

# Tiny Tapeout 03 Datasheet

Project Repository

<https://github.com/TinyTapeout/tinytapeout-03>

April 25, 2023

## Contents

Render of whole chip	9
<b>Projects</b>	<b>10</b>
0 : Test Inverter Project . . . . .	10
1 : ChipTune . . . . .	11
The ChipTune Project . . . . .	12
2 : 7 Segment Life . . . . .	17
3 : Another Piece of Pi . . . . .	20
4 : Wormy . . . . .	22
5 : Knight Rider Sensor Lights . . . . .	26
6 : Single digit latch . . . . .	28

7 : 4x4 Memory . . . . .	29
8 : KS-Signal . . . . .	30
9 : Hovalaa CPU . . . . .	32
10 : SKINNY SBOX . . . . .	33
11 : Stateful Lock . . . . .	34
12 : Ascon's 5-bit S-box . . . . .	35
13 : 8bit configurable galois Lfsr . . . . .	36
14 : Sbox SKINNY 8 Bit . . . . .	37
15 : BinaryDoorLock . . . . .	38
16 : bad apple . . . . .	39
17 : TinyFPGA attempt for TinyTapeout3 . . . . .	40
18 : 4bit Adder . . . . .	41
19 : 12-bit PDP8 . . . . .	42
20 : CTF - Catch the fish . . . . .	44
21 : Dot operation calculator . . . . .	45
22 : Random number guess game . . . . .	46
23 : Desperate Tapeout . . . . .	47
24 : Simple multiply . . . . .	48
25 : Parallel Nibble to UART . . . . .	49
26 : tiny logic analyzer . . . . .	50
27 : XOR Stream Cipher . . . . .	52
28 : LED Panel Driver . . . . .	55
29 : 6-bit FIFO . . . . .	57
30 : ezm_cpu . . . . .	59
31 : 31b-PrimeDetector . . . . .	60
32 : 4-bit ALU . . . . .	62
33 : Pulse-Density Modulators . . . . .	64
34 : CRC Decelerator . . . . .	67
35 : Simple clock . . . . .	71
36 : Binary to DEC and HEX . . . . .	72
37 : Simon Says . . . . .	73
38 : Shift Register Ram . . . . .	76
39 : tinysat . . . . .	77
40 : POV display . . . . .	78
41 : Toy CPU . . . . .	79
42 : Base-10 grey counter counts from zero to a trillion . . . . .	81
43 : ttFIR: Digital Finite Impulse Response (FIR) Filter . . . . .	85
44 : QTCore-A1 . . . . .	87
45 : MicroTapeout (of sky130 cells) . . . . .	98
46 : nipple multiplier . . . . .	104
47 : Synthesizable Digital Temperature Sensor . . . . .	106
48 : 4-bits sequential ALU . . . . .	108

49 : Brightness control of LED with PWM . . . . .	109
50 : Microtapeout . . . . .	110
Shift Register . . . . .	110
ALU . . . . .	110
Adder . . . . .	110
Seven Segment . . . . .	110
51 : Neptune guitar tuner (fixed window) . . . . .	112
52 : Neptune guitar tuner (proportional window) . . . . .	114
53 : M segments . . . . .	116
54 : 7 segment seconds . . . . .	117
55 : 7 segment wokwi counter . . . . .	118
56 : Straight through test . . . . .	120
57 : Combo lock . . . . .	121
58 : ro-based_tempsense . . . . .	122
59 : FSM_LAT . . . . .	123
60 : 7 segment seconds . . . . .	124
61 : MMM Finite State Machine (4 States) . . . . .	125
States . . . . .	125
Outputs . . . . .	125
62 : FSK modem . . . . .	127
63 : QTChallenges . . . . .	128
64 : HiddenCPU . . . . .	146
65 : Simple UART interface . . . . .	147
66 : MSF Clock . . . . .	149
67 : Hamming(7,4) encoder decoder . . . . .	151
68 : I2S reeceiver, data mix and transmitter . . . . .	152
69 : S4GA: Super Slow Serial SRAM FPGA . . . . .	153
70 : 7 Segment Random Walk . . . . .	155
71 : Tiny binarized neural network . . . . .	156
72 : RTL Locked QTCore-A1 . . . . .	157
73 : Arbiter Game . . . . .	164
74 : Frequency Divider . . . . .	166
75 : FullAdderusing4is1 . . . . .	167
76 : BCDtoDECIMAL . . . . .	169
77 : AI Decelerator . . . . .	171
78 : 3BitParallelAdder . . . . .	172
79 : Asynchronous 3-Bit Down Counter . . . . .	174
80 : Ring oscillator with skew correction . . . . .	176
81 : 3 bit multiplier . . . . .	178
82 : Tiny Teeth Toothbrush Timer . . . . .	180
83 : 2's Compliment Subtractor . . . . .	182
84 : Customizable Padlock . . . . .	183

85 : Customizable UART String . . . . .	184
86 : Customizable UART Character . . . . .	185
87 : 7-Seg ‘Tiny Tapeout’ Display . . . . .	186
88 : Hola . . . . .	187
89 : 5_1MUX . . . . .	188
90 : 3-bit 4-position register . . . . .	189
91 : Simple adder used for educational purposes . . . . .	190
92 : SIMON Cipher . . . . .	191
93 : HD74480 Clock . . . . .	192
94 : Scrolling Binary Matrix display . . . . .	194
95 : Power supply sequencer . . . . .	196
96 : Duty Controller . . . . .	197
97 : ALU . . . . .	198
98 : The McCoy 8-bit Microprocessor . . . . .	199
99 : binary clock . . . . .	200
100 : TinySensor . . . . .	202
101 : Water-Level-Indicator-With-Auto-Motor-Control . . . . .	204
102 : 16x8 SRAM & Streaming Signal Generator . . . . .	205
103 : German Traffic Light State Machine . . . . .	208
104 : 4-spin Ising Chain Simulation . . . . .	209
105 : Avalon Semiconductors ‘5401’ 4-bit Microprocessor . . . . .	211
106 : small FFT . . . . .	213
107 : Stream Integrator . . . . .	214
108 : tiny-fir . . . . .	216
109 : Configurable SR . . . . .	217
110 : LUTRAM . . . . .	219
111 : chase the beat . . . . .	220
112 : BCD to 7-segment encoder . . . . .	221
113 : A LED Flasher . . . . .	222
114 : 4-bit Multiplier . . . . .	223
115 : Avalon Semiconductors ‘TBB1143’ Programmable Sound Generator .	224
116 : RGB LED Matrix Driver . . . . .	225
117 : Tiny Phase/Frequency Detector . . . . .	226
118 : Loading Animation . . . . .	227
119 : tiny egg timer . . . . .	229
120 : Potato-1 (Brainfuck CPU) . . . . .	230
121 : heart zoe mom dad . . . . .	233
122 : Tiny Synth . . . . .	234
123 : 5-bit Galois LFSR . . . . .	235
124 : prbs15 . . . . .	236
125 : 4-bit badge ALU . . . . .	237
126 : Pi ( ) to 1000+ decimal places . . . . .	238

127 : Siren . . . . .	240
128 : YaFPGA . . . . .	241
129 : M0: A 16-bit SUBLEQ Microprocessor . . . . .	242
130 : bitslam . . . . .	244
131 : 8x8 Bit Pattern Player . . . . .	246
132 : XLS: bit population count . . . . .	248
133 : RC5 decoder . . . . .	249
134 : chiDOM . . . . .	250
135 : Super Mario Tune on A Piezo Speaker . . . . .	251
136 : Tiny rot13 . . . . .	253
137 : 4 bit counter on steamdeck . . . . .	255
138 : Shiftregister Challenge 40 Bit . . . . .	256
139 : TinyTapeout2 4-bit multiplier. . . . .	258
140 : TinyTapeout2 multiplexed segment display timer. . . . .	260
141 : XLS: 8-bit counter . . . . .	261
142 : XorShift32 . . . . .	262
143 : XorShift32 . . . . .	263
144 : Multiple Tunes on A Piezo Speaker . . . . .	264
145 : TinyTapeout 2 LCD Nametag . . . . .	265
146 : UART-CC . . . . .	266
147 : 3-bit 8-channel PWM driver . . . . .	267
148 : LEDChaser from LiteX test . . . . .	268
149 : 8-bit (E4M3) Floating Point Multiplier . . . . .	269
150 : Dice roll . . . . .	271
151 : CNS TT02 Test 1:Score Board . . . . .	272
152 : CNS002 (TT02-Test 2) . . . . .	273
153 : Test2 . . . . .	274
154 : 7-segment LED flasher . . . . .	276
155 : Nano-neuron . . . . .	277
156 : SQRT1 Square Root Engine . . . . .	278
157 : Breathing LED . . . . .	279
158 : Fibonacci & Gold Code . . . . .	280
159 : tinytapeout2-HELLo-3orLd-7seg . . . . .	282
160 : Non-restoring Square Root . . . . .	283
161 : GOL-Cell . . . . .	285
162 : 7-channel PWM driver controlled via SPI bus . . . . .	287
163 : hex shift register . . . . .	288
164 : Ring OSC Speed Test . . . . .	289
165 : TinyPID . . . . .	291
166 : TrainLED2 - RGB-LED driver with 8 bit PWM engine . . . . .	292
167 : Zinnia+ (MCPU5+) 8 Bit CPU . . . . .	294
168 : 4 bit CPU . . . . .	295

169 : Stack Calculator . . . . .	297
170 : 1-bit ALU . . . . .	302
171 : SPI Flash State Machine . . . . .	304
172 : r2rdac . . . . .	306
173 : Worm in a Maze . . . . .	307
174 : 8 bit CPU . . . . .	308
175 : Pseudo-random number generator . . . . .	310
176 : BCD to 7-Segment Decoder . . . . .	311
177 : Frequency Counter . . . . .	312
178 : Taillight controller of a 1965 Ford Thunderbird . . . . .	313
179 : FPGA test . . . . .	314
180 : chi 2 shares . . . . .	315
181 : chi 3 shares . . . . .	316
182 : Whisk: 16-bit Serial RISC CPU . . . . .	317
183 : Scalable synchronous 4-bit tri-directional loadable counter . . . . .	319
184 : Asynchronous Binary to Ternary Converter and Comparator . . . . .	321
185 : Vector dot product . . . . .	323
186 : Monte Carlo Pi Integrator . . . . .	324
187 : Funny Blinky . . . . .	325
188 : GPS C/A PRN Generator . . . . .	326
189 : Sigma-Delta ADC/DAC . . . . .	327
190 : BCD to Hex 7-Segment Decoder . . . . .	329
191 : SRLD . . . . .	330
192 : Counter . . . . .	331
193 : 2bitALU . . . . .	332
194 : A (7, 1/2) Convolutional Encoder . . . . .	333
195 : Tiny PIC-like MCU . . . . .	334
196 : RV8U - 8-bit RISC-V Microcore Processor . . . . .	335
197 : Logic-2G97-2G98 . . . . .	336
198 : Melody Generator . . . . .	337
199 : Rotary Encoder Counter . . . . .	338
200 : Wolf sheep cabbage river crossing puzzle ASIC design . . . . .	339
201 : Low-speed UART transmitter with limited character set loading . . . . .	341
202 : Rotary encoder . . . . .	343
203 : FROG 4-Bit CPU . . . . .	345
204 : Configurable Gray Code Counter . . . . .	346
205 : Baudot Converter . . . . .	349
206 : Marquee . . . . .	350
207 : channel coding . . . . .	351
208 : Chisel 16-bit GCD with scan in and out . . . . .	352
209 : Adder with 7-segment decoder . . . . .	353
210 : Hex to 7 Segment Decoder . . . . .	355

211 : Multiple seven-segment digit buffer . . . . .	356
212 : LED Chaser . . . . .	358
213 : Rolling Average - 5 bit, 8 bank . . . . .	359
214 : w5s8: universal turing machine core . . . . .	360
215 : Test3 . . . . .	361
216 : Seven Segment Clock . . . . .	363
217 : serv - Serial RISCV CPU . . . . .	364
218 : 4:2 Compressor . . . . .	365
219 : PS2 keyboard Interface . . . . .	366
220 : Hello Generator . . . . .	367
221 : MicroASIC VI . . . . .	369
222 : Optimised Euclidean Algorithm . . . . .	370
223 : CRC-16 and Parity calculator . . . . .	371
224 : SevSegFX . . . . .	372
225 : LAB11 . . . . .	373
226 : Option23 Serial . . . . .	375
227 : Option23 . . . . .	376
228 : Option22 . . . . .	377
229 : 4x4 RAM . . . . .	378
230 : Digital padlock . . . . .	379
231 : FFT Butterfly in Wokwi . . . . .	381
232 : Femto 4-bit CPU . . . . .	382
233 : Logisim demo - LED blinker . . . . .	383
234 : Secret File . . . . .	384
235 : Basic 4 bit cpu . . . . .	385
236 : Adi counter . . . . .	387
237 : Clock divider ASIC . . . . .	388
238 : Amaranth 6 Bits Gray counter . . . . .	389
239 : Laura's L . . . . .	390
240 : Two Bit Universal FSM . . . . .	391
241 : Super Mario Tune on A Piezo Speaker . . . . .	393
242 : PSRANDOM . . . . .	395
243 : Gameshow Buzzer . . . . .	396
244 : Balanced Ternary Calculator . . . . .	397
245 : RiscV Scan Chain based CPU – block 1 – clocking . . . . .	398
246 : RiscV Scan Chain based CPU – block 2 – instructions . . . . .	399
247 : RiscV Scan Chain based CPU – block 3 – registers . . . . .	400
248 : RiscV Scan Chain based CPU – block 4 – ALU . . . . .	401

<b>Technical info</b>	<b>402</b>
Scan chain . . . . .	402
Clocking . . . . .	403

Clock divider . . . . .	404
Wait states . . . . .	404
Pinout . . . . .	404
Instructions to build GDS . . . . .	405
Changing macro block size . . . . .	406
<b>Verification</b>	<b>407</b>
Setup . . . . .	407
Simulations . . . . .	407
Top level tests setup . . . . .	408
Formal Verification . . . . .	409
Timing constraints . . . . .	409
Physical tests . . . . .	410
<b>Toplevel STA and Spice analysis for TinyTapeout-02 and 03.</b>	<b>411</b>
Introduction . . . . .	411
GitHub action . . . . .	411
Prerequisites . . . . .	411
Assumptions . . . . .	412
Issues . . . . .	412
Invoking STA analysis . . . . .	412
How it works . . . . .	412
Invoking Spice simulation: . . . . .	413
<b>Sponsored by</b>	<b>415</b>
<b>Team</b>	<b>415</b>

# Render of whole chip



Figure 1: Full GDS

# Projects

## 0 : Test Inverter Project

- Author: Matt Venn
- Description: Inverts every line. This project is also used to fill any empty design spaces.
- 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	a	segment a
1	b	segment b
2	c	segment c
3	d	segment d
4	e	segment e
5	f	segment f
6	g	segment g
7	dot	dot

# 1 : ChipTune

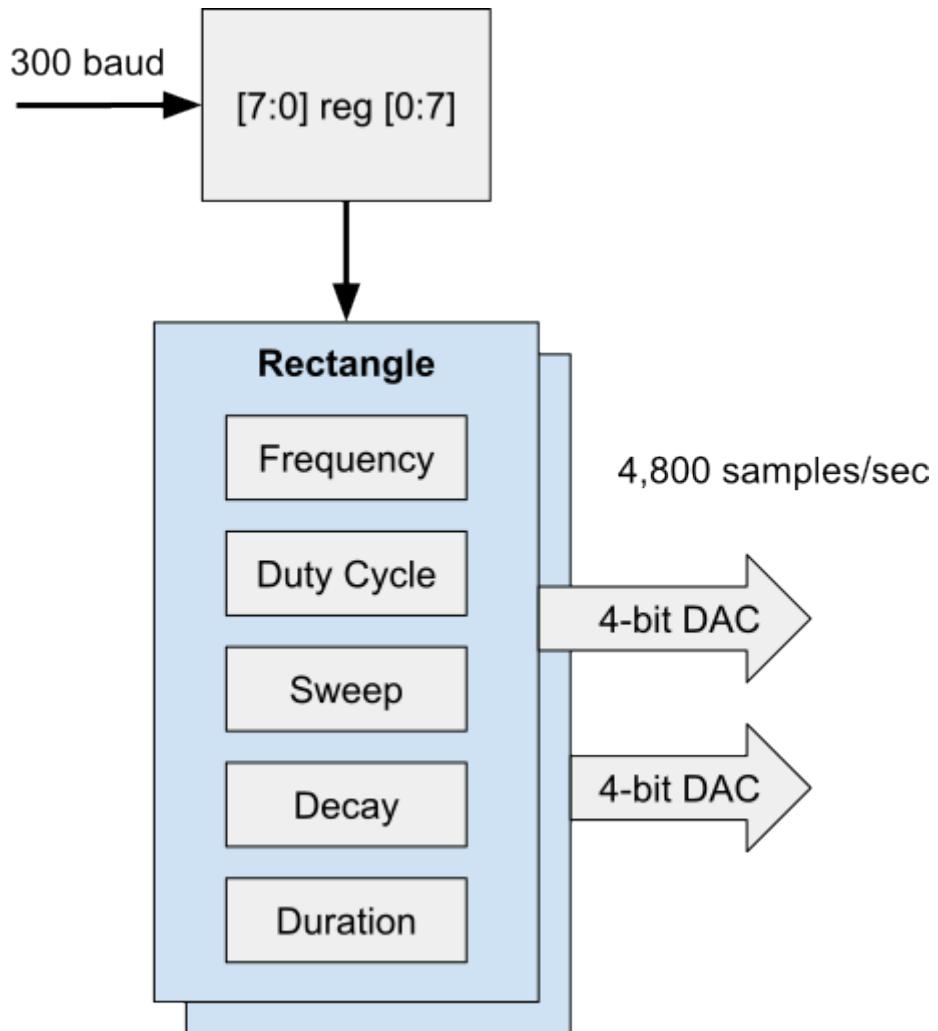


Figure 2: picture

- Author: Wallie Everest
- Description: Vintage 8-bit sound generator
- GitHub repository
- HDL project
- Extra docs
- Clock: 9600 Hz
- External hardware: Computer COM port

## How it works

Chiptune implements an 8-bit Programmable Sound Generator. Input is from a serial UART interface. Output is 4-bit DAC audio.

# The ChipTune Project

This is a two-in-one project. First, an audio device replicates the square-wave sound generators of vintage video games. Second, a re-imagined version of the scan controller is implemented to demonstrate faster data acquisition.

## TinyTapeout 3 Configuration

For this third Multi Project Chip (MPC) tapeout, the original scan chain will be configured in ‘external mode’. The inputs and outputs for user projects will be derived externally from scanchain data, not from the chip’s I/O pins. The scanchain control signals will occupy pins designated `io_in` and `io_out`. Expected throughput is better than 10k byte/second. The ChipTune project will configure the shift clock to attain 9600 bytes/sec. This speed provides a 4800 Hz clock and 300 baud communication to the tiny user project.

[Top Level Drawing]

Devices from the eFabless Multi-Project Wafer (MPW) shuttle are delivered in two package options, each with 64 pins. TinyTapeout 2 will be packaged in a QFN, whereas TinyTapout 3 will be packaged as a WCSP. In both delivered configurations, the device is mounted to a daughter board PCB with 10 castellated pins per side. The daughter board provides basic clock, configuration and power management for the Caravel SoC.

## MPRJ\_IO Pin Assignments

Signal	Name	Dir	WCSP	QFN	PCB
<code>mprj_io[0]</code>	JTAG	In	D7	31	J3.14
<code>mprj_io[1]</code>	SDO	Out	E9	32	J3.13
<code>mprj_io[2]</code>	SDI	In	F9	33	J3.12
<code>mprj_io[3]</code>	CSB	In	E8	34	J3.11
<code>mprj_io[4]</code>	SCK	In	F8	35	J3.10
<code>mprj_io[5]</code>	SER_RX	In	E7	36	J3.9
<code>mprj_io[6]</code>	SER_TX	Out	F7	37	J3.8
<code>mprj_io[7]</code>	IRQ	In	E5	41	J3.7
<code>mprj_io[8]</code>	DRIVER_SEL[0]	In	F5	42	J3.6
<code>mprj_io[9]</code>	DRIVER_SEL[1]	In	E4	43	J3.5
<code>mprj_io[10]</code>	SLOW_CLK	Out	F4	44	J3.4
<code>mprj_io[11]</code>	SET_CLK_DIV	In	E3	45	J3.3
<code>mprj_io[12]</code>	ACTIVE_SELECT[0]	In	F3	46	J3.2
<code>mprj_io[13]</code>	ACTIVE_SELECT[1]	In	D3	48	J3.1
<code>mprj_io[14]</code>	ACTIVE_SELECT[2]	In	E2	50	J2.14

Signal	Name	Dir	WCSP	QFN	PCB
mpj_io[15]	ACTIVE_SELECT[3]	In	F1	51	J2.13
mpj_io[16]	ACTIVE_SELECT[4]	In	E1	53	J2.12
mpj_io[17]	ACTIVE_SELECT[5]	In	D2	54	J2.11
mpj_io[18]	ACTIVE_SELECT[6]	In	D1	55	J2.10
mpj_io[19]	ACTIVE_SELECT[7]	In	C10	57	J2.9
mpj_io[20]	ACTIVE_SELECT[8]	In	C2	58	J2.8
mpj_io[21]	IO_IN[0] / EXT_SCAN_CLK_OUT	In	B1	59	J2.7
mpj_io[22]	IO_IN[1] / EXT_SCAN_DATA_OUT	In	B2	60	J2.6
mpj_io[23]	IO_IN[2] / EXT_SCAN_SELECT	In	A1	61	J2.5
mpj_io[24]	IO_IN[3] / EXT_SCAN_LATCH_EN	In	C3	62	J2.4
mpj_io[25]	IO_IN[4]	In	A3	2	J2.3
mpj_io[26]	IO_IN[5]	In	B4	3	J2.2
mpj_io[27]	IO_IN[6]	In	A4	4	J2.1
mpj_io[28]	IO_IN[7]	In	B5	5	J1.14
mpj_io[29]	IO_OUT[0] / EXT_SCAN_DATA_IN	Out	A5	6	J1.13
mpj_io[30]	IO_OUT[1] / EXT_SCAN_DATA_IN	Out	B6	7	J1.12
mpj_io[31]	IO_OUT[2]	Out	A6	8	J1.11
mpj_io[32]	IO_OUT[3]	Out	A7	11	J1.10
mpj_io[33]	IO_OUT[4]	Out	C8	12	J1.9
mpj_io[34]	IO_OUT[5]	Out	B8	13	J1.4
mpj_io[35]	IO_OUT[6]	Out	A8	14	J1.3
mpj_io[36]	IO_OUT[7]	Out	B9	15	J1.2
mpj_io[37]	READY	Out	A9	16	J1.1

## Caravel Connections

Within the Caravel SoC, the TinyTapeout project has configured the user space into 250 sub-projects. Each project is interconnected by a scanchain that serpentine through the chip. Control of the 4-wire chain provides access to each project.

### [Caravel]

The scanchain topology has pros and cons, as would any interconnect scheme. This project presents an alternative topology based on a JTAG implementation. The advantage is a reduction in the length of the register latency.

## Scanchain V1

- Pro: Economical use of resources. Readily hardened in ASIC fabric. Testable on an FPGA platform.
- Con: Very long register chain (2,000) impacts overall acquisition rate.

## **Scanchain V2**

- Pro: Short register chain (10) minimizes latency. Economical use of resources. Testable on an FPGA platform.
- Con: Speed limited by length of multiplexer propagation (250 instances).

## **Multiplexer**

- Pro: Pure combinatorial output. Potential to be the fastest option.
- Con: Requires a large number of long routing resources. Internal tri-state busses not testable on an FPGA platform.

## **Tiny Project Configuration**

This tiny project embeds the revised scanchain (version 2) to investigate its low latency and testability. The penalty for this implementation is 10 clock cycles per transfer. The delay is mitigated by a serial UART expanding the internal registers of the audio generator. Four extra scanchain endpoints are included for demonstration purposes.

### [Project]

A clock generator is used to recover a bit-clock from asynchronous serial data. An external 16x baud clock is required by the design. This planned 4800 Hz project clock will permit a 300 baud serial link.

## **Scanchain Version 2**

The re-imagined scanchain uses a bypass technique commonly seen with JTAG devices. Each ‘tap’ in the chain routes data through a combinatorial multiplexer until the module is activated. Individual taps are assigned a unique 8-bit address during tapeout elaboration. A tap is activated when it receives a matching address message, enabling its’ 8-bit shift register. Unselected taps drive logic 0 into the projects’ inputs to keep them in a quiescent state.

### [Scanchain V2]

Encoding of serial data is compatible with the ubiquitous UART format. The waveform is one-start, eight-data, one-stop (300,8,N,1). Least significant bits are transmitted first. An immediate advantage is the use of a computer COM port to generate and analyze functional data. The serial interface is in addition to decoded parallel data available on the I/O ports.

## **ChipTune Operation**

The audio portion of the project consists of two rectangular pulse generators. Each module is controlled by four 8-bit registers. Configurable parameters are the frequency, duty cycle, sweep, decay, and note duration.

### [Operation]

The frequency range of the project is limited by the legacy scanchain, but mid-range frequencies are acceptable. Additional triangle and noise modules will be added in future work when more bandwidth is available.

## **Design For Test Considerations**

An isolated instance of scanchain version 2 has been tested on an FPGA platform with good results. A shift rate of 3.7 MHz permits communication with the computer at 115,200 baud. Longer scan chains do not affect throughput until multiplexer delays become dominant. For an FPGA, 75% of the timing delays are attributed to routing resources.

### [Test Configuration]

Output of the sub-project is always available at both the parallel output port and the serial data. A ‘Mode’ signal driven by the DTS line controls whether the project’s input is derived from the parallel input port or serial data.

## **Summary**

The next shuttle for TinyTapeout is planning a multiplexer for selecting between the 250 projects. This will alleviate latency in the present design. The revised scanchain offered here is an alternative for other group projects. The scanchain topology still holds merit in many applications, especially for the application of a computer-based logic analyzer via a USB COM port.

## **How to test**

The ChipTune project can be interfaced to a computer COM port at 300 baud. The project’s scanchain is updated at 9600 Hz such that a 4800 Hz UART clock is available at `io_in[4]`. The system clock on `io_in[0]` may also be 4800 Hz. An external bit clock must be provided on `io_in[5]` with a rising edge in the center of the bit period. This clock may be sourced from the bit-aligned reference clock on `io_out[5]`. Serial data is received on `io_in[7]`. Serial data is decoded by the device as an address when `io_in[6] = 0`, and as data when `io_in[6] = 1`. A 4-bit DAC output is available

on `io_out[4:0]`. The mode pin, `io_in[3]` must stay high for this version of the design.

## IO

#	Input	Output
0	<code>io_in[0]</code> (CLOCK)	<code>io_out[0]</code> (DAC_0)
1	<code>io_in[1]</code> (IN_1)	<code>io_out[1]</code> (DAC_1)
2	<code>io_in[2]</code> (IN_2)	<code>io_out[2]</code> (DAC_2)
3	<code>io_in[3]</code> (MODE)	<code>io_out[3]</code> (DAC_3)
4	<code>io_in[4]</code> (BAUD_CLK)	<code>io_out[4]</code> (DAC_4)
5	<code>io_in[5]</code> (TCK)	<code>io_out[5]</code> (REF_CLK)
6	<code>io_in[6]</code> (TMS)	<code>io_out[6]</code> (RTCK)
7	<code>io_in[7]</code> (TDI)	<code>io_out[7]</code> (TDO)

## 2 : 7 Segment Life

- Author: icegoat9
- Description: Simple 7-segment cellular automaton
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 6250 Hz
- External hardware: None. Could add debounced momentary pushbuttons in parallel with dip switches 1,2,3 to make loading in new patterns and stepping through a run easier.

### How it works

This implements a very simple 7-segment cellular automaton from ~200 basic logic gates. At each clock cycle, each of the seven segments is set to “alive” or “dead” based on a simple set of rules:

1. If a segment was “alive” in the previous time step and has exactly one living neighbor, it survives.
2. If a segment was “dead” (or “empty”) in the previous time step and has exactly two living neighbors, it becomes alive (“gives birth”)

A “neighbor” is any segment it touches, tip to tip. This means that the top and bottom segments only have two neighbors, while the side segments have three neighbors and the center segment has four neighbors.

### Implementation

See the Wokwi gate layout and simulation. At a high level:

- Seven flip-flops hold the cellular automaton’s internal state, which is also wired to the seven-segment display.
- Combinatorial logic generates the next state for each segment based on its neighbors.
- When either the system clock or a user-toggled clock input goes high, this new state is latched in to the flip-flops.
- There’s minor additional support logic to let the user manually shift in an initial condition and handle clock dividing.

### How to test

For full details with some examples, see the github README doc link. At a high level, assuming the IC is mounted on the standard tinytapeout PCB which provides dip

switches, clock, and a seven-segment display for output:

1. Set all dip switches off, and the clock slide switch to the ‘manual’ clock side.
2. Power on the system. An arbitrary state may appear on the 7-segment display.
3. Set dip switch 4 on ('run mode').
4. Toggle dip switch 1 on and off to advance the automaton to the next state, you should see the 7-segment display update.

If you want to watch it run automatically (which may quickly settle on an empty state or a static pattern, depending on start state):

1. Set the PCB clock divider to the maximum clock division (255). With a system clock of 6.25kHz, the clock input should now be ~24.5Hz.
2. Set dip switches 5 and 7 on to add an additional 16x clock divider.
3. Set dip switch 4 on.
4. Switch the clock slide switch to the ‘system clock’ side. The display should advance at roughly 1.5Hz (if I’ve done math correctly)
5. To run faster or slower, set a combination of dip switches 5 (8x clock divider), 6 (4x divider), and 7 (2x divider)

If you want to load a custom initial state:

1. Set dip switch 4 off ('load mode').
2. Toggle dip switches 2 and/or 3 on and off up to seven times total, to shift in 0 and 1 values to the automaton’s internal state (see github README for examples).
3. Set dip switch 4 on and run manually or automatically as above.

### **Exercises / puzzles for the reader:**

1. How many unique initial states are there, disregarding equivalent mirrored/rotated states? (there are  $2^7 = 128$  possible initial states, but many are equivalent)
2. What fraction of these initial states survive? (i.e. don’t eventually die out)
3. What fraction settle into a static living pattern vs an infinite cycle between multiple different patterns?
4. What is the longest sequence of unique states a pattern travels through (stop counting once it reaches a previously-visited state, beginning an infinite loop)?
5. What is the longest cycle of unique states that repeats in a loop?

## **IO**

#	Input	Output
0	clock	7segmentA
1	load0	7segmentB

#	Input	Output
2	load1	7segmentC
3	runmode	7segmentD
4	clockdiv8	7segmentE
5	clockdiv4	7segmentF
6	clockdiv2	7segmentG
7	unused	7segmentDP

### 3 : Another Piece of Pi

- Author: Meinhard Kissich, EAS Group, Graz University of Technology
- Description: This design takes up the idea of James Ross [1], who submitted a circuit to Tiny Tapeout 02 that stores and outputs the first 1024 decimal digits of the number Pi (including the decimal point) to a 7-segment display. In contrast to his approach, a densely packed decimal encoding is used to store the data. With this approach, 1400 digits can be stored and output within the design area of 150um x 170um. However, at 1400 decimals and utilization of 38.99%, the limitation seems to be routing. Like James, I'm also interested to hear about better strategies to fit more information into the design with synthesizable Verilog code. [1] [https://github.com/jar/tt02\\_freespeech](https://github.com/jar/tt02_freespeech)
- GitHub repository
- HDL project
- Extra docs
- Clock: 0 Hz
- External hardware: 7-segment display

#### How it works

The circuit stores each triplet of decimals in a 10-bit vector encoded as densely packed decimals. An index vector selects the current digits to be output to the 7-segment display. It consists of an upper part `index[11:2]` that selects the triplet and a lower part `index[1:0]` that specifies the digit within the triplet. First, the upper part decides on the triplet, which is then decoded into three decimals. Afterwards, the lower part selects one of the three decimals to be decoded into 7-segment display logic and applied to the outputs. The index is incremented at each primary clock edge. However, when the lower part equals three, i.e., `index[1:0]==1'b10`, two is added, as the triplet consists of three (not four) digits.

- `index == 'b0000000000|00: triplet[0], digit 0 within triplet`
- `index == 'b0000000000|01: triplet[0], digit 1 within triplet`
- `index == 'b0000000000|10: triplet[0], digit 2 within triplet`
- `index == 'b0000000001|00: triplet[1], digit 0 within triplet`
- `index == 'b0000000001|01: triplet[1], digit 1 within triplet`
- `index == 'b0000000001|10: triplet[1], digit 2 within triplet`

There is one exception to the rule above: the decimal point. Another multiplexer

at the input of the 7-segment decoder can either forward a digit from the decoded triplet or a constant – the decimal point. Once the lower part of the index counter, i.e., `index[1:0]` reaches 2'b10 for the first time, the multiplexer selects the decimal point and pauses incrementing the index for one clock cycle.

- `index == 'b0000000000|00`: triplet[0], digit 0 within triplet
- `index == 'b0000000000|01`: triplet[0], decimal point
- `index == 'b0000000000|01`: triplet[0], digit 1 within triplet
- `index == 'b0000000000|10`: triplet[0], digit 2 within triplet
- `index == 'b0000000001|00`: triplet[1], digit 0 within triplet

## How to test

For simulation, please use the provided testbench and Makefile. It is important to run the `genmux.py` Python script first, as it generates the test vectors required by the Verilog testbench. For testing the physical chip, release the reset and compare the digits of Pi against a reference.

## 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

## 4 : Wormy

- Author: nqbit
- Description: MC Wormy Pants squirms like a worm and grows just as fast.
- GitHub repository
- HDL project
- Extra docs
- Clock: 300 Hz
- External hardware:

### How it works

Wormy is a very simple, addictive last person video game. This last person, open-world game takes you down the path of an earthworm. Wormy's world is made up of a 4x4 grid represented by 3x16-bit arrays: Direction[0], Direction[1], and Occupied. The Direction[x] maps keep track of which way a segment of worm moves, if it is on, and Occupied keeps track of if the grid location is occupied.

Example Occupied Grid:

	X	X	X	X	
			X		
			X		

In addition to direction and occupied there are also pointers to the head and the tail. The same grid would look something like the following:

Example Occupied Grid with head(H) and tail(T) highlighted:

	T	X	X	X	
			X		
			H		

BOOM! Wormy shouldn't run into itself. If its head hits any part of its body, that causes a collision. There is a collision if the location of the Wormy's head is occupied by another segment of the Wormy. To determine this, we keep track of the worm location, and specifically the current and future locations of the worm head (H) and tail (T). If the future location of H will occupy a location that will already be occupied, this causes a collision.

This is made a bit trickier with growth (see below), because if the future state of H is set to occupy the current state of T - there is only a collision if growth is set to occur

in the next cycle, so be careful if you plan to have Wormy chase its tail! SPLAT!

NOM NOM! Wormy eats the tasty earth around it and grows. Every so often Wormy, after having eaten all of its lunch, grows a whole segment. During a growth cycle, the state of T simply persists (remains occupied and heading in the same direction as it is).

Last Person Input - see user\_input.v To move Wormy along, the last player needs to push buttons to help Wormy find more tasty earth: up, right, down or left.

Buttons are nasty little bugs in general. When pushed the button generates an analog signal that might not look exactly like a single rising edge.

It might look something like this:

In order to help protect our logic from this scary looking signal that might introduce metastability (<- WHAT?), we can filter it with a couple flippy-floppies and keep the metastability at bay. ARGH.

Once we have a clear pushed or not-pushed, we can suggest that Wormy move in a specific direction. If the last player tries button mashing, Wormy won't listen. Once a second Wormy checks what the last button press was and tries really hard to go that way (see BOOM!).

Earthworms don't have eyes - see multiplexer.v The game's display is made up of a 4x4

grid of LEDs controlled by a multiplexer. Why multiplexing? With a multiplexed LED setup, we can control more display units (LEDs), with a limited number of outputs (8 on this TinyTapeout project).

To get multiplexing working the network of outputs is mapped to each display unit. This allows us to manipulate assigned outputs to control the state of each display unit, one at a time. We then cycle through each display unit quickly enough to display a persistent image to the last player.

Wires (A1-4, B1-4) map to each location on the game arena (4x4 grid):

A1				
-----				
A2				
-----				
A3				
-----				
A4				
-----				
	B1	B2	B3	B4

When B's voltage is OFF the LED's state changes to ON if A is also ON. - ON LED state: A(ON) — >| — B(OFF) - OFF LED state: A(ON) — >| — B(ON)

Example: The 3 filled squares below each represent a Wormy segment in the ON state as controlled by the multiplexer. Notice how each is lighter than the last. This is because the multiplexer cycles through each LED to update the state, creating one persistent image even though the LEDS are not on over the entire period of time.

A1		0		
-----				
A2		o		
-----				
A3		.		
-----				
A4				
-----				
	B1	B2	B3	B4

Another Example: If you enter a dark cave and point a flashlight straight ahead at one point on the wall you have a very small visual field that is contained within the beam of light. However, you can expand your visual field in the cave by waving the flashlight back and forth across the wall. Despite the fact that the beam is moving over individual points on the wall, the entire wall can be seen at once. This is similar to the concept used in the Wormy display, since the multiplexer changes the state of

the worm occupied locations to ON one at a time, but in a cycle. The result is a solid image, made up of LEDs cycling through ON states to produce a persistent image of Wormy (that beautiful Lumbricina).

## How to test

After reset, you should see a single pixel moving along the display and it should grow every now and then.

