: Full GDS

Projects

0 : Test Inverter Project

- Author: Matt Venn
- Description: inverts every line
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

uses 8 inverters to invert every line

How to test

setting the input switch to on should turn the corresponding led off

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

1 : SIMON Cipher

- Author: Fraser Price
- Description: Simon32/64 Encryption
- GitHub repository
- HDL project
- 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

2 : HD74480 Clock

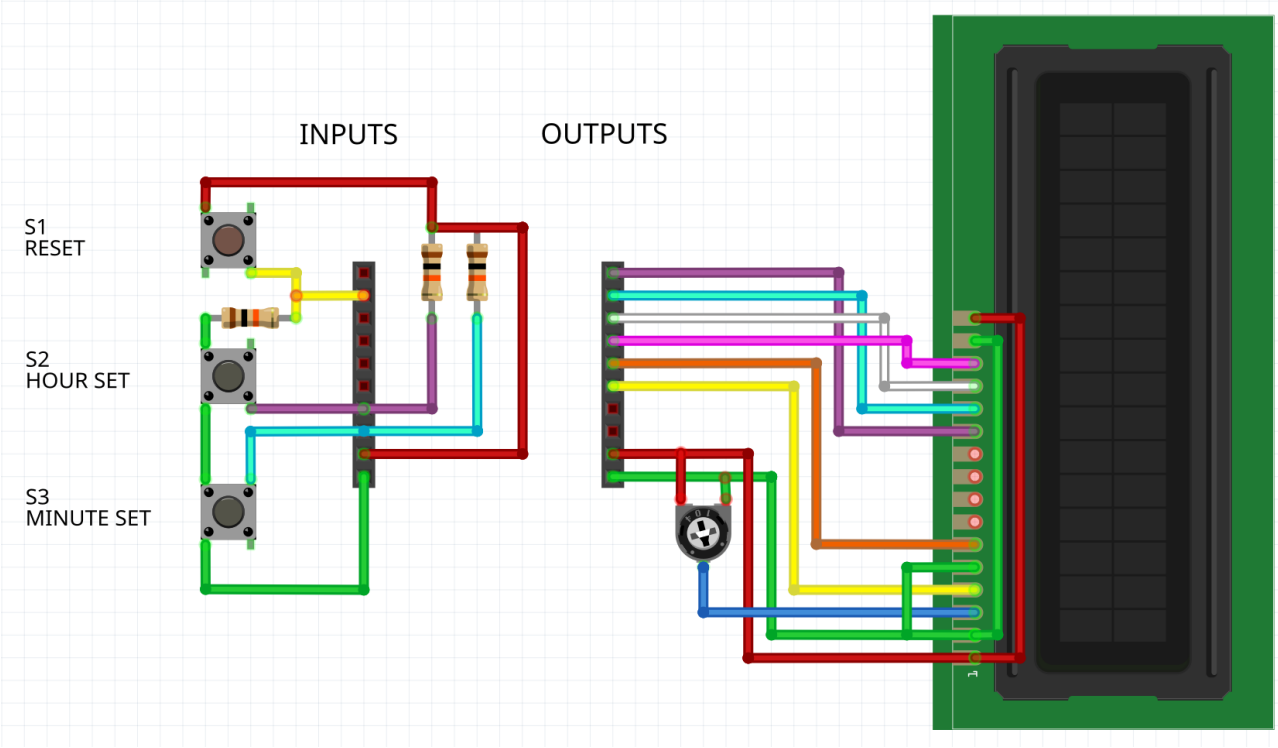

Figure 2: picture

- Author: Tom Keddie
- Description: Displays a clock on a attached HD74480
- GitHub repository
- HDL project
- 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

3 : Scrolling Binary Matrix display

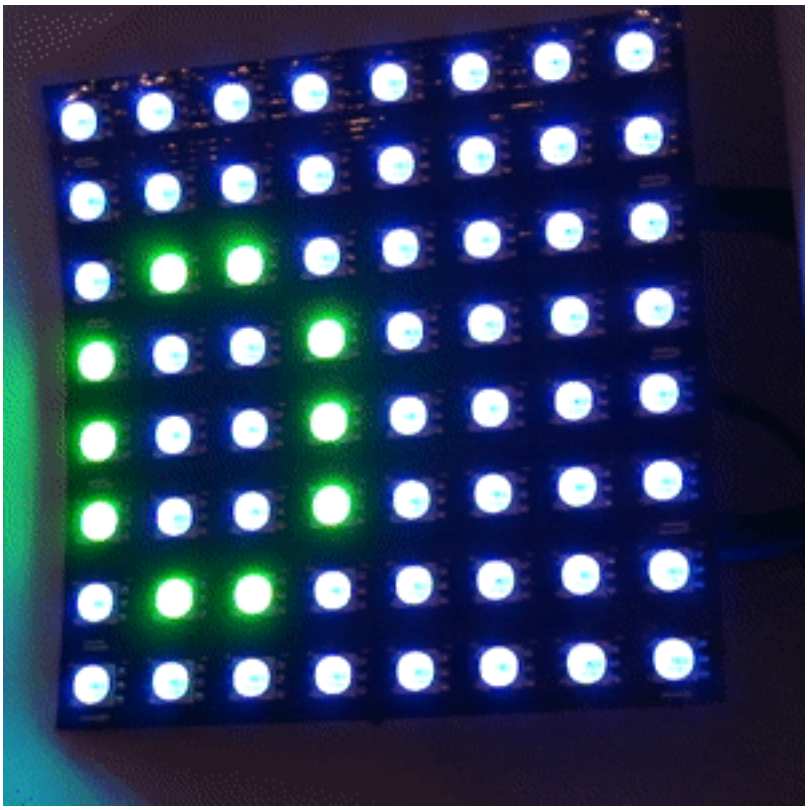

Figure 3: picture

- Author: Chris
- Description: Display scrolling binary data from input pin on 8x8 SK9822 LED matrix display
- GitHub repository
- HDL project
- Extra docs
- Clock: 6000 Hz
- External hardware: Requires 8x8 matrix SK9822 LED display and 3.3V to 5V logic level shifter to convert the data and clock signals to the correct voltage for the display.

How it works

Uses 8x8 matrix SK9822 LED display to scroll binary data as 0s and 1s in a simple font, from the input pin. Designed in verilog and tested using iCEstick FPGA Evaluation Kit. Each LED takes a 32 bit value, consisting of r,g,b and brightness.

How to test

Need 8x8 matrix SK9822 LED 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

4 : Power supply sequencer

- Author: Jon Klein
- Description: Sequentially enable and disable channels with configurable delay
- GitHub repository
- HDL project
- 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

5 : Duty Controller

- Author: Marcelo Pouso / Miguel Correia
- Description: Increase/Decrease a duty cycle of square signal.
- GitHub repository
- HDL project
- 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

6 : S4GA: Super Slow Serial SRAM FPGA

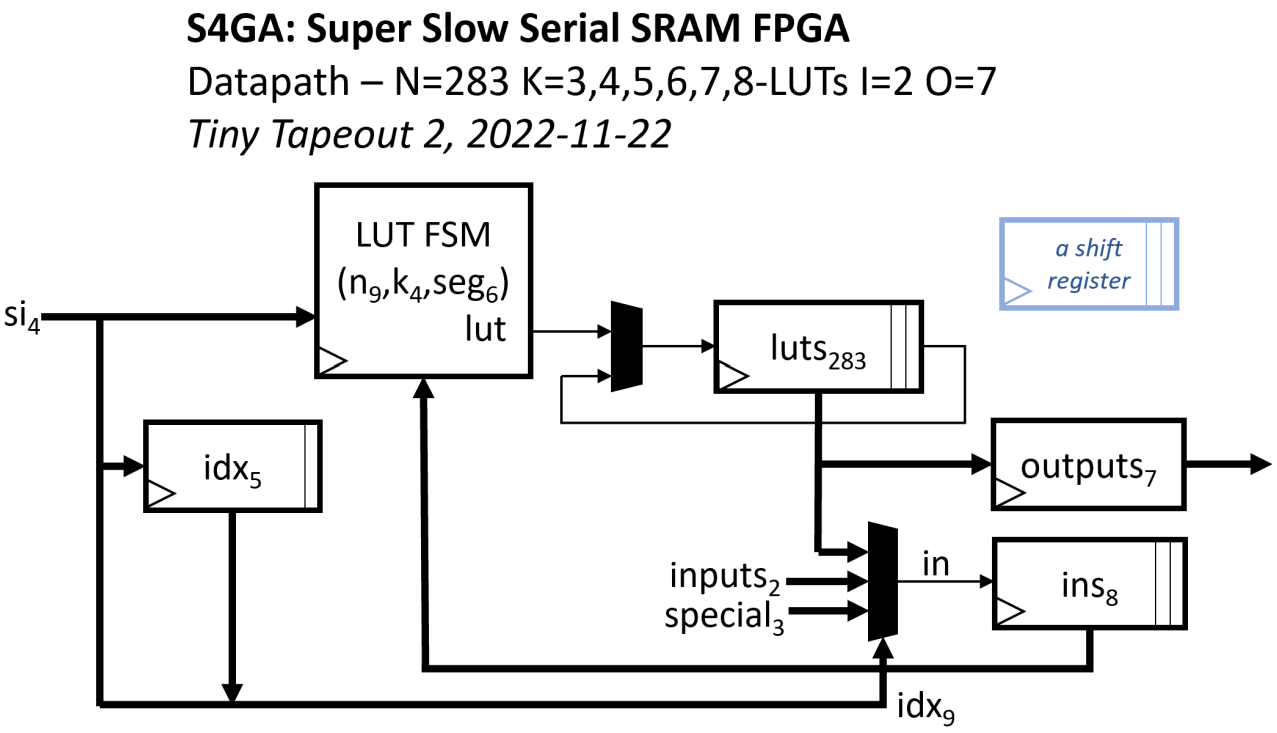

Figure 4: picture

- Author: Jan Gray
- Description: one fracturable 5-LUT that receives FPGA LUT configuration frames, serially evaluates LUT inputs and LUT outputs
- GitHub repository
- HDL project
- 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 92b LUT configuration frames, four bits per cycle. Every 23 clock cycles it evaluates a 5-input LUT. The last N=283 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

7 : ALU

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

How it works

When 4 bits total are input'd into the ALU, it goes through 3 ripple carries and two finite state machines that adds to a temporary value that gets included in the basic ALU calculations

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

8 : The McCoy 8-bit Microprocessor

picture

- Author: Aidan Good
- Description: Custom RISC-V inspired microprocessor capable of simple arithmatic, branching, and jumps through a custom ISA.
- GitHub repository
- HDL project
- Extra docs
- Clock: None Hz
- External hardware: Any source that allows for 16 GPIO pins. 8 to set the input pins, 8 to read the output pins.

How it works

This chip contains an opcode decoder, 8-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	out6
7	in5	out7

9 : binary clock

- Author: Azdle
- Description: A binary clock using multiplexed LEDs
- GitHub repository
- HDL project
- 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

10 : TinySensor

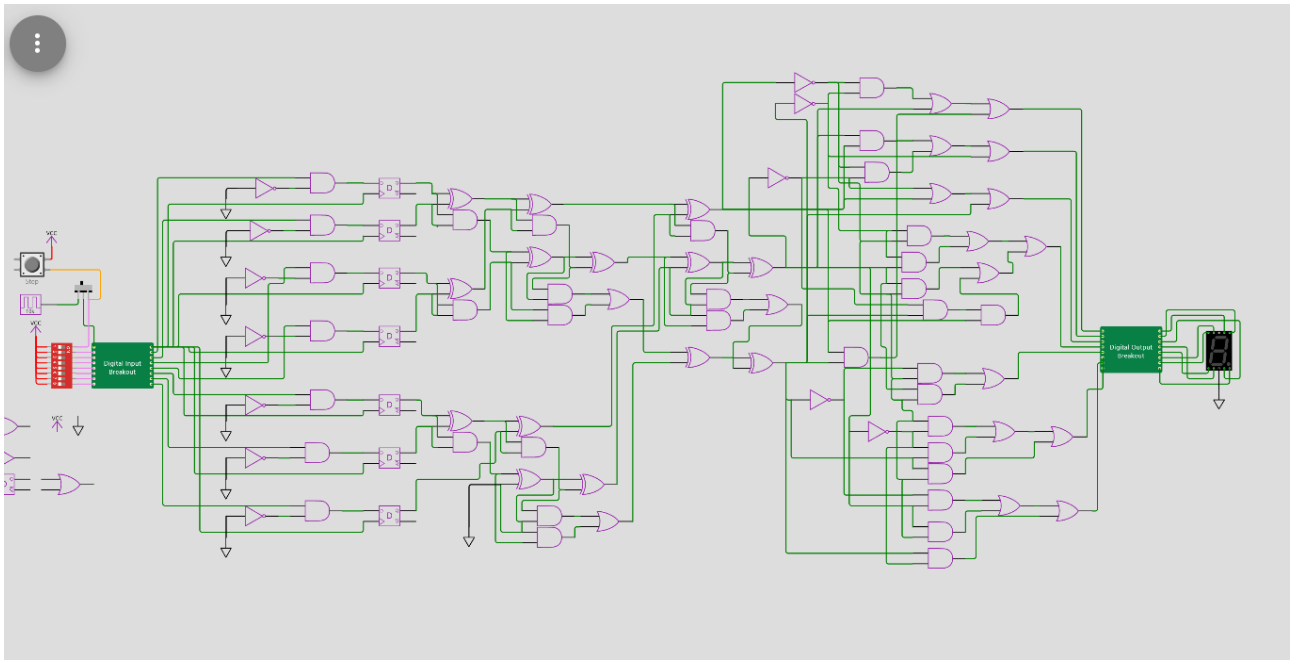

Figure 5: picture

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

How it works

Inputs 1-6 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

Dip switches 1-6 can be used instead of external hw to provide inputs, and 7 is used to switch between Step or Continous sample mode. 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

11 : 16x8 SRAM & Streaming Signal Generator

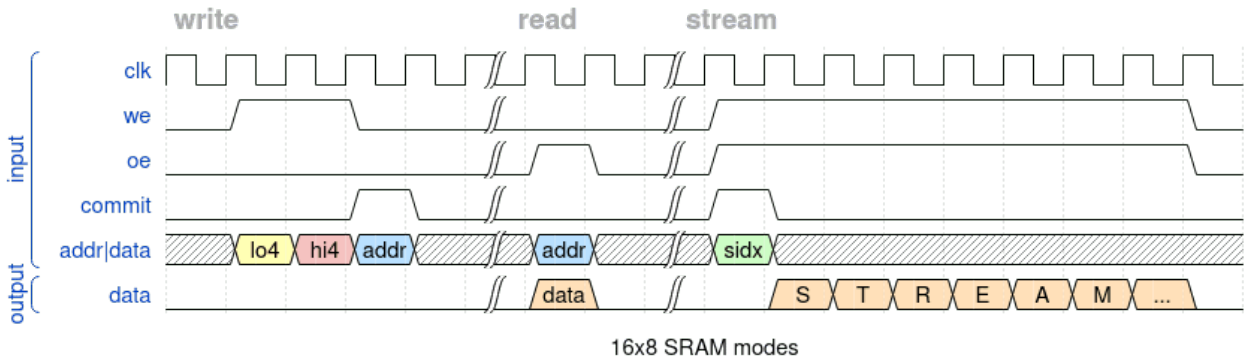

Figure 6: picture

- Author: James Ross
- Description: Write to, Read from, and Stream 16 addressable 8-bit words of memory
- GitHub repository
- HDL project
- 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 4-bit address will place 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 4-bit address will place 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.

How to test

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

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	addr[3]/high[3]/low[3]	data[7]

12 : German Traffic Light State Machine

- Author: Jens Schleusner
- Description: A state machine to control german traffic lights at an intersection.
- GitHub repository
- Wokwi project
- 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

13 : 4-spin Ising Chain Simulation

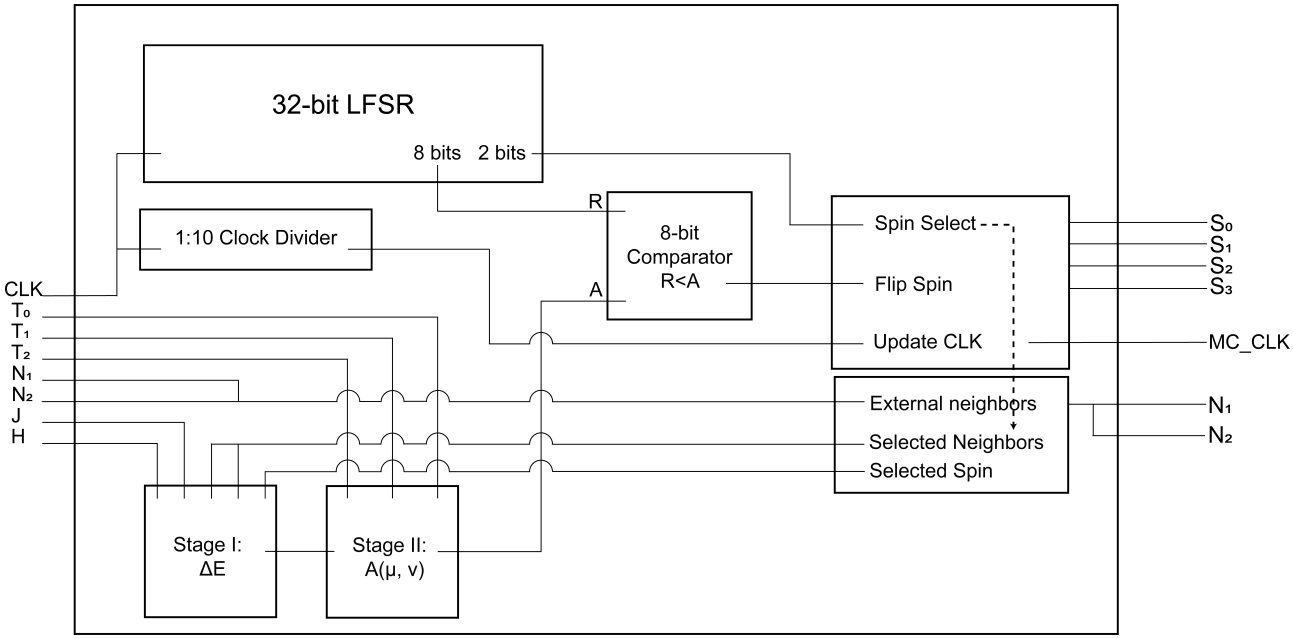

Figure 7: picture

- 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 project
- Extra docs
- Clock: 20 Hz
- External hardware: None

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. Using the inputs for external neighbors, N of these circuits can be chained together to create a 4N spin Ising chain.

How to test

The design can be tested by enabling one of the neighbours (input 4 or 5) and leave all other inputs low, 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.	none
7	H, Value of the coupling to the external field H.	segment h, MC Step Indicator.