## IO

#	Input	Output
0	clock	A0 - Multiplexer channel A to be tied to a an array of 16 multiplexed LEDs
1	reset	A1
2	button0	A2
3	button1	A3
4	button2	B0 - Multiplexer channel B to be tied to a an array of 16 multiplexed LEDs
5	button3	B1
6	none	B2
7	none	B3

## 5 : Knight Rider Sensor Lights

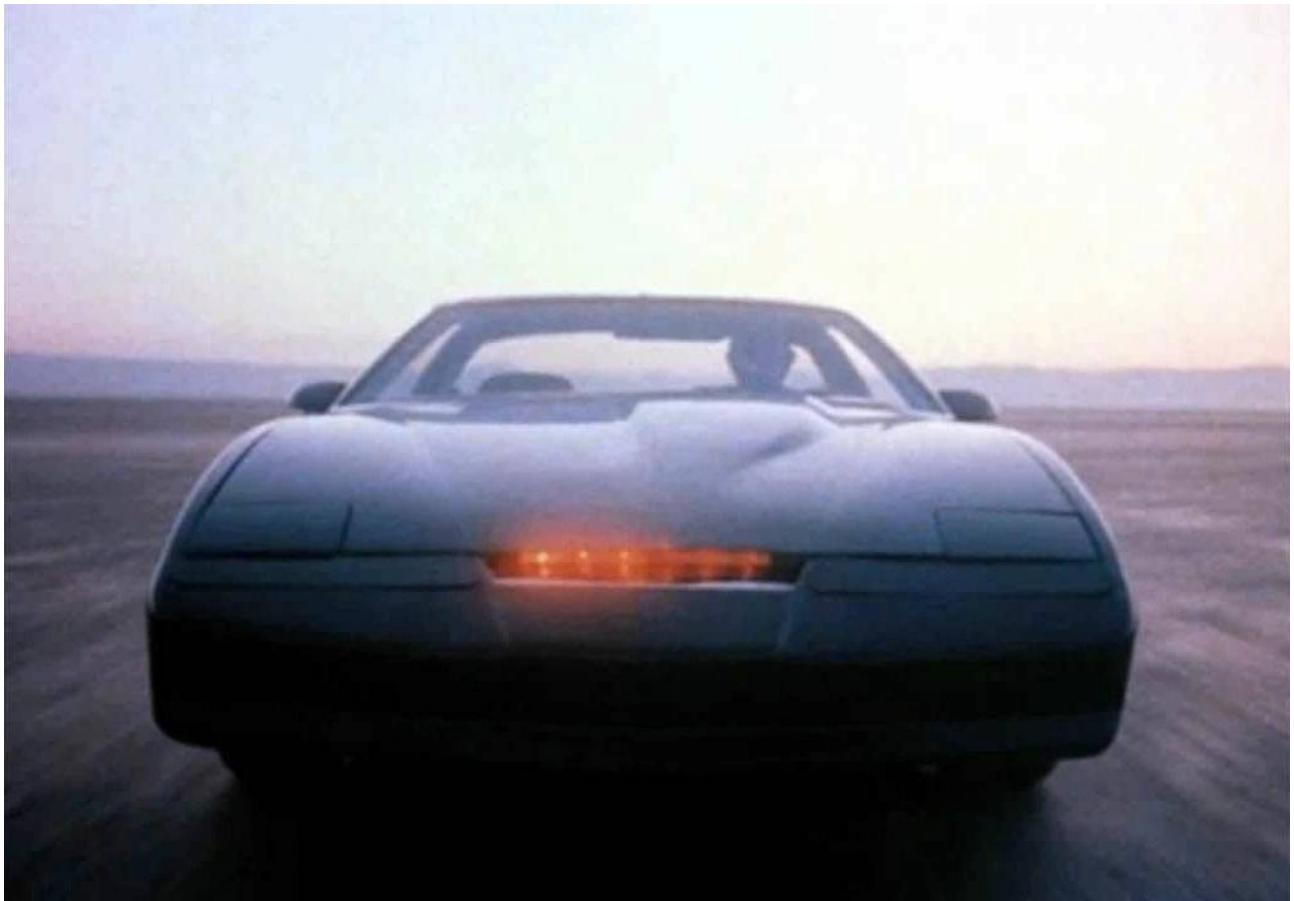


Figure 3: picture

- Author: Kolos Koblasz
- Description: The logic assertes output bits one by one, like KITT's sensor lights in Knight Rider.
- GitHub repository
- HDL project
- Extra docs
- Clock: 6000 Hz
- External hardware: Connect LEDs with ~1K-10K Ohm serial resistors to output pins and connect push button switches to Input[2] and Input[3] which drive the inputs with logic zeros when idle and with logic 1 when pressed. Rising edge on these inputs selects the next settings.

### How it works

Uses several counters, shiftregisters to create a moving light. Input[2] and Input[3] can control speed and brightness respectively. Brightness control is achieved by PWM of the output bits at 50Hz. Simulated with 6KHz clock signal.

## How to test

After reset it starts moving the switched on LED. Input[0] is clk and Input[1] is reset (1=reset on, 0=reset off). By creating rising edges on Input[2] and Input[3] the two config spaces can be discovered. Connect LEDs with ~1K-10K Ohm serial resistors to output pins and connect push button switches to Input[2] and Input[3] which drive the inputs with logic zeros when idle and with logic 1 when pressed. Rising edge on these inputs selects the next settings.

## IO

#	Input	Output
0	clock	LED 0
1	reset	LED 1
2	speed control	LED 2
3	brightness control	LED 3
4	none	LED 4
5	none	LED 5
6	none	LED 6
7	none	LED 7

## 6 : Single digit latch

- Author: Dylan Garrett
- Description: Store a single digit 0-9 and display it on a 7-segment display
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

Explain how your project works

### How to test

Explain how to test your project

### 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	dot

## 7 : 4x4 Memory

- Author: Yannick Reiß
- Description: Store 4x4 bits of memory.
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 1000 Hz
- External hardware:

### How it works

On write enable, the four data-inputs are saved to the d-flipflops selected by the address inputs. The output is always read from the current selected flipflops.

### How to test

Connect a clock, buttons for reset and write\_enable and switches for address and data like shown below. The output can be read by using 4 LEDs or any other kind of binary output device.

### IO

#	Input	Output
0	clock	data1
1	reset	data2
2	write_enable	data3
3	addr1	data4
4	addr2	none
5	data1	none
6	data2	none
7	data3	none

## 8 : KS-Signal

- Author: Yannick Reiß
- Description: Set KS-Signal based on track information.
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 10000 Hz
- External hardware: Input: Clock, 9 Buttons, Output: 3 yellow LEDs, 3 white LEDs, 1 green LED, 1 red LED, 1 orange LED

### How it works

Simply switches lamps on and off on toggle.

### How to test

Connect the clock, and the buttons as inputs. Connect the LEDs as outputs in the order shown below.

### IO

#	Input	Output
0	track1_free	Fahrt! (Green)
1	track2_free	Halt Erwarten! (Orange)
2	pre_signal	Halt! (Red)
3	none	Vorsicht! (All yellow)
4	none	pre signal indicator (white)
5	allow shunting	lower shunting indicator (white)
6	Vorsicht!	shorter breaking distance (white)
7	shorter breaking distance	none

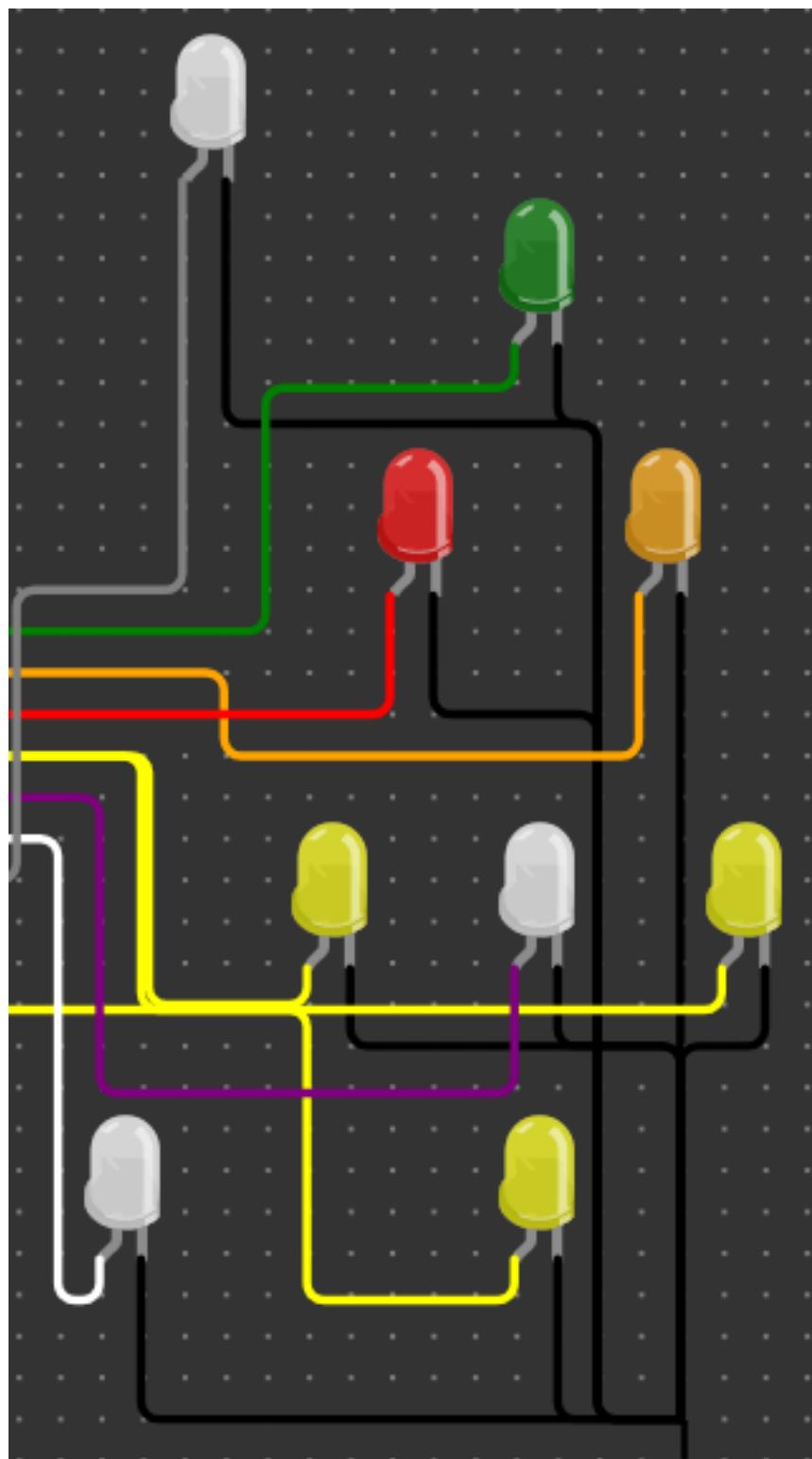


Figure 4: picture

## 9 : Hovalaag CPU

- Author: Mike Bell
- Description: Implementation of the CPU from HOVALAAG
- GitHub repository
- HDL project
- Extra docs
- Clock: 12500 Hz
- External hardware:

### How it works

HOVALAAG (Hand-Optimizing VLIW Assembly Language as a Game) is a free Zachlike game.

This is an implementation of the VLIW processor from the game. Thank you to @nothings for the fun game, making the assembler public domain, and for permission to create this hardware implementation.

The processor uses 32-bit instructions and has 12-bit I/O. The instruction and data are therefore passed in and out over 10 clocks per processor clock.

More details in the github repo.

### How to test

The assembler can be downloaded to generate programs.

The subcycle counter can be reset independently of the rest of the processor, to ensure you can get to a known state without clearing all registers.

### IO

#	Input	Output
0	Clock	Output 0
1	Reset disable (resets enabled when low)	Output 1
2	Input 0 or Reset (when high)	Output 2
3	Input 1 or Reset subcycle count (when high)	Output 3
4	Input 2 or enable ROSC (when high and reset enabled)	Output 4
5	Input 3	Output 5
6	Input 4	Output 6
7	Input 5	Output 7

## 10 : SKINNY SBOX

- Author: Niklas Fassbender
- Description: Implementation of a 4-Bit Sbox for SKINNY
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

Explain how your project works

### How to test

Explain how to test your project

### IO

#	Input	Output
0	bit_0_lsb	bit_0_lsb
1	bit_1	bit_1
2	bit_2	bit_2
3	bit_3_msb	bit_3_msb
4	none	none
5	none	none
6	none	none
7	none	none

## 11 : Stateful Lock

- Author: Tim Henkes
- Description: A little combination lock which requires three codes in the correct order to unlock
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

A little two-bit state machine decides which code is required to enter the next state. The fourth state equals to the lock being open. A wrong code in any state resets to the first state.

### How to test

To test the project, refer to its Wokwi, which is public.

### IO

#	Input	Output
0	unused	lock status
1	reset	unused
2	enter	unused
3	code digit 0	unused
4	code digit 1	unused
5	code digit 2	unused
6	code digit 3	unused
7	code digit 4	unused

## 12 : Ascon's 5-bit S-box

- Author: Fabio Campos
- Description: Ascon's 5-bit S-box
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

See Figure 4a in <https://eprint.iacr.org/2021/1574.pdf>

### How to test

See Section 3.2 and Table 4 in <https://eprint.iacr.org/2021/1574.pdf>

### IO

#	Input	Output
0	x0	x0
1	x1	x1
2	x2	x2
3	x3	x3
4	x4	x4
5	none	none
6	none	none
7	none	none

## 13 : 8bit configurable galois Ifsr

- Author: Alexander Schönborn
- Description: A 8bit configurable galois Ifsr.
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware:

### How it works

Uses 2 cycles each to set the shift and xor mask registers. It has 3 operation modes: normal shift,

### How to test

After reset, set the shift register and xor state, after that normal shift mode

### IO

#	Input	Output
0	clock	output[0] lsb normal Ifsr output
1	reset	output[1] other 7 bits to see the full state
2	mode[0] 00 normal shift mode, 01 set register mode,	output[2]
3	mode[1] 10 set mode registers, 11 unused	output[3]
4	data_in[0] is used for both filling register and xor mask state	output[4]
5	data_in[1] needs 2 cycles to fill all 8 bits	output[5]
6	data_in[2] first cycle is lower 4 bits, 2nd upper 4 bits	output[6]
7	data_in[3] see above	output[7]

## 14 : Sbox SKINNY 8 Bit

- Author: Thorsten Knoll
- Description: A circuit for the substitution of 8 bits. Made for the SKINNY cipher algorithm.
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

Explain how your project works

### How to test

Explain how to test your project

### IO

#	Input	Output
0	in_0	out_0
1	in_1	out_1
2	in_2	out_2
3	in_3	out_3
4	in_4	out_4
5	in_5	out_5
6	in_6	out_6
7	in_7	out_7

## 15 : BinaryDoorLock

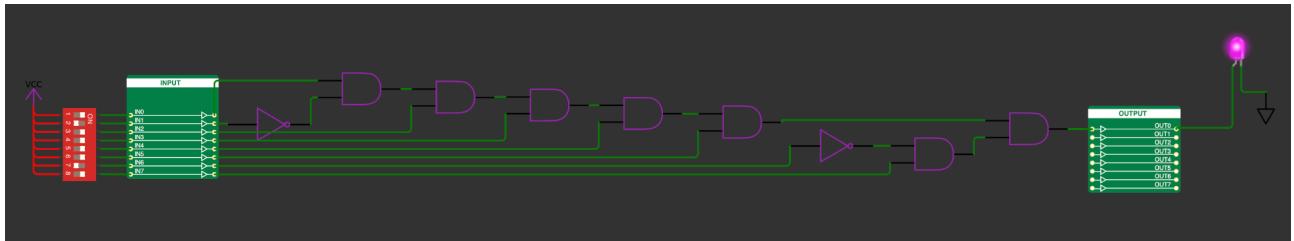


Figure 5: picture

- Author: Marcus Michaely
- Description: Input is 8-Bit and only one combination opens the door
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

The idea was to build a simple BinaryDoorLock with 8-Bits as input. With 8 Bits there are 256 possible combinations which can be switched on with the input switches. Only one combination opens the BinaryDoorLock by setting the output pin high.

### How to test

You can test this project on its **WOKWI**:

<https://wokwi.com/projects/359387860730498049>

### IO

#	Input	Output
0	Bit_0	Output
1	Bit_1	none
2	Bit_2	none
3	Bit_3	none
4	Bit_4	none
5	Bit_5	none
6	Bit_6	none
7	Bit_7	none

## 16 : bad apple

- Author: shadow1229
- Description: Plays bad apple over a Piezo Speaker connected across io\_out[1:0]. Based on <https://github.com/meriac/tt02-play-tune>
- GitHub repository
- HDL project
- Extra docs
- Clock: 12000 Hz
- External hardware: Piezo speaker connected across io\_out[1:0]

### How it works

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

### How to test

Provide 12kHz 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 (Bad Apple)

### 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

## 17 : TinyFPGA attempt for TinyTapeout3

- Author: Emilian Miron
- Description: FPGA attempt
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware:

### How it works

TODO

### How to test

After reset, the counter should increase by one every second.

### 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	slow clock output

## 18 : 4bit Adder

- Author: Carin Schreiner
- Description: This tiny tape out project takes two four bit numbers and adds them.
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

The halfadder uses simple logic gates.

### How to test

Input bits 0-3 are used for the first number and bits 4-7 for the second number. The output bits 0-3 are the resulting number an bit 4 the carry. All numbers should be in little endian format.

### IO

#	Input	Output
0	a0	r0
1	a1	r1
2	a2	r2
3	a3	r3
4	b0	carry
5	b1	none
6	b2	none
7	b3	none

## 19 : 12-bit PDP8

- Author: Paul Campnell
- Description: PDP8 core
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware:

### How it works

This is a 12-bit basic PDP8 cpu - it doesn't have the extended arithmetic unit (so no multiply or divide). Included is an assembler (mostly for test). Bus interface is a 5-clock to get 12 bits of address and 12 bits of data though 8-bit interfaces. Address is 2 beats of 6 bits each, data is 3 beats of 4 bits each, I/O cycles have an extra beat

output bits									
7	6	5	4	3	2	1	0		
1	0	A	A	A	A	A	A		address hi
1	1	A	A	A	A	A	A		address lo
0 1 1 I I 4 2 1								I/O cycle intro	
either									
0	0	0	0	-	-	-	-	-	read data high nibble
0	0	1	0	-	-	-	-	-	read data med nibble
0	1	0	0	-	-	-	-	-	read data low nibble
or									
0	0	0	1	D	D	D	D		write data high nibble
0	0	1	1	D	D	D	D		write data med nibble
0	1	0	1	D	D	D	D		write data low nibble

Input bits are ignored except during read beats, interrupts are sampled during the first address beat

### How to test

code in test-bench, assembler in asm dir

### IO

#	Input	Output
0	clock	address_0_data_mux_0
1	reset	address_1_data_mux_1
2	none	address_2_data_mux_2
3	none	address_3_data_mux_3
4	data_in_0	address_4_rw_mux
5	data_in_1	address_5_phase_lo_select_mux
6	data_in_2	phase_hi_select
7	data_in_3	address_data_select

## 20 : CTF - Catch the fish

- Author: Carin Schreiner
- Description: Catch the fish is a whac-a-mole game.
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

The project consists of \* a fibonacci lfsr of degree 6. \* a simple timer: the timer consists of two flip flops. It slows down the game and further amplifies the pseudo-randomness \* a button press detection for each button \* a simple reward detection \* Output state Handlers \* Output Selection: The last bit of the lsfr is used as the decision whether to give an output or not. The second and third to last bits are used to determine on which pin the output should be given. If both bits are zero, no output is given.

### How to test

To play the game, press the start button and make sure the clock is set to a frequency of one or two. The higher the frequency the more difficult the game gets. To score, you need to press the button while the respectivly numbered feedback output is one. You then get a reward feedback

### IO

#	Input	Output
0	clock	feedback 1
1	button 1	feedback 2
2	button 2	feedback 3
3	button 3	reward feedback
4	none	none
5	none	none
6	none	none
7	none	none

## 21 : Dot operation calculator

- Author: Yannick Reiß
- Description: Can calculate the result for 3 bit multiplication and division.
- GitHub repository
- HDL project
- Extra docs
- Clock: 0 Hz
- External hardware: 7 switches, 6 leds

### How it works

It takes input 1 to determine operator and two 3 bit inputs as operands. The result is put to output 0-5. In multiplication mode it just outputs the number binary encoded. In division mode the output pins 0-2 are the quotient and 3-5 are the remainder.

### How to test

Connect input 1-7 with switches and output 0-6 with leds.

### IO

#	Input	Output
0	clk_dummy	product/quotient
1	opcode	product/quotient
2	operand1_1	product/quotient
3	operand1_2	product/remainder
4	operand1_3	product/remainder
5	operand2_1	product/remainder
6	operand2_2	none
7	operand2_3	none

## 22 : Random number guess game

- Author: Yufei Zhen, Elaina Zodiatis, Khadijatou Dibba
- Description: Guess 6-bit random number, 3 attempts in one round.
- GitHub repository
- HDL project
- Extra docs
- Clock: manual Hz
- External hardware:

### How it works

Uses '\$random' to generate 6-bit random numbers, and input [7:2] to make a guess.

Some combinatorial logics are used to check if the guessed bit matches the random number bit on positive clock edges. If they match, the corresponding bit in the output [5:0] is set to 1; otherwise, it is set to 0.

Of course a correct guess will output 6'b111111 and reset random number to enter a new round. If the player uses all 3 guesses, the game outputs 6'b000000 and resets with a new random number as well.

### How to test

After reset, output [5:0] should zero out.

### IO

#	Input	Output
0	clock	result[0]
1	rst	result[1]
2	guess[0]	result[2]
3	guess[1]	result[3]
4	guess[2]	result[4]
5	guess[3]	result[5]
6	guess[4]	none
7	guess[5]	none

## 23 : Desperate Tapeout

- Author: Etienne de Maricourt
- Description: Customized UART string transmitter
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 300 Hz
- External hardware: MCU with UART receiver, UART-to-USB adapter

### How it works

Shameless copy of TT02 “Customizable UART String” by J. Rosenthal, since getting started the day of the submission deadline is NOT a good idea.

This circuit implements five shift registers with 21 bits: seven idle bits, one start bit, eight data bits, one stop bit, and four more idle bits. The circuit supports transmitting the string “EDM!”

### How to test

Connect an Arduino serial RX pin to the eight output pin (Output[7]). In the Arduino code, set the serial baud rate Serial.begin(); in the \*.ino file to 300.

Set the PCB clock frequency to 300Hz. Set the slide switch to the clock. Set SW7 to OFF ('Load'). Set SW8 to ON ('Output Enable'). Set SW7 to ON ('TX').

### IO

#	Input	Output
0	none	Load TX data
1	none	Output enable
2	none	none
3	none	none
4	none	none
5	none	none
6	Load TX data	none
7	Output enable	TX

## 24 : Simple multiply

- Author: Anton Maurovic
- Description: Multiply two 8-bit numbers, get a 16-bit result.
- GitHub repository
- HDL project
- Extra docs
- Clock: Any Hz
- External hardware:

### How it works

Do synchronous reset first, then on each rising clock edge...

Clock in each of two 8-bit values, one nibble at a time (high to low).

Then clock out a 16-bit value, one byte (via `result`) at a time (high to low).

### How to test

After synchronous reset, expect `result` output to be 0.

Set nibble to a value of your choice, then pulse the clock.

Repeat 3 more times.

Then pulse the clock 1 more times, each time expecting to get a byte at the output `result`.

### IO

#	Input	Output
0	clock	<code>result[0]</code>
1	reset	<code>result[1]</code>
2	none	<code>result[2]</code>
3	none	<code>result[3]</code>
4	nibble[0]	<code>result[4]</code>
5	nibble[1]	<code>result[5]</code>
6	nibble[2]	<code>result[6]</code>
7	nibble[3]	<code>result[7]</code>

## 25 : Parallel Nibble to UART

- Author: Andrew M
- Description: Loads two half-bytes into registers, then sends over UART
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware: Arduino can be used for testing UART RX

### How it works

4 bits should be loaded into registers through the data pins twice - once for the LSB of a byte, once for the MSB of a byte (specified using the mode pins). Once ready, mode can be set to transmit to send the loaded byte over UART.

### How to test

When mode[1,2]=01, data is clocked in as the least-significant 4-bits for a UART transmissions. When mode[1,2]=10, the same occurs for the most-significant 4-bits. When mode[1,2]=11, the complete data should get sent over UART.

### IO

#	Input	Output
0	clock	uart_tx
1	reset	none
2	data1	none
3	data2	none
4	data3	none
5	data4	none
6	mode1	none
7	mode2	none

## 26 : tiny logic analyzer

- Author: yubex
- Description: The design samples one data input and shows the current state and edge events using the 7 segment display.
- GitHub repository
- HDL project
- Extra docs
- Clock: 12500 Hz
- External hardware: no external HW required

### How it works

The samples of the data input pin are shifted into a shift register first. After that the 2 most significant bits of the shift register are used to detect the current signal state (high, low) and the edge events (rising edge, falling edge). There are 4 different states/events displayed on the 7 segment display.

state/event	segments on
high	a
low	d
rising edge	e and f
falling edge	b and c

The edge events duration is extended by counters, so the events can actually be seen by the human eye.

### How to test

You can test by using the dip switch connected to io\_in[2] as data input. I plan to connect a wire to the pin on the PCB by PMOD connector or custom soldering to be able to have a “measurement probe”. ;-)

### IO

#	Input	Output
0	clock	segment a
1	reset	segment b
2	data input	segment c
3	none	segment d

#	Input	Output
4	none	segment e
5	none	segment f
6	none	segment g
7	none	dot

## 27 : XOR Stream Cipher

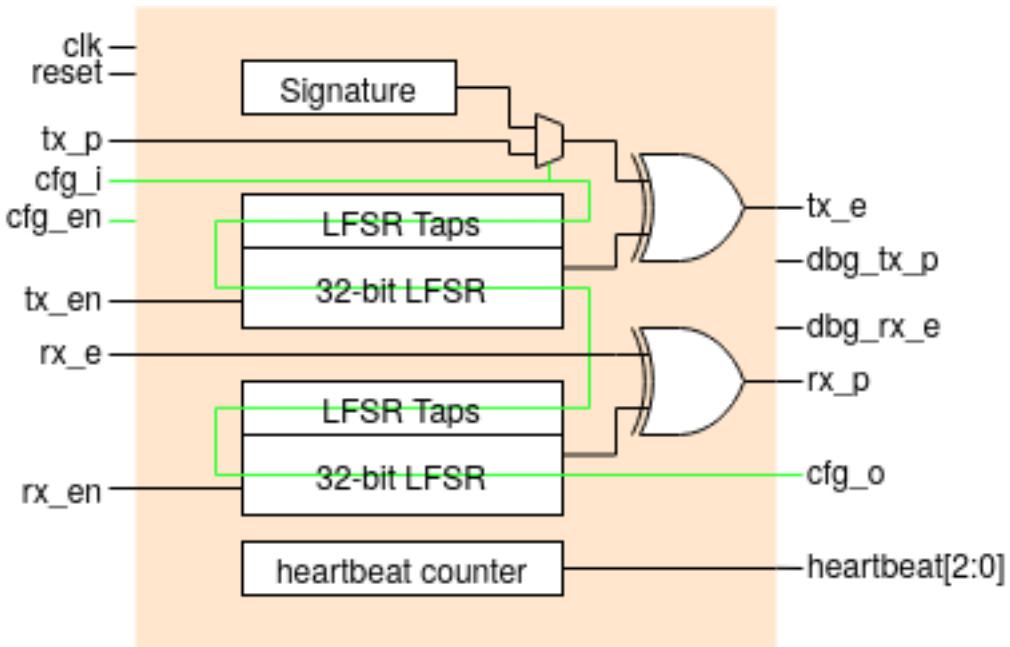


Figure 6: picture

- Author: Luke Vassallo
- Description: An two channel XOR stream cipher with fully programmable 32-bit galois LFSRs.
- GitHub repository
- HDL project
- Extra docs
- Clock: 12500 Hz
- External hardware: General puporse FPGA with PMOD connector for detailed testing./

### How it works

An XOR (exclusive or) cipher is a type of encryption that uses a bitwise exclusive or operation to combine a plaintext message with a secret key. This process generates a ciphertext that can only be decrypted by someone with the same secret key. XOR ciphers are commonly used in computer security and are popular due to their simplicity and efficiency. They can be implemented in hardware using XOR cipher chips, which typically use a Galois LFSR (linear feedback shift register) to generate a key stream that is XORed with the plaintext to produce the ciphertext. XOR ciphers are considered relatively secure as long as the secret key is kept secret and is not easily guessable.

The system uses a Galois Linear Feedback Shift Register (LFSR) to produce a key stream that is combined with the incoming bitstream through an XORing process to

create the cipher stream that appears at the output. A concurrent channel is additionally provided such that a transmission and reception bitstream can be encrypted and decrypted simultaneously. When the system is reset, a default configuration is applied that will encrypt the bitsream on tx\_p and decrypt the bitsream on rx\_e when te\_en and rx\_en are respectively driven high.

To configure the chip (optional), a 130-bit configuration vector is serial shifted through the pin cfg\_i while the current configuration is simultaneously outputted on pin cfg\_o. This configuration method functions as a lengthy shift register with its input connected to cfg\_i and its output connected to cfg\_o. It only operates synchronously to clk when cfg\_en is asserted. The configuration vector consists of bits 95->64, 31->0 for the tx/rx LFSR state, 127->96, 63->32 bits for the tx/rx LFSR taps. Power-on-reset state has the taps configured for PRBS-31 and LFSR state holding 0x55. Bit 128 is used to internally route the output bitstreams through an XOR that returns the original bitstreams provided at the input albeit placed on debug output pins (disabled by default). Bit 129 selects between the internal or externally provided plaintext generator (default).

## How to test

Reset the module and optionally configure the LFSR taps, state and other options using the serial shift configuration register. When the enable pin (tx\_en/rx\_en) is asserted the corresponding channel is activate and the plain/cipher text bitstream will be encrypted/decrypted. Detailed simulation, FPGA and ASIC design examples are available on GitHub (<https://github.com/LukeVassallo/tt03-xor-cipher>). The GitHub repository is mirrored on my private GitLab instance (<https://gitlab.lukevassallo.com/luke/tt03-xor-cipher>) that incorporates automated builds and pre-built bitstreams.

## IO

---

#	Input	Output
0	clk (12.5KHz system clock)	tx_e ( Encrypted bitstream for transmission)
1	rst (Active high synchronous reset)	dbg_tx_p (Decrypted transmit bitstream, pin disabled by default)
2	tx_p (Plaintext bitstream for transmission)	dbg_rx_e (Encrypted receive bitstream, pin disabled by default)
3	cfg_i (Configuration input to the 130-bit serial shift register)	rx_p (Decrypted bitstream for reception)
4	cfg_en (Active high configuration enable)	cfg_o (Configuration output from the 130-bit shift register)

#	Input	Output
5	tx_en (Transmit channel enable)	heartbeat[7] (bit from heartbeat counter)
6	rx_e (Encrypted bitstream for reception)	heartbeat[8] (bit from heartbeat counter)
7	rx_en (Receive channel enable)	heartbeat[9] (bit from heartbeat counter)

## 28 : LED Panel Driver



Figure 7: picture

- Author: Tom Keddie
- Description: Drives a 16x16 P10 LED panel
- GitHub repository
- HDL project
- Extra docs
- Clock: 6000 Hz
- External hardware: led panel, level converter to 5V logic

## How it works

- The circuit updates half of a P10 16x16 LED display module
- It initially displays the string TT03
- It provides a 600baud uart input to
  - paint pixels
  - erase pixels
  - clear the display
  - change the displayed colour
- Functionality is limited by resource availability and clock rate
  - single colour at once
  - no double buffer, updates may have artifacts
  - refresh rate is low, some flicker is observed ( $16*8=128$  pixels at 6kHz is ~46Hz, plus overhead)
- Mode pin to allow for 2 different clocking patterns

## How to test

- Connect the display module as per the outputs
- Connect the uart
- Power on and see the TT03 text
- If the display is swapped by quadrant change the mode pin
- Use the script(s) in the software directory to control the display

## IO

#	Input	Output
0	clock	red
1	reset	blue
2	uart	b
3	mode	blank
4	none	green
5	none	a
6	none	clk
7	none	latch

## 29 : 6-bit FIFO

- Author: Mike Bell
- Description: Implementation of a FIFO
- GitHub repository
- HDL project
- Extra docs
- Clock: 50000 Hz
- External hardware:

### How it works

The design implements a 52 entry 6-bit FIFO. The oldest 4 entries are accessible by setting the peek address. The first 48 entries in the FIFO are implemented as a chain of latches. This allows for high data density in the limited area. The last 4 entries in the FIFO are implemented by a ring of flip-flops. This allows random access to the last four entries.

Because of the way the chain of latches works, when entries are popped from the FIFO it takes time for data in the latch part of the FIFO to move down the chain. If the FIFO is fairly empty this shouldn't be noticeable, but when the FIFO is quite full this can cause the FIFO to refuse writes even though it is not full.

A ready output indicates whether the latch chain is ready for more data. If the FIFO is completely full and one entry is popped then it takes 48 cycles after the pop for the chain to be ready again. Therefore, if ready stays low when the FIFO is clocked 49 or more times without any data being popped then the FIFO is full.

There are minimal delays on the read side - if any data is in the FIFO then it can always be read, this works because empty latch entries can pass the data through them without needing to be clocked. The only exception to this is that due to input buffering newly written entries take 2 cycles to appear on the output.

### How to test

New entries, taken from inputs 2-7, are written to the FIFO on a rising clock edge when input 1 is high. Data writes are ignored when the Ready output is low.

Because of limited inputs, the write enable is used to determine the mode of inputs 2-5. When not writing, these control reset (active low), pop (active high) and the peek address.

The peek address controls whether the oldest to 4th oldest entry in the FIFO is presented on the data outputs.

Reading the oldest entry when the FIFO is empty always reads 0. However, peeking at previous entries when the FIFO is empty or has fewer occupied entries reads stale data - the values should not be relied upon.

The oldest entry is popped from the FIFO by setting pop. It is valid to set the peek address to a non-zero value when popping, but it is always the oldest entry that is popped, which is not the entry being read if peek address is non-zero. The peek address is considered prior to the pop.

When writing, it is always the last entry in the FIFO that is read (peek address is considered to be zero). The new value written when the FIFO is empty is not presented on the outputs until the following cycle.

## IO

#	Input	Output
0	Clock	Ready
1	Mode (Write enable)	Empty_n
2	Reset_n / Data 0	Data 0
3	Pop / Data 1	Data 1
4	Peek A0 / Data 2	Data 2
5	Peek A1 / Data 3	Data 3
6	Unused / Data 4	Data 4
7	Unused / Data 5	Data 5

## 30 : ezm\_cpu

- Author: guianmonezm#4787
- Description: basic 8bit CPU
- GitHub repository
- HDL project
- Extra docs
- Clock: 6000 Hz
- External hardware: FPGA or ram and demultiplexer

### How it works

you connect an external ram and a decoder to it or an FPGA with a rom programed.

### How to test

connect an external ram and a decoder to it.

### IO

#	Input	Output
0	clock	pc[0]/c
1	reset	pc[1]/c
2	in1	pc[2]/c
3	in2	pc[3]/c
4	in3	pc[4]/c
5	in4	pc[5]/c
6	in5	pc[6]/c
7	in6	pc[7]/c

## 31 : 31b-PrimeDetector

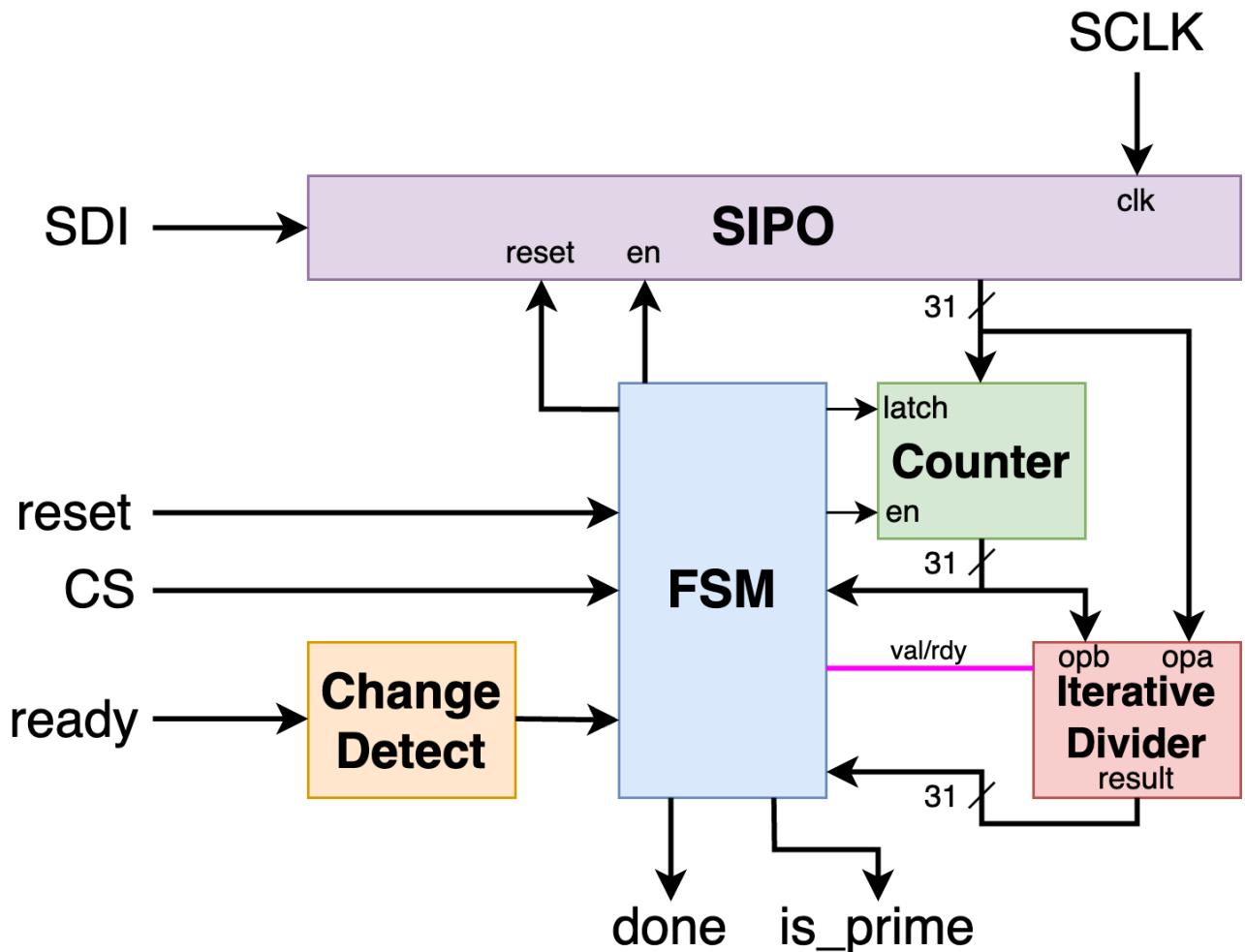


Figure 8: picture

- Author: Aidan McNay
- Description: Detects whether a 31-bit number is prime or not
- GitHub repository
- HDL project
- Extra docs
- Clock: 50000 (though could probably be faster) Hz
- External hardware: External Clock, 5 buttons for inputs, 2 LEDs for outputs

### How it works

The 31-bit Prime Detector takes in a 31-bit number (shifted in serially). Once the number is obtained, the FSM control logic takes over. It attempts to divide the value by all numbers less than it; if it finds one that divides evenly, the logic stops and declares the number not prime. If it doesn't divide evenly by any of these, the number is declared prime.

Due to space constraints, the design uses an iterative divider and FSM logic to minimize space usage. Further information can be taken from the README.md on the GitHub page

## How to test

This design requires an external clock. Before testing, the design should be reset with the appropriate pin (active high), which resets the stored value to 0. To shift in a value, use the SPI-like interface; when the CS line is enabled (active low), on rising edges of SCLK, the data present at SDI is shifted in. Data is shifted into the LSB, and progressively shifted to more significant bits as new data is received (with the data at MSB being shifted out and disregarded).

Once you have the desired number stored, start the calculations by enabling the ready pin (active high). Note that the stored value cannot change while calculations are ongoing.

Once the calculations are finished, the done pin will be driven high. The result will be shown on the is\_prime pin; a value of 1 indicates that the value inputted is prime.

## IO

#	Input	Output
0	clock	done
1	reset	is_prime
2	SDI	waiting
3	SCLK	GND
4	CS	GND
5	ready	GND
6	NC	GND
7	NC	GND

## 32 : 4-bit ALU

- Author: ReJ aka Renaldas Zioma
- Description: Digital design for a 4-bit ALU supporting 8 different operations and built-in 4-bit accumulator register
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware: push-button, debouncer, DIP-switch, 5 LEDs

### How it works

Each clock cycle ALU performs one of the 8 possible operations and stores result in the 4-bit accumulator register.

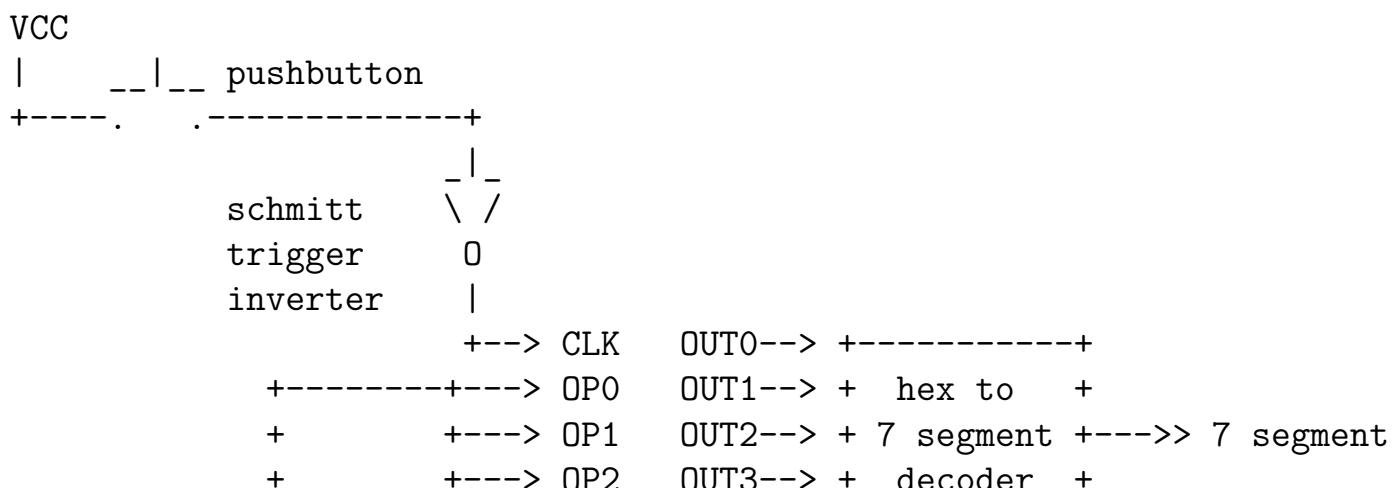
accumulator [4 bit] = accumulator [4 bit] (operation) operand [4 bit]

Supported operations: lda imm :: imm -> accumulator  
neg imm :: 0x0F - imm -> accumulator  
shr :: accumulator / 2 -> accumulator  
sub imm :: accumulator - imm -> accumulator  
and imm :: accumulator & imm -> accumulator  
xor imm :: accumulator ^ imm -> accumulator  
or imm :: accumulator | imm -> accumulator  
add imm :: accumulator add imm -> accumulator  
+ imm -> accumulator

Matrix mapping of operation opcode to internal control signals muxA muxB muxC  
AtoX negX setC outC invC 000 lda -- 1 0 0 - 0 - 001 neg -- 1 0 1 - 0 - 010 shr -- 1  
1 0 - 0 - 011 sub 1 1 0 0 1 1 1 1 100 and 0 0 0 0 0 - 0 - 101 xor 0 1 0 0 0 - 0 - 110  
or 1 0 0 0 0 - 0 - 111 add 1 1 0 0 0 0 1 0

### How to test

The following diagram shows a simple test setup that can be used to test ALU



+ DIP	---->	IMM0	-----+
+ switch	---->	IMM1	
+	---->	IMM2	
+	---->	IMM3	CARRY--> LED
-----+--			

To reset ALU set all input pins to 0 which corresponds to lda 0 operation loading Accumulator register with 0.

## IO

#	Input	Output
0	clock	accumulator value 0th bit
1	opcode 0th bit	accumulator value 1st bit
2	opcode 1st bit	accumulator value 2nd bit
3	opcode 2nd bit	accumulator value 3rd bit
4	operand 0th bit	{‘unused (TODO’: ‘negative flag’)’}
5	operand 1st bit	{‘unused (TODO’: ‘overflow flag’)’}
6	operand 2nd bit	{‘unused (TODO’: ‘zero flag’)’}
7	operand 3rd bit	carry flag

### 33 : Pulse-Density Modulators

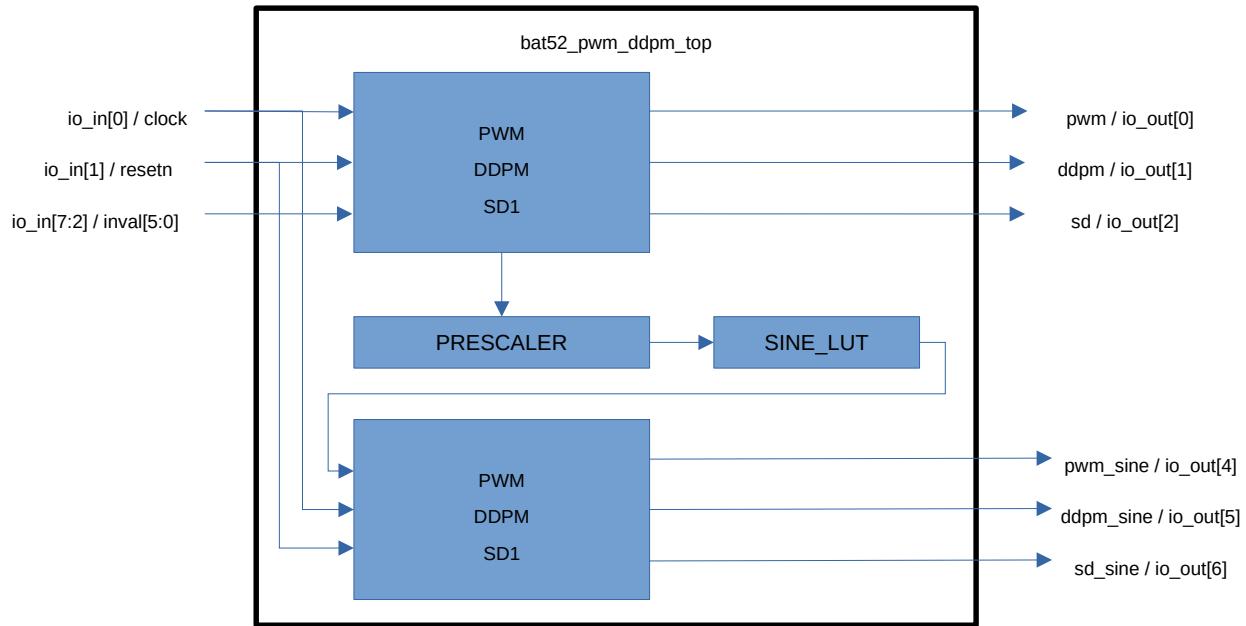


Figure 9: picture

- Author: Marco Merlin
- Description: An implementation of a DDPM, PWM and Sigma-Delta Pulse-Density Modulators with python libraries myHDL and PuEDA.
- GitHub repository
- HDL project
- Extra docs
- Clock: 12500 Hz
- External hardware: clock source

#### How it works

This project implements three different architectures of Pulse Density Modulators (PDM), to compare performances and implementation complexity of the different schemes.

PDM modulators are of particular interest because exploiting oversampling, they allow to implement Digital-to-Analog-Conversion (DAC) schemes with an equivalent resolu-

tion of multiple bits, based on a single-bit digital output, by means of a straightforward analogue low-pass filter.

The PDM modulators implemented in this project are the following: 1) Pulse Width Modulation (PWM) 2) Dyadic Digital Pulse Modulation (DDPM) 3) Sigma-Delta (SD)

PWM is arguably the simplest and possibly most widespread PDM technology that is praised for its low complexity.

DDPM [1][2][3] is a type of digital modulation technique in which the pulse width are quantized in a dyadic manner, meaning that they are quantized in powers of two, which allows for efficient implementation using binary arithmetic. As a consequence, DDPM modulators are relatively inexpensive to deploy, with a complexity comparable to that of widespread Pulse-Width Modulation (PWM) modulators.

SD modulators are perhaps the best-performing PDM technology, that is particularly well suited for higher resolution data conversion, and typically find use in audio DACs and fractional PLLs. In spite their good performances, SD modulators are Infinite-Impulse-Response (IIR) closed-loop systems which behavior depends on the input signal, resulting in a number of concerns (most notably stability), that typically require careful modeling of the systems and condition under which they will be required to operate.

This design is separated into two sections: 1) 6 bits resolution instance of PWM, DDPM and SD fed by a static DC value from the input pins 2) 8 bits resolution instance of PWM, DDPM and SD fed sine look-up-table (LUT) that allows to evaluate the spectral content of the modulated signals.

## How to test

Common: The circuit needs to be fed with a clock on pin `io_in[0]`. Reset signal needs to be released by raising to 1 pin `io_in[1]`.

Static DC:

The input `inval` of the first set of DC modulators is fed through pins `io_in[7:2]`. The low-passed dc component of the outputs on pins `io_out[0]` (PWM), `io_out[1]`, and `io_out[2]` is proportional to the decimal value of the input `inval`.

Sinusoidal output: The low-passed outputs on pins `io_out[4]` (PWM), `io_out[5]`, and `io_out[6]` is a sinusoidal wave. When clocking the chip with a clock frequency of 12.5kHz, the frequency of the sine is of 0.76z ( $f_{sin} = f_{clock} / 2^{14}$ ), so that it should be visible at naked eye. The different designs should achieve an ENOB of 8 bits in the band 0-40Hz, with different level of out-of-band emission between each other.

## IO

#	Input	Output
0	clock	pwm / segment a
1	resetn	ddpm / segment b
2	inval[0]	sd / segment c
3	inval[1]	0'b1 / segment d
4	inval[2]	pwm_sine / segment e
5	inval[3]	ddpm_sine / segment f
6	inval[4]	sd_sine / segment g
7	inval[5]	0'b1 / dot

## 34 : CRC Decelerator

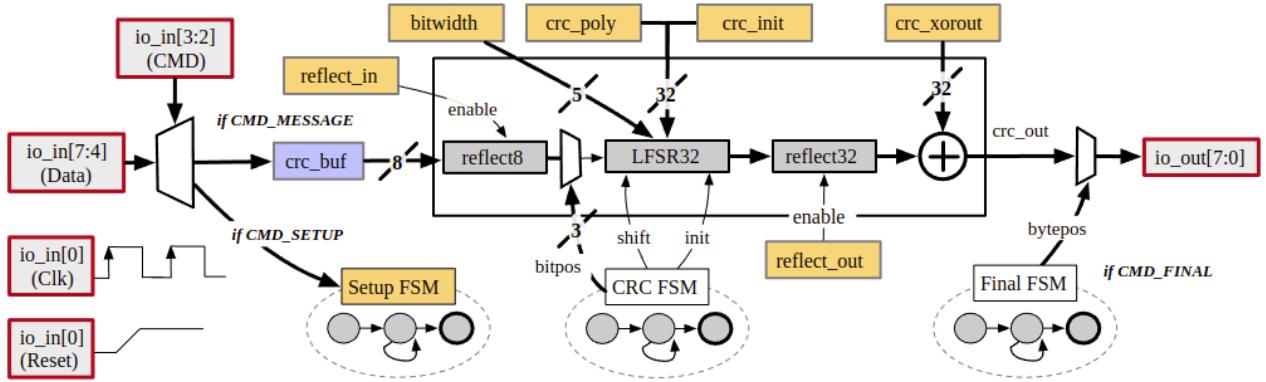


Figure 10: picture

- Author: Grant Hernandez (@grant-h)
- Description: A reconfigurable CRC engine
- GitHub repository
- HDL project
- Extra docs
- Clock: any Hz
- External hardware: none

### How it works

*“The world’s slowest CRC!”*

The Cyclic Redundancy Check Decelerator is a reconfigurable CRC block that can be programmed to calculate any CRC up to 32-bits with arbitrary length, streamed input data. Since TinyTapeout 3 (TT03) I/O speeds are low, its unlikely that this CRC engine will be faster than the CPU/microcontroller streaming in data, hence “decelerator”.

This TT03 project is the follow up to my earlier, full-custom VLSI version of a CRC-32 datapath, built in Cadence, and fabricated using the MOSIS service while attending my university. Read more about my original the CRC-32 design and check out the die shots.

[3D-View]

### I/O Interface

The CRC IP has 8 input pins. Two are for the `clk` (`io_in[1]`) and reset (`rst / io_in[1]`). Then there is a two-bit command input, `cmd` (`io_in[3:2]`), and the remaining 4-bits are for data input, `data_in` (`io_in[7:4]`). The design uses

only the positive edge of the clock and has a synchronous, active high, reset line. The data input is limited to passing a single nibble at a time. How this data is used depends on the current command.

Here is a table showing the I/O pins:

#	Input	Output
0	clk	data_out[0]
1	rst	data_out[1]
2	cmd[0]	data_out[2]
3	cmd[1]	data_out[3]
4	data_in[0]	data_out[4]
5	data_in[1]	data_out[5]
6	data_in[2]	data_out[6]
7	data_in[3]	data_out[7]

There are 4 supported commands:

#	Name	Description
2'b00	CMD_RESET	Restarts the CRC calculations using the parameters from the last SETUP bitstream
2'b01	CMD_SETUP	Streams in a CRC-bitwidth dependent bitstream to configure the CRC parameters
2'b10	CMD_MESSAGE	Stream in a message to CRC 4-bits at a time. First cycle is lower 4-bits. Second cycle is upper 4-bits. Wait 8 cycles for byte to be processed. Repeat.
2'b11	CMD_FINAL	Continually stream out the final CRC value 8-bits at a time in a loop until deasserted

The overall flow is, a SETUP bitstream containing the CRC bitwidth, reflect in/out parameters, CRC poly, initial value, and XOR out is streamed in. Then the MESSAGE is streamed in 4-bits at a time until the message is complete. Finally, the FINAL is asserted and the final CRC value is streamed out on the output pins. To restart another CRC fresh, send RESET or resume adding additional data to the existing CRC by using MESSAGE.

## CRC Setup Bitstream

The most complex portion of the using this CRC IP is streaming in the configuration bitstream. This bitstream can be from 20-bits in a CRC-4 or less case and up to 104 bits in a CRC-32 case. The bitstream format is roughly: [config\_lo - 4 bits] [config\_hi - 4 bits] [poly - 4N bits] [init - 4N bits] [xor - 4N-bits]. Each nibble is packed with the MSB on data\_in[3]. config\_lo is 4-bits and defines the first 4-bits of the CRC bitwidth bitwidth[3:0]. config\_hi is also 4-bits and it contains the top-2 bits of the bitwidth and the reflect\_out and reflect\_in parameters of the CRC: [bitwidth[5]] [bitwidth[4]] [reflect\_out] [reflect\_in].

This initial configuration is always 8-bits, but the remaining bitstream is variable length dependent on the bitwidth. Following the initial parameters is the poly, init value, and XOR out values. These are equal length and are streamed one nibble at a time from least-significant nibble to most (least significant being bits[3:0]).

This process is best demonstrated using a timing diagram. For the example we'll be using the CRC-16/USB parameters which are width=16 poly=0x8005 init=0xffff refin=true refout=true xorout=0xffff check=0xb4c8:

### CRC-16/USB Setup Bitstream

The fully packed bitstream in hex is f35008fffffffff. The first nibble corresponds to bitwidth[3:0] = 0xf. This represents the value 15, which is one less than the bitwidth. This is because the CRC treats a bitwidth of zero as a CRC-1. You must pass in a bitwidth one less than the desired bitwidth to account for this. The second nibble unpacks as 4'b0011 in binary which makes the top 2-bits of the bitwidth be zero and sets reflect in and out to be true. The remaining nibbles are the configuration parameters least-significant nibble to most. Note that setup\_fsm is internal to the design.

## CRC Message Streaming

Once the CRC's parameters have been initialized, you can switch to CMD\_MESSAGE to stream in data to be CRC'd one nibble at a time. CRCs typically use the check message of 123456789 which is 0x31 0x32 0x33 0x34 0x35 0x36 0x37 0x38 0x39 in individual bytes. Here is a timing diagram showing this message being streamed in and the CRC's result register:

### CRC-16/USB 123456789 Check Message

The first break skips the 8 clock cycles for 0x31. The second break skips the middle 6 bytes and the final break skips the last shifting clock cycles. Following CMD\_MESSAGE you would signal CMD\_FINAL on cmd to have the CRC value streamed out on the data

output pins. Note that `crc_state` is internal to the design.

The CRC decelerator is a reconfigurable CRC block that can be programmed to calculate different CRC values up to 64-bits with arbitrary length streamed input data. Since clock speeds are low, its unlikely that this CRC engine will be faster than the CPU streaming in data, hence “decelerator”.

To begin, a SETUP bitstream containing the bitwidth, reflect in/out, CRC poly, init, and XOR out is sent. Then the MESSAGE is streamed in 4-bits at a time until the message is complete. Finally, the FINAL is signaled, leading the final CRC value to be streamed out. To calculate another CRC fresh, send RESET.

## How to test

See documentation.

## IO

#	Input	Output
0	<code>clk</code>	<code>data_out[0]</code>
1	<code>rst</code>	<code>data_out[1]</code>
2	<code>cmd[0]</code>	<code>data_out[2]</code>
3	<code>cmd[1]</code>	<code>data_out[3]</code>
4	<code>data_in[0]</code>	<code>data_out[4]</code>
5	<code>data_in[1]</code>	<code>data_out[5]</code>
6	<code>data_in[2]</code>	<code>data_out[6]</code>
7	<code>data_in[3]</code>	<code>data_out[7]</code>

## 35 : Simple clock

- Author: Søren Poulsen
- Description: Shows time of day.
- GitHub repository
- HDL project
- Extra docs
- Clock: 32 Hz
- External hardware:

### How it works

Takes a clock as input and output data to shift registers connected to four 7 segments.

### How to test

After reset, the 7 segments should show the clock.

### IO

#	Input	Output
0	clock	srclk
1	reset	rclk
2	minutes	ser
3	hours	none
4	none	none
5	none	none
6	none	none
7	none	none

## 36 : Binary to DEC and HEX

- Author: Norberto Hernandez-Como
- Description: Converts a 4 digit binary number to decimal or to hexadecimal using a 7-segment display
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

Converts a 4 digit binary number to decimal or to hexadecimal using a 7-segment display

### How to test

Select binary to decimal with input 6 (named Dec, the dot is displayed) or binary to hexadecimal with input 7 (named Hex). Input a 4 digit binary number using inputs B3, B2, B1, B0 and see in the 7-segment display the converted number in decimal or hexadecimal. All forbidden combinations are not displayed.

### IO

#	Input	Output
0	B0	segment a
1	B1	segment b
2	B2	segment c
3	B3	segment d
4	none	segment e
5	none	segment f
6	Dec	segment g
7	Hex	dot

## 37 : Simon Says

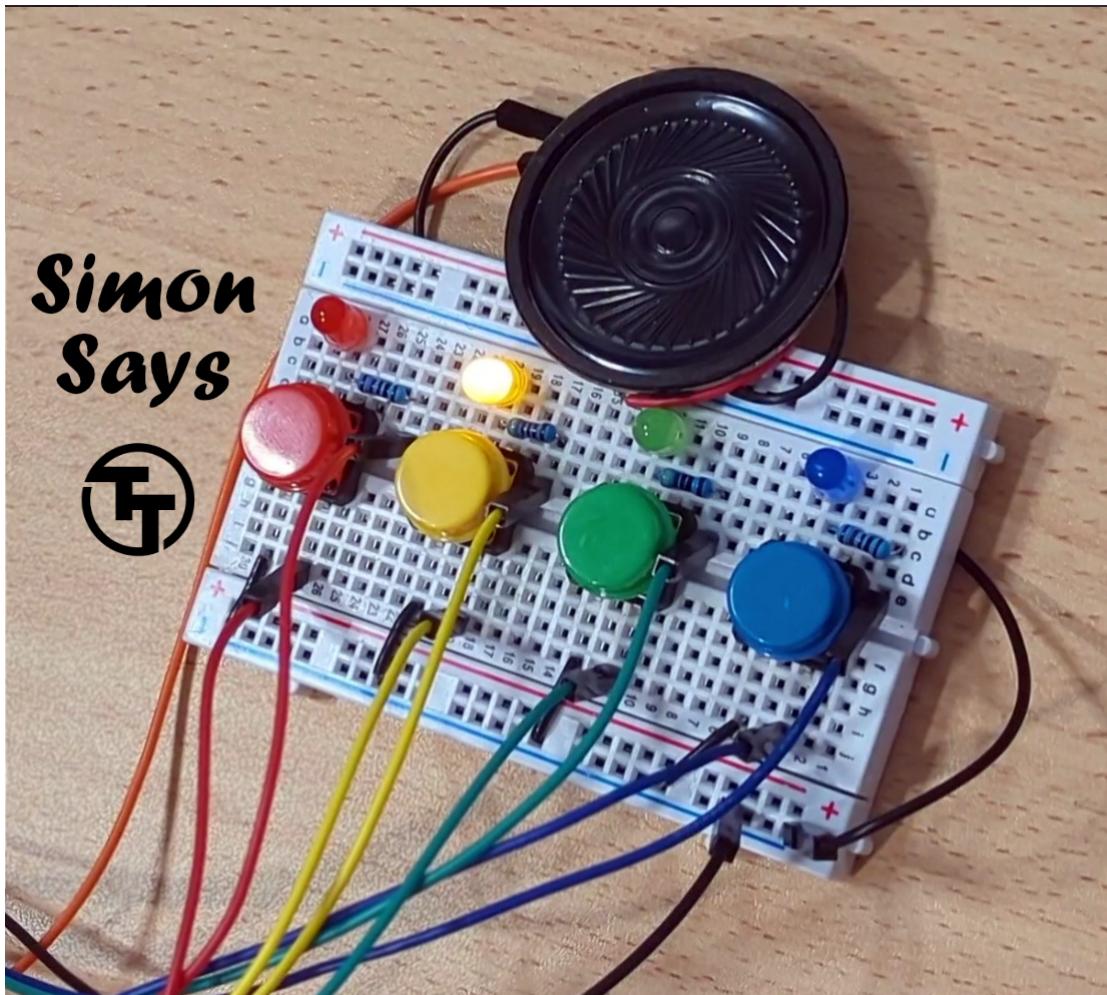


Figure 11: picture

- Author: Uri Shaked
- Description: A simple memory game
- GitHub repository
- HDL project
- Extra docs
- Clock: 4000 Hz
- External hardware: Four push buttons (with pull-down resistors), four LEDs, and optionally a speaker/buzzer

### How it works

Simon says is a simple electronic memory game: the user has to repeat a growing sequence of colors. The sequence is displayed by lighting up the LEDs. Each color also has a corresponding tone.

In each turn, the game will play the sequence, and then wait for the user to repeat the

sequence by pressing the buttons according to the color sequence. If the user repeated the sequence correctly, the game will play a “leveling-up” sound, add a new color at the end of the sequence, and move to the next turn.

The game continues until the user has made a mistake. Then a game over sound is played, and the game restarts.

The game supports four clock speeds, which can be selected using the clk3 and clk1 inputs:

clk3	clk1	Clock Speed
0	0	4KHz
0	1	6KHz
1	0	12KHz
1	1	14KHz

Setting the clock speed affects the speed of the game and the tone generator.

Check out the online simulation at <https://wokwi.com/projects/352319274216569857> (including wiring diagram).

## How to test

You need four buttons, four LEDs, resistors, and optionally a speaker/buzzer. Ideally, you want to use 4 different colors for the buttons/LEDs (red, green, blue, yellow). 1. Connect the buttons to pins btn1, btn2, btn3, and btn4, and also connect each button to a pull down resistor. 2. Connect the LEDs to pins led1, led2, led3, and led4, matching the colors of the buttons (so led1 and btn1 have the same color, etc.) 3. Connect the speaker to the speaker pin. 4. Select the clock frequency (using the clk3 and clk1 inputs). 5. Reset the game, and then press any button to start it. Enjoy!

## IO

#	Input	Output
0	clock	led1
1	reset	led2
2	btn1	led3
3	btn2	led4
4	btn3	speaker
5	btn4	none

#	Input	Output
6	clk1	none
7	clk3	none

## 38 : Shift Register Ram

- Author: Dakotath
- Description: The device holds bits in shift registers to remember crap
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 5000 Hz
- External hardware:

### How it works

It's ram. It stores stuff. This device can hold 8 Bytes of stuff. Yes, I know, It's not a lot.

### How to test

Explain how to test your project

### IO

#	Input	Output
0	Clock Data	D0
1	Data Input	D1
2	Clock Address	D2
3	Address Input	D3
4	Output Enable	D4
5	None	D5
6	None	D6
7	None	D7

## 39 : tinysat

- Author: Emmanouel Matigakis
- Description: Tiny sat solver.
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware:

### How it works

uses a counter to iterate over all possible assignments to a simple sat problem.

### How to test

I'm thinking of using an RP2040 to basically implement the same tests as in test.py in the physical thing.

### IO

#	Input	Output
0	clock	x[0]
1	reset	x[1]
2	run	x[2]
3	load	x[3]
4	data[0]	sol
5	data[1]	done
6	data[2]	0
7	data[3]	0

## 40 : POV display

- Author: Balint Kovacs
- Description: Small POV display
- GitHub repository
- HDL project
- Extra docs
- Clock: 12500 Hz
- External hardware: A line of LEDs, an MCU to load the image data, and some means of timing

### How it works

The image is stored in a 8x32 loop, this can be updated over the SPI bus. Additionally, there is a clock generator that generates 48-128 pulses for every cycle of the hall effect sensor. Finally, a controller passes 32 of those pulses each cycle to the loop memory, and also handles blanking.

Relevant registers are reset by transitions of the hall effect sensor and the CS lines.

### How to test

- Supply a regular clock, up to f\_clk/1024, on hall\_in.
- Load an image in a single, 32 byte SPI transaction.
- Move the device quickly

### IO

#	Input	Output
0	clock	led0
1	cs_n	led1
2	sck	led2
3	mosi	led3
4	hall_in	led4
5	hall_invert	led5
6	divider[0]	led6
7	divider[1]	led7

# 41 : Toy CPU

- Author: jordan336
- Description: Toy CPU is an 8 bit toy CPU for the Tiny Tapeout project.
- GitHub repository
- HDL project
- Extra docs
- Clock: 10000 Hz
- External hardware:

## How it works

### ISA

Opcode	Mnemonic	Name	Description
000	DISP	Display	$\text{data\_out} = \text{reg}[\text{src\_a}]$
001	ADD	Add	$\text{reg}[\text{dest}] = \text{reg}[\text{src\_a}] + \text{reg}[\text{src\_b}]$
010	ADD_I	Add (immediate)	$\text{reg}[\text{dest}] = \text{reg}[\text{src\_a}] + \text{imm}$
011	AND	And	$\text{reg}[\text{dest}] = \text{reg}[\text{src\_a}] \& \text{reg}[\text{src\_b}]$
100	AND_I	And (immediate)	$\text{reg}[\text{dest}] = \text{reg}[\text{src\_a}] \& \text{imm}$
101	OR	Or	$\text{reg}[\text{dest}] = \text{reg}[\text{src\_a}]   \text{reg}[\text{src\_b}]$
110	OR_I	Or (immediate)	$\text{reg}[\text{dest}] = \text{reg}[\text{src\_a}]   \text{imm}$
111	STRE	Store	$\text{reg}[\text{dest}] = \text{imm}$

### Instruction format

Instructions are passed using the upper 6 bits of the inputs. Depending on the opcode, the full instruction with opcode and all arguments is passed using one, two, or three 6 bit instruction words.

Word	Input [7:5]	Input [4:2]	Input [1]	Input [0]
0	opcode[2:0]	src_a[2:0]	rst	clk
1	dest[2:0]	src_b[2:0] or imm[7:5]	rst	clk
2	{X,imm[4:3]}	imm[2:0]	rst	clk

Opcode	Mnemonic	Number of Instruction Words
000	DISP	1
001	ADD	2
010	ADD_I	3
011	AND	2
100	AND_I	3

Opcode	Mnemonic	Number of Instruction Words
101	OR	2
110	OR_I	3
111	STRE	3

## Start input

After exiting reset, the Toy CPU looks for a start input to begin processing the instruction stream. The start input is all 1s in the 6 bit instruction word (0x3F). After sampling the start sequence, the CPU will interpret the next 6 bit instruction word as the first word in the instruction stream.

## How to test

Drive a clock on input pin 0 and perform a reset using pin 1. Drive the start input on the 6 bit instruction word, then encode your instructions in the above format on the 6 bit instruction word interface.

## IO

#	Input	Output
0	clock	data_out[0]
1	reset	data_out[1]
2	instruction[0]	data_out[2]
3	instruction[1]	data_out[3]
4	instruction[2]	data_out[4]
5	instruction[3]	data_out[5]
6	instruction[4]	data_out[6]
7	instruction[5]	data_out[7]

## 42 : Base-10 grey counter counts from zero to a trillion

- Author: Daniel Wisehart
- Description: Change only one output bit per count, but count with decimal digits instead of the usual reverse bit order grey counter.
- GitHub repository
- HDL project
- Extra docs
- Clock: any Hz
- External hardware:

### How it works

Like a standard grey counter, this counter will only change one bit per time, but the bits are grouped into decimal digits. (This also makes for easy bits -> decimal decoding, which is done in the test.py file.)

Each decimal digit uses five bits. As with all grey counters, you have to scan the counter output fast enough that either the output value is the same or it has only increased by 1 between scans. If you scan too slowly and the value changes by 2 or more, then the grey counter can and will give you bad counts.

As an example of how this grey counter works, this is the progression of grey bits when counting from decimal 0 to 12:

10's	1's	count
10001	10001	0
10001	00001	1
10001	00011	2
10001	00010	3
10001	00110	4
10001	00100	5
10001	01100	6
10001	01000	7
10001	11000	8
10001	10000	9
00001	10001	10
00001	00001	11
00001	00011	12

But wait, you say, at decimal value 10, two bits change at once. This is where your deciphering of the bits has to be smart.

The bits in the 10's digit change 10x more slowly than the 1's digit. So when you decipher the combined value, you look first at the 10's digit. If the 10's digit has

changed, then you can ignore the 1's digit, because you know the combined value is 10 or 20 or ... 90 or back to 00. If the 10's digit has not changed, then you look at both digits to decipher the combined value: 1 or 2 or ... 98 or 99.

Some work has been done to insure that the 10's digit changes before the 1's digit, but if the physical routes between the outputs of this circuit and the inputs to your scanning circuit make the 10's bits arrive at your inputs later than the 1's bits, you can and will receive 1's bits that change from 9 ('b10000) to 0 ('b00000) even though you have not yet seen the 10's bits change. That can be worked around because you can deduce that the 10's digit has rolled over, so you update your internal copy of the 10's bits. It is probably better to keep the input paths the same length or make the 1's bits use a little longer path than the 10's bits.

Note that in this implementation, the TinyTapeout PCB hardware—which only has 8 outputs—outputs a combination of eight bits depending on the input selection. This is the purpose of the input selection: to determine which counter values you can see on the outputs. To see all of the digits at once, you have to run this in the simulator or with cocotb. Here are the hardware output selections that are available. The selection uses input bits 7 to 2, in that order.

select [5:0]	output [7:0]		
000101	rollover	100B[5:0]	10B[5:4]
000110	100B[0]	10B[5:0]	1B[5:4]
000111	10B[0]	1B[5:0]	100M[5:4]
001001	1B[0]	100M[5:0]	10M[5:4]
001010	100M[0]	10M[5:0]	1M[5:4]
001011	10M[0]	1M[5:0]	100T[5:4]
010001	1M[0]	100T[5:0]	10T[5:4]
010010	100T[0]	10T[5:0]	1T[5:4]
010011	10T[0]	1T[5:0]	100[5:4]
100001	1T[0]	100[5:0]	10[5:4]
100010	100[0]	10[5:0]	1[5:4]
default	10[1:0]	1[5:0]	half_clk

Here are the meanings of the abbreviations

abbrev	meaning
100B	100 billions digit
10B	10 billions digit
1B	billions digit
100M	100 millions digit
10M	10 millions digit
1M	millions digit
100T	100 thousands digit
10T	10 thousands digit
1T	thousands digit
100	hundreds digit
10	tens digit
1	singles digit
rollover	one trillion
half_clk	clock divided by two

## How to test

After reset goes low, the counter should increase by one with each rising edge of the clock. You are encouraged to try different clock rates. If you load a 60 bit value into the init input before you assert reset, that bit pattern will be loaded into the grey counter, as shown in test.py.