14 : Avalon Semiconductors '5401' 4-bit Microprocessor

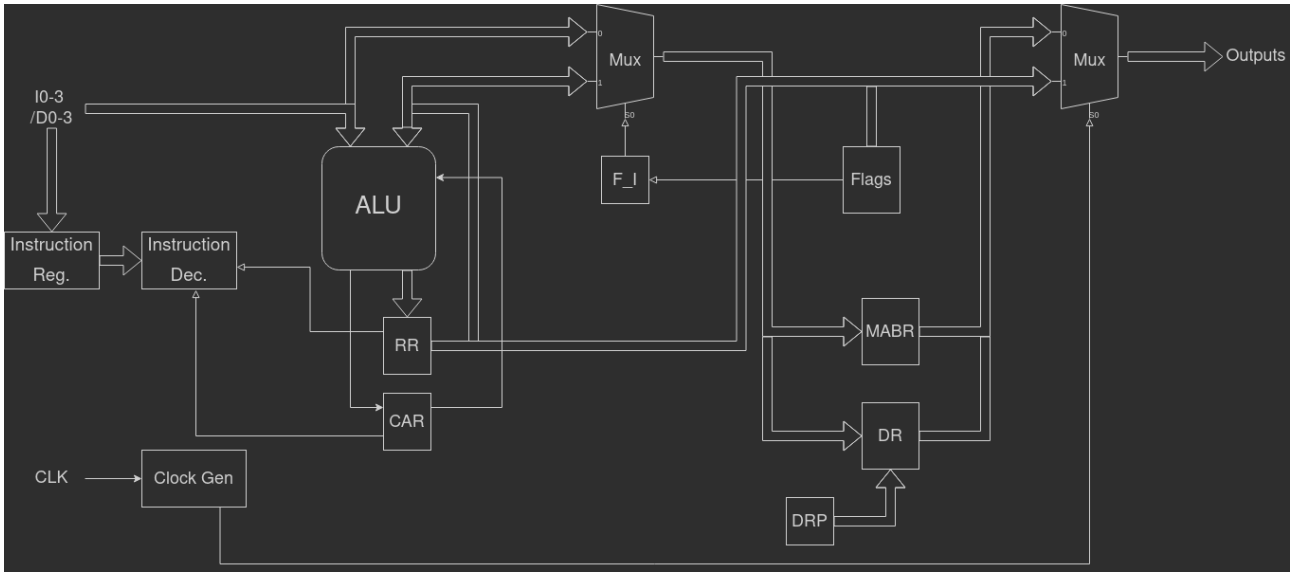

Figure 8: picture

- 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 project
- 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

15 : small FFT

- Author: Rice Shelley
- Description: Computes a small fft
- GitHub repository
- HDL project
- 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

16 : Stream Integrator

- Author: William Moyes
- Description: A silicon implementation of a simple optical computation
- GitHub repository
- Wokwi project
- 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

17 : tiny-fir

- Author: Tom Schucker
- Description: 4bit 2-stage FIR filter
- GitHub repository
- Wokwi project
- 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

18 : Configurable SR

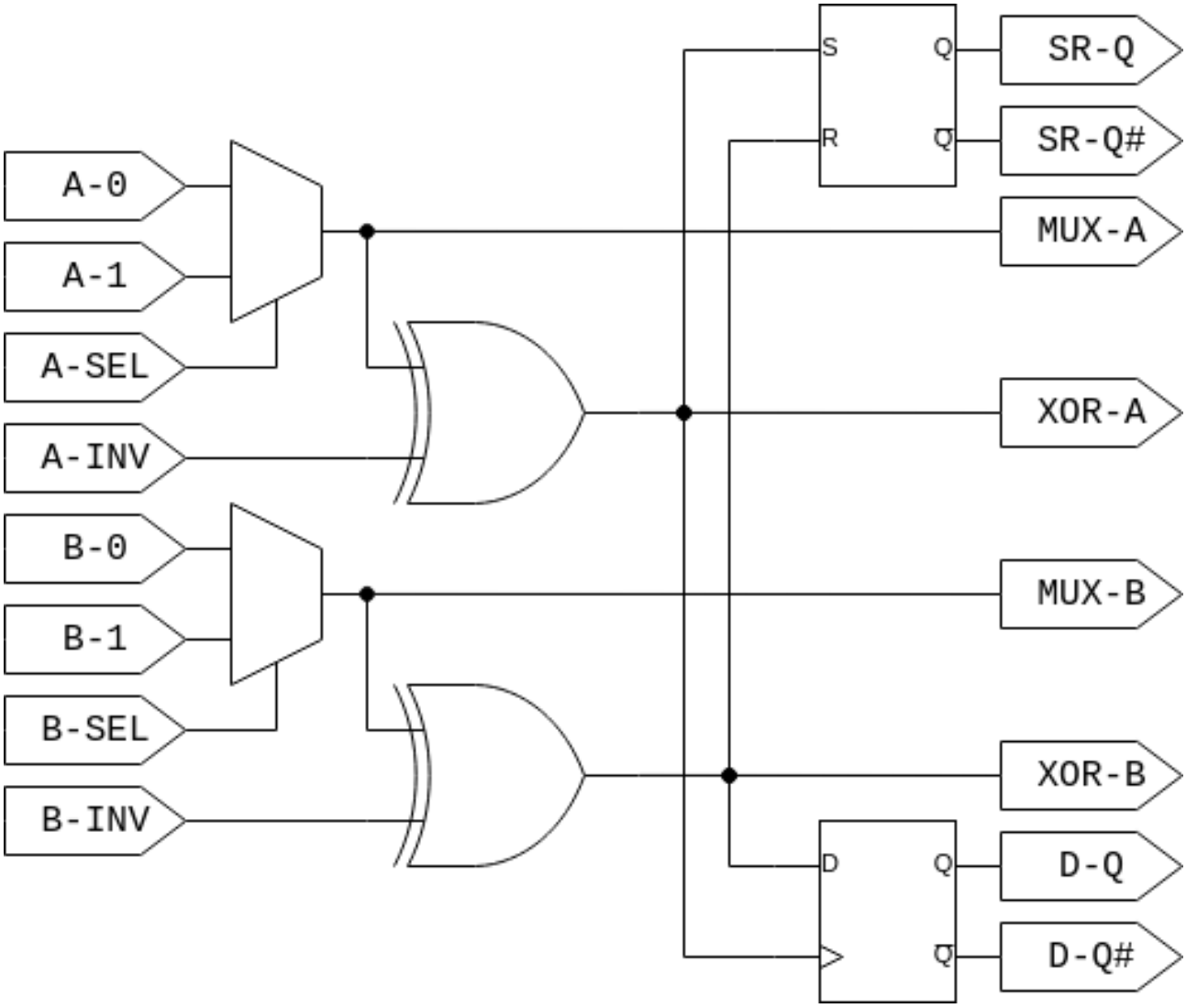

Figure 9: picture

- Author: Greg Steiert
- Description: Configurable gates driving SR and D flip-flops
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware: none

How it works

Two configurable gates enable a variety of complex flip-flop functions

How to test

When SEL and INV are low, the 0 inputs directly drive the flip-flops. A-0 can be connected to the clock for use with the D flip-flop.

IO

#	Input	Output
0	A-0	MUX-A
1	A-1	XOR-A
2	A-SEL	SR-Q
3	A-INV	D-Q
4	B-0	MUX-B
5	B-1	XOR-B
6	B-SEL	SR-Q#
7	B-INV	D-Q#

19 : 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 project
- 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

20 : 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 project
- 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

21 : 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 project
- 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

22 : A LED Flasher

picture

- Author: Ben Everard
- Description: Select different inputs to generate different LED patterns
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware: A LED on each pin

How it works

see How To Test

How to test

input 1 - clock input input 2 - feed NOT LED 1 back into the shift register – this creates a pattern where every other LED is switched on input 3 - feed 1 into the shift register if both the first two LEDs are off. This creates a pattern where every third LED is on input 4 - feed 1 into the shift register if the first three LEDs are off. This creates a pattern where every fourth LED is on input 5 - feed 1 into the shift register if all the LEDs are off. This creates a pattern of one light scanning across the LEDs input 6 - set the direction of the shift register input 7 - toggles fixed direction or alternating direction. If alternating direction is set, the direction of the shift register will flip if all the LEDs are off input 8 - enable the clock divider

IO

#	Input	Output
0	clock	LED1
1	not_1	LED2
2	not_1_2	LED3
3	not_1_2_3	LED4
4	not_all	LED5
5	direction	LED6
6	toggle_direction	LED7
7	clock_div_enable	LED8

23 : 4-bit Multiplier

- Author: Fernando Dominguez Pousa
- Description: 4-bit Multiplier based on single bit full adders
- GitHub repository
- HDL project
- 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)

24 : Avalon Semiconductors 'TBB1143' Programmable Sound Generator

- Author: Tholin
- Description: Sound generator with two square-wave voices, one sawtooth voice and one noise channel. Can also be used as a general-purpose frequency generator.
- GitHub repository
- HDL project
- Extra docs
- Clock: 6000 Hz
- External hardware: Lots of resistors or some other DAC, as well as a microprocessor or microcontroller to program the 1143.

How it works

All tone generators simply take the input clock frequency, multiplied by 256 and divide it by 16 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. The output is a 6-bit digital sample, but can easily be converted to an analog signal using a resistor chain. Also uses the leftover output pins as general-purpose outputs.

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 15 will cause a ~440Hz tone to appear on the output.

IO

#	Input	Output
0	CLK	SOUT0
1	RST	SOUT1
2	D0	T0
3	D1	T1
4	D2	T2
5	D3	T3
6	A0	LED0
7	WRT	LED1

25 : Transmit UART

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

How it works

Sends an async uart message on severals pins

How to test

Attach a uart receiver to each pins, set the baud rate to 1200 and read the messages

IO

#	Input	Output
0	clock	uart_tx_0
1	reset	uart_tx_1
2	none	uart_tx_2
3	none	none
4	none	none
5	none	none
6	none	none
7	none	none

26 : RGB LED Matrix Driver

- Author: Matt M
- Description: Drives a simple animation on SparkFun's RGB LED 8x8 matrix backpack
- GitHub repository
- HDL project
- 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)

27 : Tiny Phase/Frequency Detector

- Author: argunda
- Description: Detect phase shifts between 2 square waves.
- GitHub repository
- Wokwi project
- 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

28 : Loading Animation

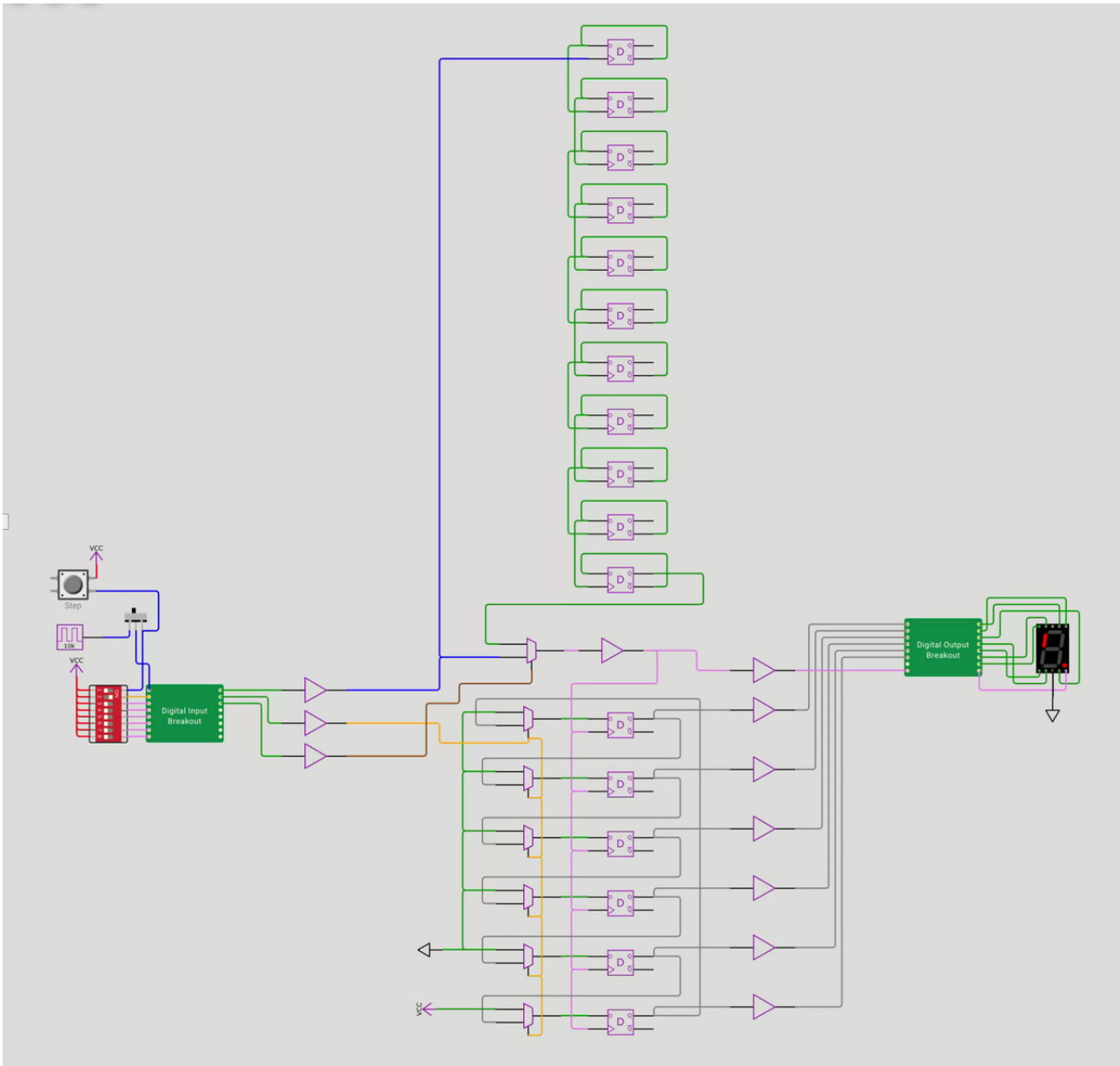

Figure 10: picture

- Author: Andre & Milosch Meriac
- Description: Submission for tt02 - Rotating Dash
- GitHub repository
- Wokwi project
- 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

29 : tiny egg timer

- Author: yubex
- Description: tiny egg timer is a configurable small timer
- GitHub repository
- HDL project
- 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

30 : Potato-1 (Brainfuck CPU)

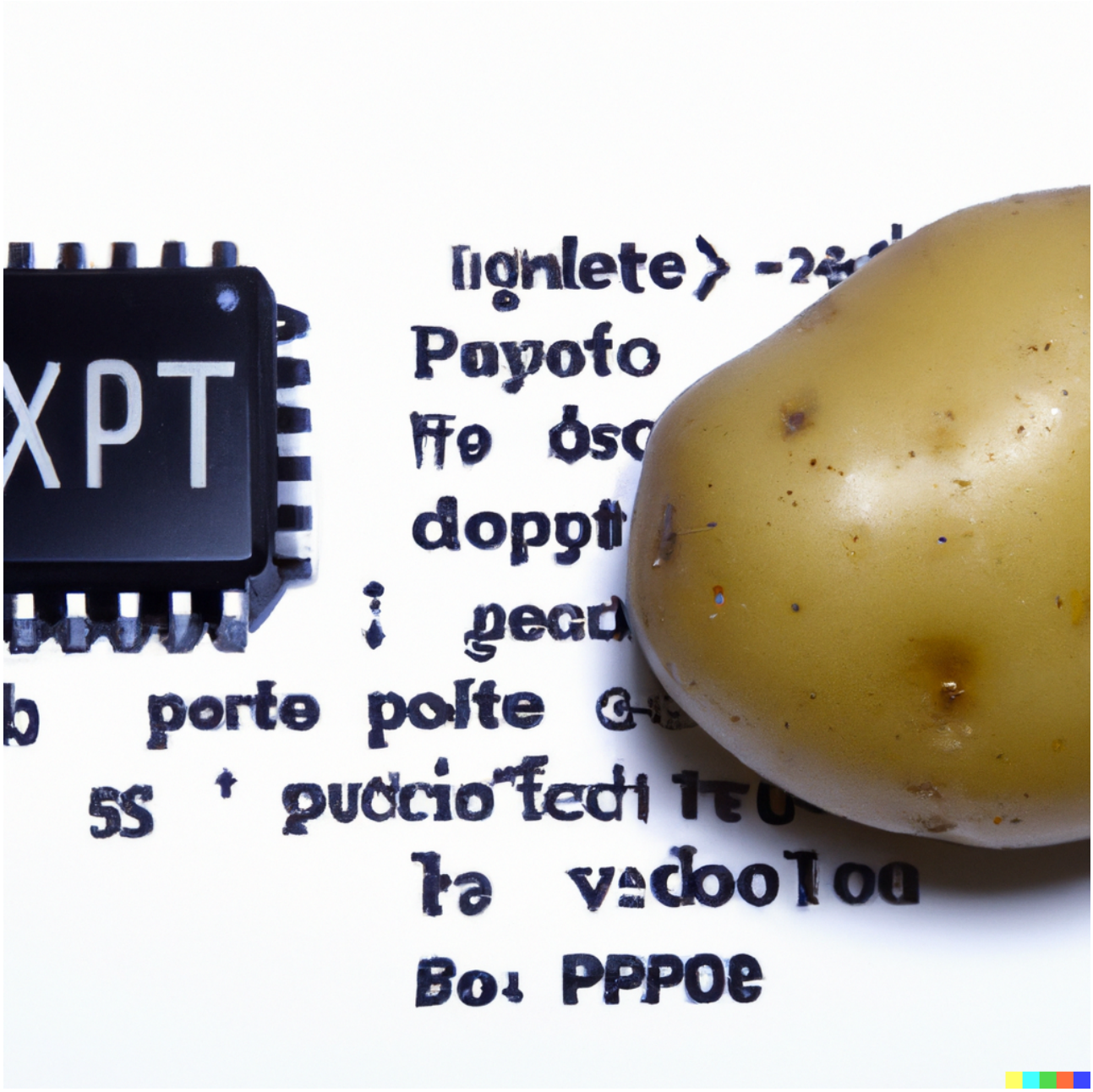

Figure 11: picture

- 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 project

- 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

31 : heart zoe mom dad

- Author: zoe nguyen. taylor
- Description: outputs my name and my age (zoe 4)
- GitHub repository
- HDL project
- 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

32 : 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 project
- 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	Pitch + 1 Octave
1	C	Pitch
2	D	Pitch - 1 Octave
3	E	Pitch - 2 Octave
4	F	none
5	G	none
6	A	none
7	B	none

33 : 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 project
- Extra docs
- Clock: 625 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.
8. Did you know there is a secret CPU inside?

IO

#	Input	Output
0	clock	segment a
1	reset_lfsr	segment b
2	reset_taps	segment c
3	data_in1	segment d
4	data_in2	segment e
5	data_in3	segment f
6	data_in4	segment g
7	data_in5	segment p

34 : prbs15

- Author: Tom Schucker
- Description: generates and checks prbs15 sequences
- GitHub repository
- Wokwi project
- 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

35 : 4-bit badge ALU

- Author: Rolf Widenfelt
- Description: A 4-bit ALU inspired by Supercon.6 badge
- GitHub repository
- HDL project
- 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

36 : Pi ( ) to 1000+ decimal places

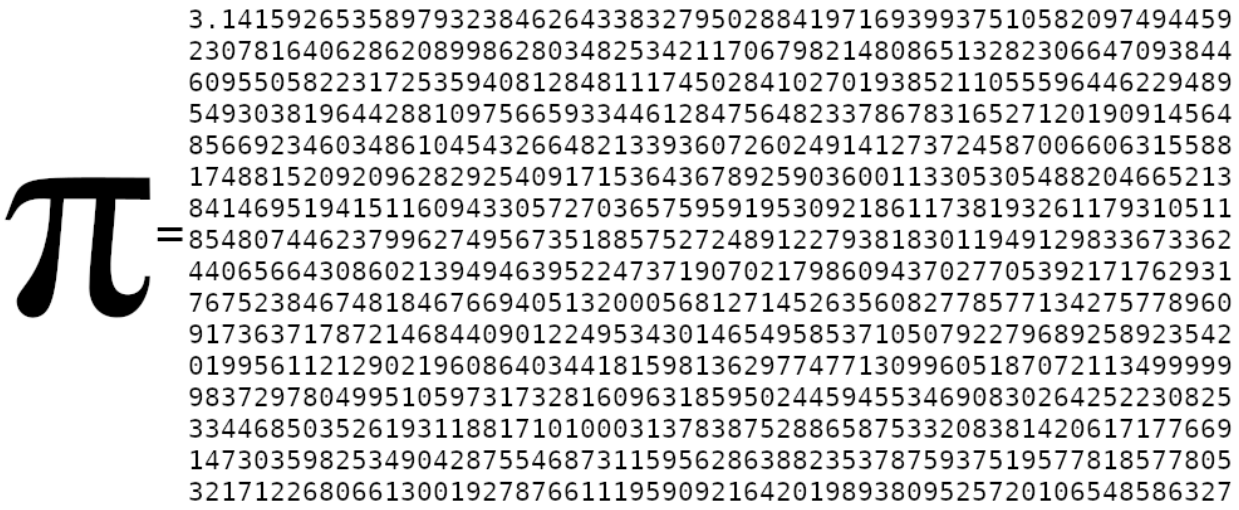

Figure 12: picture

- Author: James Ross
- Description: This circuit outputs the first 1024 decimal digits of Pi ( ), including the decimal after the three. The repository started out as something else, but after completing the 16x8 SRAM circiut (128 bits), I became curious about just how much information could be packed into the circuit area. The D flip flops in SRAM aren't particularly dense and the circuit has other functionality beyond information storage. For this demonstration, I needed something without a logical pattern, something familiar, and something which would exercise all the LEDs in the seven segment display. The Pi constant was perfect. After a number of experiments in Verilog, trying the Espresso Heuristic Logic Minimizer tool, the best results ended up being a large boring block of case statements and letting the toolchain figure it out. The information limit I found was 1023*log2(10)+1 $\sim=$ 3,400 bits, after which the toolchain struggled. However, it appears in this case that the layout is limited by metal, not combinatorial logic. I am interested to hear about better strategies to do something like this with synthesizable Verilog.
- GitHub repository
- HDL project
- Extra docs
- Clock: 0 Hz
- External hardware: The seven segment display is used directly.

How it works

There is some combinatorial logic which generates the first 1024 decimal digits and then decodes those digits (and the decimal) to the 7 segment display.

How to test

The clock is used to drive the incremental changes in the display. The reset pin is used to zero the index.