If you enter `make` from the `src` directory, a cocotb test will run and report the results. Inside of `test.py` there is a `RANGE` constant at the top of the file which you can use to tell cocotb how many different values to check. Here are some representative test times:

RANGE = 10,000	2 sec
RANGE = 100,000	19 sec
RANGE = 1,000,000	189 sec
RANGE = 10,000,000	2193 sec

## IO

#	Input	Output
0	clock	output[7]
1	reset	output[6]
2	select[5]	output[5]
3	select[4]	output[4]
4	select[3]	output[3]
5	select[2]	output[2]

#	Input	Output
6	select[1]	output[1]
7	select[0]	output[0]

## 43 : ttFIR: Digital Finite Impulse Response (FIR) Filter

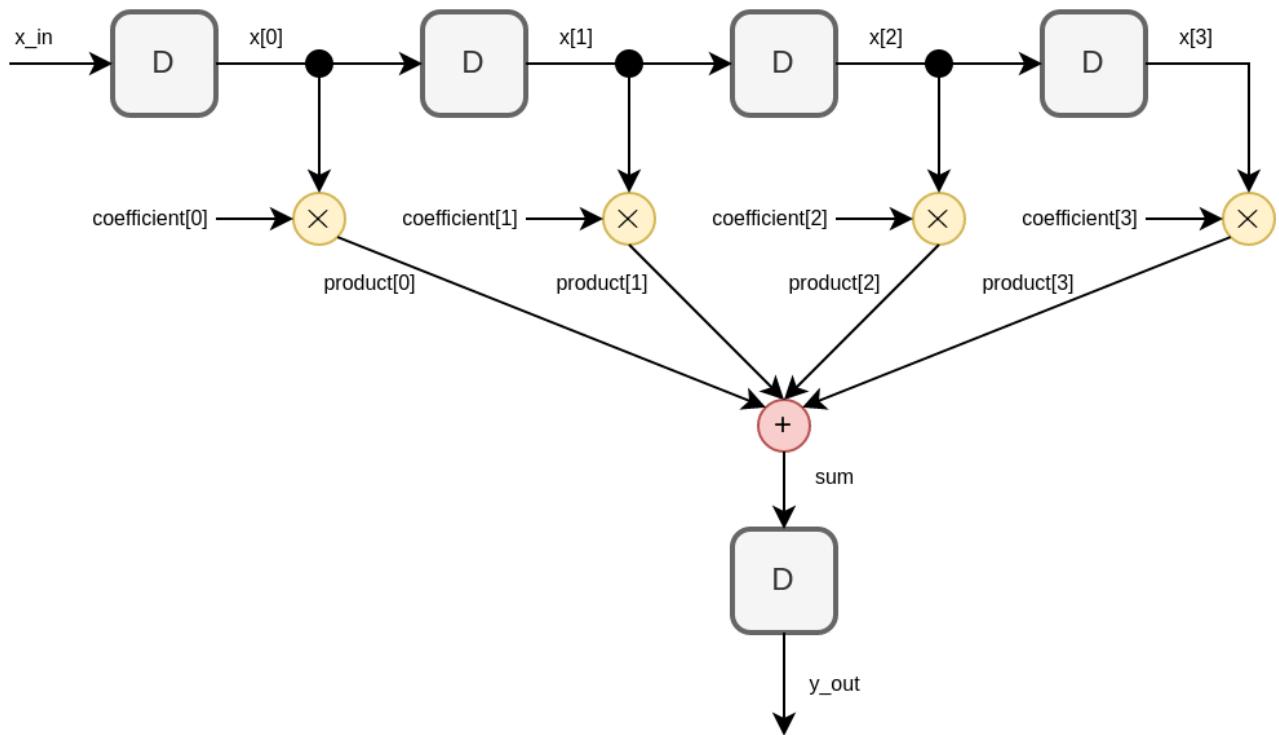


Figure 12: picture

- Author: Georg
- Description: A 4-tap Finite Impulse Response (FIR) filter with 6 bit input signal, 6 bit filter coefficients, and 8 bit output signal. **Challenge for future tinytapeouts:** Increase number of taps.
- GitHub repository
- HDL project
- Extra docs
- Clock: maximum available Hz
- External hardware: FPGA/microcontroller for providing and reading input signal and output signal, respectively

### How it works

ttFIR implements a digital 4-tap Finite Impulse Response (FIR) filter. Inputs, coefficients, and outputs are 6bit, 6bit, and 8bit values, respectively, in 2's complement format. Internally, intermediate products are in 12bit and the final sum is in 14bit. The 8 most significant bits (MSB) of the final sum are output. The 6 least significant bits (LSB) are discarded, which corresponds to a division by 64.

## How to test

- reset high: shift registers for coefficient and output are set to zero.
- reset low:
  - 4 clock cycles: 6bit coefficients in 2's complement format are loaded into registers. The coefficients are loaded in reverse order, i.e., coefficient[3], coefficient[2], coefficient[1], coefficient[0] must be provided in clock cycles 0, 1, 2, 3, respectively.
  - input at each clock cycle: 6bit inputs in 2's complement format are loaded into shift register.
  - output at each clock cycle: coefficients and input values in shift register are multiplied, added and output in 8bit 2's complement format.
- relative to the input, the output is delayed by input register + output register = 2 clock cycles.
- test inputs and expected outputs are defined in the cocotb testbench.

## IO

#	Input	Output
0	clock	bit0 LSB of 2's complement output.
1	reset	bit1
2	bit0 LSB of 2's complement coefficient/input.	bit2
3	bit1	bit3
4	bit2	bit4
5	bit3	bit5
6	bit4	bit6
7	bit5 MSB.	bit7 MSB.

## 44 : QTCore-A1

- Author: Hammond Pearce
- Description: An accumulator-based 8-bit microarchitecture designed via GPT-4 conversations.
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware: Any microcontroller with an SPI peripheral

### How it works

The QTCore-A1 is a basic accumulator-based 8-bit microarchitecture (with an emphasis on the micro). It is a Von Neumann design (shared data and instruction memory).

Although primarily designed for Tiny Tapeout 3, it is parameterized and may be synthesized to FPGAs (and a project file for CMOD A7 is provided on the project GitHub).

Probably the most interesting thing about this design is that all functional Verilog beyond the Tiny Tapeout wrapper was written by GPT-4, i.e. not a human! The author (Hammond Pearce) developed with GPT-4 first the ISA, then the processor, fully conversationally. Hammond wrote the test-benches to validate the design, and then had the appropriate back-and-forth with GPT-4 to have it fix all bugs. For your interest, we will provide all conversation logs in the project repository.

The architecture defines a processor with the following components:

- Control Unit: 2-cycle FSM for driving the processor (3 bit one-hot encoded state register)
- Program Counter: 5-bit register containing the current address of the program
- Instruction Register: 8-bit register containing the current instruction to execute
- Accumulator: 8-bit register used for data storage, manipulation, and logic
- Memory Bank: 17 8-bit registers which store instructions and data. The 17th register is used for I/O.

In order to interact with the processor, all registers are connected via one large scan chain. As such, you can use an external microcontroller's SPI peripheral to read and write the status of the processor. We use the SPI clock SPI\_SCK as the clock to drive both the QTCore-A1 and the scan chain. You choose which you are using by asserting the appropriate chip select. This makes it quite easy to interact with the processor.

The ISA description follows. For your convenience, we also provide an assembler in Python (also written by GPT-4) which makes it quite easy to write simple programs.

## Immediate Data Manipulation Instructions

- ADDI: Add 4-bit Immediate to Accumulator
  - Opcode (4 bits): 1110
  - Immediate (4 bits): 4-bit Immediate
  - Register Effects:  $ACC \leftarrow ACC + IMM$

## Instructions with Variable-Data Operands

- LDA: Load Accumulator with memory contents
  - Opcode (3 bits): 000
  - Operand (5 bits): Memory Address
  - Register Effects:  $ACC \leftarrow M[Address]$
- STA: Store Accumulator to memory
  - Opcode (3 bits): 001
  - Operand (5 bits): Memory Address
  - Register Effects:  $M[Address] \leftarrow ACC$
- ADD: Add memory contents to Accumulator
  - Opcode (3 bits): 010
  - Operand (5 bits): Memory Address
  - Register Effects:  $ACC \leftarrow ACC + M[Address]$
- SUB: Subtract memory contents from Accumulator
  - Opcode (3 bits): 011
  - Operand (5 bits): Memory Address
  - Register Effects:  $ACC \leftarrow ACC - M[Address]$
- AND: AND memory contents with Accumulator
  - Opcode (3 bits): 100
  - Operand (5 bits): Memory Address
  - Register Effects:  $ACC \leftarrow ACC \& M[Address]$
- OR: OR memory contents with Accumulator
  - Opcode (3 bits): 101
  - Operand (5 bits): Memory Address
  - Register Effects:  $ACC \leftarrow ACC | M[Address]$
- XOR: XOR memory contents with Accumulator
  - Opcode (3 bits): 110
  - Operand (5 bits): Memory Address
  - Register Effects:  $ACC \leftarrow ACC \wedge M[Address]$

## Control and Branching Instructions

- JMP: Jump to memory address
  - Opcode (8 bits): 11110000

- PC Behavior:  $PC \leftarrow ACC$  (Load the PC with the address stored in the accumulator)
- JSR: Jump to Subroutine (save address to ACC)
  - Opcode (8 bits): 11110001
  - PC Behavior:  $ACC \leftarrow PC + 1$ ,  $PC \leftarrow ACC$  (Save the next address in ACC, then jump to the address in ACC)
- BEQ\_FWD: Branch if equal, forward (branch if  $ACC == 0$ )
  - Opcode (8 bits): 11110010
  - PC Behavior: If  $ACC == 0$ , then  $PC \leftarrow PC + 3$  (Jump 2 instructions forward if ACC is zero)
- BEQ\_BWD: Branch if equal, backward (branch if  $ACC == 0$ )
  - Opcode (8 bits): 11110011
  - PC Behavior: If  $ACC == 0$ , then  $PC \leftarrow PC - 2$  (Jump 1 instruction backward if ACC is zero)
- BNE\_FWD: Branch if not equal, forward (branch if  $ACC != 0$ )
  - Opcode (8 bits): 11110100
  - PC Behavior: If  $ACC != 0$ , then  $PC \leftarrow PC + 3$  (Jump 2 instructions forward if ACC is non-zero)
- BNE\_BWD: Branch if not equal, backward (branch if  $ACC != 0$ )
  - Opcode (8 bits): 11110101
  - PC Behavior: If  $ACC != 0$ , then  $PC \leftarrow PC - 2$  (Jump 1 instruction backward if ACC is non-zero)
- HLT: Halt the processor until reset
  - Opcode (8 bits): 11111111
  - PC Behavior: Stop execution (PC does not change until a reset occurs)

## Data Manipulation Instructions

- SHL: Shift Accumulator left
  - Opcode (8 bits): 11110110
  - Register Effects:  $ACC \leftarrow ACC \ll 1$
- SHR: Shift Accumulator right
  - Opcode (8 bits): 11110111
  - Register Effects:  $ACC \leftarrow ACC \gg 1$
- SHL4: Shift Accumulator left by 4 bits
  - Opcode (8 bits): 11111000
  - Register Effects:  $ACC \leftarrow ACC \ll 4$
- ROL: Rotate Accumulator left
  - Opcode (8 bits): 11111001
  - Register Effects:  $ACC \leftarrow (ACC \ll 1) \text{ OR } (ACC \gg 7)$
- ROR: Rotate Accumulator right
  - Opcode (8 bits): 11111010

- Register Effects:  $ACC \leftarrow (ACC >> 1) \text{ OR } (ACC << 7)$
- LDAR: Load Accumulator via indirect memory access (ACC as ptr)
  - Opcode (8 bits): 11111011
  - Register Effects:  $ACC \leftarrow M[ACC]$
- DEC: Decrement Accumulator
  - Opcode (8 bits): 11111100
  - Register Effects:  $ACC \leftarrow ACC - 1$
- CLR: Clear (Zero) Accumulator
  - Opcode (8 bits): 11111101
  - Register Effects:  $ACC \leftarrow 0$
- INV: Invert (NOT) Accumulator
  - Opcode (8 bits): 11111110
  - Register Effects:  $ACC \leftarrow \sim ACC$

## **Example programming using the assembler**

Writing assembly programs for QTCore-A1 is simplified by the assembler produced by GPT-4. First, we define two additional meta-instructions:

### **Meta-instructions:**

- NOP: Do nothing
  - Implemented as ADDI 0
- DATA: Define raw data to be loaded at the current address
  - Operand (8 bits): 8-bit data value

### **Presenting programs to the assembler:**

1. Programs are presented in the format [address]: [mnemonic] [optional operand]
2. There is a special meta-instruction called DATA, which is followed by a number. If this is used, just place that number at that address.
3. Programs cannot exceed the size of the memory (in Tiny Tapeout 3, this is 17 bytes including the IO register).
4. The memory contains both instructions and data.

### **Example program:**

An interesting example program is presented. This assumes a button is connected to the general purpose input, and some LEDs are connected to the LED output.

We assume this file is called test\_btn\_led.asm:

```

; This program tests the btn and LEDs
; It will wait for low->high transitions on the button input.
; After receiving this, it will toggle a set of LEDs.
;
; BTNs and LEDS are at address 17, btn at LSB
0: LDA 17 ; load the btn and LEDS
1: AND 16 ; mask the btn
2: BNE_BWD ; if btn&1 is not zero then branch back to 0
3: LDA 17 ; load the btn and LEDS
4: AND 16 ; mask the btn
5: BEQ_BWD ; if btn&1 is zero then branch back to 3
;
; the button has now done a transition from low to high
;
6: LDA 14 ; load the counter toggle
7: XOR 16 ; toggle the counter using the btn mask
8: STA 14 ; store the counter value
9: ADDI 14 ; get the counter value offset
;           (if 0, will be 14 (which is 0), if 1, 15)
10: LDAR ; load the counter LED pattern
11: STA 17 ; store the LED pattern
12: CLR
13: JMP
;
; data
;
14: DATA 0; toggle and LED pattern 0
15: DATA 24; LED pattern of ON
;           (test for led_out=value 12, since 24>>1 == 12)
16: DATA 1 ; btn and counter mask

```

To compile this program, we would invoke the assembler (provided in the associated repository) as follows:

```
$ ./assembler.py test_btn_led.asm
```

The assembler will generate the following files for us:

- `test_btn_led.bin`: This is a binary representation of the assembly program. It can be used, or we can use one of the helper formats...
- `test_btn_led.memarray.v`: This provides the binary ready for use in a Verilog test bench to directly write to the memory of the processor.
- `test_btn_led.scanchain.v`: This provides the binary ready for use in the provided Verilog test bench which loads and unloads programs via the scan chain.

- `test_btn_led.c`: This provides the binary as an array suitable for use in a C program on an external microcontroller which can load it into the processor via SPI.

## Processor operation

The processor executes all instructions via 2 stages (multi-cycle).

The timing is as follows. Note the branch instructions are +2/-3 due to the already-incremented PC in the fetch stage.

### **FETCH cycle (all instructions)**

1.  $IR \leftarrow M[PC]$
2.  $PC \leftarrow PC + 1$

### **EXECUTE cycle**

For **Immediate Data Manipulation Instructions**:

- ADDI:  $ACC \leftarrow ACC + IMM$

For **Instructions with Variable-Data Operands**:

- LDA:  $ACC \leftarrow M[Address]$
- STA:  $M[Address] \leftarrow ACC$
- ADD:  $ACC \leftarrow ACC + M[Address]$
- SUB:  $ACC \leftarrow ACC - M[Address]$
- AND:  $ACC \leftarrow ACC \& M[Address]$
- OR:  $ACC \leftarrow ACC | M[Address]$
- XOR:  $ACC \leftarrow ACC \wedge M[Address]$

For **Control and Branching Instructions**:

- JMP:  $PC \leftarrow ACC$
- JSR:  $ACC \leftarrow PC, PC \leftarrow ACC$
- BEQ\_FWD: If  $ACC == 0$ , then  $PC \leftarrow PC + 2$
- BEQ\_BWD: If  $ACC == 0$ , then  $PC \leftarrow PC - 3$
- BNE\_FWD: If  $ACC != 0$ , then  $PC \leftarrow PC + 2$
- BNE\_BWD: If  $ACC != 0$ , then  $PC \leftarrow PC - 3$
- HLT: (No operation, processor halted)

For **Data Manipulation Instructions**:

- SHL:  $ACC \leftarrow ACC \ll 1$
- SHR:  $ACC \leftarrow ACC \gg 1$
- SHL4:  $ACC \leftarrow ACC \ll 4$
- ROL:  $ACC \leftarrow (ACC \ll 1) \text{ OR } (ACC \gg 7)$

- ROR:  $ACC \leftarrow (ACC >> 1) \text{ OR } (ACC << 7)$
- LDAR:  $ACC \leftarrow M[ACC]$
- DEC:  $ACC \leftarrow ACC - 1$
- CLR:  $ACC \leftarrow 0$
- INV:  $ACC \leftarrow \sim ACC$

## Processor Memory Map

Address	Description
0-16	17 bytes of general purpose Instruction/Data memory
17	I/O: $\{OUT[7:1], IN[0]\}$
18	Constant value: 19 (See "7seg" note below)
19	Constant value: Bits for 7seg "0"
20	Constant value: Bits for 7seg "1"
21	Constant value: Bits for 7seg "2"
22	Constant value: Bits for 7seg "3"
23	Constant value: Bits for 7seg "4"
24	Constant value: Bits for 7seg "5"
25	Constant value: Bits for 7seg "6"
26	Constant value: Bits for 7seg "7"
27	Constant value: Bits for 7seg "8"
28	Constant value: Bits for 7seg "9"
29-31	Constant value: 1

## 7seg

Tiny Tapeout 3 has a 7-segment display on the board. To make it useful with the QTCore-A1, the processor helpfully includes the bit patterns stored at addresses 19-28. The easiest way to make use of these is with the LDAR instruction. Here's an example snippet, which assumes a value between 0 and 9 is stored at address 16:

```
0: LDA 16 ; load the value we want to display
1: ADD 18 ; add it to the constant 19 to get the 7 segment pattern offset
2: LDAR   ; load the 7 segment pattern
3: STA 17 ; emit the 7 segment pattern
```

## How to test

The processor will not run unless a program is scanned into it. Fortunately, this is easy using the SPI peripheral of an external microcontroller.

## Wiring the SPI

The QTCore-A1 is designed to be connected to an SPI peripheral along with two chip selects. This is because we use the SPI SCK as the clock for both the scan chain and the processor.

Connect the I/O according to the provided wiring table. Then, set the SPI peripheral to the following settings:

- SPI Mode 0
- 8-bit data
- MSB first
- A very slow clock (I used 1kHz)
- The scan chain chip select is active low
- The processor chip select is active low

## Loading a program

During the scan chain mode (when the scan enable is low), the entire processor acts as a giant shift register, with the bits in the following order:

All elements of the scan chain are presented MSB first.

- `scan_chain[2:0]` - 3-bit state register
- `scan_chain[7:3]` - 5-bit PC
- `scan_chain[15:8]` - 8-bit Instruction Register
- `scan_chain[23:16]` - 8-bit Accumulator
- `scan_chain[31 -: 8]` - Memory[0]
- `scan_chain[39 -: 8]` - Memory[1]
- `scan_chain[47 -: 8]` - Memory[2]
- `scan_chain[55 -: 8]` - Memory[3]
- `scan_chain[63 -: 8]` - Memory[4]
- `scan_chain[71 -: 8]` - Memory[5]
- `scan_chain[79 -: 8]` - Memory[6]
- `scan_chain[87 -: 8]` - Memory[7]
- `scan_chain[95 -: 8]` - Memory[8]
- `scan_chain[103 -: 8]` - Memory[9]
- `scan_chain[111 -: 8]` - Memory[10]
- `scan_chain[119 -: 8]` - Memory[11]
- `scan_chain[127 -: 8]` - Memory[12]
- `scan_chain[135 -: 8]` - Memory[13]
- `scan_chain[143 -: 8]` - Memory[14]
- `scan_chain[151 -: 8]` - Memory[15]
- `scan_chain[159 -: 8]` - Memory[16]

- scan\_chain[167 -: 8] - Memory[17] (the IO register)

C code to load the previously-discussed example program (e.g. via the STM32 HAL) is provided:

```
//The registers are presented in reverse order to
// the table as we load them MSB first.
uint8_t program_led_btn[21] = {
    0b00000000, //IOREG
    0b00000001, //MEM[16]
    0b00011000, //MEM[15]
    0b00000000, //MEM[14]
    0b11110000, //MEM[13]
    0b11111101, //MEM[12]
    0b00110001, //MEM[11]
    0b11111011, //MEM[10]
    0b11101110, //MEM[9]
    0b00101110, //MEM[8]
    0b11010000, //MEM[7]
    0b00001110, //MEM[6]
    0b11110011, //MEM[5]
    0b10010000, //MEM[4]
    0b00010001, //MEM[3]
    0b11110101, //MEM[2]
    0b10010000, //MEM[1]
    0b00010001, //MEM[0]
    0b00000000, //ACC
    0b00000000, //IR
    0b00000001 //PC[5bit], CU[3bit]
};

//... this will go in the SPI peripheral initialization code
hspi1.Init.Mode = SPI_MODE_MASTER;
hspi1.Init.Direction = SPI_DIRECTION_2LINES;
hspi1.Init.DataSize = SPI_DATASIZE_8BIT;
hspi1.Init.CLKPolarity = SPI_POLARITY_LOW;
hspi1.Init.CLKPhase = SPI_PHASE_1EDGE;
hspi1.Init.NSS = SPI_NSS_SOFT;
//...
hspi1.Init.FirstBit = SPI_FIRSTBIT_MSB;
hspi1.Init.TIMode = SPI_TIMODE_DISABLE;
//...
```

```

//... this will go in your main or similar
uint8_t *program = program_led_btn;
//on the first run, scan_out will give us the reset value of the
// processor
HAL_GPIO_WritePin(SPI_SCAN_CS_GPIO_Port, SPI_SCAN_CS_Pin, 0);
HAL_Delay(100);
HAL_SPI_TransmitReceive(&hspi1, program, scan_out, 21, HAL_MAX_DELAY);
HAL_Delay(100);
HAL_GPIO_WritePin(SPI_SCAN_CS_GPIO_Port, SPI_SCAN_CS_Pin, 1);
//we can check if the program loaded correctly by immediately scanning
// it back in again, it will be unloaded to scan_out
HAL_GPIO_WritePin(SPI_SCAN_CS_GPIO_Port, SPI_SCAN_CS_Pin, 0);
HAL_Delay(100);
HAL_SPI_TransmitReceive(&hspi1, program, scan_out, 21, HAL_MAX_DELAY);
HAL_Delay(100);
HAL_GPIO_WritePin(SPI_SCAN_CS_GPIO_Port, SPI_SCAN_CS_Pin, 1);
//Check if the program loaded correctly
for(int i = 0; i < 21; i++) {
    if(program[i] != scan_out[i]) {
        while(1); //it failed
    }
}

```

## Running a program

Once the program is loaded (using the above code or similar), we can run a program. This is as easy as providing a clock signal to the processor and setting the processor enable line high. We do this using the SPI as follows:

```

uint8_t dummy; //a dummy value
HAL_GPIO_WritePin(SPI_PROC_CS_GPIO_Port, SPI_PROC_CS_Pin, 0);
//...
//run this in a loop
HAL_SPI_TransmitReceive(&hspi1, &dummy, &dummy, 1, HAL_MAX_DELAY);

```

The processor will ignore any value being shifted in on the MOSI data line during operation. However, it does provide a nice feature in that the processor will emit the current value of the processor\_halt signal on the MISO line. This means that we can improve the code to run this process in a loop and catch when the program reaches a HLT instruction:

```

HAL_Delay(100);
HAL_GPIO_WritePin(SPI_PROC_CS_GPIO_Port, SPI_PROC_CS_Pin, 0);

```

```

dummy = 0;
while(1) {
    HAL_SPI_TransmitReceive(&hspi1, &dummy, &dummy, 1, HAL_MAX_DELAY);
    if(dummy == 0xFF)
        break;
}
HAL_GPIO_WritePin(SPI_PROC_CS_GPIO_Port, SPI_PROC_CS_Pin, 1);
HAL_Delay(100);

```

Of course, the example program does not HLT, so we will not reach this point in this code. But, it works for other programs that do contain a HLT instruction.

Once you have the provided example program running, you will be able to press the button and see how the LEDs are toggled.

## IO

---

#	Input	Output
0	clock - connect to an SPI SCK.	general purpose output 0 (e.g. LED segment a). This output comes from the I/O register bit 1.
1	reset (active high)	general purpose output 1 (e.g. LED segment b). This output comes from the I/O register bit 2.
2	scan enable (active low) - connect to an SPI chip select.	general purpose output 2 (e.g. LED segment c). This output comes from the I/O register bit 3.
3	processor enable (active low) - connect to an SPI chip select.	general purpose output 3 (e.g. LED segment d). This output comes from the I/O register bit 4.
4	scan data in - connect to an SPI MOSI.	general purpose output 4 (e.g. LED segment e). This output comes from the I/O register bit 5.
5	general purpose input (e.g. Button). This input will be provided to the I/O register bit 0.	general purpose output 5 (e.g. LED segment f). This output comes from the I/O register bit 6.
6	none	general purpose output 6 (e.g. LED segment g). This output comes from the I/O register bit 7.
7	none	scan data out - connect to an SPI MISO.

---

## 45 : MicroTapeout (of sky130 cells)

- Author: htfab
- Description: 395 standard cells with a mux to select between them
- GitHub repository
- HDL project
- Extra docs
- Clock: 10000 Hz
- External hardware:

### How it works

Digital chip designs are usually written in a hardware description language like RTL Verilog and then synthesized into a set of mask layers suitable for fabrication. In order to make both synthesis and verification robust for huge designs, a modular approach is used where the functionality of the circuit is decomposed into pre-built blocks called *standard cells* with well-known and thoroughly tested behaviour and layout.

This design contains a copy of most standard cells in the sky130\_fd\_sc\_hd library along with a multiplexing mechanism that allows exposing any of them to the input/output pins.

An MPW shuttle fabricates multiple designs on the same wafer. TinyTapeout merges several projects in a single shuttle submission. MicroTapeout pushes the limit with each block containing just a single cell. Apart from the geek factor the fabricated chip can be used by low-level digital design engineers to better understand the behaviour of the individual standard cells and might even provide some timing insights.

There are 437 standard cells in our library, of which 42 don't produce output or require special power handling. This leaves us with 395 cells. Each cell has up to 6 inputs and up to 2 outputs for a total of 427 outputs. The same 6 inputs are fed into each cell in parallel while the 427 outputs are divided into 54 pages of 8 outputs each with a multiplexer deciding which page is mapped to the output pins.

In order to drive the 6 cell inputs and the 6 bits of input to the mux from a total of 8 input pins we use some registered logic. Input pin 0 is a clock signal while input pin 1 selects *page mode*. On each rising clock edge we save input pins 2 to 7 into a page register if page mode is on and into an input register if page mode is off. Cell inputs are then supplied from the input register and the mux operates on the page register.

Mapping of outputs to pages:

page	pin	pin 0/4	pin 1/5	pin 2/6	pin 3/7
000000	0-3	conb_1.h	conb_1.l	buf_1	buf_2

page	pin	pin 0/4	pin 1/5	pin 2/6	pin 3/7
	4-7	buf_4	buf_6	buf_8	buf_12
000001	0-3	buf_16	bufbuf_8	bufbuf_16	inv_1
	4-7	inv_2	inv_4	inv_6	inv_8
000010	0-3	inv_12	inv_16	bufinv_8	bufinv_16
	4-7	and2_0	and2_1	and2_2	and2_4
000011	0-3	and2b_1	and2b_2	and2b_4	and3_1
	4-7	and3_2	and3_4	and3b_1	and3b_2
000100	0-3	and3b_4	and4_1	and4_2	and4_4
	4-7	and4b_1	and4b_2	and4b_4	and4bb_1
000101	0-3	and4bb_2	and4bb_4	nand2_1	nand2_2
	4-7	nand2_4	nand2_8	nand2b_1	nand2b_2
000110	0-3	nand2b_4	nand3_1	nand3_2	nand3_4
	4-7	nand3b_1	nand3b_2	nand3b_4	nand4_1
000111	0-3	nand4_2	nand4_4	nand4b_1	nand4b_2
	4-7	nand4b_4	nand4bb_1	nand4bb_2	nand4bb_4
001000	0-3	or2_0	or2_1	or2_2	or2_4
	4-7	or2b_1	or2b_2	or2b_4	or3_1
001001	0-3	or3_2	or3_4	or3b_1	or3b_2
	4-7	or3b_4	or4_1	or4_2	or4_4
001010	0-3	or4b_1	or4b_2	or4b_4	or4bb_1
	4-7	or4bb_2	or4bb_4	nor2_1	nor2_2
001011	0-3	nor2_4	nor2_8	nor2b_1	nor2b_2
	4-7	nor2b_4	nor3_1	nor3_2	nor3_4
001100	0-3	nor3b_1	nor3b_2	nor3b_4	nor4_1
	4-7	nor4_2	nor4_4	nor4b_1	nor4b_2
001101	0-3	nor4b_4	nor4bb_1	nor4bb_2	nor4bb_4
	4-7	xor2_1	xor2_2	xor2_4	xor3_1
001110	0-3	xor3_2	xor3_4	xnor2_1	xnor2_2
	4-7	xnor2_4	xnor3_1	xnor3_2	xnor3_4
001111	0-3	a2111o_1	a2111o_2	a2111o_4	a2111oi_0
	4-7	a2111oi_1	a2111oi_2	a2111oi_4	a211o_1
010000	0-3	a211o_2	a211o_4	a211oi_1	a211oi_2
	4-7	a211oi_4	a21bo_1	a21bo_2	a21bo_4
010001	0-3	a21boi_0	a21boi_1	a21boi_2	a21boi_4
	4-7	a21o_1	a21o_2	a21o_4	a21oi_1
010010	0-3	a21oi_2	a21oi_4	a221o_1	a221o_2
	4-7	a221o_4	a221oi_1	a221oi_2	a221oi_4
010011	0-3	a222oi_1	a22o_1	a22o_2	a22o_4
	4-7	a22oi_1	a22oi_2	a22oi_4	a2bb2o_1
010100	0-3	a2bb2o_2	a2bb2o_4	a2bb2oi_1	a2bb2oi_2

page	pin	pin 0/4	pin 1/5	pin 2/6	pin 3/7
	4-7	a2bb2oi_4	a311o_1	a311o_2	a311o_4
010101	0-3	a311oi_1	a311oi_2	a311oi_4	a31o_1
	4-7	a31o_2	a31o_4	a31oi_1	a31oi_2
010110	0-3	a31oi_4	a32o_1	a32o_2	a32o_4
	4-7	a32oi_1	a32oi_2	a32oi_4	a41o_1
010111	0-3	a41o_2	a41o_4	a41oi_1	a41oi_2
	4-7	a41oi_4	o2111a_1	o2111a_2	o2111a_4
011000	0-3	o2111ai_1	o2111ai_2	o2111ai_4	o211a_1
	4-7	o211a_2	o211a_4	o211ai_1	o211ai_2
011001	0-3	o211ai_4	o21a_1	o21a_2	o21a_4
	4-7	o21ai_0	o21ai_1	o21ai_2	o21ai_4
011010	0-3	o21ba_1	o21ba_2	o21ba_4	o21bai_1
	4-7	o21bai_2	o21bai_4	o221a_1	o221a_2
011011	0-3	o221a_4	o221ai_1	o221ai_2	o221ai_4
	4-7	o22a_1	o22a_2	o22a_4	o22ai_1
011100	0-3	o22ai_2	o22ai_4	o2bb2a_1	o2bb2a_2
	4-7	o2bb2a_4	o2bb2ai_1	o2bb2ai_2	o2bb2ai_4
011101	0-3	o311a_1	o311a_2	o311a_4	o311ai_0
	4-7	o311ai_1	o311ai_2	o311ai_4	o31a_1
011110	0-3	o31a_2	o31a_4	o31ai_1	o31ai_2
	4-7	o31ai_4	o32a_1	o32a_2	o32a_4
011111	0-3	o32ai_1	o32ai_2	o32ai_4	o41a_1
	4-7	o41a_2	o41a_4	o41ai_1	o41ai_2
100000	0-3	o41ai_4	maj3_1	maj3_2	maj3_4
	4-7	mux2_1	mux2_2	mux2_4	mux2_8
100001	0-3	mux2i_1	mux2i_2	mux2i_4	mux4_1
	4-7	mux4_2	mux4_4	ha_1.c	ha_1.s
100010	0-3	ha_2.c	ha_2.s	ha_4.c	ha_4.s
	4-7	fa_1.c	fa_1.s	fa_2.c	fa_2.s
100011	0-3	fa_4.c	fa_4.s	fah_1.c	fah_1.s
	4-7	fahcin_1.c	fahcin_1.s	fahcon_1.c	fahcon_1.s
100100	0-3	dlxtp_1	dlxbp_1.q	dlxbp_1.n	dlxtn_1
	4-7	dlxtn_2	dlxtn_4	dlxbn_1.q	dlxbn_1.n
100101	0-3	dlxbn_2.q	dlxbn_2.n	dlrtp_1	dlrtp_2
	4-7	dlrtp_4	dlrbp_1.q	dlrbp_1.n	dlrbp_2.q
100110	0-3	dlrbp_2.n	dlrtn_1	dlrtn_2	dlrtn_4
	4-7	dlrbn_1.q	dlrbn_1.n	dlrbn_2.q	dlrbn_2.n
100111	0-3	dfxtp_1	dfxtp_2	dfxtp_4	dfxbp_1.q
	4-7	dfxbp_1.n	dfxbp_2.q	dfxbp_2.n	dfrtp_1
101000	0-3	dfrtp_2	dfrtp_4	dfrbp_1.q	dfrbp_1.n

page	pin	pin 0/4	pin 1/5	pin 2/6	pin 3/7
		4-7 dfrbp_2.q	dfrbp_2.n	dfrtn_1	dfstp_1
101001	0-3 dfstp_2	dfstp_4	dfsdp_1.q	dfsdp_1.n	
		4-7 dfsdp_2.q	dfsdp_2.n	dfbbp_1.q	dfbbp_1.n
101010	0-3 dfbbn_1.q	dfbbn_1.n	dfbbn_2.q	dfbbn_2.n	
		4-7 edfxtp_1	edfxtp_1.q	edfxtp_1.n	sdfxtp_1
101011	0-3 sdfxtp_2	sdfxtp_4	sdfxtp_1.q	sdfxtp_1.n	
		4-7 sdfxtp_2.q	sdfxtp_2.n	sdfrtp_1	sdfrtp_2
101100	0-3 sdfrtp_4	sdfrbp_1.q	sdfrbp_1.n	sdfrbp_2.q	
		4-7 sdfrbp_2.n	sdfrtn_1	sdfstp_1	sdfstp_2
101101	0-3 sdfstp_4	sdfsdp_1.q	sdfsdp_1.n	sdfsdp_2.q	
		4-7 sdfsdp_2.n	sdffbp_1.q	sdffbp_1.n	sdfbbn_1.q
101110	0-3 sdfbbn_1.n	sdfbbn_2.q	sdfbbn_2.n	sedfxtp_1	
		4-7 sedfxtp_2	sedfxtp_4	sedfxtp_1.q	sedfxtp_1.n
101111	0-3 sedfxtp_2.q	sedfxtp_2.n	ebufn_1/_2	ebufn_4/_8	
		4-7 einvp_1/n_0	einvp_1/n_1	einvp_2/n_2	einvp_4/n_4
110000	0-3 einvp_8/n_8	dg~sd1_1	dg~4sd2_1	dg~4sd3_1	
		4-7 dm~6s2s_1	dm~6s4s_1	dm~6s6s_1	clkbuf_1
110001	0-3 clkbuf_2	clkbuf_4	clkbuf_8	clkbuf_16	
		4-7 clkinv_1	clkinv_2	clkinv_4	clkinv_8
110010	0-3 clkinv_16	clkinvlp_2	clkinvlp_4	cdb~4s15_1	
		4-7 cdb~4s15_2	cdb~4s18_1	cdb~4s18_2	cdb~4s25_1
110011	0-3 cbd~4s25_2	cbd~4s50_1	cbd~4s50_2	dlclkp_1	
		4-7 dlclkp_2	dlclkp_4	sdlclkp_1	sdlclkp_2
110100	0-3 sdlclkp_4	lpfii~0p_1	lpfii~0n_1	lpfii~1p_1	
		4-7 lpfii~1n_1	lpfii~latch_1	lpfib~_1	lpfib~_2
110101	0-3 lpfib~_4	lpfib~_8	lpfib~_16		

where  $dg\sim$  = dlygate,  $dm\sim$  = dlymetal,  $cdb\sim$  = clkbuf,  $lpfii\sim$  = lpflow\_inputiso,  $lpfib\sim$  = lpflow\_isobufsrc.

The design also contains an experimental timing circuit for measuring the switching times of the individual standard cells using a ring oscillator. This is complicated by (1) a ring oscillator built from standard cells being necessarily slower than the time to be measured, and (2) several buffering and multiplexing cells plus wires also included in the measurement.

To offset (1), we can repeat a measurement several times and average them. Since the ring oscillator is not synchronized to the rest of the chip, this should result in higher timing resolution. To combat (2), we can compare the results to gate-level simulations and finetune the models until the results match up.

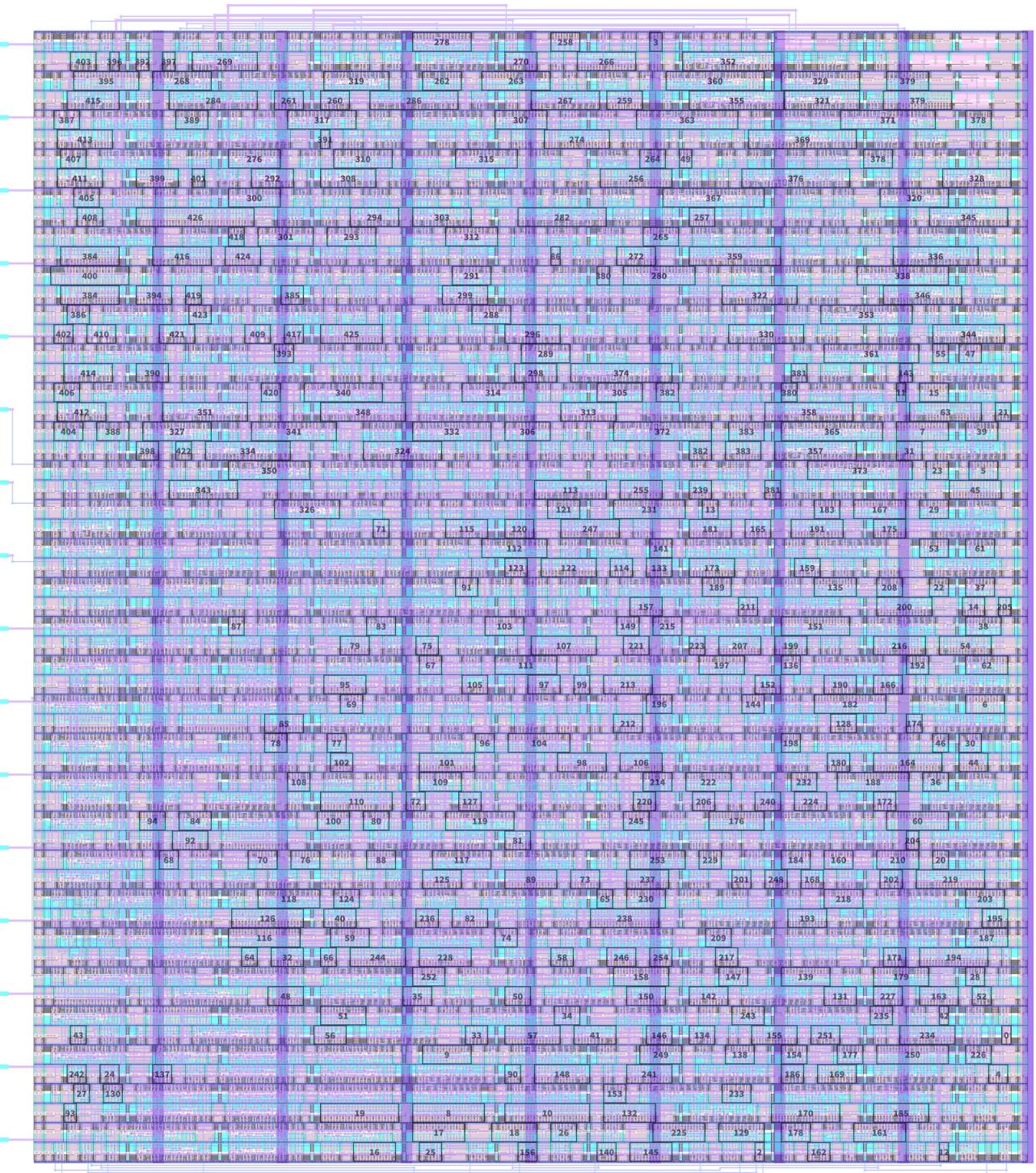


Figure 13: picture

## How to test

Set pin 1 high to switch to page mode. Find the standard cell you would like to test in the table above and set pins 2-7 to the 6 bit binary page number indicated in the first column. If pin 0 is not connected to the clock, manually toggle it low and then high to force a clock cycle. Set pin 1 low to switch to input mode. Set pins 2 and up to the values that should be supplied to the selected standard cell's input pins. Once again, you may need to manually trigger a clock cycle. The result should appear on the output pin corresponding to the table column.

To use the experimental timing circuit, make sure pin 0 is in manual mode (not connected to a clock). First set the page number the same way as above. While still in page mode, set pins 2-7 to the virtual page number 111pqr where pqr is the cell index within the page. Trigger another clock cycle using pin 0. Now set pin 1 low to switch to input mode. Set pins 2 and up to the *initial* cell input values and toggle pin 0 low and high again. Set pins 2 and up to the *modified* cell input values and toggle pin 0 low and high once more. This will latch the standard cell's previous output (i.e. the one for the *initial* input) and will connect the ring oscillator to a counter while the output is the same as the latched value. The counter is connected to output pins 0-5. If the cell output for the *initial* and *modified* inputs are different, this should settle to a value based on the cell switching time. Otherwise it will keep running indefinitely. Pin 6 is connected to the same gated clock as the counter but through a massive clock divider, resulting in visible blinking if the counter is still running. The blinking speed can also be measured to calculate the frequency of the ring oscillator (which depends on temperature, voltage and process parameters). Pin 7 shows the latched cell output to help debugging.

## IO

#	Input	Output
0	clock	output[8*page+0] / counter[0]
1	page mode	output[8*page+1] / counter[1]
2	input[0] / page[0] / cell[0]	output[8*page+2] / counter[2]
3	input[1] / page[1] / cell[1]	output[8*page+3] / counter[3]
4	input[2] / page[2] / cell[2]	output[8*page+4] / counter[4]
5	input[3] / page[3] / 1 (timing)	output[8*page+5] / counter[5]
6	input[4] / page[4] / 1 (timing)	output[8*page+6] / strobe
7	input[5] / page[5] / 1 (timing)	output[8*page+7] / latched value

## 46 : nibble multiplier

- Author: 'Mohamed Nasser
- Description: multiply two 8-b numbers in 4 chunks
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware:

### How it works

assign the switches with the first nibble of the first number, make valid high, at the next cycle it will be latched. de-assert valid and assign the switches with the second nibble of the first number, make valid high, at the next cycle it will be latched. assign the switches with the first nibble of the second number, make valid high, at the next cycle it will be latched. de-assert valid and assign the switches with the second nibble of the second number, make valid high, at the next cycle it will be latched. the result will be 16-bits however we have only 8-b output so, if the toggle bit is high the first 8-b of the result will be shown at the output. if the toggle bit is low the second 8-b of the result will be shown at the output.

### How to test

assign the switches with the first nibble of the first number, make valid high, at the next cycle it will be latched. de-assert valid and assign the switches with the second nibble of the first number, make valid high, at the next cycle it will be latched. assign the switches with the first nibble of the second number, make valid high, at the next cycle it will be latched. de-assert valid and assign the switches with the second nibble of the second number, make valid high, at the next cycle it will be latched. the result will be 16-bits however we have only 8-b output so, if the toggle bit is high the first 8-b of the result will be shown at the output. if the toggle bit is low the second 8-b of the result will be shown at the output.

### IO

#	Input	Output
0	data0	dout0
1	data1	dout1
2	data2	dout2
3	data3	dout3

#	Input	Output
4	clk	dout4
5	reset_n	dout5
6	valid	dout6
7	toggle	dout7

## 47 : Synthesizable Digital Temperature Sensor

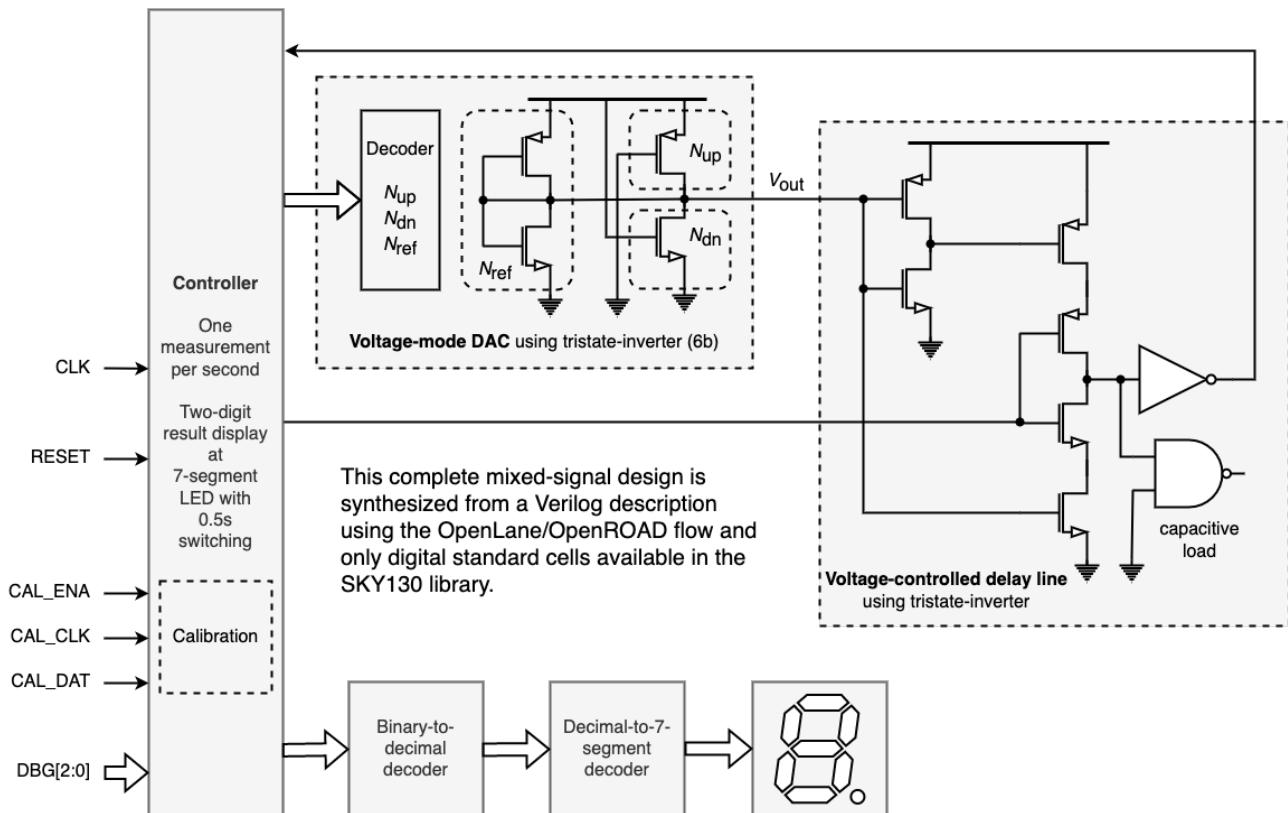


Figure 14: picture

- Author: Harald Pretl
- Description: Measure the on-chip temperature and display on the LED display.
- GitHub repository
- HDL project
- Extra docs
- Clock: 10000 Hz
- External hardware:

### How it works

By creatively twisting the use of a tristate-inverter (EINVP) a voltage DAC is built. This voltage-mode DAC is used in another twisted arrangement of an EINVP to bias an NMOS into subthreshold operation to discharge a pre-charged capacitor (the input capacitor of an inverter). Since the subthreshold current of a MOSFET is a strong function of temperature, the resulting delay time is also a strong function of temperature, thus a digital temperature sensor is built.

The temperature-dependent digital signal is output at the LED display, showing tens (dot off) and ones (dot on).

A calibration engine via a LUT is included, to allow to linearize and calibrate the shown temperature code.

io\_in[0] is used as a CLK signal, and io\_in[1] is used a RESET.

io\_in[4:2] is used to load and enable the calibration engine.

io\_in[7:5] is used to enable various debug modes, presenting internal state signals to the io\_out.

## How to test

After reset, one temperature measurement is taken per second and displayed using the LEDs. A code ranging 0...63 is displayed with tens first (dot off) and ones later (dot on).

During normal operation io\_in[7:5] have to be set to 000. The different debug modes are documented in the Verilog code.

For calibration, the internal LUT can be serially loaded by using CAL\_CLK (io\_in[2]) and CAL\_DAT (io\_in[3]). Once fully loaded the calibration engine is enabled by setting CAL\_ENA (io\_in[4]) to 1 (setting it to 0 displays the raw sensor code).

## IO

#	Input	Output
0	clock	segment a (or debug information)
1	reset	segment b (or debug information)
2	cal_clk	segment c (or debug information)
3	cal_dat	segment d (or debug information)
4	cal_ena	segment e (or debug information)
5	debug_mode[0]	segment f (or debug information)
6	debug_mode[1]	segment g (or debug information)
7	debug_mode[2]	indicate ones or tens (or debug information)

## 48 : 4-bits sequential ALU

- Author: Diego Satizabal
- Description: A 4-bits sequential ALU that takes operands and opcode sequentially and performs operations and outputs results
- GitHub repository
- HDL project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

The ALU takes 4-bits wide Operators and Operation sequentially with every clock cycle if the enabled signal is set to high, it takes an additional clock to output the result

### How to test

Run `make` from the command line in the `src` directory to perform all tests suites, you must have Python, cocotb and Icarus Verilog installed If you have GTK Wave installed you can run the command `make test_gtkwave` to generate the VCD output, then run `gtkwave tb.vcd` to see the waveforms

### IO

#	Input	Output
0	clock	result_0
1	reset	result_1
2	enabled	result_2
3	none	result_3
4	Opx_opcode_0	done_flag
5	Opx_opcode_1	carry_flag
6	Opx_opcode_2	zero_flag
7	Opx_opcode_3	sign_flag

## 49 : Brightness control of LED with PWM

- Author: Ioannis G. Intzes
- Description: Increase and Decrease the PWM (Pulse-Width) to dim a LED.
- GitHub repository
- HDL project
- Extra docs
- Clock: 12500 Hz
- External hardware:

### How it works

This Verilog code implements a simple PWM (Pulse-Width Modulation) controller module that takes an input clock signal of 12500 Hz and generates a PWM output signal based on the input from two buttons to increase or decrease the PWM duty cycle. The module also includes debouncing functionality for the button inputs to eliminate any bouncing that may occur when the button is pressed. The PWM output is controlled by a duty cycle variable that is updated based on the button inputs and ranges from 0% to 100%. The module also includes output signals that indicate when the maximum and minimum duty cycle values have been reached and a 1 Hz clock signal.

### How to test

After reset, the counter should increase by one every second.

### IO

#	Input	Output
0	clk	inled
1	btn_incrPWM	deled
2	btn_decrPWM	led
3	none	clock_1Hz
4	none	none
5	none	none
6	none	none
7	none	none

## 50 : Microtapeout

- Author: Enno Schnackenberg
- Description: A Shift Register, A seven segment encoder (Hexadecimal), 1 Bit ALU, 3 Bit Adder and a surprise
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

You can select the project over Input 0 and 1 (e.g. 01 for the seven segment encoder) and then have the 6 remaining pins for the project

### How to test

#### Shift Register

Set Input 0 and 1 to LOW. Input 2 is the Data In Pin, Input 3 is the Shift Register Clock and Input 4 the Ooutput Latch

#### ALU

Set Input 0 and 1 to HIGH and then you can use Input 5 to 7 for the multiplexer Inputs 2 to 4 are for the Inputs of the ALU the Output is connected to Output 1

#### Adder

Set the Inputs like for the ALU, but now ise Inputs 2 to 4 for the first number and Inputs 5 to 7 for the second. Outputs 1 to 4 are the Resulting number

#### Seven Segment

Set Input 0 to LOW and Input 1 to HIGH and then use Inputs 3 - 6 for the number input (MSB first)

### IO

#	Input	Output
0	Project Select 0	segment a
1	Project Select 1	segment b
2	I1	segment c
3	I2	segment d
4	I3	segment e
5	I4	segment f
6	I5	segment g
7	I6	dot

## 51 : Neptune guitar tuner (fixed window)



Figure 15: picture

- Author: Pat Deegan
- Description: It's a guitar tuner! and so much more... (TODO)
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware: The analog input signal must be massaged into clean digital. See the Neptune project for details <https://github.com/psychogenic/neptune>

### How it works

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

Puts a slower square wave output on output 7.

## How to test

After reset, the counter should increase by one every second.

## IO

#	Input	Output
0	clock	segment a
1	reset	segment b
2	clock config A	segment c
3	clock config B	segment d
4	measured frequency input pulses	segment e
5	none	segment f
6	none	segment g
7	none	display select

## 52 : Neptune guitar tuner (proportional window)



Figure 16: picture

- Author: Pat Deegan
- Description: It's a guitar tuner! and so much more... (TODO)
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware: The analog input signal must be massaged into clean digital. See the Neptune project for details <https://github.com/psychogenic/neptune>

### How it works

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

Puts a slower square wave output on output 7.

## How to test

After reset, the counter should increase by one every second.

## IO

#	Input	Output
0	clock	segment a
1	reset	segment b
2	clock config A	segment c
3	clock config B	segment d
4	measured frequency input pulses	segment e
5	none	segment f
6	none	segment g
7	none	display select

## 53 : M segments

- Author: Matt Venn
- Description: pressing the first 4 buttons will put an M on the LEDs
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

Uses 3 AND gates

### How to test

Press the first 4 buttons and an M will show on the 7 segment display

### IO

#	Input	Output
0	in1	segment a
1	in2	segment b
2	in3	segment c
3	in4	segment d
4	none	segment e
5	none	segment f
6	none	segment g
7	none	dot

## 54 : 7 segment seconds

- Author: Matt Venn
- Description: Count up to 10, one second at a time.
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware:

### How it works

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

Puts a slower square wave output on output 7.

### How to test

After reset, the counter should increase by one every second.

### 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	slow clock output

## 55 : 7 segment wokwi counter

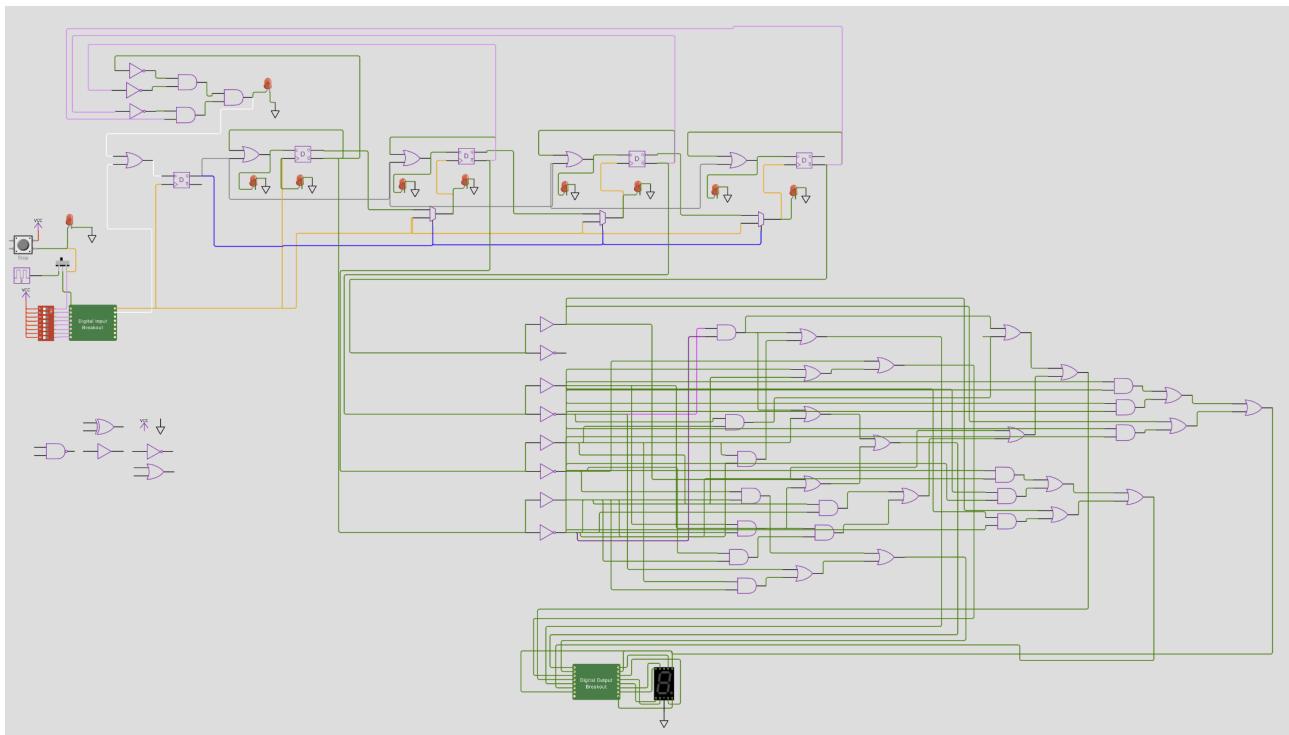


Figure 17: picture

- Author: Matt Venn
- Description: counts up from 0 to 9, incrementing once per second
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 1 Hz
- External hardware:

### How it works

Uses a 4 bit counter to sequence, the combinational logic to drive the 7 segment display.

### How to test

Set the clock to 1Hz.

### 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	none	dot

## 56 : Straight through test

- Author: Matt Venn
- Description: Just connects inputs to outputs
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

8 wires connect the 8 inputs to the 8 outputs

### How to test

Set any DIP switch high and the corresponding LED should light on the 7 segment display

### IO

#	Input	Output
0	a	segment a
1	b	segment b
2	c	segment c
3	d	segment d
4	e	segment e
5	f	segment f
6	g	segment g
7	dot	dot

## 57 : Combo lock

- Author: Benjamin Collier
- Description: set, reset, and check for a combo
- GitHub repository
- HDL project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

this projects works by taking 4 inputs for number 0-9. The first 4 number inputed will be the new combo (0000 is not an input zero should be 0001). Then to “unlock” press your combo again. to reset and enter a new combo the unlock must be high and reset must be pressed, or master reset must be pressed.

### How to test

Enter 4 single digits in binary into input 0-4, each input must be followed by 0000. Then input the same 4 digits in each followed by 0000 to make unlock high. a valid input would look some think like this: 0001, 0000, 0101, 0000, 1001, 0000, 0011, 0000, then 0001, 0000, 0101, 0000, 1001, 0000, 0011, 0000, will make unlock high.

### IO

#	Input	Output
0	input 0	unlock
1	input 1	none
2	input 2	none
3	input 3	none
4	reset	none
5	master reset	none
6	none	none
7	none	none

## 58 : ro-based\_tempsense

- Author: Jorge Marin, Daniel Arevalos
- Description: Ring oscillator whose frequency depends on temperature.
- GitHub repository
- HDL project
- Extra docs
- Clock: 10000 Hz
- External hardware:

### How it works

Uses counters to determine the number of cycles of the ring oscillator within a clock period, which will then be sent through the UART to determine the temperature vs frequency characteristic.

### How to test

After reset and enable are set, the ring oscillator should start and then when a START code is received by UART, a value is sent back.

### IO

#	Input	Output
0	clk_internal	tx
1	clk_external	sum[8]
2	clk_sel	sum[10]
3	enable_inv_osc	sum[13]
4	enable_nand_osc	sum[15]
5	reset	sum[17]
6	rx	sum[19]
7	osc_sel	sum[21]

## 59 : FSM\_LAT

- Author: Juan Sanchez
- Description: FSM look at table
- GitHub repository
- HDL project
- Extra docs
- Clock: 0 Hz
- External hardware: un fpga que permita generar un relog una salida serial para programar la secuencia de la maquina de estados

### How it works

el proyecto es una maquina de estados a la cual se le puede cambiar la secuencia de los estados por medio de una entrada serial

### How to test

para cargar la secuencia de estados se debe conectar un relog el cual este sincronizado con la secuencia serial para programar la maquina de estados = y para pasar de estado la entrada conectada a cada estado debe estar en 1

### IO

#	Input	Output
0	oi_in[0]	oi_out[1]
1	oi_in[1]	oi_out[2]
2	oi_in[2]	oi_out[3]
3	oi_in[3]	oi_out[4]
4	oi_in[4]	oi_out[5]
5	oi_in[5]	oi_out[6]
6	oi_in[6]	oi_out[7]
7	oi_in[7]	oi_out[0]

## 60 : 7 segment seconds

- Author: Matt Venn
- Description: Count up to 10, one second at a time.
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware:

### How it works

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

Puts a slower square wave output on output 7.

### How to test

After reset, the counter should increase by one every second.

### 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	slow clock output

## 61 : MMM Finite State Machine (4 States)

- Author: Alexandra Zhang Jiang
- Description: Finite State Machine for the Magnetic Microsystems and Micro-robotics Research Lab at UCI
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 1000 Hz
- External hardware: 3-electrode sensor and motors

### How it works

This chip will be incorporated in the microrobot that is being developed in lab.

### Inputs

The inputs consist of two bits I0 and I1, each one for a sensor that detects contamination in water to the left and right of the robot respectively. The input bus goes to logic gates along with the Current State to calculate the Next State, which is then saved in a register (D-FlipFlop).

### States

There are a total of four states: - 00 Start (default initial state of the robot, both motors are off) - 01 Go Right (which enables Left Motor in order to go right) - 10 Go Left (which enables Right Motor in order to go left) - 11 Straight (both motors are enabled to go straight)

The register will store the Current State (which is given by S0 and S1) at any given time.

### Outputs

This is a Moore Finite State Machine, meaning that the output only depends on the current state. The outputs consist of two bits M0 and M1, each one turns on the motor on the right or left of the robot respectively.

### How to test

The truth table for the FSM can be found below.

I0	I1	S0	S1	S0+	S1+	M0	M1
0	0	0	0	0	0	0	0
0	0	0	1	1	1	0	1
0	0	1	0	1	1	1	0
0	0	1	1	1	1	1	1
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	1
0	1	1	0	0	1	1	0
0	1	1	1	0	1	1	1
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	1
1	0	1	0	1	0	1	0
1	0	1	1	1	0	1	1
1	1	0	0	1	1	0	0
1	1	0	1	1	1	0	1
1	1	1	0	1	1	1	0
1	1	1	1	1	1	1	1

## IO

#	Input	Output
0	clock	Enable Right Motor
1	reset	Enable Left Motor
2	none	segment c
3	none	segment d
4	none	segment e
5	none	segment f
6	Sensor Left Input	segment g
7	Sensor Right Input	dot

## 62 : FSK modem

- Author: Balint Kovacs
- Description: Very simple FSK modem, comparable to Bell 103
- GitHub repository
- HDL project
- Extra docs
- Clock: 12500 Hz
- External hardware:

### How it works

On the transmit side, it has two triangle wave synthesizers. The frequency of one can be altered by the data in pin, and the other one can be mixed in for DTMF.

On the receive side, it takes the input, heterodynes it with some square waves, and then filters I and Q over a 31 tap FIR filter. Finally, a phase detector is applied, that output is differentiated, and filtered for glitches. The noise immunity of this approach is questionable.

### How to test

- Set registers over SPI.
- Connect the “incoming” phone line to a comparator, and to samples\_in.
- Connect the “outgoing” phone line to an R-2R DAC, and to samples\_out.
- Connect a serial port to data\_in and data\_out.
- Enjoy squawking 300bps serial!

### IO

#	Input	Output
0	clock	data_out
1	reset	valid_out
2	cs_n	samples_out[0]
3	sck	samples_out[1]
4	mosi	samples_out[2]
5	data_in	samples_out[3]
6	none	samples_out[4]
7	samples_in	samples_out[5]

## 63 : QTChallenges

- Author: Jason Blocklove
- Description: This project implements 8 different benchmark circuits created 100% with ChatGPT-4.
- GitHub repository
- HDL project
- Extra docs
- Clock: 15000 Hz
- External hardware:

### How it works

This design implements a series of 8 benchmark circuits selectable by 3 bits of the design input, all of which written by ChatGPT-4. A series of prompts for each circuit were created which had ChatGPT design the module itself as well as a Verilog testbench for the design, and the design was considered finalized when there were no errors from simulation or synthesis. As much of the feedback as possible was given by the tools – Icarus Verilog for simulation and the Tiny Tapeout OpenLane/yosys toolchain for synthesis.

As a result of the designs being 100% created by ChatGPT, they all passed their ChatGPT-created testbenches, but several are not functionally correct as the generated testbenches are insufficient or incorrect. The best example of this is the **Dice Roller** benchmark, which has a constant output but passes its testbench fully.

The only design with a human-made testbench is the **Wrapper Module**, whose testbench uses dummy modules just to ensure the multiplexing works as expected. It seemed unrealistic to have ChatGPT create a testbench for every benchmark all in one given the token limits and how it struggled to make some of the standalone testbenches.

The complete transcripts of the ChatGPT conversations can be found at [https://github.com/JBlocklove/tt03-chatgpt-4\\_benchmarks/tree/main/conversations](https://github.com/JBlocklove/tt03-chatgpt-4_benchmarks/tree/main/conversations)

---

### Wrapper Module/Multiplexer

#### ChatGPT Prompt

I am trying to create a Verilog model for a wrapper around several benchmarks, specifically for the Tiny Tapeout project. It must meet the following specifications:

- Inputs:
  - io\_in (8-bits)
- Outputs:
  - io\_out (8-bits)

The design should instantiate the following modules, and use 3 bits of the 8-bit input to select which one will output from the module.

Benchmarks:

- shift\_register:
  - Inputs:
    - clk
    - reset\_n
    - data\_in
    - shift\_enable
  - Outputs:
    - [7:0] data\_out
- sequence\_generator:
  - Inputs:
    - clock
    - reset\_n
    - enable
  - Outputs:
    - [7:0] data
- sequence\_detector:
  - Inputs:
    - clk
    - reset\_n
    - [2:0] data
  - Outputs:
    - sequence\_found
- abro\_state\_machine:
  - Inputs:
    - clk
    - reset\_n
    - A
    - B
  - Outputs:

- 0
- [3:0] state
  
- binary\_to\_bcd:
  - Inputs:
    - [4:0] binary\_input
  - Outputs:
    - [3:0] bcd\_tens
    - [3:0] bcd\_units
  
- lfsr:
  - Inputs:
    - clk
    - reset\_n
  - Outputs:
    - [7:0] data
  
- traffic\_light:
  - Inputs:
    - clk
    - reset\_n
    - enable
  - Outputs:
    - red
    - yellow
    - green
  
- dice\_roller:
  - Inputs:
    - clk
    - reset\_n
    - [1:0] die\_select
    - roll
  - Outputs:
    - rolled\_number

How would I write a design that meets these specifications?

## Benchmark I/O Mapping

io_in[7:5]	Benchmark
000	Shift Register
001	Sequence Generator
010	Sequence Detector
011	ABRO
100	Binary to BCD
101	LFSR
110	Traffic Light
111	Dice Roller

## Expected Functionality

The top level module for the design is a wrapper module/multiplexer which allows the user to select which benchmark is being used for the output of the design. Using `io_in[7:5]`, the user can select which benchmark will output to the `io_out` pins.

This module was created after all of the other designs were finalized so their port mappings could be given

## Actual Functionality

The module functions as intended. This is the only module with a human-written testbench, as it seemed unrealistic to have ChatGPT create a full testbench that confirmed the module instantiations worked given how much it struggled with some of the other testbenches.

---

## Shift Register

### ChatGPT Prompt

I am trying to create a Verilog model for a shift register. It must meet the following specifications:

- Inputs:
  - Clock
  - Active-low reset
  - Data (1 bit)
  - Shift enable
- Outputs:
  - Data (8 bits)

How would I write a design that meets these specifications?

## Benchmark I/O Mapping

#	Input	Output
0	clk	Shifted data [0]
1	rst_n (async)	Shifted data [1]
2	data_in	Shifted data [2]
3	shift_enable	Shifted data [3]
4	Not used	Shifted data [4]
5	Select bit	Shifted data [5]
6	Select bit	Shifted data [6]
7	Select bit	Shifted data [7]

## Expected Functionality

The expected functionality of this shift register module is to shift the `data_in` bit in on the right side of the data vector on any rising `clk` edge where `shift_enable` is high.

## Actual Functionality

The module seems to function as intended.

---

## Sequence Generator

### ChatGPT Prompt

I am trying to create a Verilog model for a sequence generator. It must meet the following specifications:

- Inputs:
  - Clock
  - Active-low reset
  - Enable
- Outputs:
  - Data (8 bits)

While enabled, it should generate an output sequence of the following hexadecimal values and then repeat:

- 0xAF
- 0xBC
- 0xE2
- 0x78
- 0xFF
- 0xE2
- 0x0B
- 0x8D

How would I write a design that meets these specifications?

## Benchmark I/O Mapping

#	Input	Output
0	clk	Sequence Output [0]
1	rst_n (async)	Sequence Output [1]
2	Not used	Sequence Output [2]
3	Not used	Sequence Output [3]
4	enable	Sequence Output [4]
5	Select bit	Sequence Output [5]
6	Select bit	Sequence Output [6]
7	Select bit	Sequence Output [7]

## Expected Functionality

The expected functionality of this sequence generator is to output the following sequence, moving a step forward whenever the clk has a rising edge and the enable is high. Once the sequence has reached its end it should repeat.

0xAF  
0xBC  
0xE2  
0x78  
0xFF  
0xE2  
0x0B  
0x8D

## Actual Functionality

The module functions as intended.

---

## Sequence Detector

### ChatGPT Prompt

I am trying to create a Verilog model for a sequence detector. It must meet the following specifications:

- Inputs:
  - Clock
  - Active-low reset
  - Data (3 bits)
- Outputs:
  - Sequence found

While enabled, it should detect the following sequence of binary input values:

- 0b001
- 0b101
- 0b110
- 0b000
- 0b110
- 0b110
- 0b011
- 0b101

How would I write a design that meets these specifications?

### Benchmark I/O Mapping

#	Input	Output
0	clk	Not Used
1	rst_n (async)	Not Used
2	data[0]	Not Used
3	data[1]	Not Used
4	data[2]	Not Used
5	Select bit	Not Used

#	Input	Output
6	Select bit	Not Used
7	Select bit	Sequence Found

## Expected Functionality

The expected functionality of this sequence detector is to output a 1 if it receives the following sequence of data all on consecutive clock cycles.

```
0b001
0b101
0b110
0b000
0b110
0b110
0b011
0b101
```

## Actual Functionality

The module does not correctly detect the sequence. In trying to set the states to allow the sequence to overlap it instead skips the final value or outputs a 1 if the second to last value and final value are both 0b101.

---

## ABRO State Machine

### ChatGPT Prompt

I am trying to create a Verilog model for an ABRO state machine.  
It must meet the following specifications:

- Inputs:
  - Clock
  - Active-low reset
  - A
  - B
- Outputs:
  - O
  - State

Other than the main output from ABRO machine, it should output the current state of the machine for use in verification.

The states for this state machine should be one-hot encoded.

How would I write a design that meets these specifications?

## Benchmark I/O Mapping

#	Input	Output
0	clk	State [0]
1	reset_n (async)	State [1]
2	A	State [2]
3	B	State [3]
4	Not used	Output
5	Select bit	Not used
6	Select bit	Not used
7	Select bit	Not used

## Expected Functionality

The expected functionality of the ABRO (A, B, Reset, Output) state machine is to only reach the output state and output a 1 when both inputs A and B have been given before a reset. The order of the inputs should not matter, so long as both A and B are set.

## Actual Functionality

The module does not function fully as intended. If B is received before A then it works as intended, but if A is received first then it actually requires the sequence A, B, A in order to reach the output state. It also does not handle the case where A and B are set in the same cycle, instead interpreting it as if A was received first.

---

## Binary to BCD Converter

### ChatGPT Prompt

I am trying to create a Verilog model for a binary to

binary-coded-decimal converter. It must meet the following specifications:

- Inputs:
  - Binary input (5-bits)
- Outputs:
  - BCD (8-bits: 4-bits for the 10's place and 4-bits for the 1's place)

How would I write a design that meets these specifications?

## Benchmark I/O Mapping

#	Input	Output
0	binary_input[0]	BCD Ones [0]
1	binary_input[1]	BCD Ones [1]
2	binary_input[2]	BCD Ones [2]
3	binary_input[3]	BCD Ones [3]
4	binary_input[4]	BCD Tens [0]
5	Select bit	BCD Tens [1]
6	Select bit	BCD Tens [2]
7	Select bit	BCD Tens [3]

## Expected Functionality

The expected functionality of this module is to take a 5-bit binary number and produce a binary-coded-decimal output. The 4 most significant bits of the output encode to the tens place of the decimal number, the 4 least significant bits of the output encode the ones place of the decimal number.

## Actual Functionality

The module functions as intended.

---

## Linear Feedback Shift Register (LFSR)

### ChatGPT Prompt

I am trying to create a Verilog model for an LFSR. It must meet the following specifications:

- Inputs:
  - Clock
  - Active-low reset
- Outputs:
  - Data (8-bits)

The initial state should be 10001010, and the taps should be at locations 1, 4, 6, and 7.

How would I write a design that meets these specifications?

## Benchmark I/O Mapping

#	Input	Output
0	clk	Data Output [0]
1	rst_n (async)	Data Output [1]
2	Not used	Data Output [2]
3	Not used	Data Output [3]
4	Not used	Data Output [4]
5	Select bit	Data Output [5]
6	Select bit	Data Output [6]
7	Select bit	Data Output [7]

## Expected Functionality

The expected functionality of this module is to generate a pseudo-random output value from this LFSR based on the tap locations given in the prompt.

It was unspecified in the prompt, but originally it was expected that the LFSR would shift right as is standard amongst most documentation. It instead shifts to the left, which is not inherently incorrect but warrants mentioning.

## Actual Functionality

This module functions almost as expected, except the taps were placed on indices off-by-one. Rather than being at indices 1, 4, 6, and 7, they are at indices 0, 3, 5, and 6. This is still a valid LFSR, it is just not quite what was requested.

---

## Traffic Light State Machine

## ChatGPT Prompt

I am trying to create a Verilog model for a traffic light state machine. It must meet the following specifications:

- Inputs:
  - Clock
  - Active-low reset
  - Enable
- Outputs:
  - Red
  - Yellow
  - Green

The state machine should reset to a red light, change from red to green after 32 clock cycles, change from green to yellow after 20 clock cycles, and then change from yellow to red after 7 clock cycles.

How would I write a design that meets these specifications?

## Benchmark I/O Mapping

#	Input	Output
0	clk	Sequence Output [0]
1	rst_n (async)	Sequence Output [1]
2	Not used	Sequence Output [2]
3	enable	Sequence Output [3]
4	Not used	Sequence Output [4]
5	Select bit	Green
6	Select bit	Yellow
7	Select bit	Red

## Expected Functionality

The expected functionality of this module is to simulate the function of a timed traffic light. On a reset it outputs a red light, waits 32 clock cycles and then changes to a green light, waits 20 clock cycles and then changes to a yellow light, waits 7 clock cycles and then changes back to red. This should then repeat. If the enable is low, then it should pause the operation entirely and pick up again once the enable is brought high again.

## Actual Functionality

The module functions as intended.

---

## Dice Roller

### ChatGPT Prompt

I am trying to create a Verilog model for a simulated dice roller. It must meet the following specifications:

- Inputs:
  - Clock
  - Active-low reset
  - Die select (2-bits)
  - Roll
- Outputs:
  - Rolled number (up to 8-bits)

The design should simulate rolling either a 4-sided, 6-sided, 8-sided, or 20-sided die, based on the input die select. It should roll when the roll input goes high and output the random number based on the number of sides of the selected die.

How would I write a design that meets these specifications?

### Benchmark I/O Mapping

#	Input	Output
0	clk	Dice Roll [0]
1	rst_n (async)	Dice Roll [1]
2	die_select[1]	Dice Roll [2]
3	die_select[0]	Dice Roll [3]
4	roll	Dice Roll [4]
5	Select bit	Dice Roll [5]
6	Select bit	Dice Roll [6]
7	Select bit	Dice Roll [7]

## Expected Functionality

The expected functionality of this dice roller is to allow the user to select which die they would like to simulate rolling based on the following table:

die_select	Number of sides
00	4
01	6
10	8
11	20

When `roll` is high the module should output a new pseudo-random value in the range [1 - Number of sides]

## Actual Functionality

This module outputs 2 for the first two dice rolls and then consistently outputs a 1 regardless of what die is selected.

## How to test

Testing this design has some difficulties as there are several different functionalities that are selected between and not all work as expected.

For sequential designs the input vectors are expected to be given across consecutive clock cycles, as such for those designs `io_in[0]` will not be given. For designs with input bits that are unused, those positions will be shown as `x` to represent don't-cares.