IO

#	Input	Output
0	clk	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	decimal LED

37 : Siren

- Author: Alan Green
- Description: Pretty patterns and a siren straight from the 1970s
- GitHub repository
- Wokwi project
- 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

38 : YaFPGA

- Author: Frans Skarman
- Description: Yet another FPGA
- GitHub repository
- HDL project
- 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

39 : M0: A 16-bit SUBLEQ Microprocessor

```
int printf(char * s);

int main() {
  int i;

  printf("Hello World!\n");

  for (i = 'A'; i <= 'Z'; i++)
    printf("%c", i);
}
```

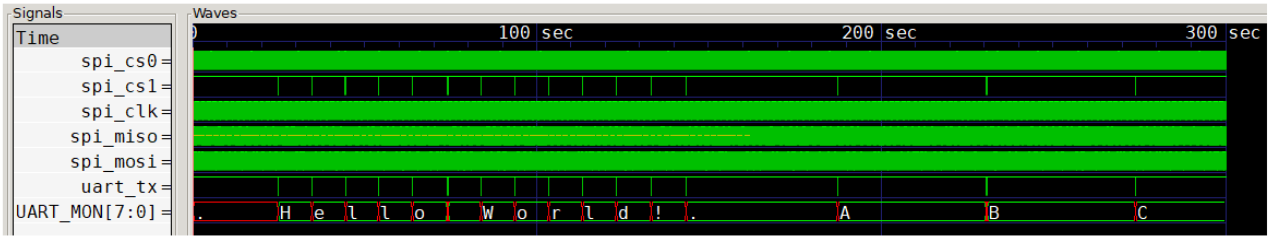

Figure 13: picture

- Author: William Moyes
- Description: A capable but slow microprocessor that fits in a very tight space
- GitHub repository
- HDL project
- Extra docs
- Clock: 12500 Hz
- External hardware: A SPI ROM and RAM for user code

How it works

The M0 is a 16-bit, bit serial microprocessor based upon the SUBLEQ architecture. The only external devices needed for operation are a SPI RAM, SPI ROM, and clock source. The entire ROM and RAM are available for user code. All registers and logic are contained within the M0 itself. A transmit UART is included for serial output. The M0 interoperates with Oleg Mazonka's HSQ C-compiler for SUBLEQ. See https://github.com/moyesw/TT02-M0/blob/main/README.md for full details on the M0.

How to test

Easy check #1 without RAM/ROM chips- Assert Reset High (input1). Hold spi_miso low (input2). Apply a slow clock to both CLK (input0) and DBG_in (input7). Bring Reset Low. Examine the inverted clock output on output7 (DBG_OUT), and compare to clk on in0 to determine io scan chain quality. Examine spi_clk on out3. There should be 40 spi clock pulses at half the clk input frequency, followed by a 2 spi clock gap where no pulses are present.

Easy check #2 without RAM/ROM chips- Assert Reset high (input2). Hold spi_miso low. Apply a clock to CLK (input0). Bring Reset Low. Allow the M0 to reach steady state (504 clock cycles from reset). Observe the UART transmits 0xFF every 504 input clock cycles on output4. Observe that the CS0 and CS1 are accessed in the pattern: CS1, CS1, CS0, CS1, CS1, CS0. Observe that the CS0+1 and the spi_mosi pin encodes the following repeating SPI access pattern: CS1:Rd(03):Addr(FFFE), CS1:Rd(03):Addr(FFFE), CS0:Rd(03):Addr(0000), CS1:Rd(03):Addr(FFFE), CS1:Wr(02):Addr(FFFE), CS0:Rd(03):Addr(8000). Note Each access will be accompanied by 16/17 bits of data movement.

Running code with RAM/ROM chips- Connect a programmed SPI ROM to CS1, and a SPI RAM to CS0. Assert Reset. Power up the ASIC and provide a clock. Lower Reset, and observe execution. The program's serial output will appear on output pin 4 at a baud rate that is one half the input clock frequency. See https://github.com/moyesw/TT02-M0/blob/main/README.md for information on external connections, ROM and RAM data formats, instruction set, and compiler usage.

IO

#	Input	Output
0	clk	spi_cs0
1	rst	spi_cs1
2	spi_miso	spi_clk
3	none	spi_mosi
4	none	uart_tx
5	none	none
6	none	none
7	dbg_in	dbg_out

40 : bitslam

- Author: Jake "ferris" Taylor
- Description: bitslam is a programmable sound chip with 2 LFSR voices.
- GitHub repository
- HDL project
- 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

41 : 8x8 Bit Pattern Player

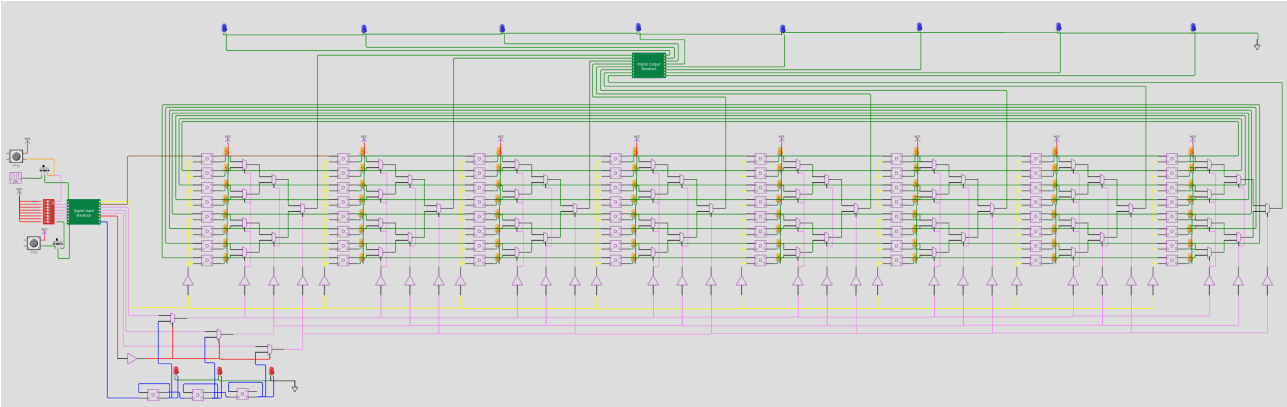

Figure 14: picture

- Author: Thorsten Knoll
- Description: 8x8 bit serial programmable, addressable and playable memory.
- GitHub repository
- Wokwi project
- 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

42 : XLS: bit population count

- Author: proppy
- Description: Count bits set in the input.
- GitHub repository
- HDL project
- 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

43 : RC5 decoder

- Author: Jean THOMAS
- Description: Increment/decrement a counter with the press of an IR remote button!
- GitHub repository
- HDL project
- 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 receiver. 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

44 : chiDOM

- Author: Maria CHiara Molteni
- Description: Chi function of Xoodoo protected at the first-order by DOM
- GitHub repository
- Wokwi project
- 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

45 : Super Mario Tune on A Piezo Speaker

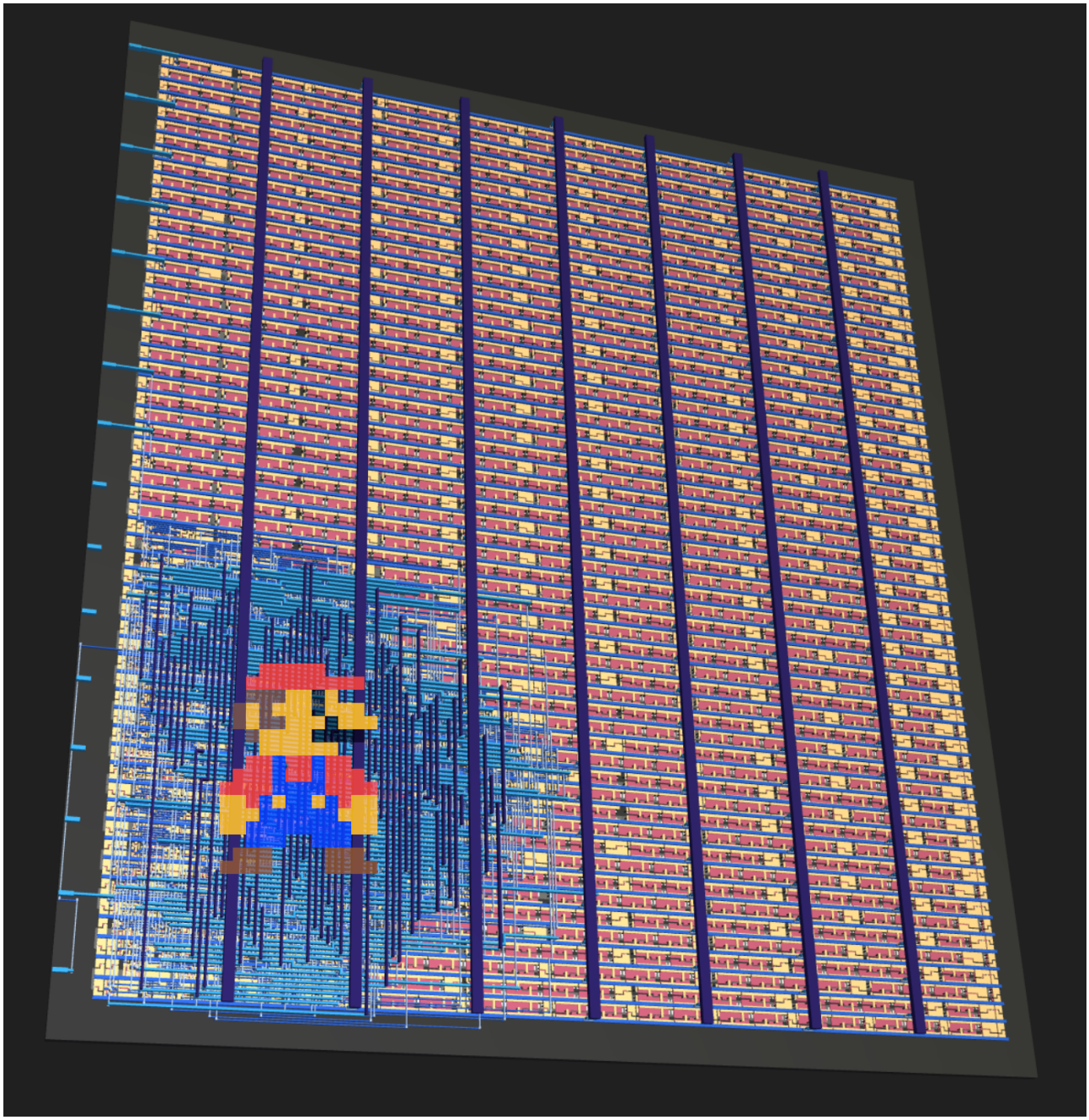

Figure 15: picture

- Author: Milosch Meriac
- Description: Plays Super Mario Tune over a Piezo Speaker connected across io_out[1:0]
- GitHub repository
- HDL project
- Extra docs
- Clock: 3125 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

46 : Tiny rot13

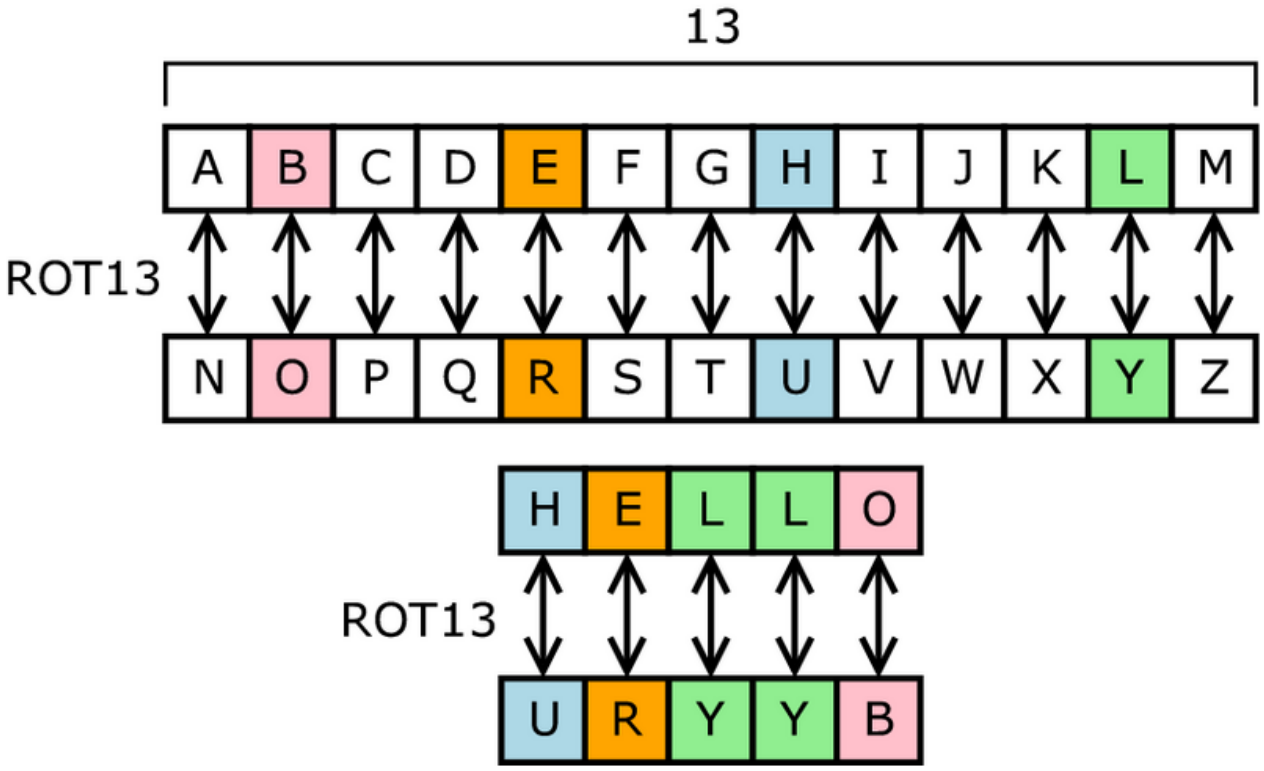

Figure 16: picture

- Author: Phase Noise
- Description: implements rot13 in the constraints of TT02
- GitHub repository
- HDL project
- 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

47 : 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 project
- 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

48 : Shiftregister Challenge 40 Bit

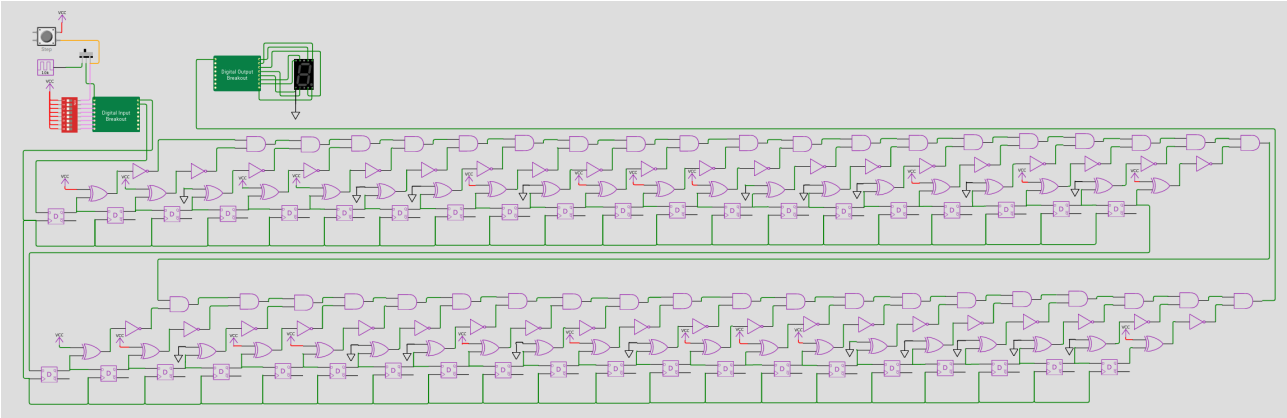

Figure 17: picture

- 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 project
- 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

49 : TinyTapeout2 4-bit multiplier.

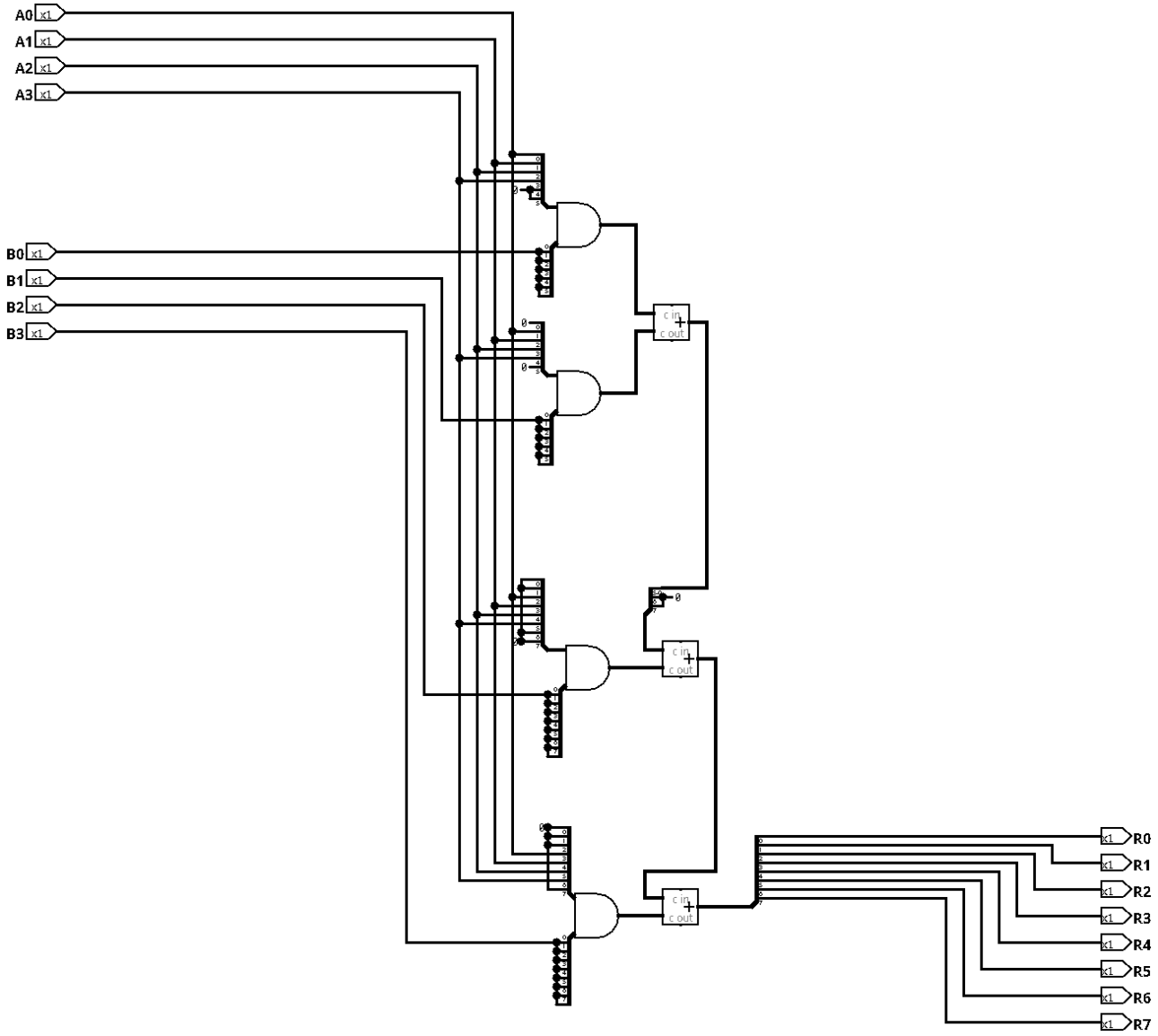

Figure 18: picture

- Author: Tholin
- Description: Multiplies two 4-bit numbers presented on the input pins and outputs an 8-bit result.
- GitHub repository
- HDL project
- 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

50 : 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 project
- 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

51 : XLS: 8-bit counter

- Author: proppy
- Description: Increment output bits
- GitHub repository
- HDL project
- Extra docs
- Clock: 10 Hz
- External hardware: LEDs, pull-up/down resistors

How it works

Implement a simple counter using https://google.github.io/xls/tutorials/intro_to_procs/

How to test

Set the reset bit once, toggle the clock once, unset the reset bit and keep toggling the clock

IO

#	Input	Output
0	clock	count0
1	reset	count1
2	none	count2
3	none	count3
4	none	count4
5	none	count5
6	none	count6
7	none	count7

52 : XorShift32

- Author: Ethan Mahintorabi
- Description: XorShift32 random number generator
- GitHub repository
- HDL project
- 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

53 : XorShift32

- Author: Ethan Mahintorabi
- Description: XorShift32 random number generator
- GitHub repository
- HDL project
- 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

54 : 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 project
- 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

55 : TinyTapeout 2 LCD Nametag

- Author: Tholin
- Description: Echoes out a predefined text onto a 20x4 character LCD.
- GitHub repository
- HDL project
- 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. Also, connect LEDs to LED0 and LED1 if you want some blinkenlights.