The following tables give input test vectors and their expected outputs:

## Shift Register

Given Input	Expected Output	Comment
000x000	00000000	Reset
000x001	00000000	Disabled
000x011	00000000	Shift in 0
000x111	00000001	Shift in 1
000x111	00000011	Shift in 1
000x101	00000110	Shift in 0
000x111	00001101	Shift in 1
000x011	00001101	Disabled

Given Input	Expected Output	Comment
000x110	00000000	Reset

## Sequence Generator

Given Input	Expected Output	Comment
0010xx0	10101111	Reset
0010xx1	10101111	Disabled
0011xx1	10101111	Enabled
0011xx1	10111100	Enabled
0011xx1	11100010	Enabled
0011xx1	01111000	Enabled
0011xx1	11111111	Enabled
0011xx1	11100010	Enabled
0011xx1	00001011	Enabled
0010xx1	10001101	Disabled
0011xx1	10001101	Enabled
0011xx0	10101111	Reset

## Sequence Detector

This module does not function as intended. These vectors are meant to represent what will be expected from the hardware, not what is expected from the initial design description.

Given Input	Expected Output	State
0100000	00000000	S0
0100001	00000000	S0
0100011	00000000	S0 -> S1
0101011	00000000	S1 -> S2
0101101	00000000	S2 -> S3
0100001	00000000	S3 -> S4
0101101	00000000	S4 -> S5
0101101	00000000	S5 -> S6
0100111	10000000	S6 -> S0
0101011	00000000	S0
0100001	00000000	S0
0100011	00000000	S0 -> S1
0101011	00000000	S1 -> S2

Given Input	Expected Output	State
0101101	00000000	S2 -> S3
0100001	00000000	S3 -> S4
0101101	00000000	S4 -> S5
0101101	00000000	S5 -> S6
0101011	10000000	S6 -> S7
0100111	00000000	S7 -> S2

## ABRO State Machine

Given Input	Expected Output	State
011x000	00000001	IDLE
011x001	00000001	IDLE
011x011	00000010	IDLE -> A
011x101	00000100	A -> B
011x111	00011000	B -> O
011x111	00000001	O -> IDLE
011x001	00000001	IDLE
011x101	00000100	IDLE -> B
011x011	00000100	B -> O
011x011	00000001	O -> IDLE

## Binary to BCD Converter

This design is purely combinational, so all 8-bits are included in the input vector. Clock cycles do not matter for this functionality.

Given Input	Expected Output	Comment
10000001	00000001	1
10011111	00110001	31
10011010	00100110	26
10010011	00011001	19

## LFSR

As the LFSR only has a clock and reset input, the table will only give the first vectors for resetting and then setting, but it will show the first several cycles of the LFSR from the initial vector.

Given Input	Expected Output	Comment
0000000	10001010	
0000001	00010101	
...	00101011	
...	01010111	
...	10101110	
...	01011100	
...	10111001	

## Traffic Light State Machine

Given the nature of this benchmark being dependent on counting clock cycles, it seems unhelpful to provide a suite of test vectors and their expected outputs. A limited set of vectors is provided with comments on their general functionality, but full testing will be left to the user's discretion.

Given Input	Expected Output	Comment
110x0x0	10000000	Reset: Sets the state machine back to the initial (RED) state
110x0x1	Depends	Disabled: Holds the FSM in its current state, pauses clock cycle count
110x1x1	Depends	Enabled: Resumes clock cycle counting and allows state transitions

## Dice Roller

This module does not function as intended. These vectors are meant to represent what will be expected from the hardware, not what is expected from the initial design description.

Given Input	Expected Output	Comment
1110000	00000000	Reset
1111001	00000010	d4
1110001	00000010	d4
1111101	00000001	d6
1110101	00000001	d6
1111011	00000001	d8
1110011	00000001	d8
1111111	00000001	d20
1110111	00000001	d20
1111001	00000001	d4
1111001	00000001	d4

## IO

#	Input	Output
0	Depends on benchmark	Depends on benchmark
1	Depends on benchmark	Depends on benchmark
2	Depends on benchmark	Depends on benchmark
3	Depends on benchmark	Depends on benchmark
4	Depends on benchmark	Depends on benchmark
5	benchmark_select[0]	Depends on benchmark
6	benchmark_select[1]	Depends on benchmark
7	benchmark_select[2]	Depends on benchmark

## 64 : HiddenCPU

- Author: HiddenRoom
- Description: Basic 8 bit CPU.
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware: No absolute requirements but external memory and hardware for addressing is not a bad idea

### How it works

Each instruction is executed in one clock cycle.

Instructions are inputed on io\_in[2:7].

### How to test

After reset, registers will initialize, in order of ascending address, to 00000000b, 00000001b, 00000010b, 00000011b

### IO

#	Input	Output
0	clock	bit zero of hardwired output reg
1	reset	number at address 11b or pc
2	opcode bit zero	bit one of hardwired output reg
3	opcode bit one	number at address 11b or pc
4	result reg address bit zero	bit two of hardwired output reg
5	result reg address bit one	number at address 11b or pc
6	non result operand reg address bit zero	bit three of hardwired output reg
7	non result operand reg address bit one	number at address 11b or pc

## 65 : Simple UART interface

- Author: Aleksandr Zlobin
- Description: UART interface with access to internal registers
- GitHub repository
- HDL project
- Extra docs
- Clock: 12500 Hz
- External hardware:

### How it works

Each transaction consists of two bytes. First byte - address and r/w bit. Second byte - data to be written (write transaction), or data read from internal registers (read transaction).

### How to test

After reset LEDs should show 3 horizontal segments.

Read transaction: first byte is send by host, device will reply with one data byte. List of implemented commands:

1. Read 15:18 bits of signature: send 0x00, device will reply with 0xDA byte
2. Read 7:0 bits of signature: send 0x02, device will reply with 0xDE byte
3. Read internal register: send 0x04, device will reply with 0x00 byte (initial value of internal register is 0)
4. Read led register: send 0x06, device will reply with 0x49 byte (initial value of led register is 0x49)
5. Read status of input4-input7: send 0x08, device will reply with byte containing state of the inputs.

Write transaction: host should send two bytes first - command, second data:

1. Write to internal register: send 0x05, send byte you want to write
2. Write to LED register: send 0x07, send byte you want to write to LED register

### IO

#	Input	Output
0	clock	segment a
1	reset	segment b
2	rx	segment c
3	input4	segment d
4	input5	segment e
5	input6	segment f
6	input7	segment g
7	input8	tx

## 66 : MSF Clock

- Author: Jamie Wood & Daniel Cannell
- Description: MSF radio clock
- GitHub repository
- HDL project
- Extra docs
- Clock: 12500 Hz
- External hardware:

### How it works

The MSF radio clock bitstream is synchronised, decoded, and the time displayed on an LED display driven by a shift-register.

### How to test

- Provide 12.5kHz clock to the clock pin, e.g. using the clock divider set to 0.
- Connect the data input to a WWVB / MSF receiver module.
- Connect the inverted input high if the module outputs a 1 when there is no carrier.
- Connect the shift\_date input to a button. If held, the display will show date instead of time.
- Connect the shift register outputs to a 42-bit shift register chain. Bit 41 is shifted out first.

Bit	Segment
41-35	Hours tens G-A (41=G, 35=A)
34-28	Hours ones G-A as above
27-21	Minutes tens G-A as above
20-14	Minutes ones G-A as above
13-7	Seconds tens G-A as above
6-0	Seconds ones G-A as above

Time and date are always both shifted out, so an 84-bit shift register will show both simultaneously.

#	Input	Output
0	clock	shift_clk
1	reset	shift_data
2	data	shift_latch
3	inverted	none
4	shift_date	none
5	none	none
6	none	none
7	none	none

## 67 : Hamming(7,4) encoder decoder

- Author: Robbert
- Description: Encodes and decodes data using Hamming 7,4 codes.
- GitHub repository
- HDL project
- Extra docs
- Clock: Hz
- External hardware:

### How it works

Encodes and decodes data using Hamming 7,4 codes.

### How to test

See `src/test.py`

### IO

#	Input	Output
0	word / codeword	codeword / word
1	word / codeword	codeword / word
2	word / codeword	codeword / word
3	word / codeword	codeword / word
4	none / codeword	codeword / error
5	none / codeword	codeword / none
6	none / codeword	codeword / none
7	mode (0: encode, 1: decode)	none

## 68 : I2S receiver, data mix and transmitter

- Author: Clemens Nasenberg
- Description: Mix two I2S streams according to selection
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware: I2S Controller/Transmitter/Receiver

### How it works

Read in 2 Channels of I2S streams, mix them according to input bits (mute, I2S input 1 or 2 or both) and send them out again. For adding the LSB of the data is truncated. This should support standard I2S transmitter/receiver.

<https://www.sparkfun.com/datasheets/BreakoutBoards/I2SBUS.pdf>

### How to test

You need to connect the word select signal (WS), continuous serial clock (SCK) and serial data for channel 1 and 2 (SD\_CH1 SD\_CH2). Configure channel\_sel (b00 is mute, b01 channel 1, b10 channel 2 and b11 is adding ch1 and ch2), reset, then let and I2S master drive the signals according to the standard and receive data on the output SD\_OUT. Some other IC needs to act as clock master and provide the clocks and signals.

### IO

#	Input	Output
0	sck	wsd
1	reset	wsp
2	ws	sd_out
3	sd_ch1	none
4	sd_ch2	none
5	channel_sel	none
6	none	none
7	none	none

## 69 : S4GA: Super Slow Serial SRAM FPGA

### S4GA: Super Slow Serial SRAM FPGA

Datapath – N=283 K=3,4,5,6,7,8-LUTs I=2 O=7

*Tiny Tapeout 2, 2022-11-22*

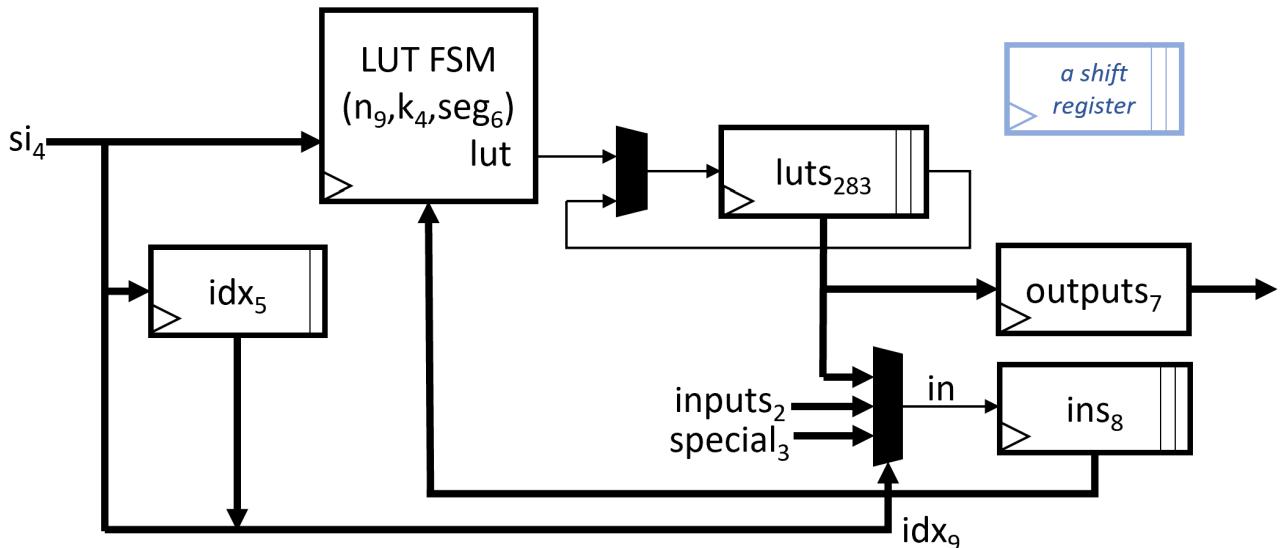


Figure 18: picture

- Author: Jan Gray(TT02), Marco Merlin (TT03)
- 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: 0 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

## 70 : 7 Segment Random Walk

- Author: Richard Miller
- Description: Random walk around the 7 segment display
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

Segments of the 7seg display are activated in a sequence controlled by a finite state machine. The state is represented by 14 flipflops. Each segment has two flipflops, one for each direction of the walk. Where a segment has two possible successors, a mux selects between them, controlled by the output from a linear feedback shift register acting as a random number generator. The FSM is self initialising so a reset signal isn't required: the number of true flipflops is summed, and if the total isn't exactly 1 the state will be initialised to have a single segment and direction active. A clock divider is used to slow the walk to a visible speed. This is done by gating the flipflop state transitions with a pulse generated every  $2^N$  clock cycles (where N is a 4-bit number set by the DIP switches.) The pulses are obtained by using a tree of muxes controlled by the bits of N to select one carry bit from within a 16-bit counting register.

### How to test

Use switches 5-8 (LSB is switch 8) to set a number N from 0 to 15; the clock will be divided by  $2^N$ . To test the clock, set switch 4 on and the 7seg dot will flash at half the clock speed.

### IO

#	Input	Output
0	clock	segment a
1	none	segment b
2	none	segment c
3	enable flashing dot	segment d
4	clock divisor bit 3 (MSB)	segment e
5	clock divisor bit 2	segment f
6	clock divisor bit 1	segment g
7	clock divisor bit 0 (LSB)	dot

## 71 : Tiny binarized neural network

- Author: ReJ aka Renaldas Zioma
- Description: 8 neurons
- GitHub repository
- HDL project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

TODO

### How to test

TODO

### IO

#	Input	Output
0	clock	output neuron 1
1	setup	output neuron 2
2	in0 / in parameter (if setup is high)	output neuron 3
3	in1	output neuron 4
4	in2	output neuron 5
5	in3	output neuron 6
6	in4	output neuron 7
7	in5	output neuron 8 / out parameter (if setup is high)

## 72 : RTL Locked QTCore-A1

- Author: Luca Collini and Hammond Pearce
- Description: A RTL locked accumulator-based 8-bit microarchitecture designed via GPT-4 conversations.
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware: Any microcontroller with an SPI peripheral

### How it works

The original QTCore-A1 from Hammond Pearce is a basic accumulator-based 8-bit microarchitecture (with an emphasis on the micro). It is a Von Neumann design (shared data and instruction memory). Its original specs can be found here: <https://github.com/kiwihi/tt03-verilog-qtcoreA1>

In order to lock the design, I had to make some space so I cut off 2 (8bit) memory registers (as a 16 bit register was added to store the locking key) and the memory mapped constants. I locked the ALU with 6 bits, locking 1 operation an 3 constants, and the control unit with 10 bits, locking an 8 bit constnt used twice and 2 branches. This locking techniques are the same one first presented in ASSURE: <https://arxiv.org/abs/2010.05344>

I also locked the ISA to ALU opcodes module using a novel RTL locking technique to lock case statements. The case variable is xored with the locking key and all case constants are replaced with the results of the original value xored with the right key. In this way the case statement works as originally intended only with the right key. To implement this locking technique I used ChatGPT with the GPT4 version. It was able to modify the `isa_to_alu_opcode` module by describing the technique, the key value to use, and the original module. The structure was correct from the first response, some additional back and forth was required for it to get the right values for the case constants (apparently gpt4 struggles a bit with xor operations).

The correct key is provided to allow everyone to use this design: 1011 1111 1111 1001  
We store the key in a scan chained register. In order to have the design working, your assembly should end with this two lines:

```
15: DATA 249      ; logic locking unlock key
16: DATA 191      ; logic locking unlock key
```

## 7seg

Tiny Tapeout 3 has a 7-segment display on the board. To make it useful with the QTCore-A1, the processor helpfully includes the bit patterns stored at addresses 16-26. Address 15 contains the Address of the 0 pattern. The easiest way to make use of these is with the LDAR instruction. Here's an example snippet, which assumes a value between 0 and 9 is stored at address 13:

```
0: LDA 13 ; load the value we want to display
1: ADD 15 ; add it to the constant 15 to get
;           the 7 segment pattern offset
2: LDAR    ; load the 7 segment pattern
3: STA 14 ; emit the 7 segment pattern
```

## How to test

The processor will not run unless a program is scanned into it. Fortunately, this is easy using the SPI peripheral of an external microcontroller.

## Wiring the SPI

The QTCore-A1 is designed to be connected to an SPI peripheral along with two chip selects. This is because we use the SPI SCK as the clock for both the scan chain and the processor.

Connect the I/O according to the provided wiring table. Then, set the SPI peripheral to the following settings:

- SPI Mode 0
- 8-bit data
- MSB first
- A very slow clock (I used 1kHz)
- The scan chain chip select is active low
- The processor chip select is active low

## Processor Memory Map

Address	Description
0-13	14 bytes of general purpose Instruction/Data memory
14	I/O: {OUT[7:1], IN[0]}
15	Constant value: 16 (See "7seg" note below)
16	Constant value: Bits for 7seg "0"

Address	Description
17	Constant value: Bits for 7seg "1"
18	Constant value: Bits for 7seg "2"
19	Constant value: Bits for 7seg "3"
21	Constant value: Bits for 7seg "4"
22	Constant value: Bits for 7seg "5"
23	Constant value: Bits for 7seg "6"
24	Constant value: Bits for 7seg "7"
25	Constant value: Bits for 7seg "8"
26	Constant value: Bits for 7seg "9"
27-31	Constant value: 1

## Loading a program

During the scan chain mode (when the scan enable is low), the entire processor acts as a giant shift register, with the bits in the following order:

All elements of the scan chain are presented MSB first.

- scan\_chain[2:0] - 3-bit state register
- scan\_chain[7:3] - 5-bit PC
- scan\_chain[15:8] - 8-bit Instruction Register
- scan\_chain[23:16] - 8-bit Accumulator
- scan\_chain[31 :- 8] - Memory[0]
- scan\_chain[39 :- 8] - Memory[1]
- scan\_chain[47 :- 8] - Memory[2]
- scan\_chain[55 :- 8] - Memory[3]
- scan\_chain[63 :- 8] - Memory[4]
- scan\_chain[71 :- 8] - Memory[5]
- scan\_chain[79 :- 8] - Memory[6]
- scan\_chain[87 :- 8] - Memory[7]
- scan\_chain[95 :- 8] - Memory[8]
- scan\_chain[103 :- 8] - Memory[9]
- scan\_chain[111 :- 8] - Memory[10]
- scan\_chain[119 :- 8] - Memory[11]
- scan\_chain[127 :- 8] - Memory[12]
- scan\_chain[135 :- 8] - Memory[13]
- scan\_chain[143 :- 8] - Memory[14]
- scan\_chain[159 :- 16] - (Locking Key)

Example program:

```
; This program tests the btn and LEDs
```

```

; will switch between 0 and 7 pressing the button
;
; BTNs and LEDS are at address 17, btn at LSB
0: ADDI 14 ; load the btn and LEDS
1: STA 14 ; load the btn and LEDS
2: LDA 14 ; load the btn and LEDS
3: AND 12 ; mask the btn
4: BNE_BWD ; if btn&1 is not zero then branch back to 0
5: LDA 13 ; load the btn and LEDS
6: STA 14 ; load the btn and LEDS
7: LDA 14 ; load the btn and LEDS
8: AND 12 ; mask the btn
9: BEQ_BWD ; if btn&1 is zero then branch back to 3
;
; the button has now done a transition from low to high
;
10: CLR
11: JMP
;
; data
12: DATA 1;
13: DATA 126 ;
15: DATA 249      ; logic locking unlock key
16: DATA 191      ; logic locking unlock key

```

C code to load the previously-discussed example program (e.g. via the STM32 HAL) is provided:

```

//The registers are presented in reverse order to
// the table as we load them MSB first.
uint8_t program_led_btn[21] = {
    0b10111111, //MEM[16]
    0b11111001, //MEM[15]
    0b00000000, // IOREG
    0b01111110, //MEM[13]
    0b00000001, //MEM[12]
    0b11110000, //MEM[11]
    0b11111101, //MEM[10]
    0b11110011, //MEM[9]
    0b10001100, //MEM[8]
    0b00001110, //MEM[7]
    0b00101110, //MEM[6]
    0b00001101, //MEM[5]

```

```

        0b11110101, //MEM[4]
        0b10001100, //MEM[3]
        0b00001110, //MEM[2]
        0b00101110, //MEM[1]
        0b11101110, //MEM[0]
        0b00000000, //ACC
        0b00000000, //IR
        0b00000001 //PC[5bit], CU[3bit]
};

//... this will go in the SPI peripheral initialization code
hspi1.Init.Mode = SPI_MODE_MASTER;
hspi1.Init.Direction = SPI_DIRECTION_2LINES;
hspi1.Init.DataSize = SPI_DATASIZE_8BIT;
hspi1.Init.CLKPolarity = SPI_POLARITY_LOW;
hspi1.Init.CLKPhase = SPI_PHASE_1EDGE;
hspi1.Init.NSS = SPI_NSS_SOFT;
//...
hspi1.Init.FirstBit = SPI_FIRSTBIT_MSB;
hspi1.Init.TIMode = SPI_TIMODE_DISABLE;
//...

//... this will go in your main or similar
uint8_t *program = program_led_btn;
//on the first run, scan_out will give us the reset value of the
// processor
HAL_GPIO_WritePin(SPI_SCAN_CS_GPIO_Port, SPI_SCAN_CS_Pin, 0);
HAL_Delay(100);
HAL_SPI_TransmitReceive(&hspi1, program, scan_out, 21, HAL_MAX_DELAY);
HAL_Delay(100);
HAL_GPIO_WritePin(SPI_SCAN_CS_GPIO_Port, SPI_SCAN_CS_Pin, 1);
//we can check if the program loaded correctly by immediately scanning
// it back in again, it will be unloaded to scan_out
HAL_GPIO_WritePin(SPI_SCAN_CS_GPIO_Port, SPI_SCAN_CS_Pin, 0);
HAL_Delay(100);
HAL_SPI_TransmitReceive(&hspi1, program, scan_out, 21, HAL_MAX_DELAY);
HAL_Delay(100);
HAL_GPIO_WritePin(SPI_SCAN_CS_GPIO_Port, SPI_SCAN_CS_Pin, 1);
//Check if the program loaded correctly
for(int i = 0; i < 21; i++) {
    if(program[i] != scan_out[i]) {
        while(1); //it failed
}

```

```
    }  
}
```

## Running a program

Once the program is loaded (using the above code or similar), we can run a program. This is as easy as providing a clock signal to the processor and setting the processor enable line high. We do this using the SPI as follows:

```
uint8_t dummy; //a dummy value  
HAL_GPIO_WritePin(SPI_PROC_CS_GPIO_Port, SPI_PROC_CS_Pin, 0);  
//...  
//run this in a loop  
HAL_SPI_TransmitReceive(&hspi1, &dummy, &dummy, 1, HAL_MAX_DELAY);
```

The processor will ignore any value being shifted in on the MOSI data line during operation. However, it does provide a nice feature in that the processor will emit the current value of the processor\_halt signal on the MISO line. This means that we can improve the code to run this process in a loop and catch when the program reaches a HLT instruction:

```
HAL_Delay(100);  
HAL_GPIO_WritePin(SPI_PROC_CS_GPIO_Port, SPI_PROC_CS_Pin, 0);  
dummy = 0;  
while(1) {  
    HAL_SPI_TransmitReceive(&hspi1, &dummy, &dummy, 1, HAL_MAX_DELAY);  
    if(dummy == 0xFF)  
        break;  
}  
HAL_GPIO_WritePin(SPI_PROC_CS_GPIO_Port, SPI_PROC_CS_Pin, 1);  
HAL_Delay(100);
```

Of course, the example program does not HLT, so we will not reach this point in this code. But, it works for other programs that do contain a HLT instruction.

Once you have the provided example program running, you will be able to press the button and see how the LEDs are toggled.

## IO

#	Input	Output
0	clock - connect to an SPI SCK.	general purpose output 0 (e.g. LED segment a). This output comes from the I/O register bit 1.
1	reset (active high)	general purpose output 1 (e.g. LED segment b). This output comes from the I/O register bit 2.
2	scan enable (active low) - connect to an SPI chip select.	general purpose output 2 (e.g. LED segment c). This output comes from the I/O register bit 3.
3	processor enable (active low) - connect to an SPI chip select.	general purpose output 3 (e.g. LED segment d). This output comes from the I/O register bit 4.
4	scan data in - connect to an SPI MOSI.	general purpose output 4 (e.g. LED segment e). This output comes from the I/O register bit 5.
5	general purpose input (e.g. Button). This input will be provided to the I/O register bit 0.	general purpose output 5 (e.g. LED segment f). This output comes from the I/O register bit 6.
6	none	general purpose output 6 (e.g. LED segment g). This output comes from the I/O register bit 7.
7	none	scan data out - connect to an SPI MISO.

## 73 : Arbiter Game

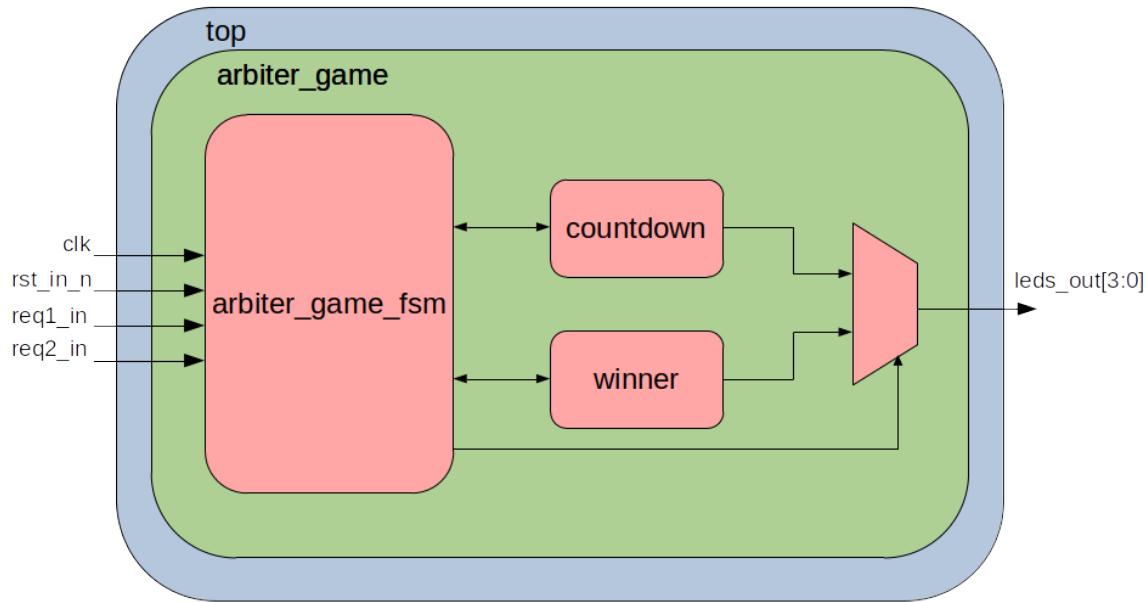


Figure 19: picture

- Author: Martin A. Heredia
- Description: This is a game for 2 players. After a countdown, the player who press his/her button first wins. Countdown and winner should be displayed in output leds.
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware: 2 active low pushbuttons for players + 1 active low pushbutton for reset. 4 LEDs connected to outputs.

### How it works

This is a 2 players game where every player has an active low pushbutton assigned (req1\_in and req2\_in inputs to top level). After applying reset to the circuit, a countdown will be displayed at the output ports (ideally connected to some LEDs). Then, when the last LED stops toggling, the players will press the buttons. The first player to press the button wins and this will be displayed in output LEDs.

## How to test

- Connect 2 active low pushbuttons to req1\_in and req2\_in (io\_in[2] and io\_in[3] respectively).
- Connect 4 LEDs to leds\_out[3:0] (io\_out[3:0]).
- Apply 1 KHz clock at clk port (io\_in[0]).
- Apply an active low reset (io\_in[1]) to start the game.

## IO

#	Input	Output
0	clk	leds_out[0] / segment a
1	rst_in_n (active low)	leds_out[1] / segment b
2	req1_in (active low)	leds_out[2] / segment c
3	req2_in (active low)	leds_out[3] / segment d
4	none	none
5	none	none
6	none	none
7	none	none

## 74 : Frequency Divider

- Author: Tanish Khanchandani
- Description: Can enter a binary number and the clock is divided by  $2^{\text{number entered}}$
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

Enter a binary number using Inputs 2-5 and the clock on output 8 should be divided.

### How to test

Enter a binary number and probe output 8 to see the clock frequency get divided.

### IO

#	Input	Output
0	clock	clock
1	Binary Input 1	none
2	Binary Input 2	none
3	Binary Input 3	none
4	Binary Input 4	none
5	none	none
6	none	none
7	none	Divided Frequency

## 75 : FullAdderusing4is1

- Author: Marushika Suri , Siya Sharma , Rudakshi Arora
- Description: This project is used to add 3-bit binary numbers
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 1 Hz
- External hardware:

### How it works

Explain how your project works : A full adder is a digital circuit that performs the addition of three binary digits: two inputs and a carry bit from a previous stage. The output of a full adder is the sum of the three inputs and a carry bit for the next stage. A 4:1 multiplexer (mux) is a digital circuit that selects one of four input signals to pass through to the output based on the value of two select inputs. Here, we are implementing a full adder using two 4:1 mux. We have taken 3 inputs:A,B,Cin. The output of first mux(i.e.,Sum function) is on OUT0 Pin and the output of second mux(Carry) is connected to OUT1 Pin, connected with a seven-segment display. The use of multiplexers simplifies the implementation of the full adder circuit.

### How to test

Explain how to test your project : At the input, we have used 4 pins. Pin IN0 is for the clock. Pin IN1, IN2, IN3 for A, B,Cin respectively. Here, we will set the clock at “1”, we’ll provide sum and carry at the OUT0 and OUT1 pins of the output respectively. Once we start the simulation, we can add two 3-bit binary numbers and the output will be displayed at the 7-segment display.

### IO

#	Input	Output
0	in1(Connected to A input)	segment a(Connected to OUT0 performing the Sum function)
1	in2(Connected to Select line S1)	segment b(Connected to OUT1 performing the Carry function)
2	in3(Connected to Select line S2)	segment c
3	none	segment d
4	none	segment e
5	none	segment f

#	Input	Output
6	none	segment g
7	none	dot

## 76 : BCDtoDECIMAL

- Author: Giresh and Aditya
- Description: The objective of this project is to create a circuit that can convert Binary Coded Decimal (BCD) numbers to their decimal equivalents. This will enable users to easily convert BCD numbers, which are commonly used in electronic devices, into the decimal format that is more commonly understood.
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

Explain how your project works : A BCD to decimal generator on a seven-segment display works by converting Binary Coded Decimal (BCD) input signals into the corresponding decimal digits that can be displayed on a seven-segment display. The BCD input consists of four binary digits, where each digit corresponds to a decimal number between 0 and 9. These four digits are converted into their corresponding decimal values by using a decoder circuit. The decoder circuit uses logic gates to decode the BCD input signals into the corresponding outputs for each of the seven segments in the seven-segment display. Each segment is either turned on or off, depending on the decimal value of the BCD input.

### How to test

Explain how to test your project : To test this project we have connected the output pins of this circuit to the seven segment pins. To display the decimal digit on the seven-segment display, the corresponding segments are turned on or off, depending on the decimal value of the BCD input. For example, if the BCD input is 0010, the decoder circuit will turn on segment A, segment F, and segment G to display the decimal digit 2 on the seven-segment display.

### IO

#	Input	Output
0	{'B': 'connected to IN0'}	{'segment a': 'connected to OUT0'}
1	{'C': 'connected to IN1'}	{'segment b': 'connected to OUT1'}
2	{'D': 'connected to IN2'}	{'segment c': 'connected to OUT2'}
3	{'E': 'connected to IN3'}	{'segment d': 'connected to OUT3'}

#	Input	Output
4	none	{'segment e': 'connected to OUT4'}
5	none	{'segment f': 'connected to OUT5'}
6	none	{'segment g': 'connected to OUT6'}
7	none	dot

## 77 : AI Decelerator

- Author: Hunter Scott
- Description: A 2x2 matrix multiplier, guaranteed to slow down your AI model training.
- GitHub repository
- HDL project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

Just performs arithmetic to do the matrix multiplication and outputs the result.

### How to test

Use the DIP switches to set the value of each position in the two matrices you're multiplying. That means that each position in each input matrix can only contain a zero or one. The output is 4 groups of 2 bits, each group representing one of the positions in the output matrix.

### IO

#	Input	Output
0	7 is Matrix 1, top left	7 is Result matrix, top left, bit 1
1	6 is Matrix 1, top right	6 is Result matrix, top left, bit 0
2	5 is Matrix 1, bottom left	5 is Result matrix, top right, bit 1
3	4 is Matrix 1, bottom right	4 is Result matrix, top right, bit 0
4	3 is Matrix 2, top left	3 is Result matrix, bottom left, bit 1
5	2 is Matrix 2, top right	2 is Result matrix, bottom left, bit 0
6	1 is Matrix 2, bottom left	1 is Result matrix, bottom right, bit 1
7	0 is Matrix 2, bottom right	0 is Result matrix, bottom right, bit 0

## 78 : 3BitParallelAdder

- Author: Anish Paul , Pancham Mittal , Ramandeep
- Description: This project is used to Add and Subtract 3-bit binary inputs
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

Explain how your project works : A 3-bit binary parallel adder-subtractor is a digital circuit that can perform both addition and subtraction of two 3-bit binary numbers in parallel. The circuit has two main inputs, which are the two 3-bit binary numbers to be added or subtracted. These inputs are represented by A2, A1, A0, and B2, B1, B0. The circuit also has a carry-in (Cin) input, which is used during addition. To understand how the 3-bit binary parallel adder-subtractor works, let's first look at how addition works in binary. In binary addition, each bit in the two binary numbers is added along with the carry from the previous addition. The carry from the current addition is then passed to the next bit addition. In our circuit the inputs IN1,IN2 and IN3 is represented as the inputs A0(Least significant bit),A1 and A2(Most significant bit). IN4,IN5,IN6 are represented as inputs B0(Least significant bit),B1 and B2(Most significant bit). IN7 is given as Cin. To make the circuit work as an adder the Carry in(Cin) is given logic 0. Similarly, the circuit can work as a 2's complement subtractor, to implement this we have connected the Cin to inputs B2,B1 B0 with an XOR gate. When the Carry in input (Cin) is given Logic 1 the XOR gate will give the 2's Complement of B. Then the result of addition between A and B will be the difference between them.

### How to test

Explain how to test your project : For testing the project , we have connected the output pins of the circuit to a seven segment display. For this we have connected the S0 (least significant bit) pin(OUT0) to the seven segment pin A, S1 pin(OUT1) to seven segment pin B, S3 (Most significant bit) pin (OUT2) to seven segment pin C and the Carry Out pin(OUT3) to seven segment pin D . The Seven Segment Display will glow in accordance to the Output on the respective pins.

### IO

#	Input	Output
0	clock	segment a - S0
1	reset	segment b - S1
2	IN1 - A0	segment c - S2
3	IN2 - A1	segment d - Cout
4	IN3 - A2	segment e
5	IN4 - B0	segment f
6	IN5 - B1	segment g
7	IN6 - B2	dot

## 79 : Asynchronous 3-Bit Down Counter

- Author: Dikshant, Mohit, Sanidhya
- Description: This Project works as a down counter which counts from 7 to 0
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

See The Wokwi gate and simulation. \* The sequence in which the counter works is as follows : 7: 0111 3: 0011 5: 0101 1: 0001 6: 0110 2: 0010 4: 0100 0: 0000 Here's how the counter works : first, all the three flip flops are set to 1 then for changing the inputs i have used the switch 1 of the Dip switch. \* When the count signal is triggered, the first flip-flop (least significant bit) toggles from 1 to 0, which produces the binary number 0111 (decimal 7). \* The second count signal should not trigger a change in the counter value, as the next count is 3, not 6. \* The third count signal triggers a change in the counter value, where the second flip-flop toggles from 1 to 0, which produces the binary number 0101 (decimal 5). \* The fourth count signal should not trigger a change in the counter value, as the next count is 1, not 4. \* The fifth count signal triggers a change in the counter value, where the first flip-flop toggles from 0 to 1, and the third flip-flop toggles from 0 to 1, which produces the binary number 0110 (decimal 6). \* The sixth count signal should not trigger a change in the counter value, as the next count is 2, not 5. \* The seventh count signal triggers a change in the counter value, where the second flip-flop toggles from 0 to 1, which produces the binary number 0100 (decimal 4). \* The eighth count signal triggers a change in the counter value, where all three flip-flops toggle from 0 to 1, which produces the binary number 0000 (decimal 0). \* The counter stays at 000 until the count signal is triggered again, which causes all three flip-flops to be set back to 1, and the counting process starts again from the beginning of the sequence. In summary, the 3-bit asynchronous down counter counts down in the sequence 7, 3, 5, 1, 6, 2, 4, 0 using three flip-flops and asynchronous inputs to trigger each flip-flop in the sequence.

### How to test

For Testing this project, the user has to use the switch 1 of the dip switch to give different inputs and then get the desired output on the 7-segment display.

### IO

#	Input	Output
0	clock	segment a - OUT0
1	reset	segment b - OUT1
2	I0 - Input	segment c - OUT2
3	I1 - None	segment d - OUT3
4	I2 - None	segment e - OUT4
5	I3 - None	segment f - OUT5
6	I4 - None	segment g - OUT6
7	Unused	dot - None

## 80 : Ring oscillator with skew correction

- Author: Daniel Wisehart
- Description: Fixing the problem of skew in a ring oscillator with a differential clock ring.
- GitHub repository
- HDL project
- Extra docs
- Clock: any Hz
- External hardware:

### How it works

Multiple buffers connected in series have a problem: they skew the incoming waveform on the output because the rise and fall time of the FETs that buffer the signal have uneven rise and fall times. The solution used here is to use a differential clock signal with two rings of buffers running opposite polarity that need to both have changed before the output is changed.