IO

#	Input	Output
0	CLK	RS
1	RST	E
2	EF0	D4
3	EF1	D5
4	EF2	D6
5	NC	D7
6	NC	LED0
7	NC	LED1

56 : UART-CC

- Author: Christina Cyr
- Description: UART Template
- GitHub repository
- Wokwi project
- 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

57 : 3-bit 8-channel PWM driver

- Author: Ivan Krasin
- Description: PWM driver with 8 channels and 8 PWM levels from 0 to 1
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware:

How it works

uses a 3-bit counter to drive PWM on 8 output channels. Each channel is controlled by a dedicated 3-bit register that specifies its PWM level: 0 means always off, 1 is for 1/7 on, 5 is for 5/7 on and 7 is 7/7 (always on)

How to test

after reset, all output pins will be low. Use set, addr and level pins to set PWM level=level0+2*level1+4*level2 on channel=addr0+2*addr1+4*addr2. The corresponding pin will start oscillating between 0 and 1 according to the clock and the set level.

IO

#	Input	Output
0	clock	out0
1	pset	out1
2	addr0	out2
3	addr1	out3
4	addr2	out4
5	level0	out5
6	level1	out6
7	level2	out7

58 : LEDChaser from LiteX test

- Author: Nick Østergaard
- Description: This is just a small demo of synthezing verilog from LiteX, this does not include any CPU.
- GitHub repository
- HDL project
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

It just implements LEDChaser from the LiteX LED core demo, where io_in[3:7] is duty cycle

How to test

Add LEDs on the outputs in a straight line – or probe all signals on a scope and check that you get a 'moving' train of pulses.

IO

#	Input	Output
0	clock	led a
1	reset	led b
2	pwm_width 0	led c
3	pwm_width 1	led d
4	pwm_width 2	led e
5	pwm_width 3	led f
6	pwm_width 4	led g
7	pwm_width 5	led h

59 : 8-bit (E4M3) Floating Point Multiplier

- Author: Clive Chan
- Description: 8-bit (E4M3) Floating Point Multiplier
- GitHub repository
- HDL project
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

A small mux in front to fill up two 8-bit buffers in halves, which feed the actual 8-bit multiplier. When ctrl0 is 0, you can clock in 4 bits from data[3:0] into the upper or lower halves of two buffers according to the values of ctrl[1] and ctrl[2]: - 00 STORE 1 LOWER - 01 STORE 1 UPPER - 10 STORE 2 LOWER - 11 STORE 2 UPPER The clock is intended for manual use instead of actually being driven by a clock, but it probably can work. The 8 bits in each of the two buffers are interpreted as an 8-bit floating point number. From MSB to LSB: - sign bit - exponent[3:0] - mantissa[2:0] These are interpreted according to an approximation of IEEE 754, i.e. (-1)**sign * 2**(exponent - EXP_BIAS) * 1.mantissa with the following implementation details / differences: - EXP_BIAS = 7, analogous to 2**(exp-1) - 1 for all IEEE-defined formats - Denormals (i.e. exponent == 0) are flushed to zero on input and output - exponent = 0b1111 is interpreted as more normal numbers instead of NaN/inf, and overflows saturate to the largest representable number (0bx1111111 = +/- 480.0) - Negative zero is interpreted as NaN instead. - Round to nearest even is implemented. The output 8 bits will always display the results of the multiplication of the two FP8's in the buffers, regardless of the clock. The module has been verified over all possible pairs of 8-bit inputs.

How to test

cd src && make

IO

#	Input	Output
0	clock	sign
1	ctrl0	exponent
2	ctrl1	exponent
3	ctrl2	exponent
4	data0	exponent

#	Input	Output
5	data1	mantissa
6	data2	mantissa
7	data3	mantissa

60 : Dice roll

- Author: Tholin
- Description: Will roll a random number from 1 - 6 on the 7-segment display, like a dice.
- GitHub repository
- HDL project
- Extra docs
- Clock: 6000 Hz
- External hardware: A 7-segment-display. The one on-board the PCB will work.

How it works

Contains a LSFR for random numbers, that constantly updates no matter if the dice is rolling or not. Pressing the 'ROLL' button will play an animation of random numbers cycling on the display, until settling on a number after a few seconds. The decimal point will light up when its done rolling.

How to test

Reset, then pulse 'ROLL' to roll the dice as many time as you like.

IO

#	Input	Output
0	CLK	segment a
1	RST	segment b
2	ROLL	segment c
3	NC	segment d
4	NC	segment e
5	NC	segment f
6	NC	segment g
7	NC	decimal point

61 : CNS TT02 Test 1:Score Board

picture

- Author: Bryan Bonilla Garay, Devin Alvarez, Ishaan Singh, Yu Feng Zhou, and N. Sertac Artan
- Description: First test run of CNS Lab. Displays an 8-bit score from one of two players as a two-digit hexadecimal value.
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware: None

How it works

Two counters keep track of user scores, which can be updated, and displayed on the 7-segment display.

How to test

Wokvi

IO

#	Input	Output
0	clock	segment a
1	reset	segment b
2	none	segment c
3	rst	segment d
4	display_digit	segment e
5	display_user	segment f
6	user	segment g
7	mode	dot

62 : Test2

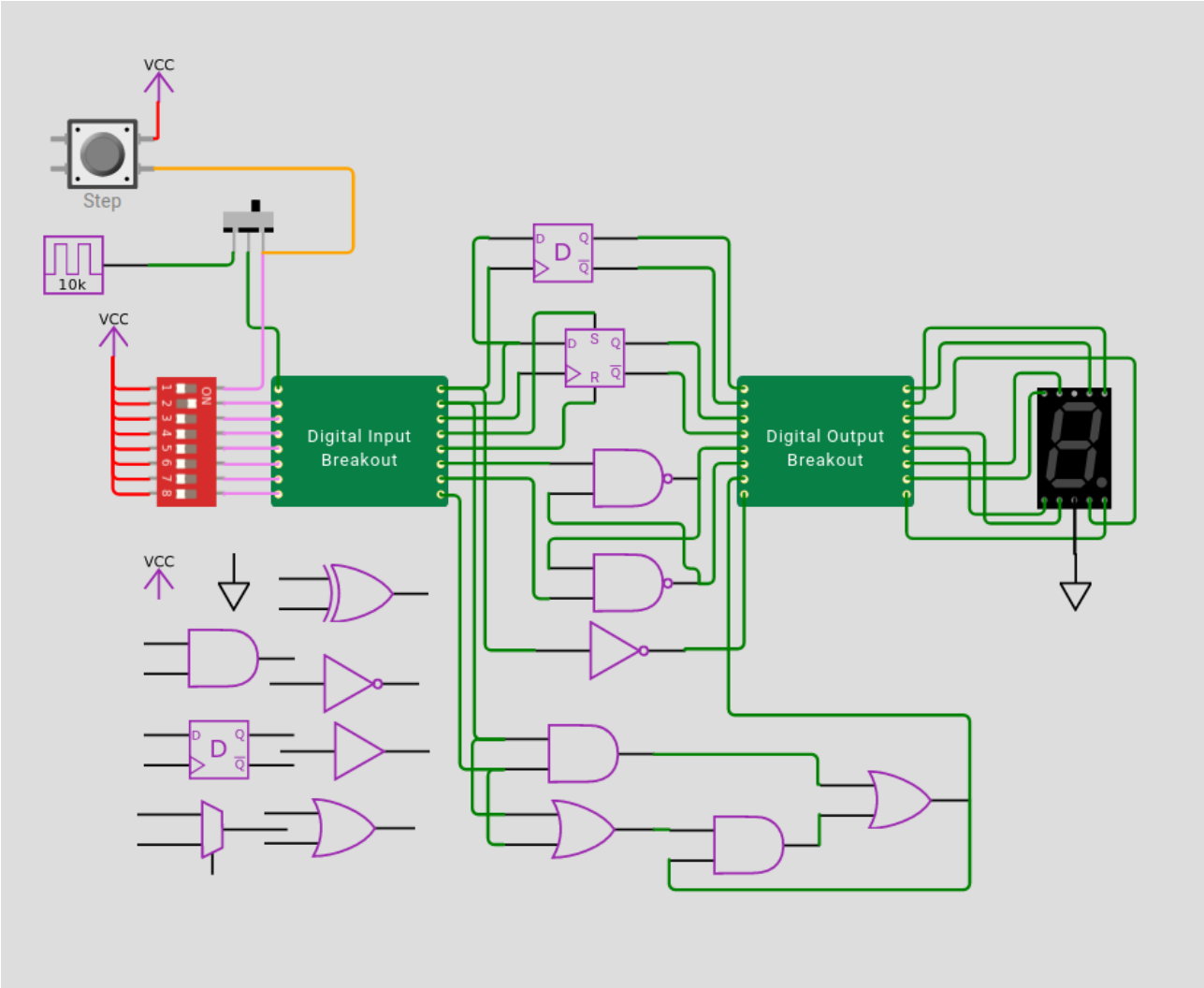

Figure 19: picture

- Author: Shaos
- Description: Testing Flip-Flops
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware: logic analyzer

How it works

nothing special - just testing

How to test

change inputs and see outputs :)

IO

#	Input	Output
0	clock	segment a
1	D	segment b
2	C	segment c
3	S	segment d
4	R	segment e
5	NAND1	segment f
6	NAND2	segment g
7	Muller	dot

63 : 7-segment LED flasher

- Author: Joseph Chiu
- Description: Drives 7-segment LED display, alternating between NIC and JAC
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware: TTO standard switches and 7-segment display

How it works

Master clock is fed through a prescaler with four tap-points which feeds a 4-bit ripple counter (there are 6 total bits, but the top two bits are discarded). 2:1 muxes are chained to act like a 8:1 mux for each LED segment position. As the counter runs, this results in each segment being turned on or off as needed to render the display sequence (NIC JAC). The highest order bit is used to blink the decimal point on/off.

How to test

IN5 and IN6 selects the clock prescaler. OUT0-OUT7 are the LED segment outputs.

IO

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

64 : Nano-neuron

- Author: Daniel Burke
- Description: minimal low vector test
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

bare influence field calculation GV open neurons

How to test

clock in reference vector (some inversions), present DUT and get outupt

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

65 : SQRT1 Square Root Engine

- Author: Davit Margarian (UDXS)
- Description: Computes 4.2 fixed-point square root for any 7-bit integer
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware: Optionally, binary to BCD converter for output

How it works

This uses Newton's method to solve sqrt in 3 cycles.

How to test

Set a 7-bit input value and toggle clock 3 times. After, the output will be correct, rounded down to the nearest 4th.

IO

#	Input	Output
0	clock	frac1
1	i1	frac2
2	i2	whole1
3	i3	whole2
4	i4	whole3
5	i5	whole4
6	i6	none
7	i7	none

66 : Breathing LED

- Author: argunda
- Description: Use the pwm output to drive an LED and it should look like it's breathing.
- GitHub repository
- HDL project
- Extra docs
- Clock: 4000 Hz
- External hardware: Clock source and external LED circuit.

How it works

A triangle wave is generated and used to determine duty cycle of pwm.

How to test

After reset, pwm should automatically be generated. The duty counter is output for debug purposes.

IO

#	Input	Output
0	clock	breathing_pwm
1	reset	duty[0]
2	none	duty[1]
3	none	duty[2]
4	none	duty[3]
5	none	duty[4]
6	none	duty[5]
7	none	duty[6]

67 : Fibonacci & Gold Code

- Author: Daniel Estevez
- Description: This project includes two independent designs: a design that calculates terms of the Fibonacci sequence and displays them in hex one character at a time on a 7-segment display, and a Gold code generator that generates the codes used by CCSDS X-band PN Delta-DOR.
- GitHub repository
- HDL project
- Extra docs
- Clock: 0 Hz
- External hardware: No external hardware is needed

How it works

The Fibonacci calculator uses 56-bit integers, so the terms of the Fibonacci sequence are displayed using 7 hex characters. Since the TinyTapeout PCB only has one 7-segment display, the terms of the Fibonacci sequence are displayed one hex character at a time, in LSB order. The dot of the 7-segment display lights up whenever the LSB is being displayed. On each clock cycle, 4-bits of the next Fibonacci term are calculated using a 4-bit adder, and 4-bits of the current term are displayed in the 7-segment display. The 7-segment display is ANDed with the project clock, so that the digits flash on the display. The Gold code generator computes a CCSDS X-band PN Delta-DOR Gold code one bit at a time using LFSRs. The output bit is shown on the 7-segment display dot. 6-bits of the second LFSR can be loaded in parallel using 6 project inputs in order to be able to generate different sequences. One of the project inputs is used to select whether the 7-segment display dot is driven by the Fibonacci calculator or by the Gold code generator.

How to test

The project can be tested by manually driving the clock using a push button or switch. Just by de-asserting the reset and driving the clock, the digits of the Fibonacci sequence terms should appear on the 7-segment display. The output select input needs to be set to Gold code (high level) in order to test the gold code generator. The load enable input (active-low), as well as the 6 inputs corresponding to the load for the B register can be used to select the sequence to generate. The load value can be set in the 6 load inputs, and then the load enable should be pulsed to perform the load. This can be done with the clock running or stopped, as the load enable is asynchronous. After the load enable is de-asserted, for each clock cycle a new bit of the Gold code sequence should appear in the 7-segment display dot.

IO

#	Input	Output
0	clock	{'segment a': 'Fibonacci hex digit'}
1	output select (high selects Gold code; low selects Fibonacci LSB marker) & Gold code load value bit 0	{'segment b': 'Fibonacci hex digit'}
2	Fibonacci reset (active-low; asynchronous) & Gold code load value bit 1	{'segment c': 'Fibonacci hex digit'}
3	Gold code load enable (active-low; asynchronous)	{'segment d': 'Fibonacci hex digit'}
4	Gold code load value bit 2	{'segment e': 'Fibonacci hex digit'}
5	Gold code load value bit 3	{'segment f': 'Fibonacci hex digit'}
6	Gold code load value bit 4	{'segment g': 'Fibonacci hex digit'}
7	Gold code load value bit 5	{'none': 'Gold code output / Fibonacci LSB digit marker'}

68 : tinytapeout2-HELLo-3orLd-7seg

Figure 20: picture

- Author: Rakesh Peter
- Description: HELLo-3orLd Runner on 7 segment Display
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 1 Hz
- External hardware:

How it works

BCD to 7seg Counter is modified to suit the Simplified SoP equation for each segments. See the repo for SoP computation.

How to test

All toggle switches in zero position and clock switch on for auto runner. Individual BCD bits can be toggled using corresponding inputs with clock switch off.

IO

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

69 : Non-restoring Square Root

Figure 21: picture

- Author: Wallace Everest
- Description: Square root for use in RMS calculations
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware: Result on 7-segment display {0x0..0xB}

How it works

7-bit input, 4-bit output, unsigned

How to test

Apply unsigned input {0x0..0x7F} to the logic pins

IO

#	Input	Output
0	clk	segment a
1	data(0)	segment b
2	data(1)	segment c
3	data(2)	segment d
4	data(3)	segment e
5	data(4)	segment f
6	data(5)	segment g
7	data(6)	segment dp

70 : GOL-Cell

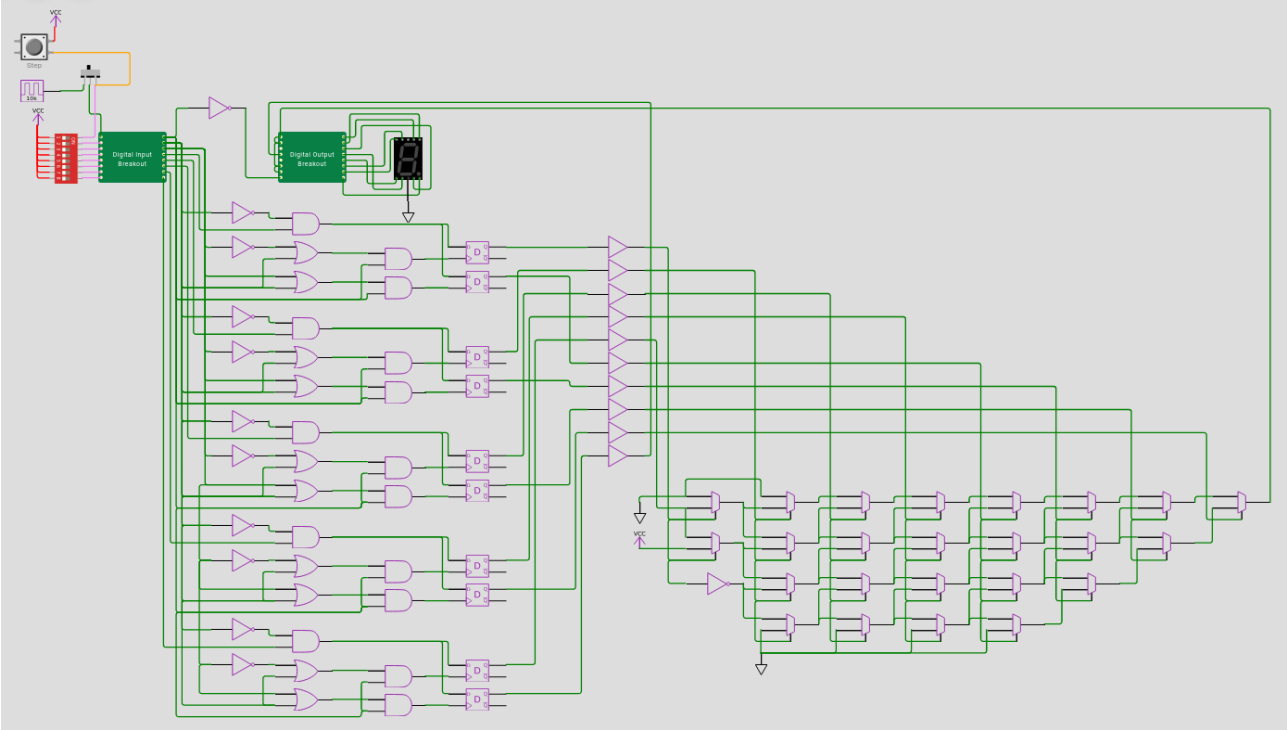

Figure 22: picture

- Author: Shaos
- Description: Game of Life Cell
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware: 8 neighbours and rerouted current state need to go in 2 stages using 5 inputs

How it works

Calculate survive/die decision based on number of neighbours and current state

How to test

Change number of neighbours and see

IO

#	Input	Output
0	clock	segment a
1	reset	segment b
2	half	segment c
3	input 0 or 5	segment d
4	input 1 or 6	segment e
5	input 2 or 7	segment f
6	input 3 or 8	segment g
7	input 4 or 9	inverted clock

71 : 7-channel PWM driver controlled via SPI bus

- Author: Ivan Krasin
- Description: PWM driver with 7 channels and 256 PWM levels from 0 to 1
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware:

How it works

uses a 8-bit counter to drive PWM on 7 output channels. Each channel is controlled by a dedicated 8-bit register that specifies its PWM level: 0 means always off, 1 is for 1/255 on, 5 is for 5/255 on and 255 is 255/255 (always on)

How to test

after reset, all output pins will be low. Use SPI writes with register addresses (0..6) to set 8-bit PWM levels. The corresponding pin will start oscillating between 0 and 1 according to the clock and the set level.

IO

#	Input	Output
0	clock	out0
1	reset	out1
2	cs	out2
3	sclk	out3
4	mosi	out4
5	reserved	out5
6	reserved	out6
7	reserved	miso

72 : hex shift register

- Author: Eric Smith
- Description: six 40-bit shift registers
- GitHub repository
- HDL project
- Extra docs
- Clock: Hz
- External hardware:

How it works

Six 40-bit shift registers. A multiplexer selects input data or recirulating output data.

How to test

on each clock n, six bits are shifted in, and the six bits that were input at clock n-4 are output

IO

#	Input	Output
0	clk	none
1	recirc	none
2	data_in[0]	data_out[0]
3	data_in[1]	data_out[1]
4	data_in[2]	data_out[2]
5	data_in[3]	data_out[3]
6	data_in[4]	data_out[4]
7	data_in[5]	data_out[5]

73 : Ring OSC Speed Test

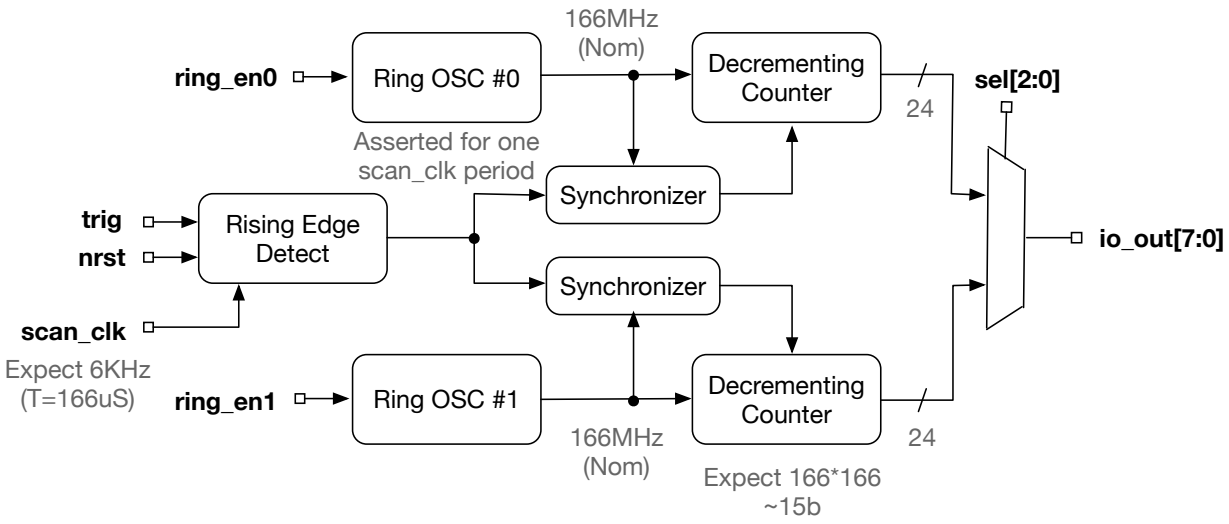

Figure 23: picture

- Author: Eric Smith
- Description: Make two rings with the same number of stages but measure how their frequency differs. Measure if they can influence eachother.
- GitHub repository
- HDL project
- Extra docs
- Clock: 6000 Hz
- External hardware: Something to sequence nrst, ring_en, trig and the sel bits

How it works

uses a register and some combinational logic

How to test

after reset, assert trigger. Use sel bits to get result

IO

#	Input	Output
0	clock	out[0]
1	nreset	out[1]
2	trig	out[2]
3	sel[0]	out[3]
4	sel[1]	out[4]

#	Input	Output
5	sel[2]	out[5]
6	ring_en[0]	out[6]
7	ring_en[1]	out[7]

74 : TinyPID

- Author: Aidan Medcalf
- Description: Tiny PID controller with SPI configuration channel, SPI ADC and DAC driver
- GitHub repository
- HDL project
- Extra docs
- Clock: 1 Hz
- External hardware: One shift register / ADC for PV read, one shift register / DAC for stimulus output.

How it works

TinyPID reads from a shift register, calculates error and PID values, and writes to a shift register. All parameters of this process are configurable.

How to test

Shift in config, then shift in PV input and see what happens. There are three bytes of configuration (setpoint, kp, ki), which are zero on startup.

IO

#	Input	Output
0	clock	pv_in_clk
1	reset	pv_in_cs
2	none	out_clk
3	cfg_clk	out_mosi
4	cfg_mosi	out_cs
5	none	none
6	cfg_cs	none
7	pv_in_miso	none

75 : TrainLED2 - RGB-LED driver with 8 bit PWM engine

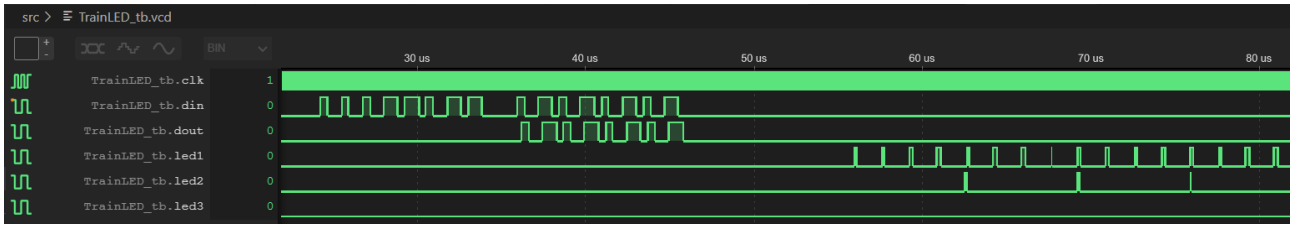

Figure 24: picture

- Author: cpldcpu
- Description: A RGB-LED driver using the WS2812 protocol
- GitHub repository
- HDL project
- Extra docs
- Clock: at least 6000 Hz
- External hardware: LEDs should be connected to the three LED outputs. The data input should be driven by a microcontroller, generating input data in a slowed down WS2812 scheme.

How it works

A fully digital implementation of an RGB LED driver that accepts the WS2812 protocol for data input. The design is fully clocked, so the timing parameters of the protocol depend on the clock rate. A pulse between 1 and 5 clock cycles on the input will be interpreted as a zero, longer pulses as a one. Each driver accepts 3x8=24 bit of input data to set the brightness of LED1,LED2 and LED3 (R,G,B). After 24 bit have been received, additional input bits are retimed and forwarded to the data output. After the data input was idle for 96 clock cycles, the input data is latched into the PWM engine and the data input is ready for the next data frame. The PWM engine uses a special dithering scheme to allow flicker free LED dimming even for relatively low clock rates.

How to test

Execute the shell script 'run.sh' in the src folder. This will invoke the test bench.

IO

#	Input	Output
0	clock	Dout Driver A
1	reset	LED1A
2	Din Driver A	LED2A

#	Input	Output
3	none	LED3A
4	none	none
5	none	none
6	none	none
7	none	none

76 : Zinnia+ (MCPU5+) 8 Bit CPU

picture

- Author: cpldcpu
- Description: A minimal 8 bit CPU
- GitHub repository
- HDL project
- Extra docs
- Clock: high Hz
- External hardware: External program memory and bus demultiplexer is required.

How it works

The CPU is based on the Harvard Architecture with separate data and program memories. The data memory is completely internal to the CPU. The program memory is external and is accessed through the I/O. All data has to be loaded as constants through machine code instructions. Two of the input pins are used for clock and reset, the remaining ones are used as program input, allowing for an instruction length of 6 bit. The output is multiplexed between the program counter (when clk is '1') and the content of the main register, the Accumulator. Interpreting the accumulator content allows reading the program output.

How to test

Execute the shell script 'run.sh primes' in the src folder. This will invoke the testbench with a rom emulator and execute a small program to compute prime numbers.

IO

#	Input	Output
0	clock	cpu_out[0]
1	reset	cpu_out[1]
2	inst_in[0]	cpu_out[2]
3	inst_in[1]	cpu_out[3]
4	inst_in[2]	cpu_out[4]
5	inst_in[3]	cpu_out[5]
6	inst_in[4]	cpu_out[6]
7	inst_in[5]	cpu_out[7]

77 : 4 bit CPU

picture

- Author: Paul Campell
- Description: simple cpu
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware: sram, latch

How it works

It has a 4-bit accumulator, a 7-bit PC, 2 7-bit index registers and a carry bit. The main limitations are the 6/8-bit bus - it's designed to run with an external SRAM and a 7-bit address latch, code is loaded externally. There are 25 instructions. each 2 or 3 nibbles: - 0 V: add a, V(x/y) - sets C - 1 V: sub a, V(x/y) - sets C - 2 V: or a, V(x/y) - 3 V: and a, V(x/y) - 4 V: xor a, V(x/y) - 5 V: mov a, V(x/y) - 6 V: movd a, V(x/y) - 7 0: swap x, y - 7 1: add a, c - 7 2: mov x.l, a - 7 3: ret - 7 4: add y, a - 7 5: add x, a - 7 6: add y, #1 - 7 6: add x, #1 - 8 V: mov a, #V - 9 V: add a, #V - a V: movd V(x/y), a - b V: mov V(x/y), a - c H L: mov x, #hl - d H L: jne a/c, hl if H[3] the test c otherwise test a - e H L: jeq a/c, hl if H[3] the test c otherwise test a - f H L: jmp/call hl if H[3] call else jmp Memory is 128/256 (128 unified or 128xcode+128xdata) 4-bit nibbles, references are a 3 bit (8 nibble) offset from the X or Y index registers - the general idea is that the Y register points to an 8 register scratch pad block (a bit like an 8051) but can also be repurposed for copies when required. There is an on-chip SRAM block for data access only (addressed with the MSB of the data address) - mostly just to soak up any additional gates. There is also a 4-deep hardware call stack.

How to test

needs a 7-bit external address latch and SRAM

IO

#	Input	Output
0	clock	data_out_0
1	reset	data_out_1
2	ram_data0	data_out_2
3	ram_data1	data_out_3

#	Input	Output
4	ram_data2	write_data_n
5	ram_data3	write_ram_n
6	io_data0	a
7	io_data1	strobe

78 : Stack Calculator

- Author: David Siaw
- Description: A stack based 4-bit calculator featuring a 4-bit wide 8 entry deep stack and 64 bits of random access memory.
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware:

How it works

The stack calculator is a 4-bit calculator. It is meant to be used in a larger circuit that will handle timing and memory. It is not a processor since it does not contain a program counter or attempt to access memory on its own. Rather, it accepts inputs in particular sequences and gives outputs depending on the instructions provided.

The stack calculator consists of a 4-bit wide stack that is 8 entries deep. The thing that makes it a stack calculator is the fact that all operations are performed against this stack. The user will provide opcodes at every upwards tick of the clock cycle to instruct the machine on what to do next.

The stack calculator can also save 16 4-bit values using the `SAVE` and `LOAD` operations.

All opcodes are 4 bits long. Some opcodes accept one additional input that define the operation. Ops take at least 2 cycles to complete and at most 4 cycles.

All input must be provided in a particular order and in the following timing:

```
CLK ___/----\___/----\___/----\___/----\___/----\___/----\___/----\___/--

OP2 _X OPC X OPER X______

OP3 _X OPC X OPER X____________

OP4 _X OPC X OPER X______________________
```

The stack machine also features an output register that can be written to using the `OUTL` and `OUTH` operations.

RESET PIN - Please hold the reset pin high and tick the clock at least 4 cycles to reset the machine.

MODE PINS - The input pins 6 and 7 are the mode pin. They can be used to set the output pins to output specific things depending on their value:

- 00 - show contents of the output register
- 01 - show 7-segment display of the top of the stack

- 10 - show 7-segment display of the value just beneath the top of the stack
- 11 - show the top 2 values on the stack on the low and high nibbles respectively.

The list opcodes are as follows:

- 0x1 PUSH
  - Pushes a value to the stack. The value must be provided in the following cycle.
  - 3 cycles - push, value, wait
- 0x2 POP
  - Pops a value from the stack. The value must be provided in the following cycle.
  - 3 cycles - pop, wait, wait
- 0x3 OUTL
  - Copies the value on the top of the stack to the lower 4 bits of the output register.
  - 2 cycles - outl, wait
- 0x4 OUTH
  - Copies the value on the top of the stack to the high 4 bits of the output register.
  - 2 cycles - outh, wait
- 0x5 SWAP
  - Swaps the top two values on the stack.
  - 3 cycles - swap, wait, wait
- 0x6 PUSF
  - Push a value on the stack depending on the operand. The operand determines the values pushed on the stack.
  - 3 cycles - peek/dupl/flag, wait, wait
    - 0x0 DUPL - pushes a copy of the value on the top of the stack to the top of the stack
    - 0x1 PEEK - pushes a copy of the value below the top of the stack to the top of the stack
    - 0x2 FLAG - pushes the contents of the status register
- 0x7 REPL
  - Removes the value at the top of the stack and pushes the value modified by an unary operation
  - 3 cycles - not/neg/incr/decr/shr1/shr2/ror1/rol1, wait, wait
    - 0x0 NOT - bitwise NOT
    - 0x1 NEG - negative, or 2's complement
    - 0x2 INCR - increment
    - 0x3 DECR - decrement
    - 0x4 SHR1 - shift right by 1
    - 0x5 SHL1 - shift left by 1
    - 0x6 ROR1 - rotate right by 1

        * 0x7 ROL1 - rotate left by 1
- 0x8 BINA
  - Binary operation - removes the top two values of the stack and pushes the result of a binary operation
  - 3 cycles - add/and/not/xor/addc/mull/mulh, wait, wait
    * 0x0 ADD - add (will set the status register carry flag if result > 15)
    * 0x1 AND - bitwise AND
    * 0x2 NOT - bitwise NOT
    * 0x3 XOR - bitwise XOR
    * 0x4 ADDC - add with carry. same as add but $+1$ if carry flag is set
    * 0x5 MULL - low nibble from result of multiplication
    * 0x6 MULH - high nibble from result of multiplication
- 0x9 MULT
  - Full multiply of the top two nibbles on the stack. Pushes the high nibble and then the low nibble to the stack in that order.
  - 4 cycles - mult, wait, wait, wait
- 0xA IDIV
  - Divide the value below the top of the stack by the value on the top of the stack. Pushes the remainder and the integer division result in order.
  - 4 cycles - idiv, wait, wait, wait
- 0xB CLFL
  - Unset all flags in flag register
  - 4 cycles - clfl, wait
- 0xC SAVE
  - Writes the value below the top of the stack to the address provided at the top of the stack.
  - 4 cycles - save, wait, wait, wait
- 0xD LOAD
  - Loads the value at the address provided at the top of the stack.
  - 4 cycles - load, wait, wait, wait

How to test

"TODO"

IO

#	Input	Output
0	clk	output0
1	rst	output1
2	input0	output2
3	input1	output3

#	Input	Output
4	input2	output4
5	input3	output5
6	mode0	output6
7	mode1	output7

79 : 1-bit ALU

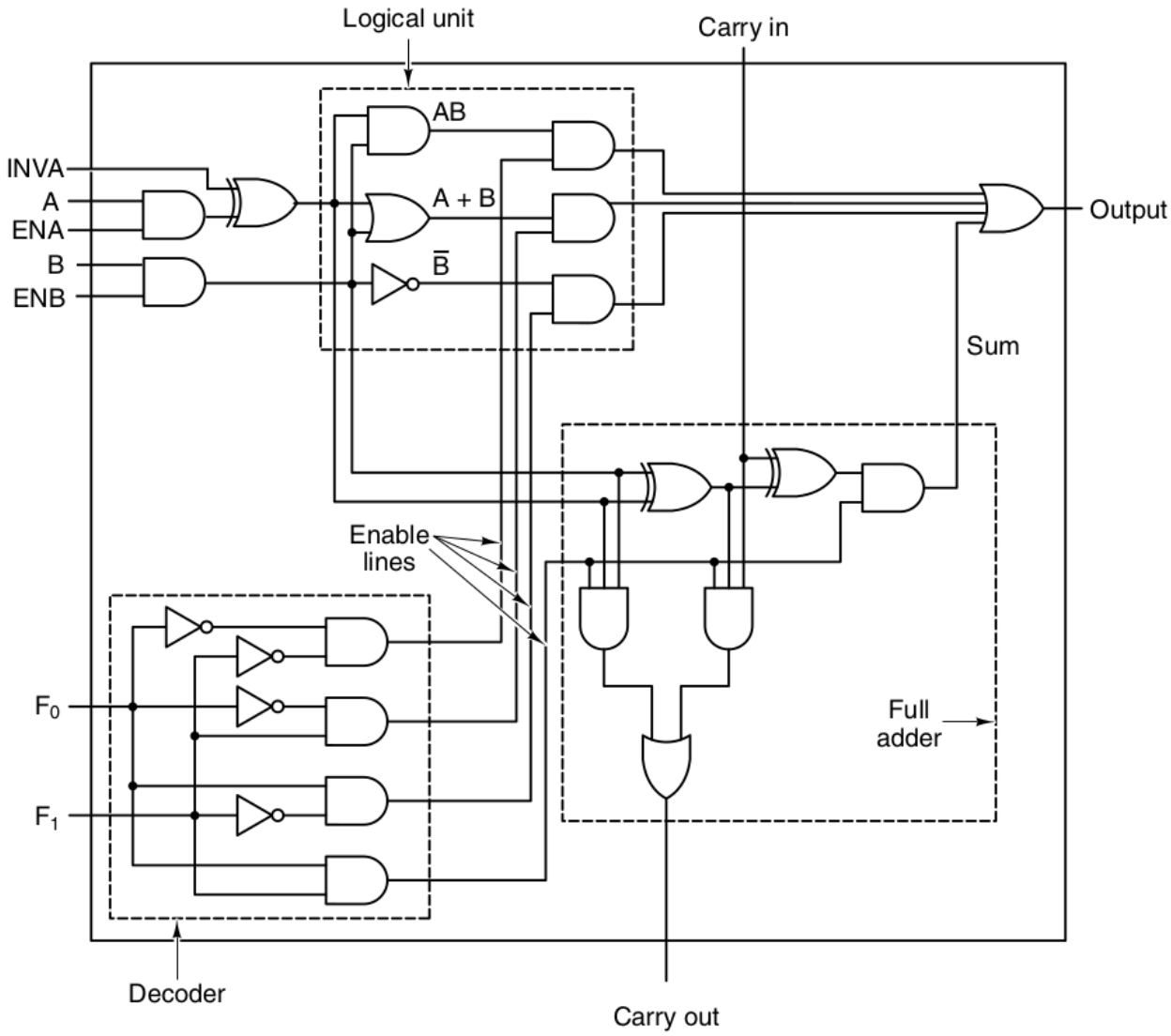

Figure 25: picture

- Author: Leo Moser
- Description: 1-bit ALU from the book `Structured Computer Organization: Andrew S. Tanenbaum`
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware: None

How it works

The 1-bit ALU implements 4 different operations: AND, NOT, OR, ADD. The current operating mode can be selected via F0 and F1. F0=0 and F1=0 results in A AND

B. F0=1 and F1=0 results in NOT B. F0=0 and F1=1 results in A OR B. F0=1 and F1=1 results in A ADD B. Where A and B are the inputs for the operation. Additional inputs can change the way of operation: `ENA` and `ENB` enable/disable the respective input. `INVA` inverts A before applying the operation. `CIN` is used as input for the full adder. Multiple 1bit ALUs could be chained to create a wider ALU.

How to test

Set the operating mode via the DIP switches with F0 and F1. Next, set the input with A and B and enable both signals with ENA=1 and ENB=1. If you choose to invert A, set INVA to 1, otherwise to 0. For F0=1 and F1=1 you can set CIN as additional input for the ADD operation. The 7-segment display shows either a `0` or a `1` depending on the output. If the ADD operation is selected, the dot of the 7-segment display represents the COUT.