The clock ring is then tested by driving grey counters to see if the resulting output is stable and at an expected rate. The original design—which is still available for testing via cocotb—used multiple rings of different lengths to detect skew with differential timings, but that extended design would not fit into hardware.

### How to test

After not reset is desasserted, set the bits of select to read one of the grey counters. The outputs should change each half second with something like “7 6 2 .” over two seconds and then repeats. If a value is not a good one, a minus sign is shown instead. A 762. says the divided ratio is 762:1. The table below shows what is being measured and what the divide ratio is. The clock number is the number of stages in its ring buffer. In hardware, the only clock that is running is clock 097, which actually has 93 stages.

Select input	counter	divisor
000001	clock 005 vs scan clock	01,300
000011	clock 011 vs scan clock	00,600
000111	clock 023 vs scan clock	00,290
000110	clock 047 vs scan clock	00,140
001110	clock 097 vs scan clock	00,069
001100	clock 005 vs clock 011	01,500
011100	clock 005 vs clock 023	03,000
011000	clock 005 vs clock 047	06,000
111000	clock 011 vs clock 023	12,000
110000	clock 011 vs clock 047	01,500
100000	clock 023 vs clock 047	03,000
default	digits 4 5 6 . repeat	n/a

## IO

#	Input	Output
0	clock	output[7] LED seg a
1	not reset	output[6] LED seg b
2	select[5]	output[5] LED seg c
3	select[4]	output[4] LED seg e
4	select[3]	output[3] LED seg f
5	select[2]	output[2] LED seg g
6	select[1]	output[1] LED seg h
7	select[0]	output[0] LED point

## 81 : 3 bit multiplier

- Author: Ananya
- Description: 3 bit multiplier
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

Explain how your project works :- A  $3 \times 3$  binary multiplier is one of the combinational logic circuits, which can perform binary multiplication of two binary numbers having a bit size of a maximum of 3 bits. The bit size of the resultant output binary product is 6. Consider the multiplicand  $A_0, A_1, A_2$  and the multiplier  $B_0, B_1, B_2$ , and the final binary product output as  $P_0, P_1, P_2$ . The 3 partial product terms are obtained in the binary multiplication because it is a 3-bit multiplier. This  $3 \times 3$  multiplier can be implemented using a 3-bit full adder and individual single-bit adders. The carry bit is raised when  $A_2B_0$  and  $A_1B_1$  are added together. By the addition of the sum obtained from that, the carry bit is obtained from the addition of  $A_0B_1$  and  $A_0B_2$  to  $A_1B_0$ , which can raise another carry bit. Hence 2 carry bits are obtained and carried over for the addition of  $A_2B_1$  and  $A_1B_2$  and the 2 more carry bits are generated in the same way. The circuit is designed with 3-bit full adders to add the 3 partial product terms. The least significant bit (LSB) of the 1st partial product is not added to the next partial product because it is taken as an LSB of the final binary product output obtained. From the above logical circuit, one 3-bit full adder is used to add the first 2 partial products together and the other 3-bit full adder adds the 3rd partial product with the sum of the first adder.

### How to test

Explain how to test your project :- For testing a 3bit multiplier put 3bit 2 binary numbers at the input and cheak the outputs coming at the 7 segmented display.

### IO

#	Input	Output
0	{'B': 'connected to IN1'}	{'segment a': 'connected to OUT0'}
1	{'C': 'connected to IN2'}	{'segment b': 'connected to OUT1'}
2	{'D': 'connected to IN3'}	{'segment c': 'connected to OUT2'}

#	Input	Output
3	{'E': 'connected to IN4'}	{'segment d': 'connected to OUT3'}
4	{'F': 'connected to IN5'}	{'segment e': 'connected to OUT4'}
5	{'G': 'connected to IN6'}	{'segment f': 'connected to OUT5'}
6	none	{'segment g': 'connected to OUT6'}
7	none	dot

## 82 : Tiny Teeth Toothbrush Timer

- Author: Noah Hoffman
- Description: Simple 2 minute timer that gives visual feedback on the seven-segment display for each 30-second increment that passes. This indicates when to move to the next quarter of the mouth. Each 30 second increment can be blocked by an input switch so the user has to acknowledge they are done with that section before moving forward. At the end of 2 minutes, all segments of the display will flash
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 10000 Hz
- External hardware:

### How it works

This project expects a 10KHz clock. It divides it by  $2^{15}$  to get about 0.305Hz clock with a period of about 3.27 seconds. This time is reflected by the blinking of the decimal on the seven segment dispaly. After 10 clock cycles, the first input goes high to signify the first 30-second increment is done and the top-right segment of the display will turn on. After the next 30-second increment, the top left segment will turn on. Then the bottom right, and finally the bottom left. Once they are all lit, the 3 segments in the middle will flash to indicate it is done. Each of the 4 increments can be blocked by an enable switch for each one. If desired, a physical input can be used to ensure the timer does not continue to the next section until the user is done the current task.

### How to test

The timing is dependent on a 10KHz clock. From a hard reset, the first increment may be less than 30 seconds, so it is best to start with at least the “reset first increment” switch off (input 3) or the “enable first increment” switch off (input 7). With the other inputs on (1,2,4,5,6), one should be able to time each segment lighting up after 30 seconds and the 3 middle segments flashing at 2 minutes. If the decimal is not flashing or is flashing at a different period than 3.27 seconds, that indicates either the clock is off (input 1 to connect the clock or 2 to enable the clock dividing) or something other than a 10KHz clock was applied. If one wants to try the gated incremental mode, make sure inputs 4,5,6 start off. As one sees the timer indicate it is time to move on, the user can toggle that section’s switch. Each section is sequentially dependent on the next, so if a user want to go back to increment 2, the timer will start from 2 and then 3 and then 4. If it is desired to freeze the process where it is at while

not resetting anything, the input 8 “pause” switch can be turned high. For normal operation it should stay low.

## IO

#	Input	Output
0	clock	segment a- complete
1	reset clock	segment b- first increment done
2	reset first increment	segment c- third increment done
3	enable fourth increment	segment d- complete
4	enable third increment	segment e- fourth increment done
5	enable second increment	segment f- second increment done
6	enable first increment	segment g- complete
7	pause	dot- clock (3.27 seconds)

## 83 : 2's Compliment Subtractor

- Author: Naman Garg and Aryan Chaudhary
- Description: Subtracts two 4-bit binary numbers using 2's complement subtraction method
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

Explain how your project work “It uses 4 full adder circuits made by using XOR and AND gates.”

### How to test

Explain how to test your project “enter two 4 bit binary numbers in the inputs to try it out.”

### IO

#	Input	Output
0	A0	segment a
1	A1	segment b
2	A2	segment c
3	A3	segment d
4	B0	segment e
5	B1	segment f
6	B2	segment g
7	B3	dot

## 84 : Customizable Padlock

- Author: Tiny Tapeout 02 (J. Rosenthal)
- Description: This design implements a customizable padlock. Set a code for your digital safe!
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware: None

### How it works

Switch 2 is used to reset the safe. Switch 8 is used to set your code (on = set, off = locked). Switches 3 to 5 are used to set the code. The push button is used to enter your code.

### How to test

Set your desired code using Switches 3 to 5. Once you've done so, toggle Switch 8 to on then back off—the safe is now set! Turn on Switch 2, and press the push button. The seven segment display should show “L” (for locked). Next turn off Switch 2 to begin entering codes. If you enter a correct code, the seven segment display should show “U” (for unlocked).

### IO

#	Input	Output
0	N/A	segment a
1	Reset	segment b
2	Code 0	segment c
3	Code 1	segment d
4	Code 2	segment e
5	N/A	segment f
6	N/A	segment g
7	Set Code	none

## 85 : Customizable UART String

- Author: Tiny Tapeout 02 (J. Rosenthal)
- Description: This design Supports sending multiple ASCII characters over UART.
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 300 Hz
- External hardware: Arduino, computer with serial monitor connected to the Arduino

### How it works

This circuit implements five shift registers with 21 bits: seven idle bits, one start bit, eight data bits, one stop bit, and four more idle bits. The circuit supports transmitting a string of ASCII characters

### How to test

Connect an Arduino serial RX pin to the eight output pin (Output[7]). In the Arduino code, set the serial baud rate Serial.begin(); in the \*.ino file to 300. Set the PCB clock frequency to 300 Hz as well. Set the slide switch to the clock. Set SW7 to OFF ('Load'). Set SW8 to ON ('Output Enable'). Set SW7 to ON ('TX').

### IO

#	Input	Output
0	clock	segment a (Output Enable)
1	N/A	segment b (Load/TX)
2	N/A	segment c
3	N/A	segment d
4	N/A	segment e
5	N/A	segment f
6	Load/TX	segment g
7	Output Enable	UART Serial Out

## 86 : Customizable UART Character

- Author: Tiny Tapeout 02 (J. Rosenthal)
- Description: This design implements a single character UART transmitter using registers made from D-flip flops and multiplexers.
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 300 Hz
- External hardware: Arduino, computer with serial monitor connected to the Arduino

### How it works

This circuit implements a shift register with 17 bits: four idle bits, one start bit, eight data bits, one stop bit, and three more idle bits. The circuit supports transmitting a user-selected ASCII character from 0x40 (@) to 0x5F (\_), including capital letters from the Latin alphabet.

### How to test

Connect an Arduino serial RX pin to the eight output pin (Output[7]). In the Arduino code, set the serial baud rate Serial.begin(); in the \*.ino file to 300. Set the PCB clock frequency to 300 Hz as well. Set SW7 to OFF (“Load”). Set SW2 to ON and SW3-6 to OFF. Set SW7 to ON (“TX”). Set SW8 to ON (“Output Enable”). Connect the Arduino via USB to your computer and run the serial monitor. If there’s no output from the Arduino serial monitor, try toggling SW7 OFF and ON again. You should see the character ‘A’ appearing repeatedly in the serial monitor.

### IO

#	Input	Output
0	clock	segment a (Load/TX)
1	Bit 0	segment b
2	Bit 1	segment c
3	Bit 2	segment d
4	Bit 3	segment e
5	Bit 4	segment f (Output Enable)
6	Load/TX	segment g
7	Output Enable	UART Serial Out

## 87 : 7-Seg ‘Tiny Tapeout’ Display

- Author: Tiny Tapeout 02 (J. Rosenthal)
- Description: This circuit will output a string of characters ('tiny tapeout') to the 7-segment display.
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 1 Hz
- External hardware: None

### How it works

The logic to light the characters appears in the bottom half of the simulation window. The top half of the simulation window implements a modulo-11 counter. In other words, the counter increments up to 11 then resets. This counter is used to determine which character we should output to the 7-segment display. The truth table for the design can be found in the Design Spreadsheet: [https://docs.google.com/spreadsheets/d/1-h9pBYtuxv6su2EC8qBc6nX\\_JqHXks6Gx5nmHFQh\\_30/edit](https://docs.google.com/spreadsheets/d/1-h9pBYtuxv6su2EC8qBc6nX_JqHXks6Gx5nmHFQh_30/edit)

### How to test

Simply turn on and watch the characters on the 7-segment display. If needed, flip Input[1] (zero-indexed) ON to reset the state machine counter.

### IO

#	Input	Output
0	clock	segment a
1	Reset Counter	segment b
2	N/A	segment c
3	Clock Disable (Test Mode)	segment d
4	Test Logic A	segment e
5	Test Logic B	segment f
6	Test Logic C	segment g
7	Test Logic D	N/A

## 88 : Hola

- Author: Pascual Bravo
- Description: Chip de prueba
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

con dos entradas va mostrando la palabra H-O-L-A Explain how your project works

### How to test

hacer la combinación 00,01,10,11 Explain how to test your project

### IO

#	Input	Output
0	A0	segment a
1	A1	segment b
2	none	segment c
3	none	segment d
4	none	segment e
5	none	segment f
6	none	segment g
7	none	dot

## **89 : 5\_1MUX**

- Author: saurabh kumar and diksha bothra
- Description: it's 5 is to 1 mux which trigger only one input and gives output according to the select lines
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

### **How it works**

when i give change the logic of clock for diffrent combinations one input is triggered nad at output that input gives output Explain how your project works

### **How to test**

when select i give logic of 000 from S1 ,S2,S3 ,I1 is trigger at 100 I2 is triggered at 001 I3 , 101 I4 and 111 I5 is triggered| Explain how to test your project

### **IO**

#	Input	Output
0	Clock - S3	segment a - OUT0
1	I1	segment b
2	I2	segment c
3	I3	segment d
4	I4	segment e
5	I5	segment f
6	S1	segment g
7	S2	dot

## 90 : 3-bit 4-position register

- Author: Chris Burton
- Description: 3-bit 4-position register using Multiplexers
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

Decodes addresses and stores the bits using multiplexers.

### How to test

To write set the 3 data bits (WriteData0-WriteData2), address of the register to be written to (WriteAddress0/WriteAddress1) and toggle WriteEnable to write the 3-bits to the register at address. To read data set the address of the register to read (ReadAddress0/ReadAddress1) and the data is output on Data0-Data2. The internal state for Data2 at all addresses and an inverted WriteEnable is also output.

### IO

#	Input	Output
0	WriteData0	OutData0
1	WriteData1	Data2Address0
2	WriteData2	Data2Address1
3	WriteAddress0	OutData2
4	WriteAddress0	Data2Address2
5	WriteEnable	Data2Address3
6	ReadAddress0	OutData1
7	ReadAddress1	NotWriteEnable

## 91 : Simple adder used for educational purposes

- Author: Francisco Brito Filho
- Description: Simple adder used for educational purposes described in VHDL and ported to verilog using ghdl plugin.
- GitHub repository
- HDL project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

The adder was adapted from its 8-bit version. See <https://github.com/britovski/adder>

### How to test

See the testbenches on the previous github repo.

### IO

#	Input	Output
0	i0[3]	s[3]
1	i0[2]	s[2]
2	i0[1]	s[1]
3	i0[0]	s[0]
4	i1[3]	co
5	i1[2]	none
6	i1[1]	none
7	i1[0]	none

## 92 : 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

## 93 : HD74480 Clock

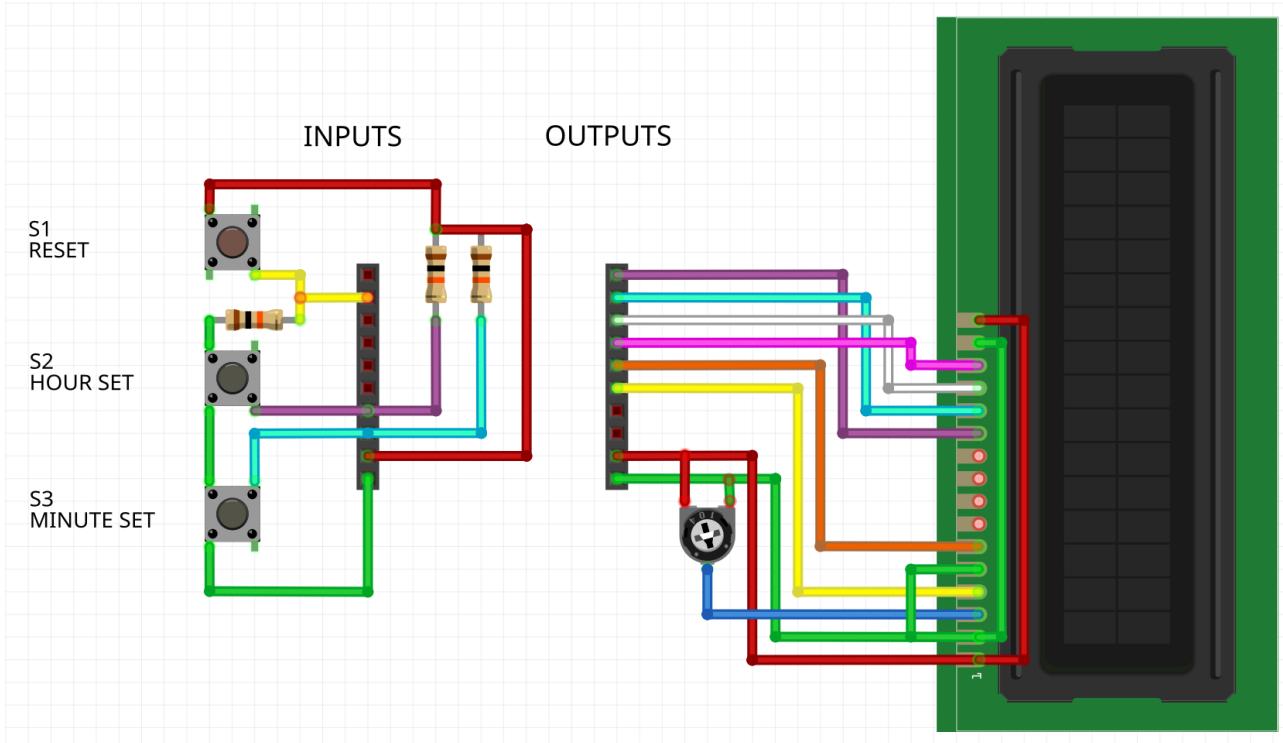


Figure 20: picture

- 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
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

## 94 : Scrolling Binary Matrix display

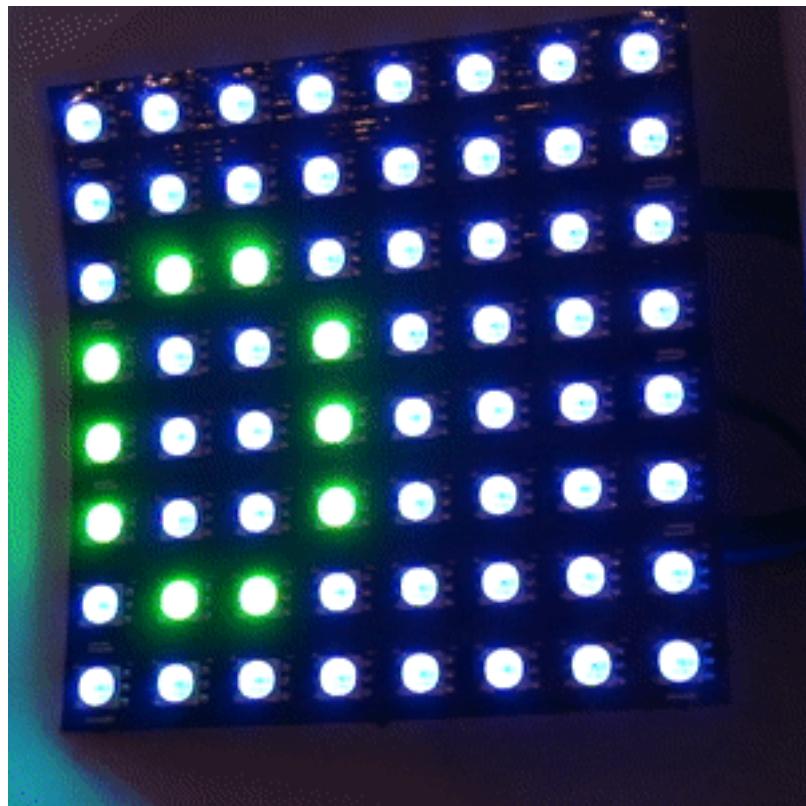


Figure 21: picture

- 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

## 95 : 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

## 96 : 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 frequency 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

## 97 : 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

## 98 : The McCoy 8-bit Microprocessor

picture

- Author: Aidan Good
- Description: Custom RISC-V inspired microprocessor capable of simple arithmetic, 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 thorough explanation 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

## 99 : 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 with 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

# 100 : TinySensor

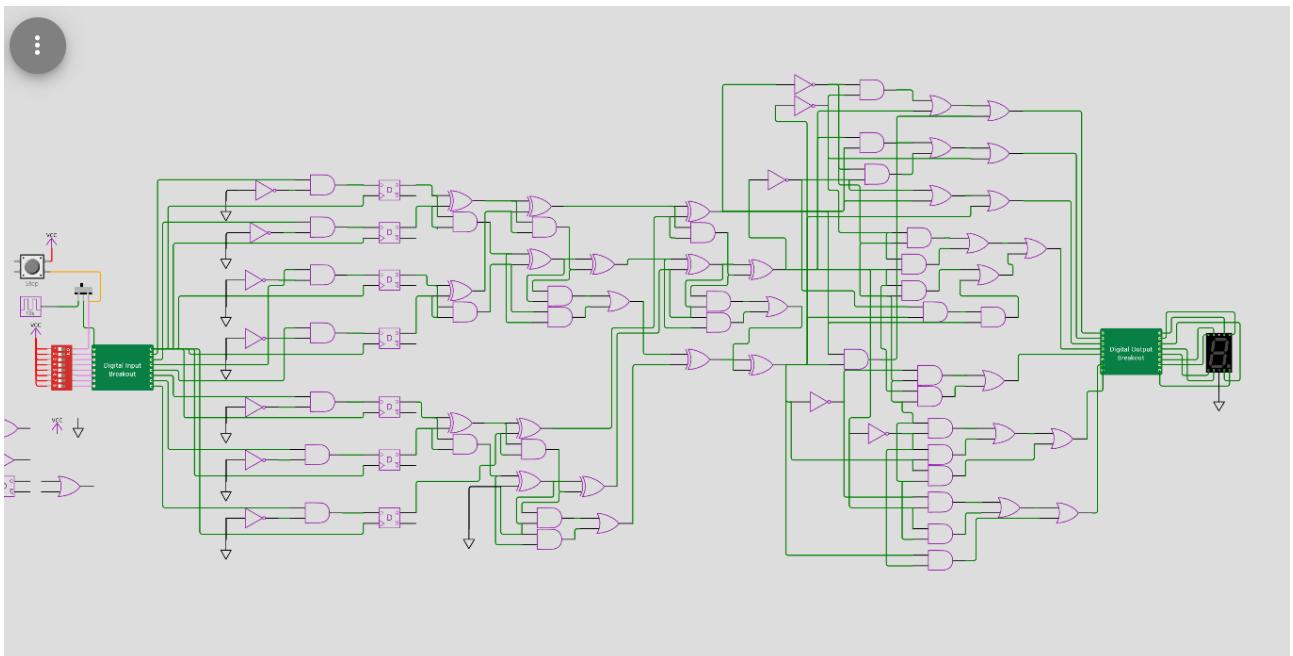


Figure 22: picture

- 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 Continuous 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

# 101 : Water-Level-Indicator-With-Auto-Motor-Control

- Author: Ashutosh Kumar
- Description: As water level of tank decreases/increases LED glow according to it.when water level touches the last then motor start and run until full the tank.
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

## How it works

First 4 flip flop uses as synchroniser. Total four state define as full Half1 Half2 Empty. These four state mapped with 00 01 10 11 and further it mapped with 4 Led. Logic circuits design to control the states using D-flip flop. Another Logic design with help of one D flip flop to control motor. Motor only start when it goes to last stage and it OFF only when water level of tank full.

## How to test

Explain how to test your project: it have 3 input clock input A input B Total 5 output out1 – connect to red Led (Full) out2 – connect to yellow1 Led (Half1) out3 – connect to yellow2 Led (Half2) out4 – connect to Red Led (Empty) out5 – connect to Red Led (moto)

## IO

#	Input	Output
0	c	segment a – Red Led (Full)
1	l	segment b – Yellow1 (Half1)
2	o	segment c – Yellow2 ( Half2)
3	c	segment d – RED (Empty)
4	k	segment e – RED (motor)
5		segment f
6	i	segment g
7	n	dot

# 102 : 16x8 SRAM & Streaming Signal Generator

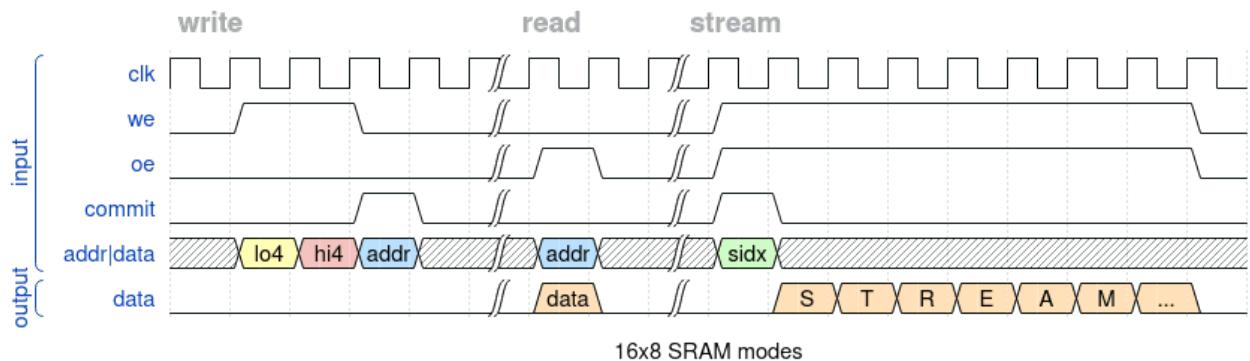


Figure 23: picture

- 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.



#	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]

## 103 : 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 you 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

## 104 : 4-spin Ising Chain Simulation

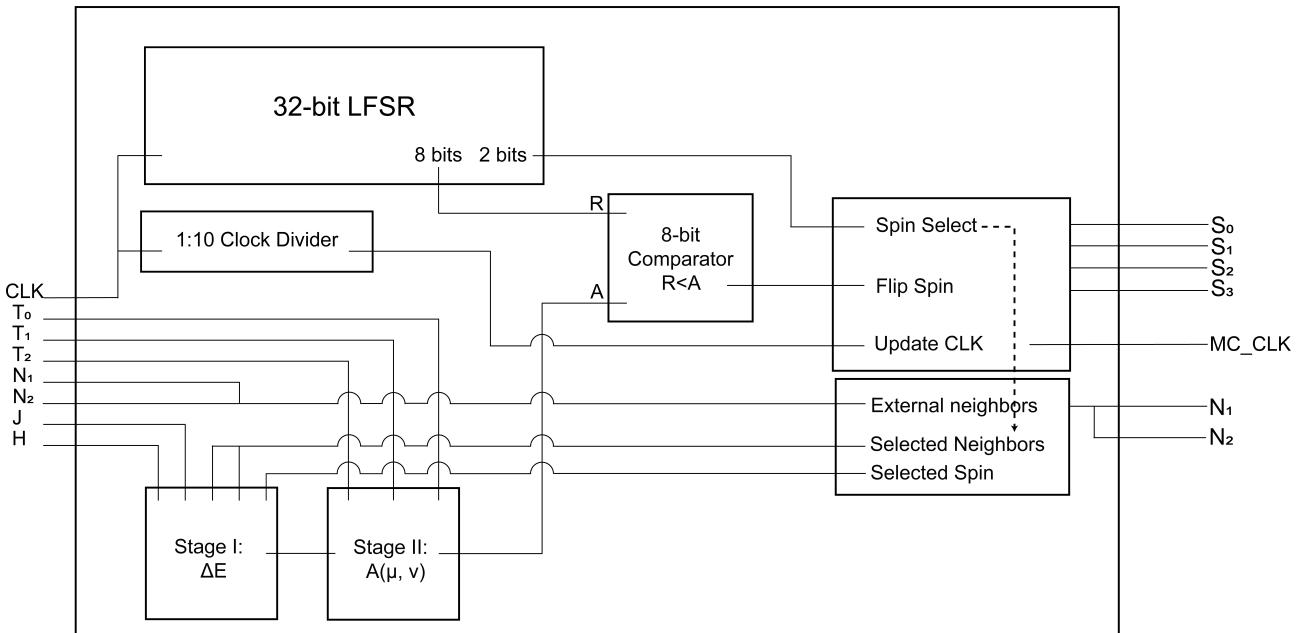


Figure 24: picture

- 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.

# 105 : Avalon Semiconductors '5401' 4-bit Microprocessor

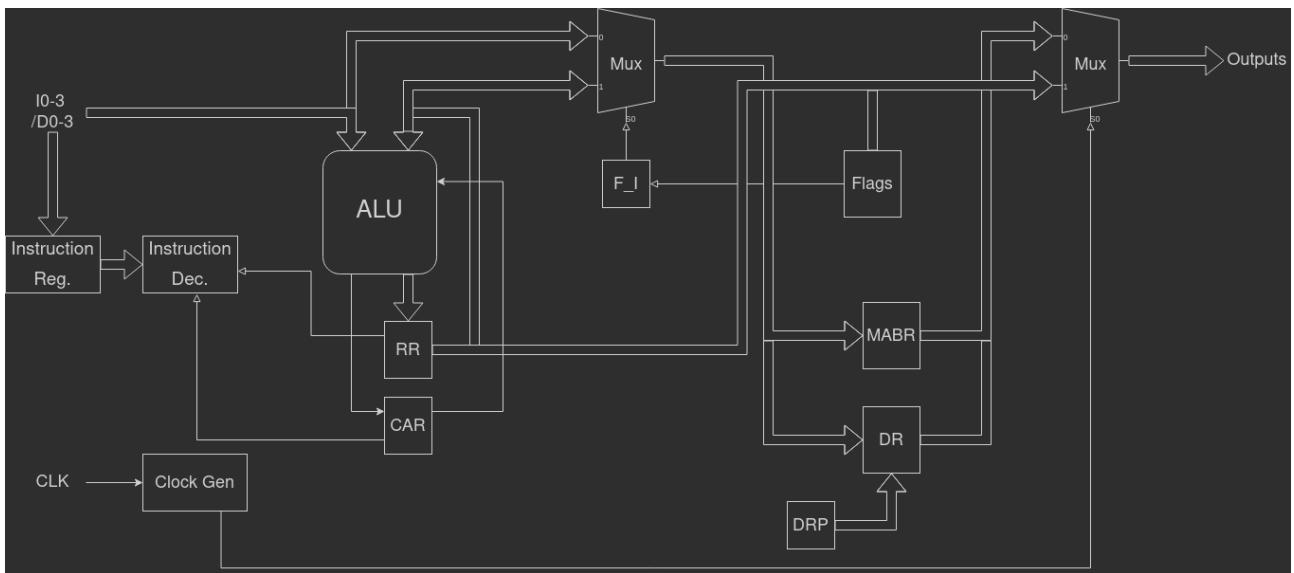


Figure 25: picture

- 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 arithmetic, 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

## 106 : 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

## 107 : 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=X000, TapeA\_1=OX00, 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  $(C ? A : B) \wedge D$  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  $(C ? A : B) \wedge D$  operation electrically instead of optomechanically. An extra  $\wedge E$  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 11111, 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

#	Input	Output
1	Value from Tape B	none
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

## 108 : 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

## 109 : Configurable SR

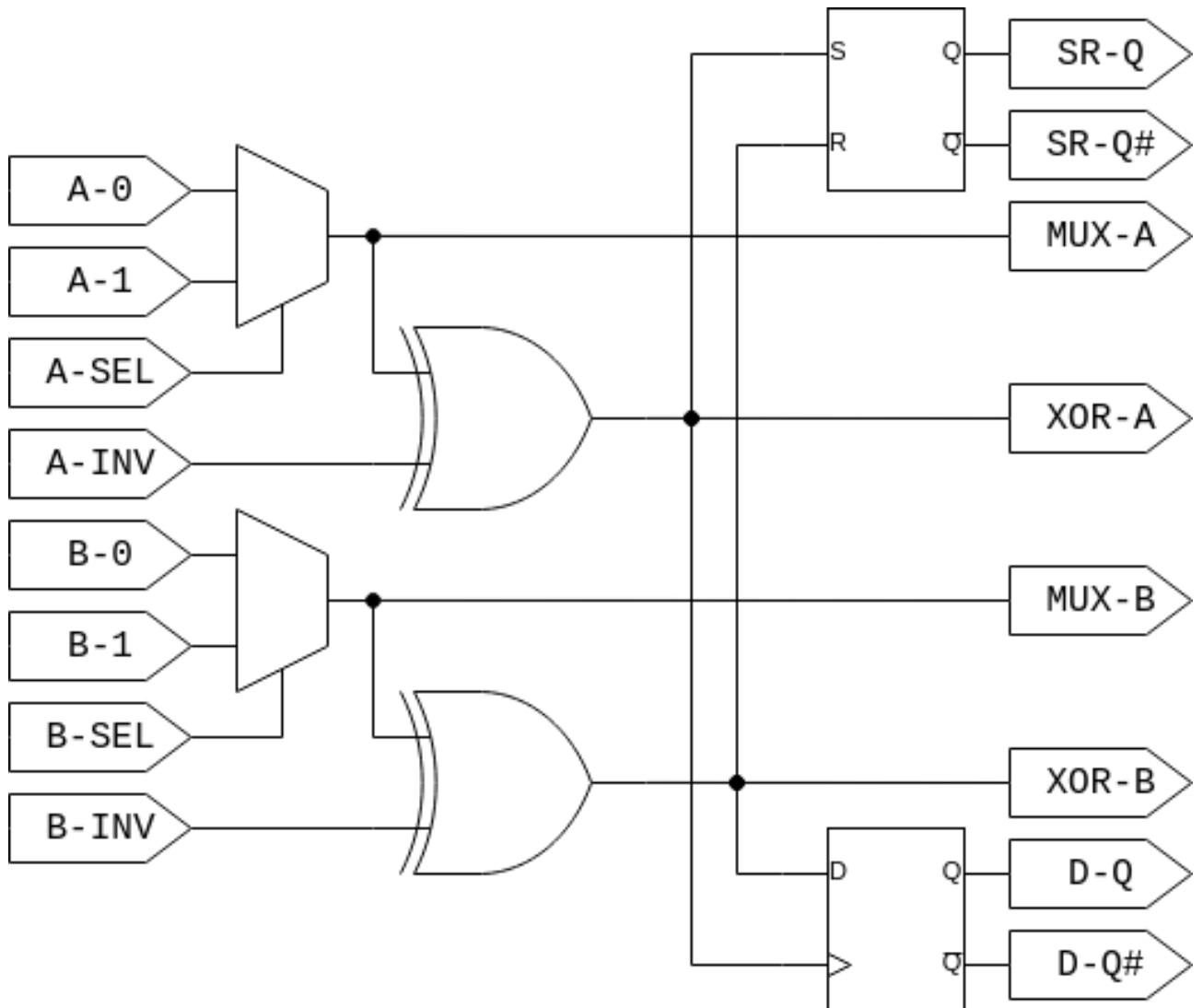


Figure 26: picture

- 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#

## 110 : 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

## 111 : 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

## 112 : 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](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

## 113 : 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

## 114 : 4-bit Multiplier

- Author: Fernando Dominguez Pouso
- 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)

# 115 : 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 still 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