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	COUT

80 : SPI Flash State Machine

- Author: Greg Steiert
- Description: Implements a state machine stored in an external SPI flash
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware: SPI Flash with 0x03 read command and 24bit address

How it works

Inputs and current state are shifted into a SPI flash to look up the next state and outputs

How to test

Connect a SPI flash device loaded with state machine values

IO

#	Input	Output
0	clock	cs
1	reset	dout
2	din	out0
3	in0	out1
4	in1	out2
5	in2	out3
6	in3	out4
7	in4	out5

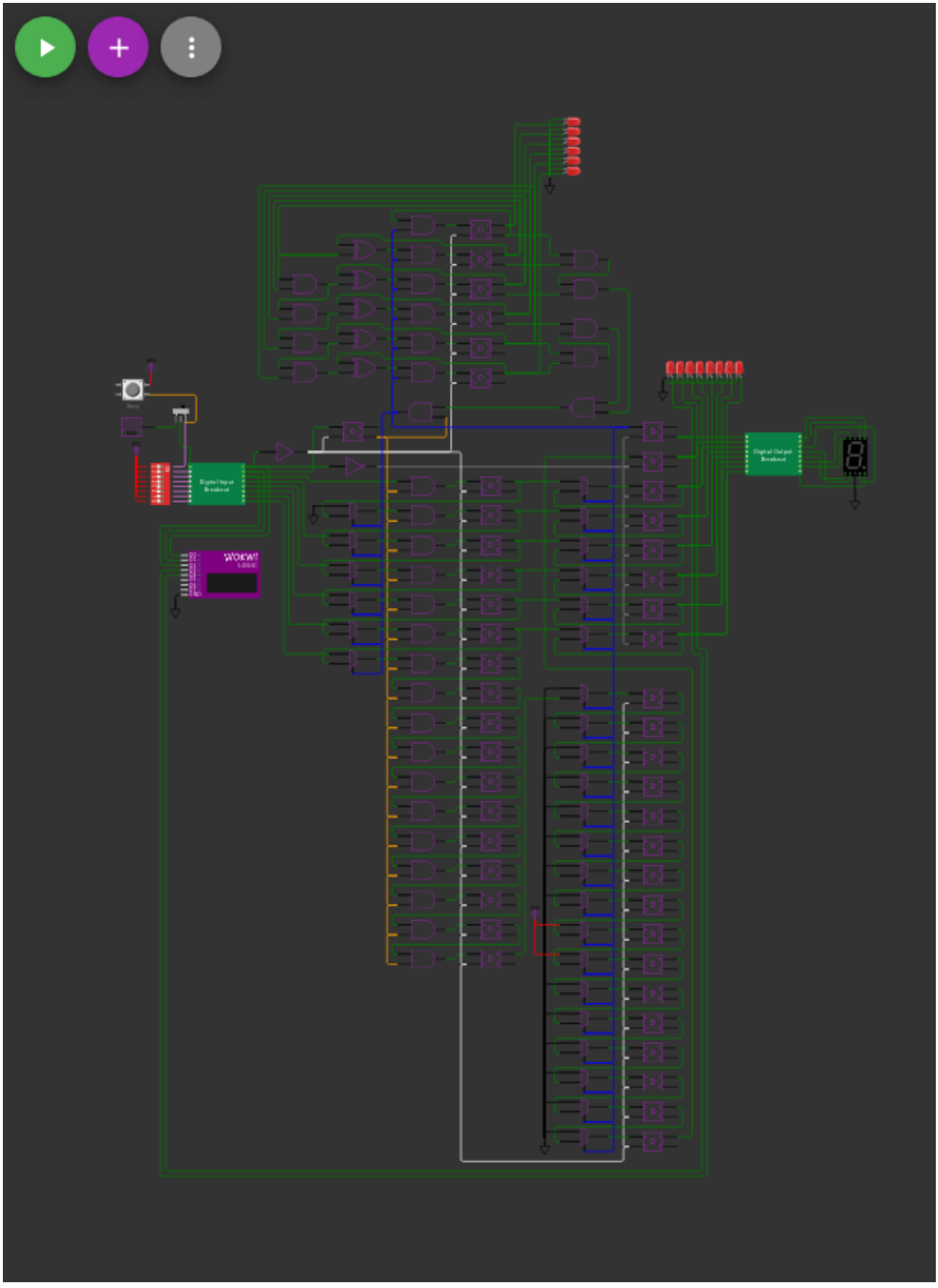

Figure 26: picture

81 : r2rdac

- Author: youngpines
- Description: small r2r
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware: resistors, opamp, and you

How it works

add a resistor ladder on the d flip flop outputs and you get a dac. AND gate is removed, pin2 is a passthrough

How to test

attach a r2r ladder and a non-inverting op-amp on the output and you can control the adc output

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

82 : Worm in a Maze

- Author: Tim Victor
- Description: Animation demo on seven-segment LED
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 25 Hz
- External hardware:

How it works

A segmented worm travels along a pseudo-random path

How to test

Maximum clock divider will probably be best

IO

#	Input	Output
0	clock	LED segment a
1	disable auto-reset	LED segment b
2	manual reset	LED segment c
3	disable /16 clock divider ("turbo mode")	LED segment d
4	display 2 or 3 worm segments	LED segment e
5	none	LED segment f
6	none	LED segment g
7	none	none

83 : 8 bit CPU

picture

- Author: Paul Campell
- Description: 8-bit version of the MoonBase 4-bit CPU
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware: sram, latch

How it works

It has an 8-bit accumulator, a 12-bit PC, 2 13-bit index registers and a carry bit. The main limitations are the 6/8-bit external bus - it's designed to run with an external SRAM and a 12-bit address latch, code is loaded externally. There are 33 instructions. each 1, 2 or 3 bytes: 0v: add a, v(x/y) - sets C 1v: sub a, v(x/y) - sets C 2v: or a, v(x/y) 3v: and a, v(x/y) 4v: xor a, v(x/y) 5v: mov a, v(x/y) 6v: movd a, v(x/y) 70: add a, c 71: inc a 72: swap x, y 73: ret 74: add y, a 75: add x, a 76: add y, #1 77: add x, #1 78: mov a, y 79: mov a, x 7a: mov b, a 7b: swap b, a 7c: mov y, a 7d: mov x, a 7e: clr a 7f: mov a, p 8v: nop 9v: nop av: movd v(x/y), a bv: mov v(x/y), a cv: nop dv: nop ev: nop f0 HL: mov a, #HL f1 HL: add a, #HL f2 HL: mov y, #EEHL f3 HL: mov x, #EEHL f4 HL: jne a/c, EEHL if EE[4] the test c otherwise test a f5 HL: jeq a/c, EEHL if EE[4] the test c otherwise test a f6 HL: jmp/call EEHL f7 HL: nop Memory is 4096 8-bit bytes, references are a 3 bit (8 byte) offset from the X or Y index registers - the general idea is that the Y register points to a register scratch pad block (a bit like an 8051) but can also be repurposed for copies when required. There is an on-chip SRAM block for data access only (addressed with the MSB of the data address) - mostly just to soak up any additional gates. There is also a 3-deep hardware call stack. Assembler is here: https://github.com/MoonbaseOtago/tt-asm

How to test

needs a 7-bit external address latch and SRAM

IO

#	Input	Output
0	clock	data_out_0
1	reset	data_out_1
2	ram_data0	data_out_2

#	Input	Output
3	ram_data1	data_out_3
4	ram_data2	write_data_n
5	ram_data3	write_ram_n
6	io_data0	a
7	io_data1	strobe

84 : Pseudo-random number generator

- Author: Thomas Böhm thomas.bohm@gmail.com
- Description: Pseudo-random number generator using a 16-bit Fibonacci linear-feedback shift register
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware: None

How it works

16 flip flops are connected in a chain, and the output of some is XORed together and fed back into the first flip flop. The outputs that are XORed together are chosen in such a way as to give the longest possible cycle (2^16-1). All bits being zero is a special case and is treated separately (all negative outputs of the flip flops are ANDed together to generate a 1 as feedback). On each clock pulse (pin 1) one new bit is generated. Setting load_en (pin 3) to HIGH allows the loading of a user defined value through the data_in pin (pin2). On each clock pulse one bit is read into the flip flop chain. When load_en (pin 3) is set to LOW the computed feedback bit is fed back into the flip flops. The outputs of the last 8 flip flops are connected to the output pins. For each clock pulse a random bit is generated and the other 7 are shifted.

How to test

Set the switch for pin 1 so that the push button generates the clock. Press on it and see the output change on the hex display. Using pin 2 and 3 a custom value can be loaded into the flip flops.

IO

#	Input	Output
0	clock	random bit 0
1	data_in	random bit 1
2	load_en	random bit 2
3	none	random bit 3
4	none	random bit 4
5	none	random bit 5
6	none	random bit 6
7	none	random bit 7

85 : BCD to 7-Segment Decoder

- Author: JinGen Lim
- Description: Converts a BCD input into a 7-segment display output
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware: 7-segment display

How it works

The IC accepts four binary-coded decimal input signals, and generates a corresponding 7-segment output signal

How to test

Connect the segment outputs to a 7-segment display. Configure the input (IN0:0, IN1:2, IN2:4, IN3:8). The input value will be shown on the 7-segment display

IO

#	Input	Output
0	input 1 (BCD 1)	segment a
1	input 2 (BCD 2)	segment b
2	input 3 (BCD 4)	segment c
3	input 4 (BCD 8)	segment d
4	decimal dot (passthrough)	segment e
5	output invert	segment f
6	none	segment g
7	none	segment dot

86 : Frequency Counter

- Author: Andrew Ramsey
- Description: Estimates the frequency of an input signal
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware: Clock and test signal generator

How it works

'Diffs' (XORs) the previous input with the current one to detect any edges (rising or falling). The edges are then fed into a windowed sum (think moving average, but without the division step). The summation is then converted into a value from 0-9 based on how close to the maximum frequency it is, where 0 is [0, 10)%, 1 is [10, 20)%, etc. which is displayed on the seven segment.

How to test

Input a clock into the clock pin, toggle reset, and verify that the seven segment reads 0. Then apply a test signal and check that the seven segment displays the expected relationship between the clock and test signals. The actual frequency of the clock doesn't matter as long as timing constraints of the chip are met. 1000 Hz makes for convenient math and is a good starting point. If needed, the design will loop the input signal back to the output for a quick sanity check.

IO

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

87 : Taillight controller of a 1965 Ford Thunderbird

- Author: Hirosh Dabui
- Description: Asic of a Taillight controller of a 1965 Ford Thunderbird
- GitHub repository
- HDL project
- Extra docs
- Clock: 6250 Hz Hz
- External hardware:

How it works

uses a moore statemachine

How to test

after reset, the statemachine runs into idle mode

IO

#	Input	Output
0	clock	r3
1	reset	r2
2	left	r1
3	right	l1
4	hazard	l2
5	none	l3
6	none	none
7	none	none

88 : FPGA test

- Author: myrtle
- Description: small mux2 fpga test
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware: TODO write up

How it works

TODO write up

How to test

TODO write up

IO

#	Input	Output
0	clock	out 0
1	cfg_frameinc	out 1
2	cfg_framestrb	out 2
3	cfg_mode	out 3
4	cfg_sel0_in0	out 4
5	cfg_sel0_in1	out 5
6	cfg_sel0_in2	out 6
7	cfg_sel0_in3	out 7

89 : chi 2 shares

- Author: Maria Chiara Molteni
- Description: Chi function of Xoodoo protected by TI with two shares
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Chi function of Xoodoo protected by TI with two shares

How to test

Set on the last 4 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

90 : chi 3 shares

- Author: Molteni Maria Chiara
- Description: Chi function of Xoodoo protected by TI with three shares
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Chi function of Xoodoo protected by TI with three shares

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

91 : Whisk: 16-bit Serial RISC CPU

- Author: Luke Wren
- Description: Execute a simple 16-bit RISC-style instruction set from up to 64 kilobytes of external SPI SRAM.
- GitHub repository
- HDL project
- Extra docs
- Clock: 6000 Hz
- External hardware: - An SPI SRAM with 16-bit addresses and support for sequential mode accesses, e.g. Microchip 23K256T-I
- A host interface for loading and initialising the SPI SRAM, e.g. Raspberry Pi Pico
- (optional) Two 74HC595 shift registers for a 16-bit output port
- (optional) A 74HC166 shift register for an 8-bit input port

All of these components will be integrated on the Whisk host board, see the project GitHub page.

How it works

Whisk uses a single SPI interface for instruction fetch, loads and stores on an external SPI SRAM. The SPI serial clock is driven at the same frequency as Whisk's clock input. The program counter, and the six general purpose registers, are all 16 bits in size, so up to 64 kilobytes of memory can be addressed.

Internally, Whisk is fully serial: registers and the program counter are read and written one bit at a time. This matches the throughput of the SPI memory interface, and leaves more area free for having more/larger general purpose registers as well as leaving room for expansion on future Tiny Tapeouts.

An optional IO port interface adds up to 16 outputs and 8 inputs, using standard parallel-in-serial-out and serial-in-parallel-out shift registers. Whisk can read or write these ports in a single instruction. These can be used for bitbanging external hardware such as displays, LEDs and buttons.

How to test

You will need a Whisk host board, with memory and the host interface to load it. See the project GitHub page.

IO

#	Input	Output
0	clk	mem_csn
1	rst_n	mem_sck
2	mem_sdi	mem_sdo
3	ioport_sdi	ioport_sck
4	none	ioport_sdo
5	none	ioport_latch_i
6	none	ioport_latch_o
7	none	none

92 : Scalable synchronous 4-bit tri-directional loadable counter

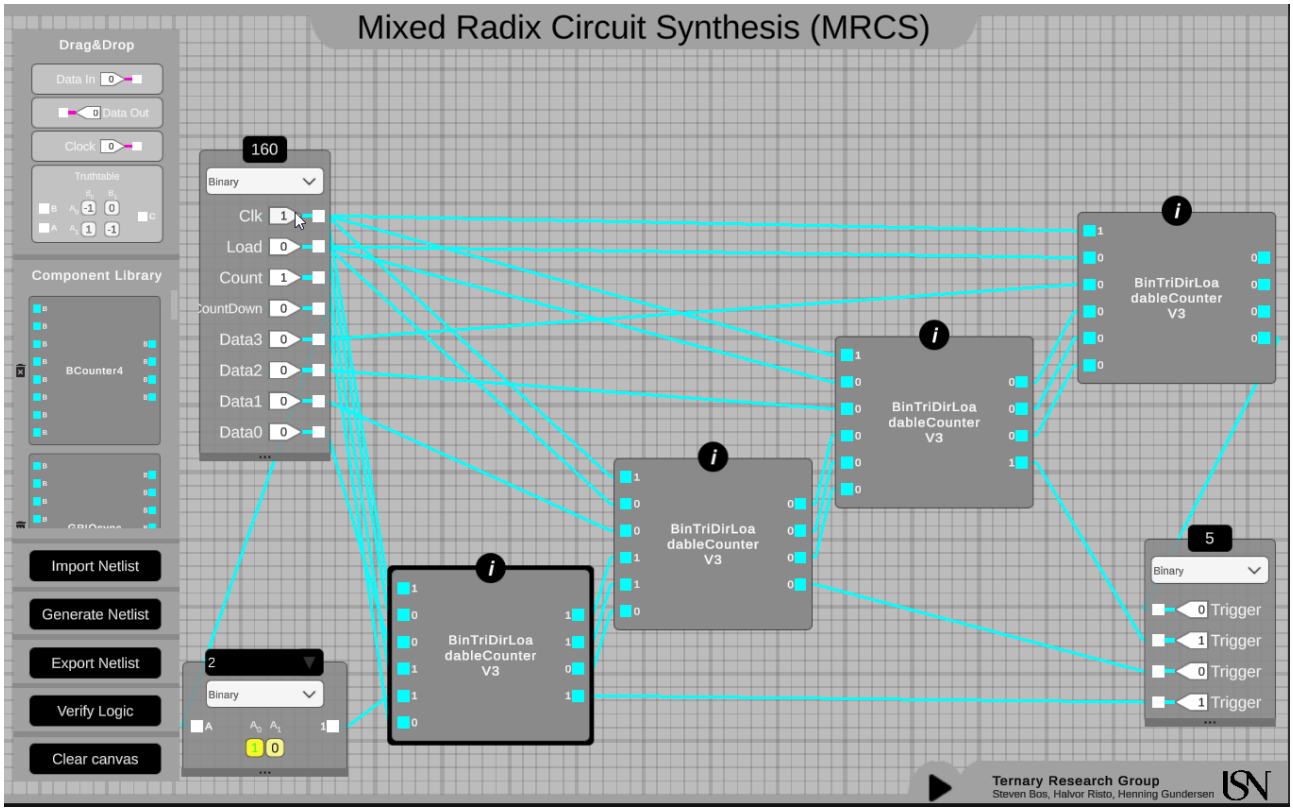

Figure 27: picture

- Author: Steven Bos
- Description: This chip offers a scalable n-bit counter design that can be used as a program counter by setting the next address (eg. for a JMP instruction). It can work in 3 directions: counting up, down and pause.
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware: no external hardware needed

How it works

See the full documentation, youtube movie and image. Each 1-bit counter has a flip-flop with count logic component reacting synchronously to the rising edge clock pulse and a count logic component that computes and setup the behavior for the next rising edge using async propagation when the level is low.

How to test

The count state is randomly initialized. Typically the first action is to reset the state to zero by setting the load switch and have one clock pulse. The second action is setting the direction by enabling count and setting countDown to true or false (and disable load). The counter overflows to all 0 when all 1 is reached and count up is set.

IO

#	Input	Output
0	clock	output3 (bits [0:3])
1	count (0 = disable/countPause, 1 = enable)	output2
2	load (0 = count mode, 1 = load mode, overwriting any count logic)	output1
3	countDown (0 = countUp, 1 = countUp)	output0
4	addr3 (bits[4:7] are used for loadable count state)	none
5	addr2	none
6	addr1	none
7	addr0	none

93 : Asynchronous Binary to Ternary Converter and Comparator

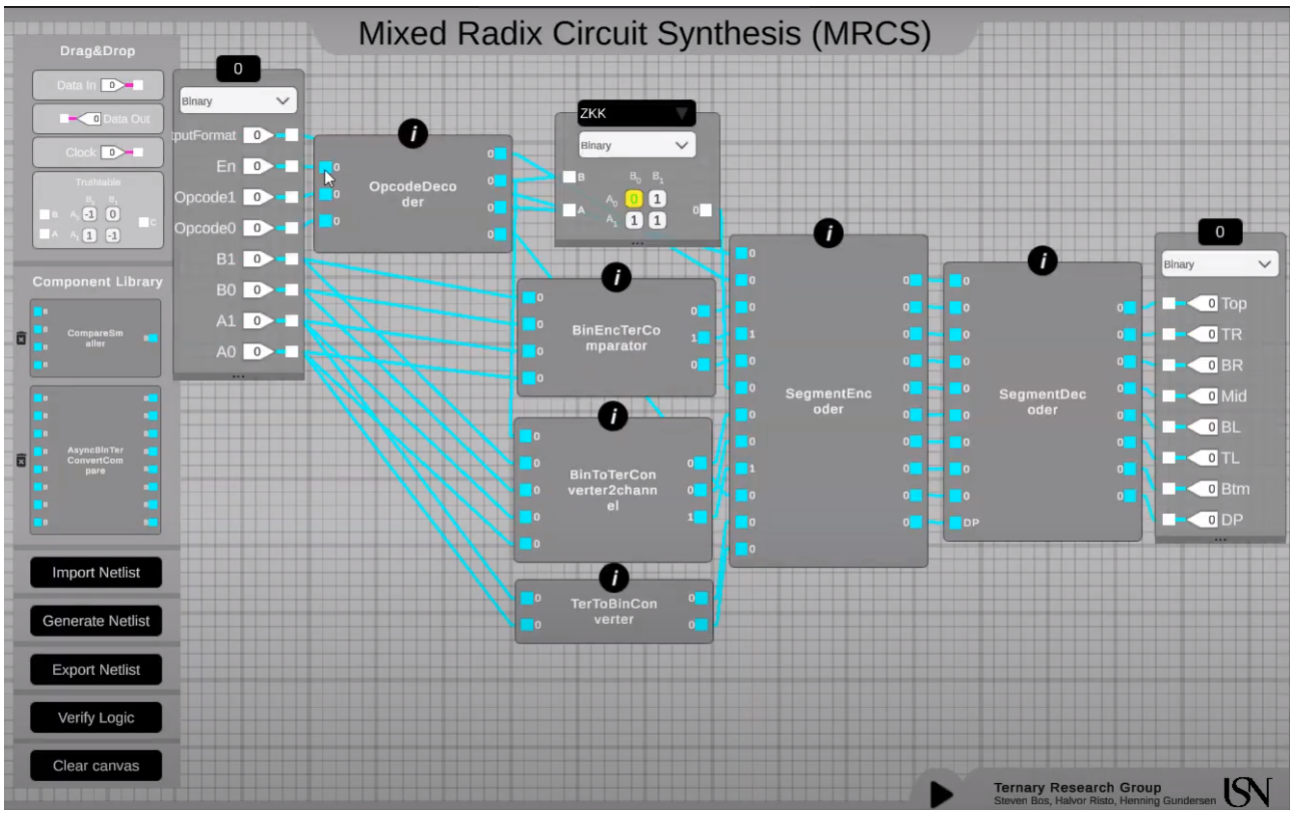

Figure 28: picture

- Author: Steven Bos
- Description: This chip offers various kinds of conversions and comparisons between binary encoded ternary and unary encoded ternary in both machine readible output and human readible (7-segment display decimal) output
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware: no external hardware needed

How it works

See the full documentation, youtube movie and image. The chip has four stages. The opcode stage set the mode of the chip. The second stage convert the input to output based on the selected mode. The third stage encoded the output to machine or human based on selected mode. Finally the encoded output is decoded on a 7 segment display.

How to test

Set the chip mode [0:3] to 0111. This enables the chip, set unary encoded ternary channel A conversion and set it to user (decimal) output. Set the input [4:7] to 0010. The Channel A input using Unary Encoded Ternary is set to 2, which the 7 segment display also shows. Note that one combination of the two bits is illegal! (see doc)

IO

#	Input	Output
0	output mode (0 = human, 1 = machine)	segment a (the 7 segment is used for human readable output, sometimes using decimals and sometimes using comparison symbols, see documentation for more details)
1	enable (0 = disable, 1 = enable)	segment b
2	opcode0 (see table in documentation for all 4 modes)	segment c
3	opcode1	segment d
4	input channel B pin0 (see table in documentation what is don't care or illegal input for which mode)	segment e
5	input channel B pin1	segment f
6	input channel A pin0	segment g
7	input channel A pin1	segment dot (the dot is an extra indicator that the output is in machine format)

94 : Vector dot product

- Author: Robert Riachi
- Description: Compute the dot product of two 2x1 vectors each containing 2 bit integers
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

a := in[0:1], b := in[2:3], c := in[4:5], d := in[6:7] - [a,b,c,d] => [ac * bd]

How to test

set input to 11011010 => which means [11,01] [10,10] => as ints [3,1] [2,2] => output should be 00001000 => *(32) + (12)* = 8

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

95 : Monte Carlo Pi Integrator

- Author: regymm
- Description: Calculate the value of Pi using the Monte Carlo method
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware: External edge counter recommended(funnyblinky is a possible choice)

How it works

Having random x and y between 0 to 1 and compare the added squares with 1. Using 8-bit fixed-point number.

How to test

SW 00: counter shows total sample points. SW 01: counter shows sample points inside 1 radius. SW 10: counter 0 and 1 will toggle, 0 for every sample point and 1 for inside point, for use with external counter.

IO

#	Input	Output
0	clock	counter 0
1	reset	counter 1
2	sw control 0	counter 2
3	sw control 1	counter 3
4	none	counter 4
5	none	counter 5
6	none	counter 6
7	none	counter 7

96 : Funny Blinky

- Author: regymm
- Description: Blink the 8 output LEDs in a funny way.
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware:

How it works

.

How to test

When function switch is turned off, all LEDs will be blinky. When switch is on, it works as a double 14-bit counter, to be used together with the mcpi module – in this case we have pause switch and two output control switches to show all bits of the counters.

IO

#	Input	Output
0	clock	led 0
1	reset	led 1
2	none	led 2
3	none	led 3
4	switch out ctrl 0	led 4
5	switch out ctrl 1	led 5
6	switch pause	led 6
7	switch function	led 7

97 : GPS C/A PRN Generator

- Author: Adam Greig
- Description: Generate the GPS C/A PRN sequences PRN1 through PRN32
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware: None

How it works

Two LFSRs are constructed per the GPS ICD, and the first is added to selected taps of the second to produce the selected final PRN sequence.

How to test

With io_in[2:7] set to 2 to select PRN2, reset and then drive the clock; the output sequence on io_out[2] will start with 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1.

IO

#	Input	Output
0	clock	G1
1	reset	G2
2	prn[0]	Selected PRN
3	prn[1]	none
4	prn[2]	none
5	prn[3]	none
6	prn[4]	none
7	none	none

98 : Sigma-Delta ADC/DAC

- Author: Adam Greig
- Description: Simple ADC and DAC
- GitHub repository
- HDL project
- Extra docs
- Clock: 6000 Hz
- External hardware: Comparator, resistor, capacitor

How it works

This project is built on a simple sigma-delta DAC. The DAC is given an n-bit control word and generates a single-bit digital output where the pulse density is proportional to that control word. By integrating this pulse train, for example with an RC filter, an analogue output voltage is produced.

The ADC operates by generating an analogue output voltage which is compared to the analogue input by an off-chip comparator. The comparator result is used as a digital input to a simple control loop that adjusts the output voltage so that it tracks the input signal. The control word for the DAC generating the output voltage is then the ADC reading. This control word is regularly transmitted as hex-encoded ASCII over a UART running at the clock rate.

A second dedicated 8-bit DAC is controlled by received words over a UART. Transmit the control word at 1/10th the clock speed into `uart_in`, and add a second external RC circuit to filter `dac_out` to an analogue voltage.

How to test

Ensure in[0] is clocked. Connect out[0] through a series resistor to both a capacitor to ground and the non-inverting input of a comparator. Connect the analogue input to measure to the inverting input, and connect the comparator output to in[2]. Connect out[1] to a UART receiver at the clock rate and receive ADC readings as hex-encoded ASCII lines.

Connect out[2] to a second RC filter, and feed one-byte DAC settings to the UART on in[3] at a baud rate 1/10th the clock. Measure the resulting analogue output.

IO

#	Input	Output
0	clock	adc_out

#	Input	Output
1	reset	uart_out
2	adc_in	dac_out
3	uart_in	none
4	none	none
5	none	none
6	none	none
7	none	none

99 : BCD to Hex 7-Segment Decoder

- Author: JinGen Lim
- Description: Converts a 4-bit BCD input into a hexadecimal 7-segment display output
- GitHub repository
- HDL project
- Extra docs
- Clock: 0 Hz
- External hardware: 7-segment display

How it works

The IC accepts four binary-coded decimal input signals, and generates a corresponding hexadecimal 7-segment output signal. Segment outputs may be inverted with the INVERT pin to support both common cathode/anode displays.

How to test

Connect the segment outputs to a 7-segment display. Configure the input (IN0:0, IN1:2, IN2:4, IN3:8). The input value will be shown on the 7-segment display

IO

#	Input	Output
0	input 1 (BCD 1)	segment a
1	input 2 (BCD 2)	segment b
2	input 3 (BCD 4)	segment c
3	input 4 (BCD 8)	segment d
4	decimal dot (passthrough)	segment e
5	output invert	segment f
6	none	segment g
7	none	segment dot

100 : SRLD

- Author: Chris Burton
- Description: 8-bit Shift Register with latch and hex decode to display alternating nibbles
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 16 Hz
- External hardware: Switches and 7-segment display

How it works

Data is inputted to an 8-bit shift register, the data can then be (optionally) latched, data can be switched around if needed based on shifted data being LSB/MSB first, cylcles between decoding high/low nibble to show on the 7-segment display.

How to test

Use shiftIn and shiftClk to clock in 8-bits of data. Toggle latch to move data from shift register to the latch. 7-seg display will show alternating high/low nibbles. If useLatch is high data comes from the latch otherwise it will be shown 'live' as it's shifted in. If cycle_display is low the display will cycle between high/low nibble otherwise it will show the nibble selected by lowHighNibble. mslLsb will switch between showing the shifted data as MSB or LSB first.

IO

#	Input	Output
0	displayClock	segment a
1	shiftIn	segment b
2	shiftClk	segment c
3	latch	segment d
4	cycle_display	segment e
5	lowHighNibble	segment f
6	useLatch	segment g
7	mslLsb	High/low nibble indicator

101 : Counter

picture

- Author: Adam Zeloof
- Description: It counts!
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 600 Hz
- External hardware: None

How it works

It counts up or displays an entered number on the seven-segment. A clock divider can be used to slow down the clock speed.

How to test

Enable the counter (input 7) and the clock divider (input 6) and it should start counting up. If you disable the counter (input 7) you can enter a number to display manually in binary (inputs 1-4).

IO

#	Input	Output
0	clock	segment a
1	b0	segment b
2	b1	segment c
3	b2	segment d
4	b3	segment e
5	none	segment f
6	clock divider enable	segment g
7	count enable	none

102 : 2bitALU

- Author: shan
- Description: 2 bit ALU which performs 16 different operations
- GitHub repository
- HDL project
- Extra docs
- Clock: none Hz
- External hardware:

How it works

Based on the 4 bit opcode, the ALU performs 16 different operations on the 2 bit inputs A & B and stores the result in 8 bit output ALU_out

How to test

Provide A, B inputs. Select opcode based on the operation to perform. Check output at ALU_out

IO

#	Input	Output
0	A1	ALU_out
1	A2	ALU_out
2	B1	ALU_out
3	B2	ALU_out
4	opcode	ALU_out
5	opcode	ALU_out
6	opcode	ALU_out
7	opcode	ALU_out

103 : A (7, 1/2) Convolutional Encoder

- Author: Jos van 't Hof
- Description: A (7, 1/2) Convolutional Encoder following the CCSDS 131.0-B-4 standard.
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

A Convolutional Encoder adds additional bits to a data stream or message that may later be used to correct errors in the transmission of the data. The specific implemented encoder is used in space applications and is a half-rate (R = 1/2) code with a constrain lenght of seven (K = 7). This means that the encoder generates two output bits (called symbols) for every input bit, and the encoder has m = K - 1 = 6 states.

How to test

Pull the write_not_shift input (IN1) high and set a 6-bit binary input (using IN2 to IN7), for example 0b100110. Provide a clock cycle on the clock input (IN0) to write the input into the shift register and clear the encoder. Pull the write_not_shift input (IN2) low to start shifting. Provide 24 clock cycles (2 each for the 6 shift registers and 6 encoder registers 2x(6+6) = 24). After each clock cycle a 0 or 1 is displayed on the 8-segment display. The encoded output for the input 0b100110 is 0b10111|0010001101000111001. The first 6 bits of the encoded output may be discared.

IO

#	Input	Output
0	clock	segment a
1	write_not_shift	segment b
2	shift_input_0	segment c
3	shift_input_1	segment d
4	shift_input_2	segment e
5	shift_input_3	segment f
6	shift_input_4	segment g
7	shift_input_5	segment dp (used to indicate clock)

104 : Tiny PIC-like MCU

- Author: myrtle
- Description: serially programmed, subset of PIC ISA, MCU
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware: A means of shifting in the program (e.g. another microcontroller, USB GPIO interface, etc) is required at startup. Once running, it is standalone.

How it works

Implements a subset of the PIC mid-range ISA (no SFR, no carry, no call/stack), 6 GPRs, 16 program words.

How to test

Program data is shifted in serially. For each program word, shift in {(1 << address), data} (28 bits total) to prog_data and then assert prog_strobe. Once loaded, deassert (bring high), reset and the program should start running. GPR 6 is GPI and GPR 7 is GPO

IO

#	Input	Output
0	clock	gpo0
1	reset	gpo1
2	prog_strobe	gpo2
3	prog_data	gpo3
4	gpi0	gpo4
5	gpi1	gpo5
6	gpi2	gpo6
7	gpi3	gpo7

105 : RV8U - 8-bit RISC-V Microcore Processor

- Author: David Richie
- Description: 8-bit processor based on RISC-V ISA
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware:

How it works

Executes reduced RISC-V based ISA

How to test

Requires interfacing to external memory

IO

#	Input	Output
0	clock	serdes output bit 0
1	reset	serdes output bit 1
2	serdes input bit 0	serdes output bit 2
3	serdes input bit 1	serdes output bit 3
4	serdes input bit 2	serdes output bit 4
5	serdes input bit 3	serdes output bit 5
6	serdes input bit 4	serdes output bit 6
7	serdes input bit 5	serdes output bit 7

106 : Logic-2G97-2G98

- Author: Sirawit Lappisatepun
- Description: Replication of TI's Little Logic 1G97 and 1G98 configurable logic gates.
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

This design replicates the circuit inside a TI configurable logic gates 74xx1G97 (and by including an inverted output, it will work as a 74xx1G98 as well). Since there are still I/O pins left, I included two of these configurables, and also one 74xx1G79 D Flip-Flop (again, an inverted output means this will also work as a 74xx1G80).

How to test

You could refer to TI's 1G79/1G80/1G97/1G98 datasheet to test the device according to the pinout listed below.

IO

#	Input	Output
0	dff_clock	dff_out
1	dff_data	dff_out_bar
2	gate1_in0	gate1_out
3	gate1_in1	gate1_out_bar
4	gate1_in2	gate2_out
5	gate2_in0	gate2_out_bar
6	gate2_in1	none
7	gate2_in2	none

107 : Melody Generator

- Author: myrtle
- Description: plays a melody, preloaded with jingle bells but re-programmable
- GitHub repository
- HDL project
- Extra docs
- Clock: 25000 Hz
- External hardware:

How it works

melody output at output 0

How to test

connect a speaker to output 0, set reload and restart to 1

IO

#	Input	Output
0	clock	melody
1	reload	none
2	restart	none
3	prog_data	none
4	prog_strobe	none
5	none	none
6	none	none
7	none	none

108 : Rotary Encoder Counter

- Author: Vaishnav Achath
- Description: Count Up/Down on the 7-segment accouring to rotary encoder input
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware: Rotary Encoder

How it works

uses a register and some combinational logic

How to test

Provides test mode enable to use input clock and inverted ip/clock as emulated encoder CLK/Data

IO

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

109 : Wolf sheep cabbage river crossing puzzle ASIC design

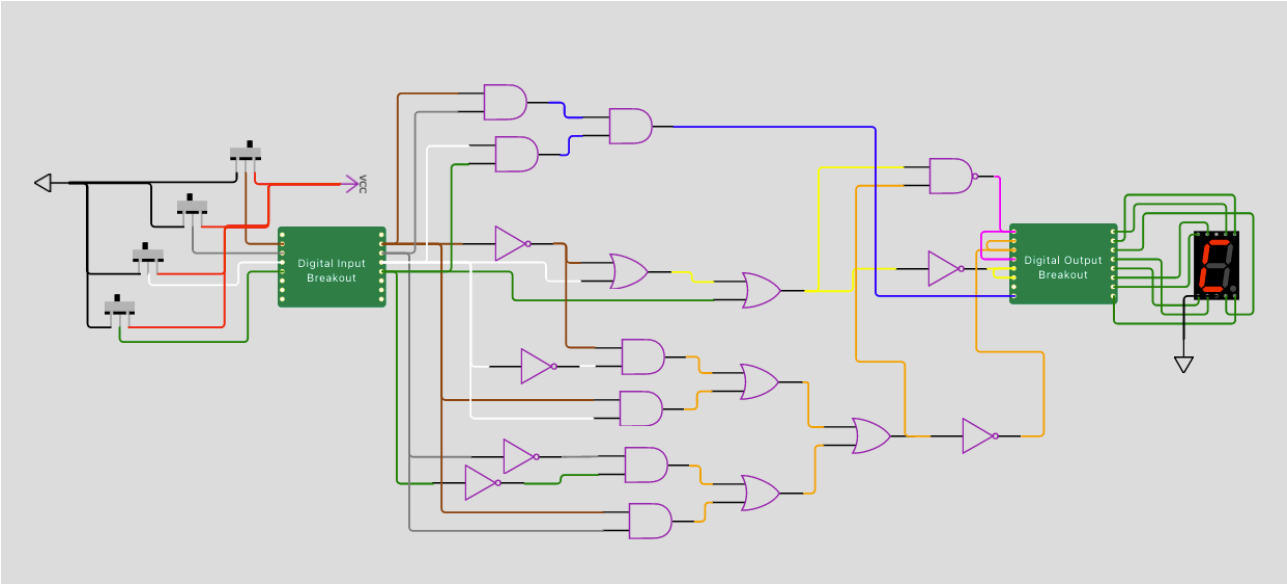

Figure 29: picture

- Author: maehw
- Description: Play the wolf, goat and cabbage puzzle interactively.
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware: Input switches and 7-segment display

How it works

Truth table with the game logic (hidden easter egg). The inputs are the positions of the farmer, wolf, goat and cabbage. The 7-segment display shows the status of the game (won or lost).

How to test

Slide the input switches, think, have a look at the 7-segment display.

IO

#	Input	Output
0	not connected because it is typically used for clocked designs and may be used in the future of this design	output signal ~E, i.e. the top and bottom segments light up, when the game is over due to an unattended situation on any river bank side
1	input signal F for the position of the farmer	output signal ~R i.e. the top-right and bottom-right segments light up, to indicate an unattended situation on the right river bank (game over)
2	input signal W for the position of the wolf	output signal ~R i.e. the top-right and bottom-right segments light up, to indicate an unattended situation on the right river bank (game over)
3	input signal G for the position of the goat	output signal ~E, i.e. the top and bottom segments light up, when the game is over due to an unattended situation on any river bank side
4	input signal C for the position of the cabbage	output signal ~L i.e. the top-left and bottom-left segments light up, to indicate an unattended situation on the left river bank (game over)
5	here be dragons or an easter egg	output signal ~L i.e. the top-left and bottom-left segments light up, to indicate an unattended situation on the left river bank (game over)
6	unused	here be dragons or an easter egg
7	unused	output signal A to light up the "dot LED" of the 7 segment display as an indicator that all objects have reached the right bank of the river and the game is won!

110 : Low-speed UART transmitter with limited character set loading

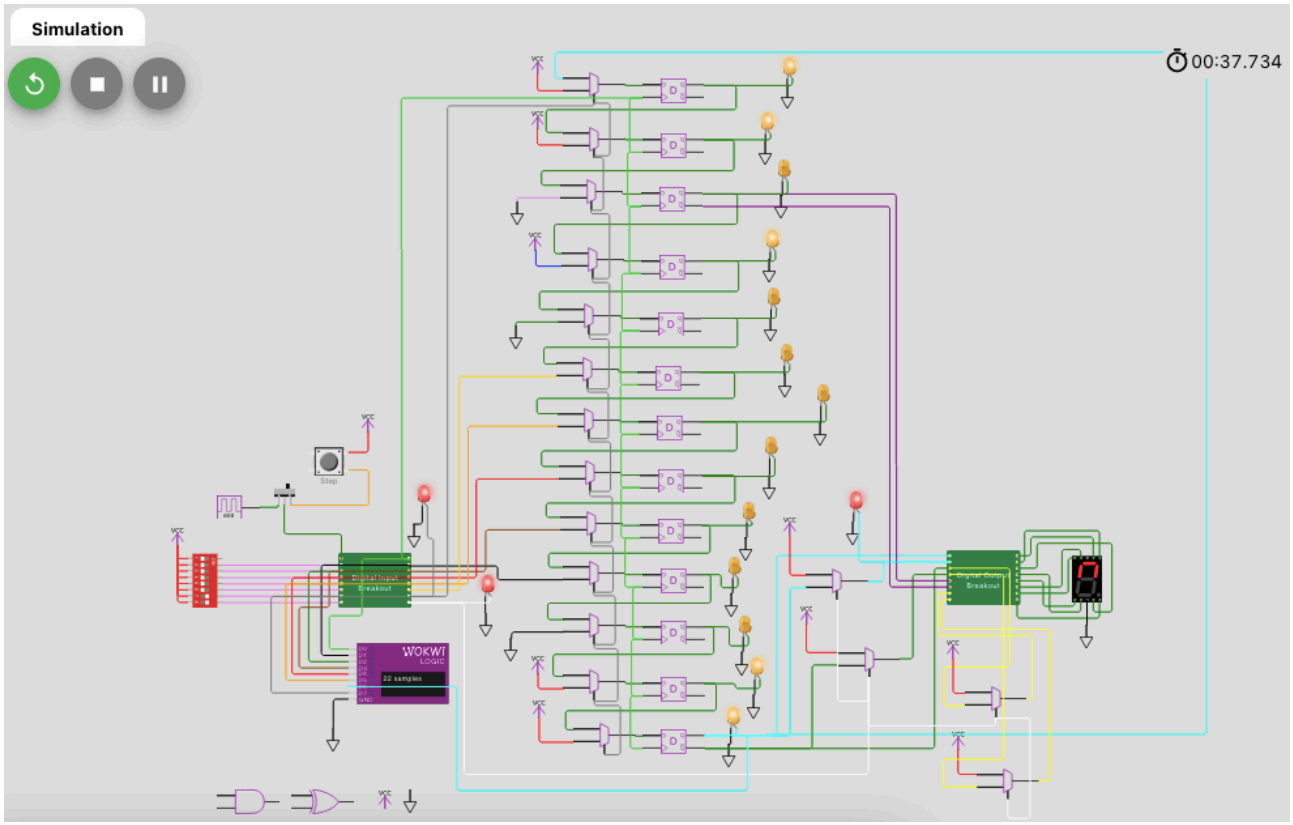

Figure 30: picture

- Author: maehw
- Description: Low baudrate UART transmitter (8N1) with limited character set (0x40..0x5F; includes all capital letters in the ASCII table) loading.
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 9600 Hz
- External hardware: UART receiver or oscilloscope or logic analyzer (optional)

How it works

The heart of the design is a 13 bit shift register (built from D flip-flops). When a word has been transmitted, it will be transmitted again and again until a new word is loaded into the shift register or the output is disabled (the word will keep on cycling internally).

How to test

Load a character into the design and attach a UART receiver (or oscilloscope or logic analyzer) on the output side.

IO

#	Input	Output
0	300 Hz input clock signal (or different value supported by the whole	UART (serial output pin, direct throughput)
1	bit b0 (the least significant bit) of the loaded and transmitted character	UART (serial output pin, gated by enable signal)
2	bit b1 of the loaded and transmitted character	UART (serial output pin, reverse polarity, direct throughput)
3	bit b2 of the loaded and transmitted character	UART (serial output pin, reverse polarity, gated by enable signal)
4	bit b3 of the loaded and transmitted character	UART (MSBit, direct throughput); typically set to 1 or can be used to sniffing the word cycling through the shift register)
5	bit b4 of the loaded and transmitted character	UART (MSBit, reverse polarity, direct throughput); same usage as above
6	load word into shift register from parallel input (IN1..IN5) (1) or cycle the existing word with start/stop bits around it (0)	UART (MSBit, gated by enable signal); typically set to 1 or can be used to sniffing the word cycling through the shift register)
7	{'output enable (for gated output signals)': '1 output is enabled, 0 output is disabled (permanently set to HIGH/1)'}	UART (MSBit, reverse polarity, gated by enable signal); same usage as above

111 : Rotary encoder

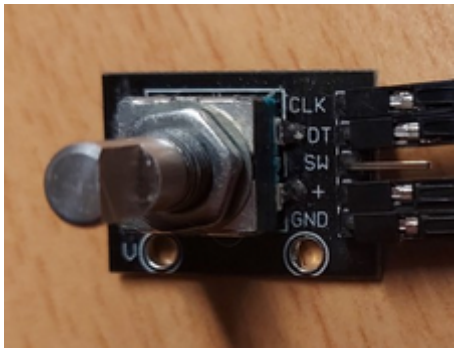

Figure 31: picture

- Author: Wim Dams
- Description: Reads in a (incremental) rotary encoder and shows the result on the seven-segment display
- GitHub repository
- HDL project
- Extra docs
- Clock: 10000 Hz
- External hardware: Rotary encoder connected to pin A and pin B

How it works

The rotary pins are sampled every clock cycle. If a rising edge is detected on pin A, the 4 bit counter will be incremented/decremented depending on pin B. The counter is put on the seven segment display and a debounce time is started (125 clk cycles)

How to test

After reset, turn the rotary encoder and the counter should increment/decrement as you turn

IO

#	Input	Output
0	clock	segment a
1	reset	segment b
2	Rotary encoder pin A (sometimes marked as CLK)	segment c
3	Rotary encoder pin B (sometimes marked as DT)	segment d
4	none	segment e
5	none	segment f

#	Input	Output
6	none	segment g
7	none	none

112 : FROG 4-Bit CPU

- Author: ChrisPVille
- Description: The FROG is an extremely minimal load-store 4-bit CPU
- GitHub repository
- HDL project
- Extra docs
- Clock: 0 Hz
- External hardware: An SRAM/ROM like memory containing instructions should be connected to addr/wcyc/data_in

How it works

The CPU addresses external memory on its addr pins and executes/interprets data on the data_in pins

How to test

Set data_in to 0x8 (NOP) and observe the addr bus count upward as the CPU executes Instructions

IO

#	Input	Output
0	clock	addr[0]
1	reset_p	addr[1]
2	data_in[0]	addr[2]
3	data_in[1]	addr[3]
4	data_in[2]	addr[4]
5	data_in[3]	addr[5]
6	none	addr[6]
7	fast - zero wait state memory mode	write memory cycle

113 : Configurable Gray Code Counter

- Author: Eric Swalens
- Description: A configurable counter driven by 2-channel Gray code
- GitHub repository
- HDL project
- Extra docs
- Clock: 5000 Hz
- External hardware: A source of Gray code; filtering and Schmitt triggers may be required if a mechanical encoder is used.

How it works

The module is an 8-bit configurable counter modified by Gray code (aka 2-bit quadrature code); it aims at easing the integration of incremental rotary encoders into projects. The counter value is given as a (truncated to 5 bits) parallel or (8 bits, no parity, 1 stop bit) serial UART output. Other outputs include the "direction" of progression of the Gray code, and a PWM signal for which the duty cycle is proportional to the counter value.

Some basic (optional) debouncing logic is included; any pulse inverting the direction must be followed by a second pulse in the same direction before the change is registered.

Additional features include support for wrapping (the counter rolls over at the minimum and maximum value), and a "gearbox" that selects the X1 (1 pulse per 4 transitions), X2 (2 pulses) or X4 (4 pulses) output of the Gray code decoder driving the counter depending on the speed at which the channels change; this can provide some form of "acceleration". The initial and maximum values of the counter can also be set.

Encoders with twice the number of detents compared to the number of pulses per round (e.g. 24 detents / 12 PPR) are supported by setting the input "update on X2" high or forcing it with the configuration parameter.

After reset the module is configured as a basic 5-bit counter which can then be further modified by sending a 32-bit word over the SPI interface. This word sets the following options (reset value between parentheses):

- gearbox enable (0)
- debounce logic enable (1)
- wrap enable (0)
- Gray code value for X1 (0)
- force update on X2 (0), this overrides a low value at the input pin (the value for X1 selects which transitions are taken into consideration)

- gearbox timer value (n/a, gearbox is disabled)
- counter initial value (0)
- counter maximum value (31)

See link to GitHub for possible errata.

How to test

For a basic test connect a device generating Gray code and retrieve the counter value at the parallel or serial outputs with a microcontroller or other circuitry.

To further configure the module send some configuration word over the SPI interface (mode 0, MSB first, CS is active low). The 32-bit configuration word is constructed a follows (bits between brackets):

- [31:24] maximum counter value
- [16:23] initial counter value after configuration
- [8:15] gearbox timer
- [6:7] unused
- [5:5] force update on X2
- [3:4] X1 value
- [2:2] debounce enable
- [1:1] wrap enable
- [0:0] gearbox enable

The gearbox is implemented with a 5-bit threshold value; it is incremented by the X4 output of the decoder and decremented by a timer (this threshold is then divided by 8 to select the gear, giving 0: X1, 1: X1, 2/3: X4). Therefore the result depends on the clock frequency and the speed at which the Gray code transitions. The gearbox timer is exposed to enable tuning the interval between two updates by the timer. For a rotary encoder with detents one can suggest using *clock_hz / (detents x transitions - 16)* as a starting point to determine a suitable value, where detents is the number per turn (e.g. 24) and transitions is the number per detent (e.g. 4). That is, 62 for a common 24 detents / 24 PPR encoder.

The 8-N-1 UART serial output shifts 1 bit out at each clock cycle. The receiving serial port therefore needs to be configured at the same speed as the clock.

The PWM frequency is derived from the maximum counter value. It might be unsuitable for visual feedback, e.g. driving a LED, for large values with a low clock frequency as the LED will appear blinking.

IO

#	Input	Output
0	clock	UART serial output
1	reset	PWM signal
2	channel A	direction
3	channel B	counter bit 0
4	update on X2	counter bit 1
5	SPI CS	counter bit 2
6	SPI SCK	counter bit 3
7	SPI SDI	counter bit 4

114 : Baudot Converter

- Author: Arthur Hazleden
- Description: This circuit will convert ASCII serial data to baudot serial data provide the reverse funtion as well
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware: The ASCII is expected to come in/out from a USB-serial adapter. The baudot is a 60mA current loop so something hefty is required.

How it works

UARTS and two conversion ROMs

How to test

Serial data in and see how it goes.

IO

#	Input	Output
0	ascii clock at 8x desired baudrate	segment a
1	baudot clock at 100x desired baudrate	segment b
2	baudot input, should be held high when line is idle but connected	segment c
3	none	segment d
4	none	segment e
5	none	segment f
6	none	segment g
7	none	none

115 : Marquee

- Author: Christopher 'ctag' Bero
- Description: Scrolls 'ctag' across the 7seg.
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 2 Hz
- External hardware: NA/Button

How it works

Uses two flip-slops to get a 4-state machine, and then just activates LEDs from the outputs.

How to test

Set clock to button and click through.

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

116 : channel coding

- Author: Asma Mohsin
- Description: Convolutional coding is widely used in modern digital communication systems.We often get encoded data by using different polynomials having same constraint lengths (K).
- GitHub repository
- HDL project
- Extra docs
- Clock: 6000 Hz
- External hardware:

How it works

We have two polynomials of 4th order and a shift register of 5 bits. we take input data of a single bit and put it in shift register on each clock edge as long as valid data bit is asserted. after this codeword is calculated by taking xor of the and of polynomial and shift register

How to test

apply clk,reset ,data valid and input data and do calculations to see if output is equal to the desired one

IO

#	Input	Output
0	clock	encoded data
1	reset	none
2	data valid	none
3	data input	none
4	none	none
5	none	none
6	none	none
7	none	none

117 : Chisel 16-bit GCD with scan in and out

- Author: Steve Burns
- Description: Simple chisel based design based on Knuth's BinaryGDC algorithm using scan chains for I/O.
- GitHub repository
- HDL project
- Extra docs
- Clock: 0 Hz
- External hardware: None

How it works

With the `ld` signal true, the `u_bit` and `v_bit` inputs are used to scan the `n`-bit numbers into the block. Simulataneously, the high-order bit of the `u` register is scanned out, allowing access to the result of the last computation. Upon lowering the `ld` signal, the Euclid iteration starts. When done, the `done` signal is raised.

How to test

Chiseltest enabled tests. Go to `chisel` and run `sbt test`.

IO

#	Input	Output
0	clock	z_bit
1	reset	done
2	ld	none
3	u_bit	none
4	v_bit	none
5	none	none
6	none	none
7	none	none

118 : Adder with 7-segment decoder

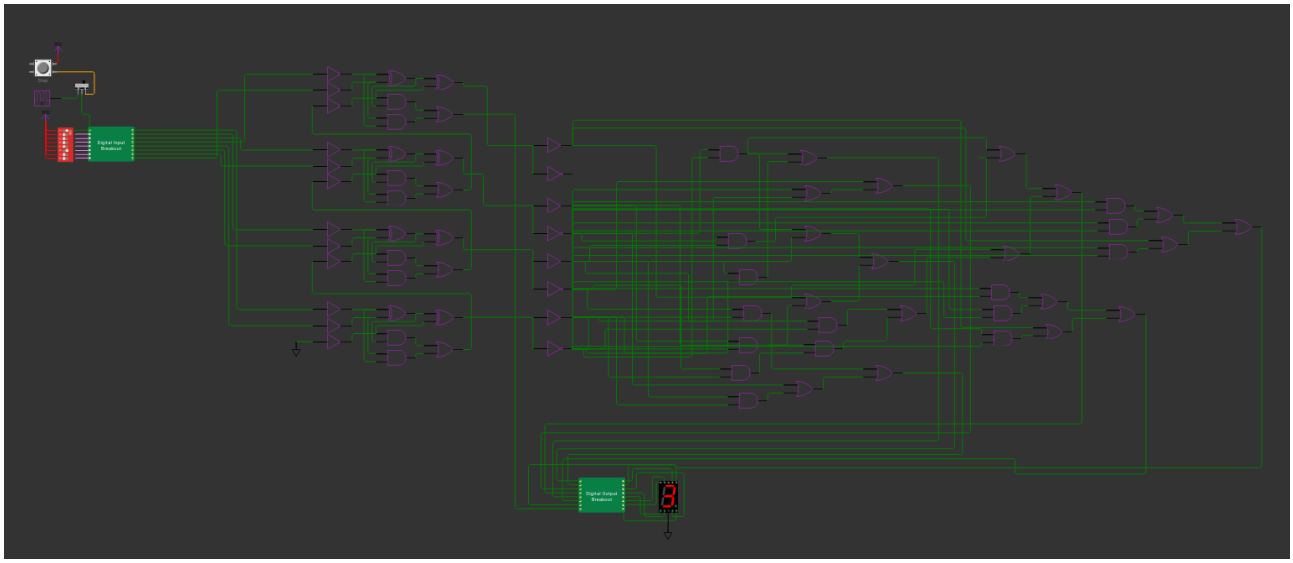

Figure 32: picture

- Author: cy384
- Description: Four bit adder with binary to 7 segment display decoder
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware: No external hardware needed.

How it works

Four full adders with carry feeding into a somewhat hairy binary to seven segment display decoder.

How to test

Use the DIP switches to enter two four bit binary numbers. Display of numbers greater than nine is questionable. The decimal point of the display is carry (i.e. a sum over 16).

IO

#	Input	Output
0	first number bit 0 (least significant)	segment a
1	first number bit 1	segment b
2	first number bit 2	segment c

#	Input	Output
3	first number bit 3	segment d
4	second number bit 0 (least significant)	segment e
5	second number bit 1	segment f
6	second number bit 2	segment g
7	second number bit 3	segment DP (carry bit)

119 : Hex to 7 Segment Decoder

- Author: Randy Glenn
- Description: Displays an input 4-bit value as a hex digit
- GitHub repository
- HDL project
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

A modern take on the classic TIL311

How to test

after reset, the counter should increase by one every second

IO

#	Input	Output
0	latch	segment a
1	blank	segment b
2	data 0	segment c
3	data 1	segment d
4	data 2	segment e
5	data 3	segment f
6	decimal	segment g
7	none	decimal

120 : Multiple seven-segment digit buffer

picture

- Author: Zach Mason
- Description: Storage and variable speed readback segment digits
- GitHub repository
- HDL project
- Extra docs
- Clock: 6250 Hz
- External hardware: None

How it works

Stores 12 seven-segment display digits in registers in write mode. In read mode, the values are sequentially displayed back, with a variable cycle rate. The segment inputs are 4-3 multiplexed and a clock divider is used to slow down the output rate. The user is responsible for tracking how many digits have been set.

How to test

First set in1-in7 low, and then reset by toggling in1 high then low. In read mode (in2 high), the decimal point will be illuminated and the first 4 segments can be changed with in4-in7. When the desired configuration is set, sel (in3) can be switched high and the remaining 3 segments can be set with in4-in6. Once the desired configuration is set, you can move to the next digit by bringing sel (in3) low. Alternatively, read mode can be entered by bringing RW (in2) low. At this point, the stored values will begin reading sequentially at a rate given by in4-in7. The base period is about 81.9ms, with in4-in7 specifying the multiplication factor for the real display rate. The slowest period is about 2.62s, where in7-in4 are all high. If in read mode, bringing in3 low will stop the cycling and keep displaying the current digit. This can be useful for changing a single digit since one could cycle through at a slow rate to find the target, enter write mode, change the stored digit, and then exit back to read mode.

IO

#	Input	Output
0	clock	segment a
1	reset	segment b
2	RW mode	segment c
3	sel, read_clk_en	segment d
4	pin0, clkspd0	segment e

#	Input	Output
5	pin1, clkspd1	segment f
6	pin2, clkspd2	segment g
7	pin3, clkspd3	segment dp

121 : LED Chaser

- Author: Bradley Boccuzzi
- Description: Push the button to fill in segments of the LED display, they will continue to shift and fill in the display until the button is released.‘
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware: None

How it works

Input is button to input of shift register. Each segment of the 7 segment display is connected to an output of the shift register.

How to test

Push switch number 8 to watch the LEDs fill in

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	shift register input	none

122 : Rolling Average - 5 bit, 8 bank

- Author: Kauna Lei
- Description: 5bit moving average
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware: gpios to connect with io_in[7:2] and to read io_out[4:0]

How it works

Using Shift Register Line and Rolling Sum Trick

How to test

Clock in the high 5 bits of io_in (io_in[7:3]) with the i_data_clk (io_in[2]) (active high), and read output on io_out[4:0]

IO

#	Input	Output
0	clock	ra_out[0]
1	reset	ra_out[1]
2	i_data_clk	ra_out[2]
3	i_value[0]	ra_out[3]
4	i_value[1]	ra_out[4]
5	i_value[2]	0
6	i_value[3]	0
7	i_value[4]	0

123 : w5s8: universal turing machine core

- Author: Andrew Foote
- Description: State transition logic for a 5-state, 8-symbol universal turing machine
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware:

How it works

Uses combinational logic to implements a (state, symbol) -> (state, symbol, direction) transition function

How to test

Provide state & symbol as inputs, and the module should output the state, symbol, and direction according to the table in test.py.

IO

#	Input	Output
0	clock	none
1	state_in[0]	next_direction
2	state_in[1]	new_sym[0]
3	state_in[2]	new_sym[1]
4	sym_in[0]	new_sym[2]
5	sym_in[1]	new_state[0]
6	sym_in[2]	new_state[1]
7	mode	new_state[2]

Technical info

Scan chain

All 250 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.

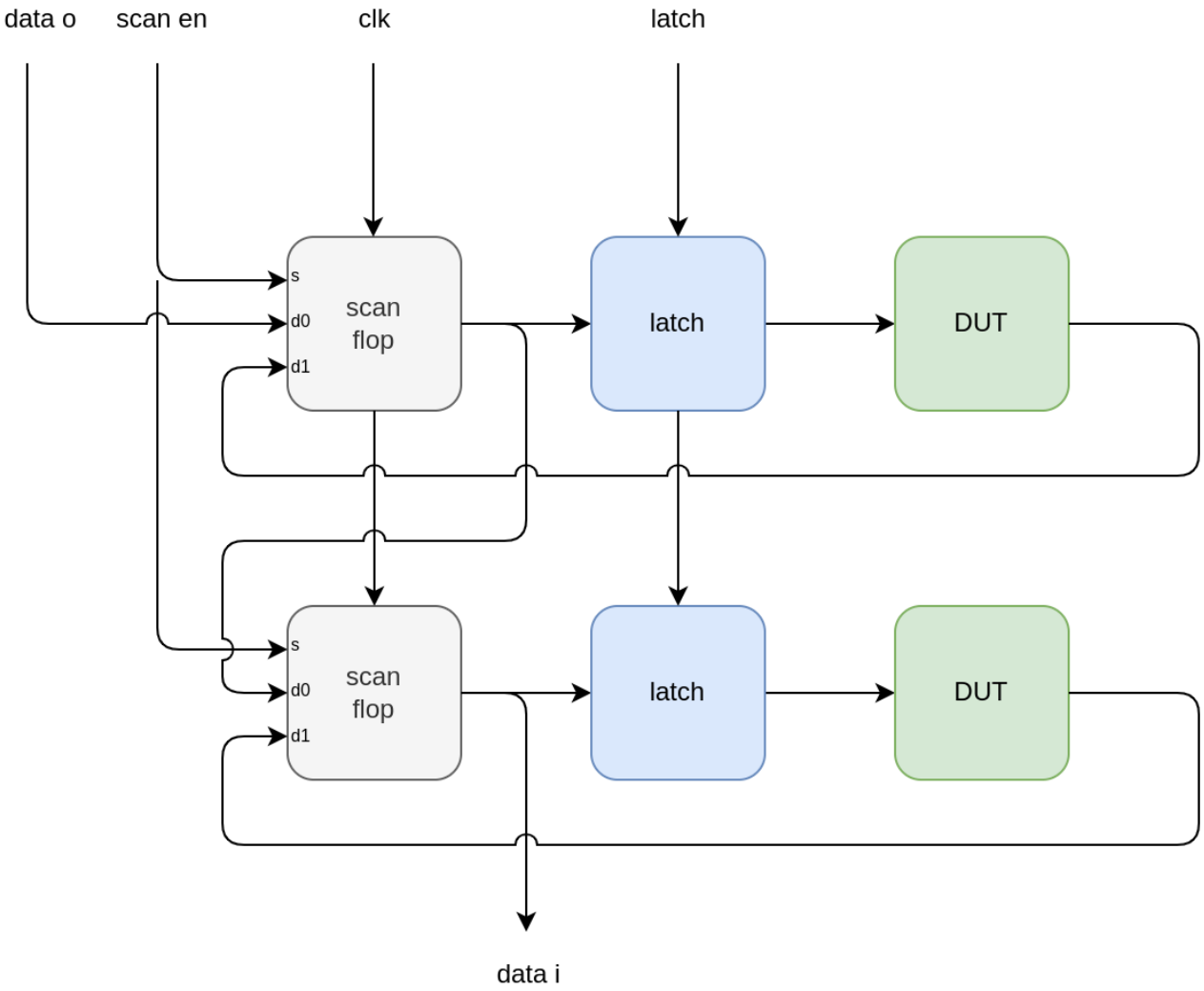

Figure 33: block diagram

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

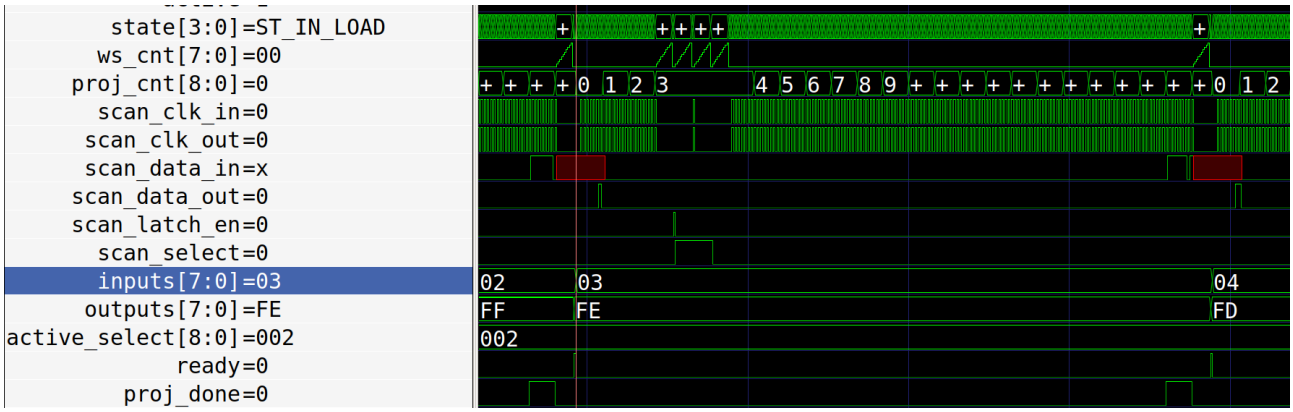

Figure 34: update cycle

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)
- 250 designs
- scan controller can do a read/write cycle in one refresh

So the max refresh rate is 100MHz / (8 * 2 * 250) = 25000Hz.

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 250 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=250 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