## 116 : 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)

## 117 : 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

## 118 : Loading Animation

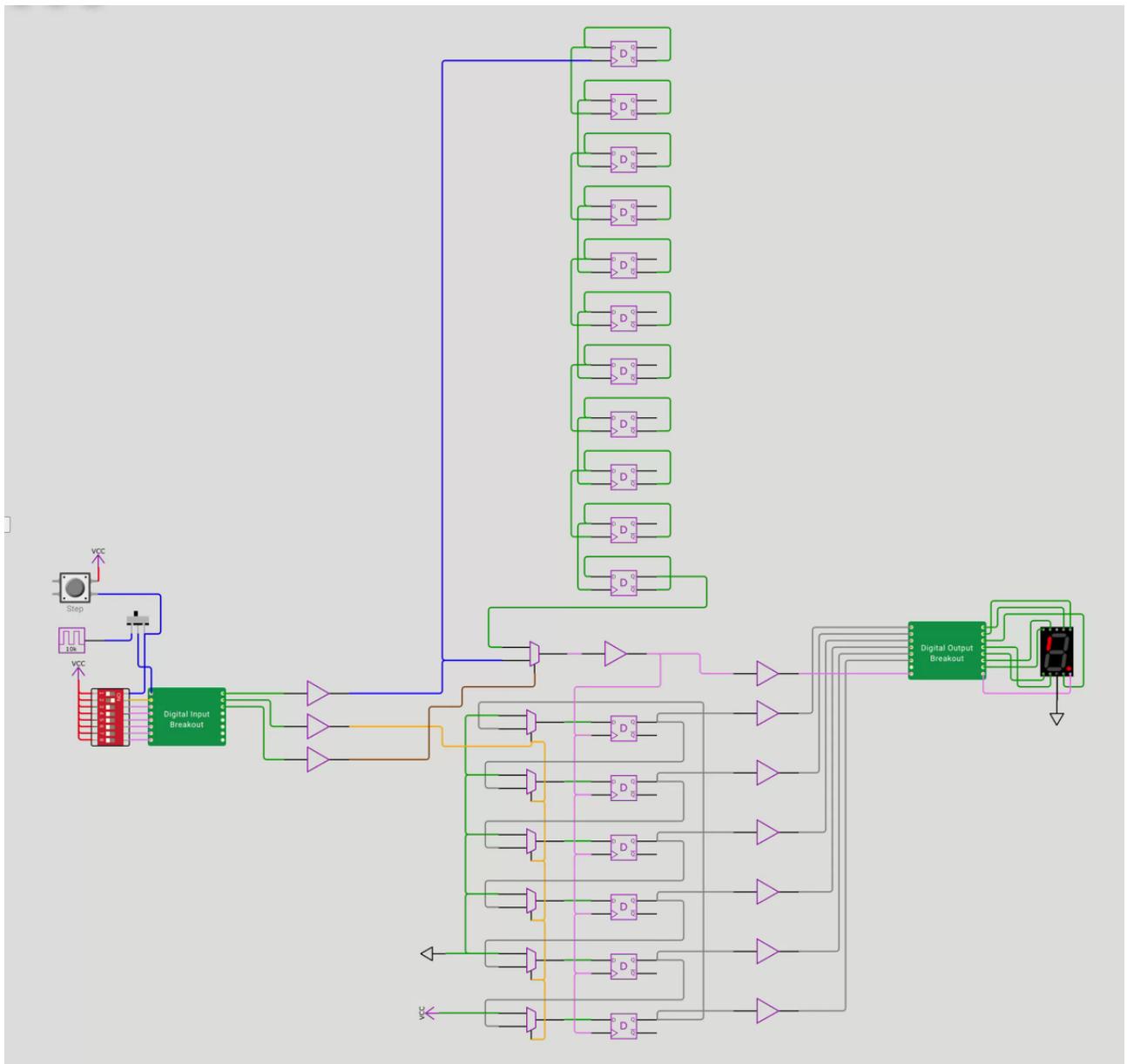


Figure 27: picture

- 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

## 119 : 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 required

### 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

## 120 : Potato-1 (Brainfuck CPU)

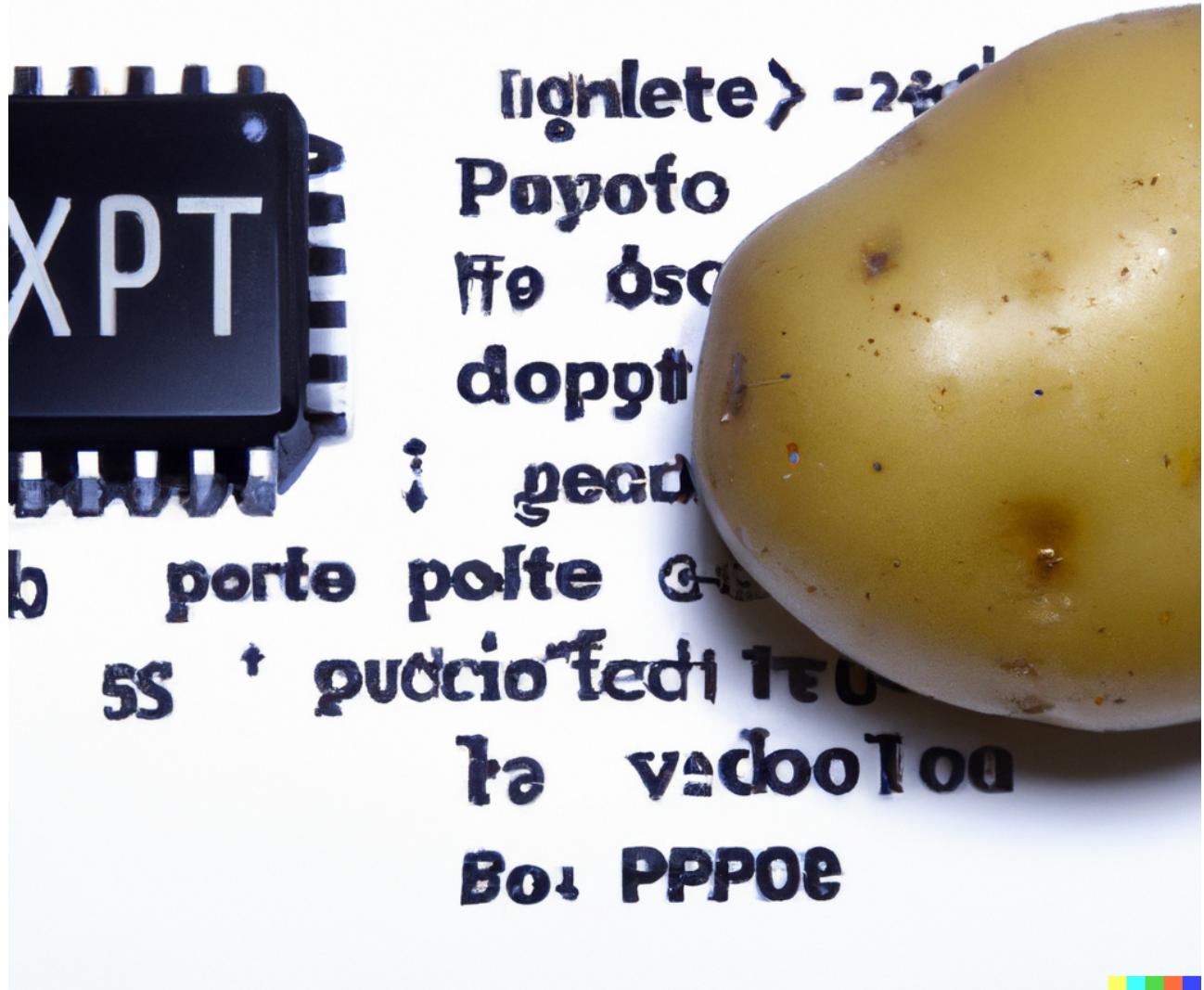


Figure 28: picture

- 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 indicate 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 immidiately 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 clockcycle

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

## 121 : 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 letters

### 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

## 122 : 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

## 123 : 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](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

## 124 : 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

## 125 : 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

## 126 : Pi ( ) to 1000+ decimal places

$\pi$

=  
3.14159265358979323846264338327950288419716939937510582097494459  
2307816406286208998628034825342117067982148086513282306647093844  
6095505822317253594081284811174502841027019385211055596446229489  
5493038196442881097566593344612847564823378678316527120190914564  
8566923460348610454326648213393607260249141273724587006606315588  
1748815209209628292540917153643678925903600113305305488204665213  
8414695194151160943305727036575959195309218611738193261179310511  
8548074462379962749567351885752724891227938183011949129833673362  
4406566430860213949463952247371907021798609437027705392171762931  
7675238467481846766940513200056812714526356082778577134275778960  
9173637178721468440901224953430146549585371050792279689258923542  
0199561121290219608640344181598136297747713099605187072113499999  
9837297804995105973173281609631859502445945534690830264252230825  
3344685035261931188171010003137838752886587533208381420617177669  
1473035982534904287554687311595628638823537875937519577818577805  
3217122680661300192787661119590921642019893809525720106548586327

Figure 29: 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 circuit (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  $1023 * \log_2(10) + 1 \approx 3,400$  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

## 127 : 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

## 128 : 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

## 129 : M0: A 16-bit SUBLEQ Microprocessor

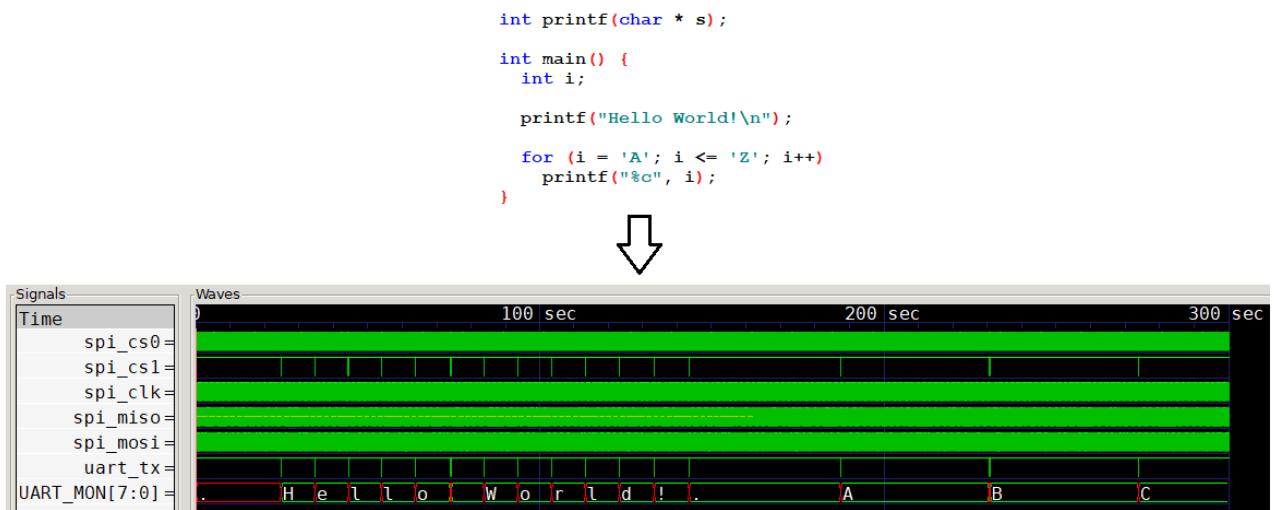


Figure 30: picture

- 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

## 130 : 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  $3\text{khz} / x$  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

#	Input	Output
2	address/data 0	digital out 2
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

## 131 : 8x8 Bit Pattern Player

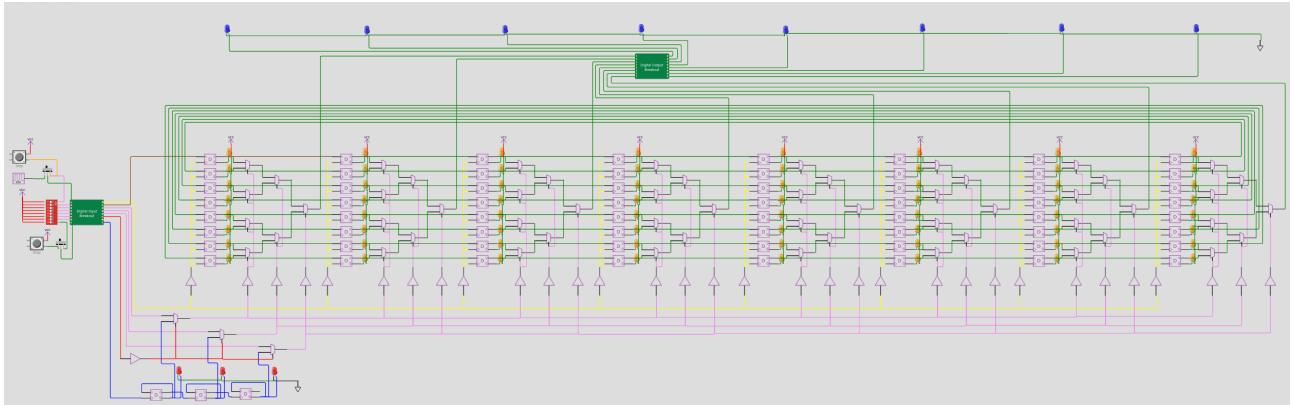


Figure 31: picture

- 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

## 132 : 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

## 133 : 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

## 134 : 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

## 135 : Super Mario Tune on A Piezo Speaker

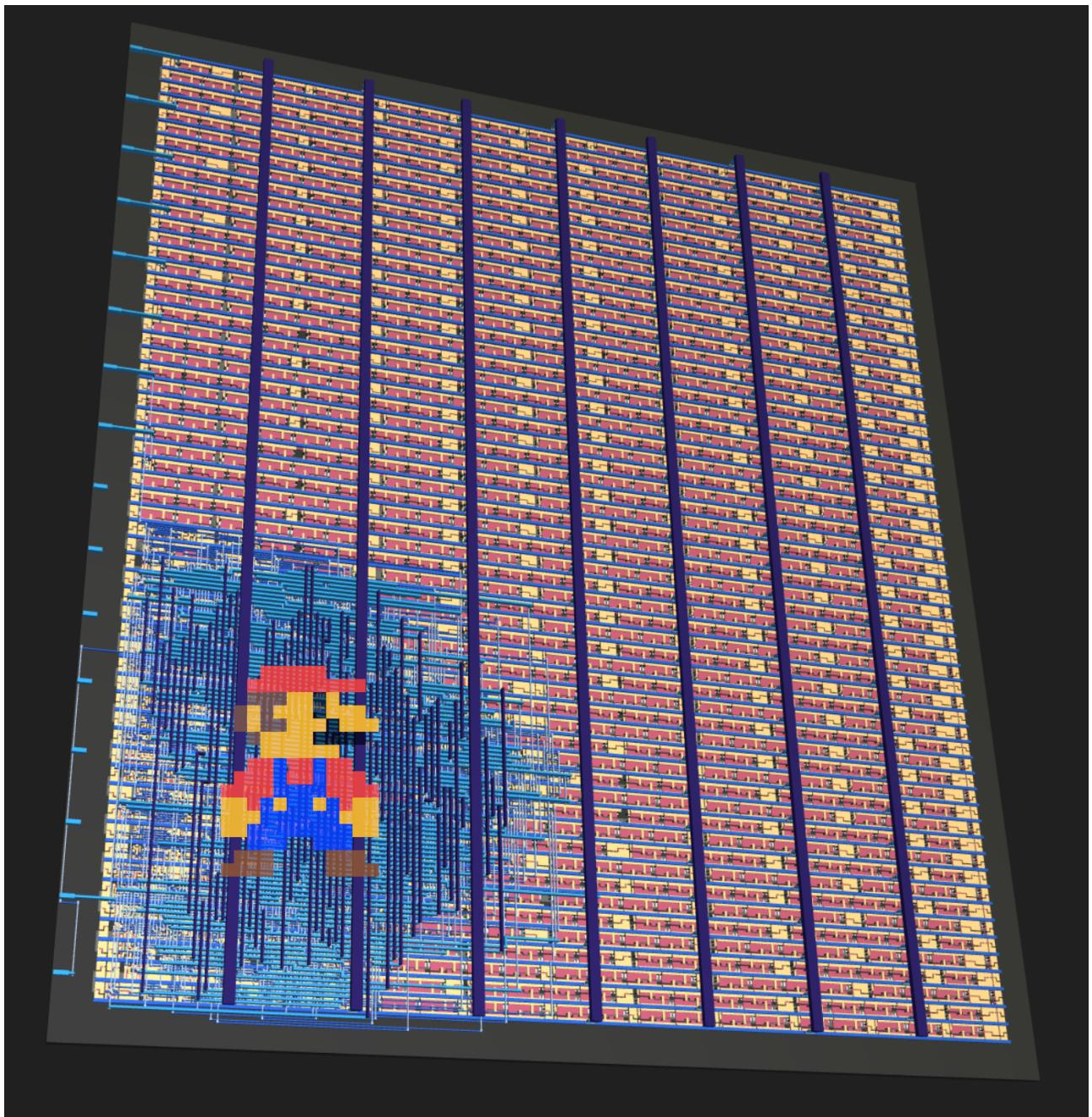


Figure 32: picture

- 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

## 136 : Tiny rot13

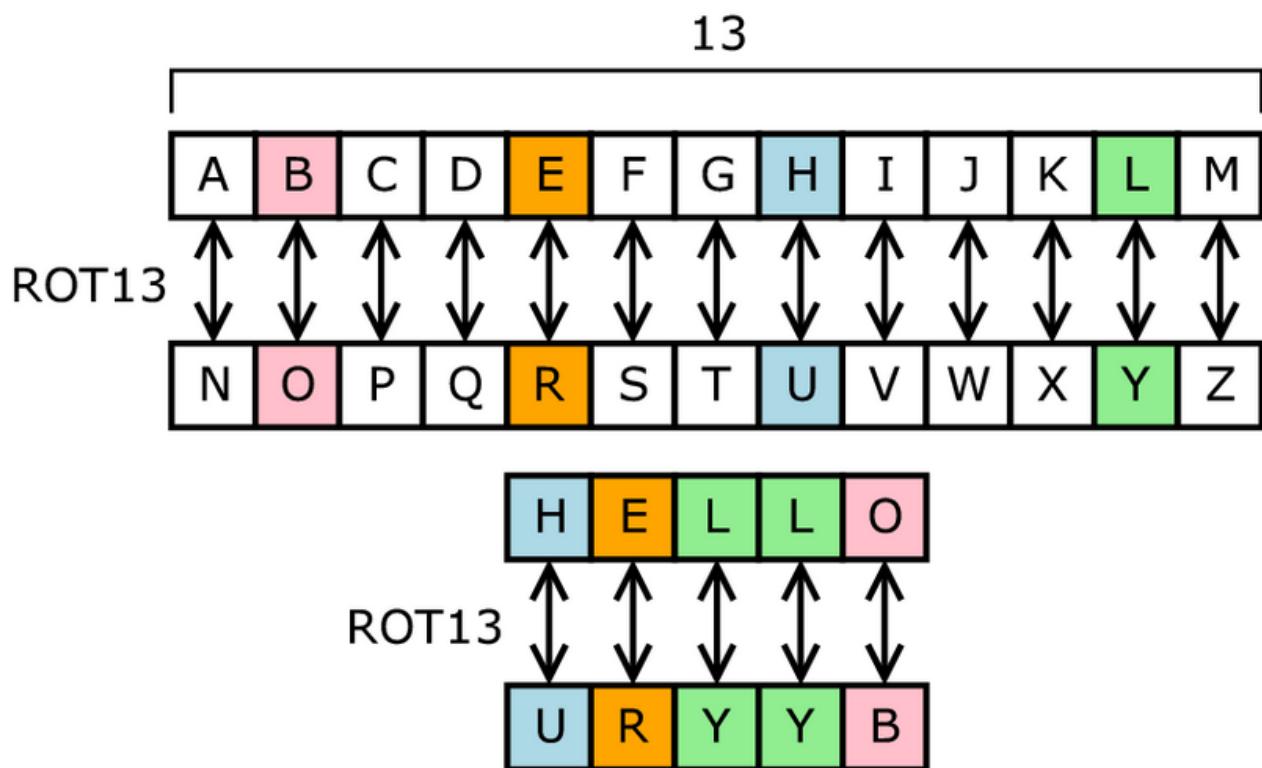


Figure 33: picture

- 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 generated 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	D00 - LSB of output
1	reset - Resets the system to a clean state	D01
2	CTL0 - LSB of control	D02
3	CTL1 - MSB of control	D03
4	D0 - LSB of input nibble	D04
5	D1	D05
6	D2	D06
7	D3 - MSB of input nibble	D07 - MSB of output

## 137 : 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

## 138 : Shiftregister Challenge 40 Bit

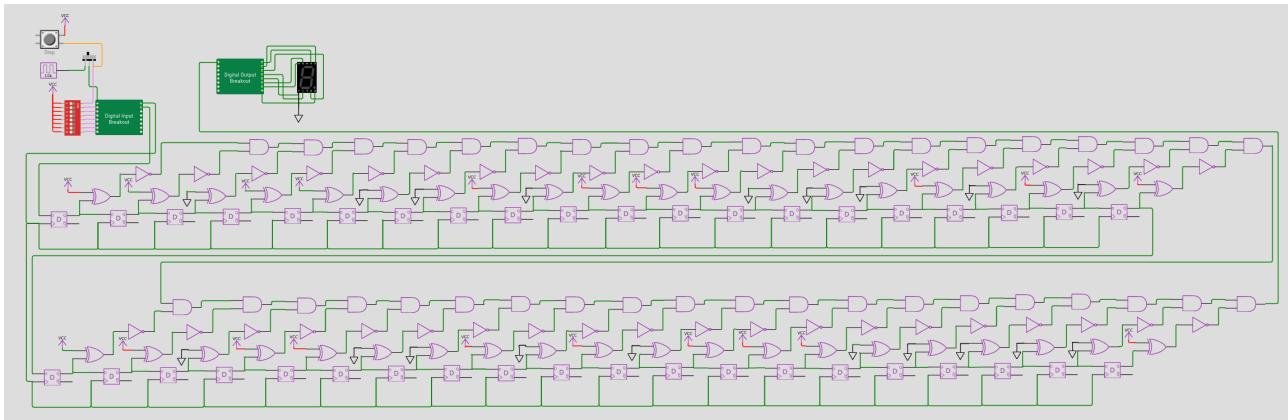


Figure 34: picture

- 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 find 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

## 139 : TinyTapeout2 4-bit multiplier.

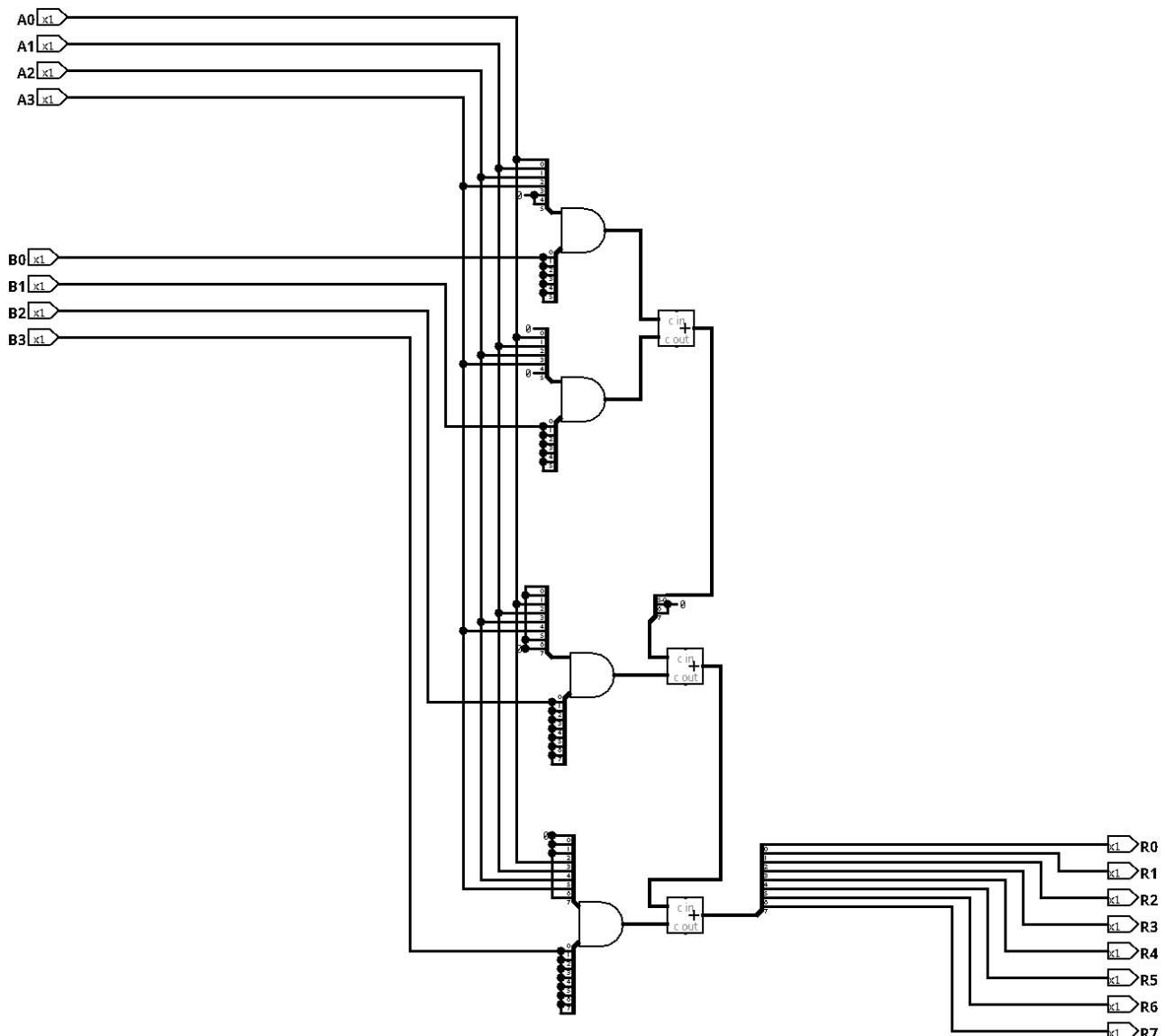


Figure 35: picture

- 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

## 140 : 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 seven 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

## 141 : 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/](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

## 142 : 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 initial 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

## 143 : 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 initial 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

## 144 : 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

## 145 : 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

## 146 : 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

## 147 : 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

## 148 : LEDChaser from LiteX test

- Author: Nick Østergaard
- Description: This is just a small demo of synthesizing 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

# 149 : 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.  $(-1)^{\text{sign}} \times 2^{(\text{exponent} - \text{EXP\_BIAS})} \times 1.\text{mantissa}$  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 (0x1111111 = +/- 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

#	Input	Output
4	data0	exponent
5	data1	mantissa
6	data2	mantissa
7	data3	mantissa

## 150 : 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

# 151 : 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

Wokwi

## 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

## 152 : CNS002 (TT02-Test 2)

picture

- Author: Bryan Bonilla Garay, Devin Alvarez, Ishaan Singh, Yu Feng Zhou, and N. Sertac Artan
- Description: First test run of CNS Lab (second design)
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware: None

### How it works

Apply inputs, get outputs

### How to test

Wokwi

### 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

## 153 : Test2

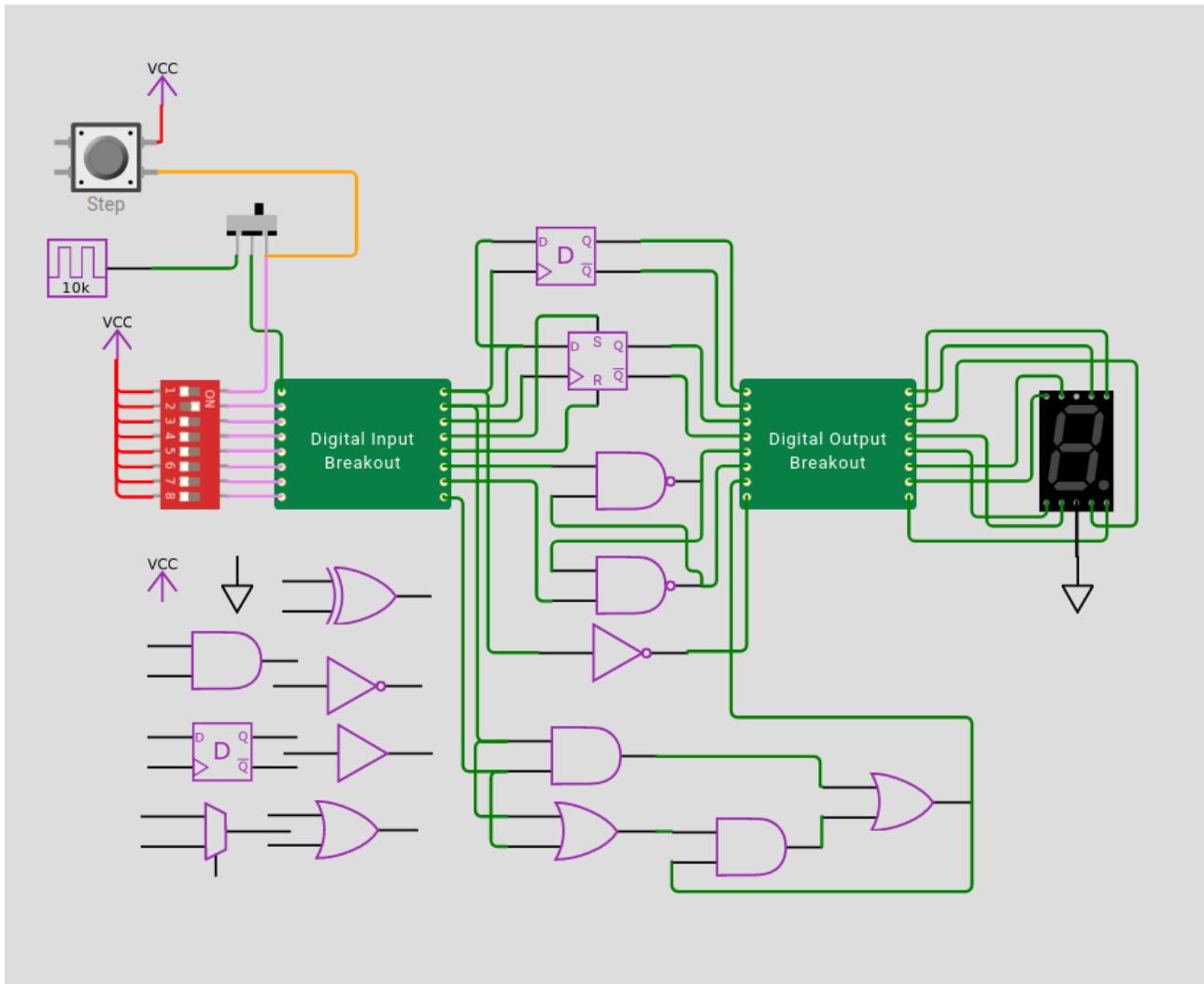


Figure 36: picture

- 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

## 154 : 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

## 155 : 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 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

## 156 : 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

## 157 : 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]

## 158 : 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’}

## 159 : tinytapeout2-HELLo-3orLd-7seg



Figure 37: 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

## 160 : Non-restoring Square Root

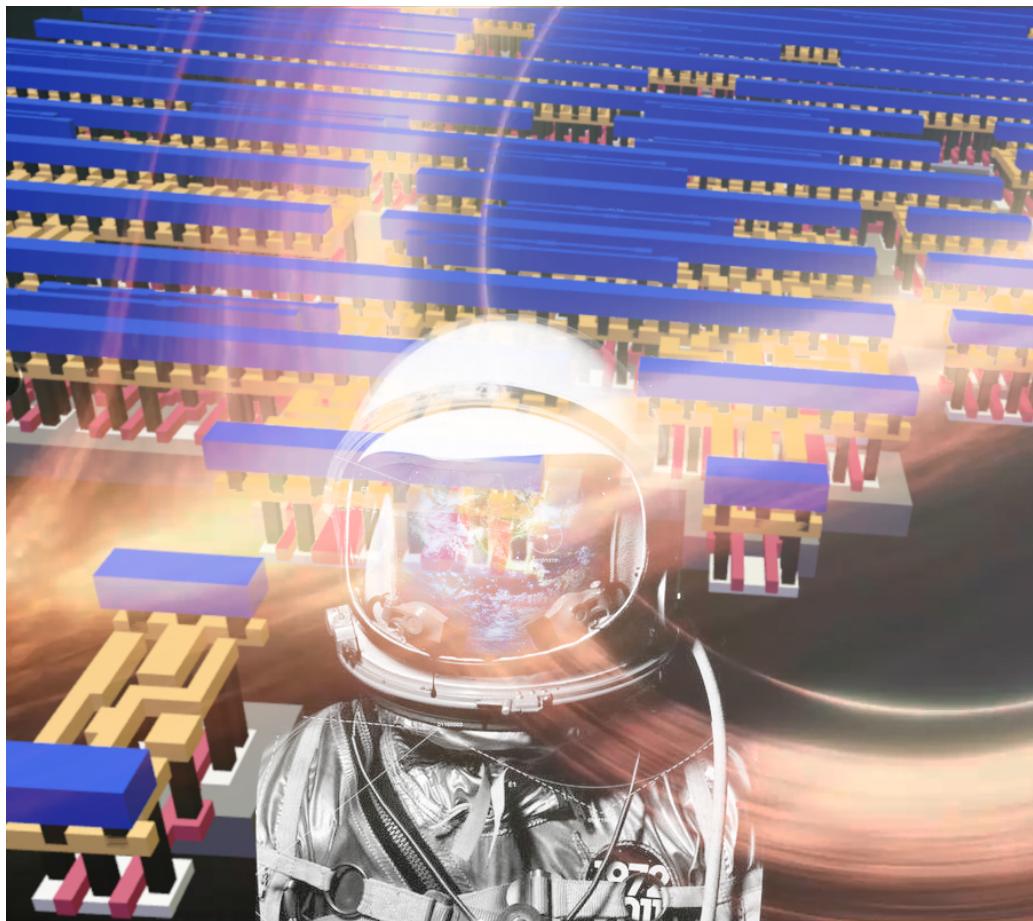


Figure 38: picture

- 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

## 161 : GOL-Cell

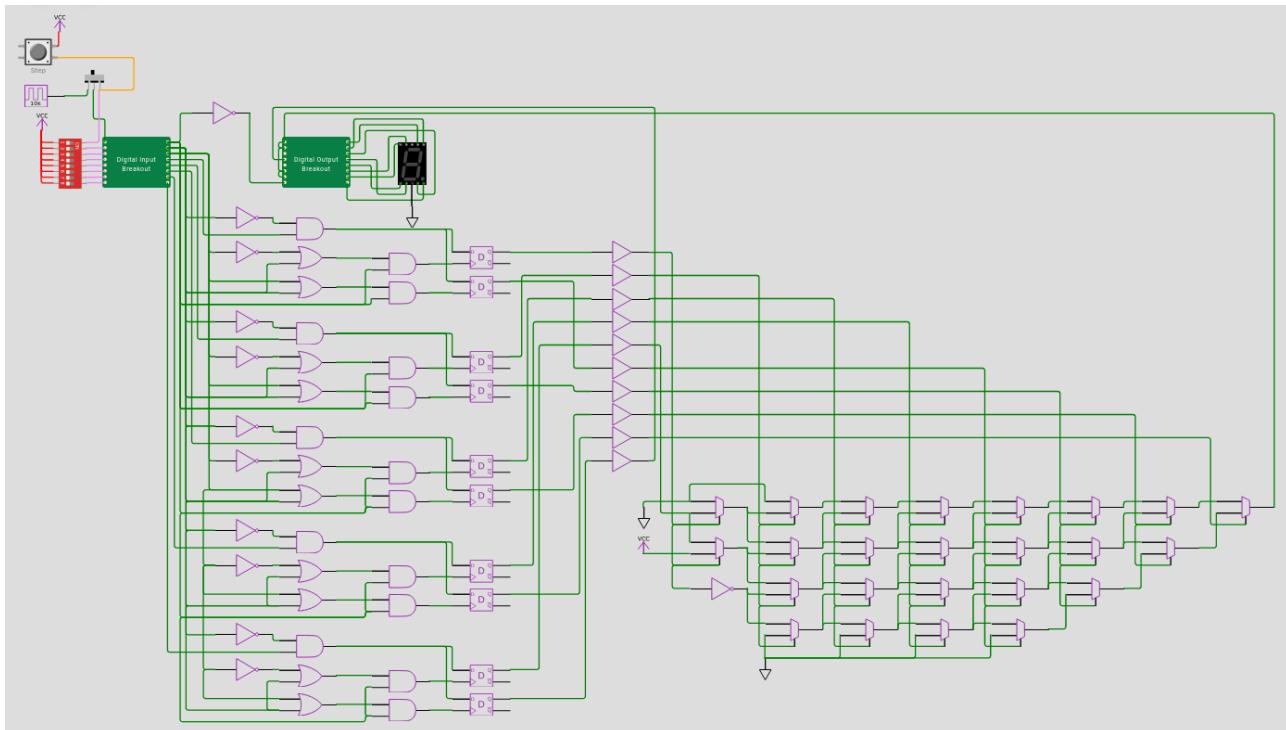


Figure 39: picture

- 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

## 162 : 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

## 163 : 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 recirculating 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]

## 164 : Ring OSC Speed Test

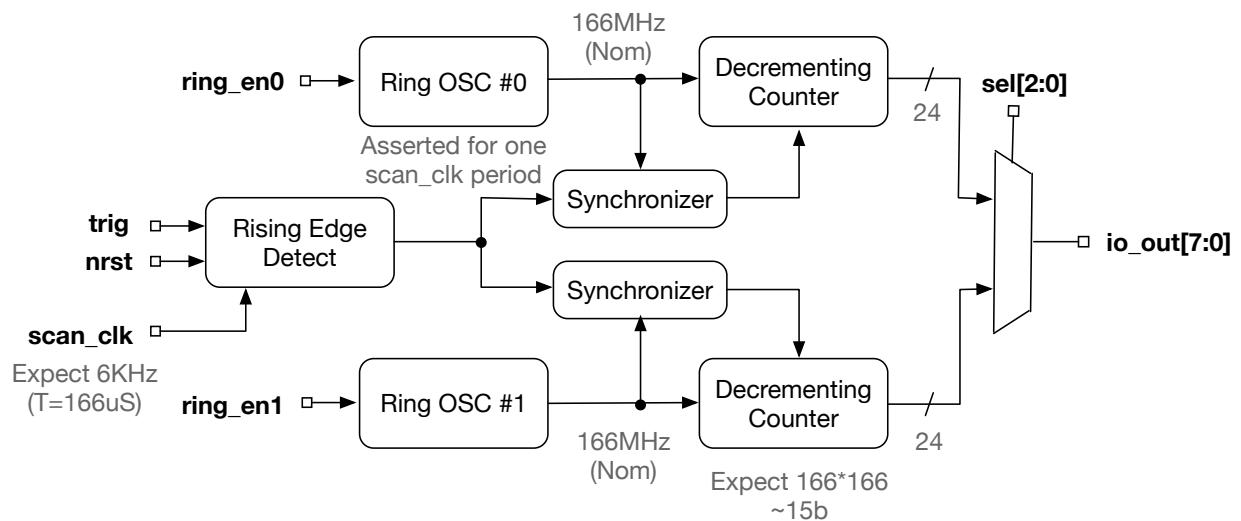


Figure 40: picture

- Author: Eric Smith
- Description: Make two rings with the same number of stages but measure how their frequency differs. Measure if they can influence each other.
- 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]

#	Input	Output
4	sel[1]	out[4]
5	sel[2]	out[5]
6	ring_en[0]	out[6]
7	ring_en[1]	out[7]

## 165 : 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

## 166 : TrainLED2 - RGB-LED driver with 8 bit PWM engine

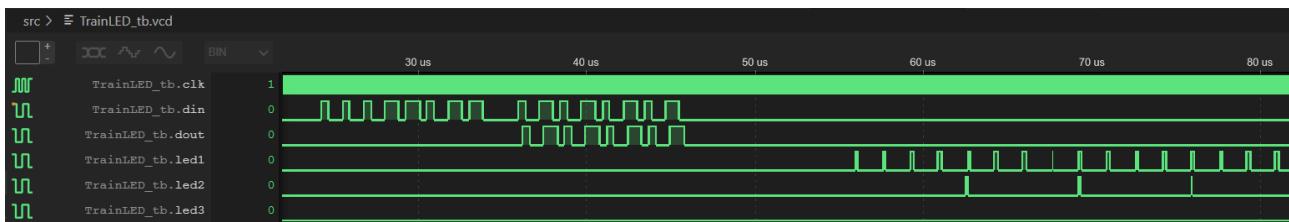


Figure 41: picture

- 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  $3 \times 8 = 24$  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

#	Input	Output
2	Din Driver A	LED2A
3	none	LED3A
4	none	none
5	none	none
6	none	none
7	none	none

# 167 : 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]

# 168 : 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

## 169 : Stack Calculator

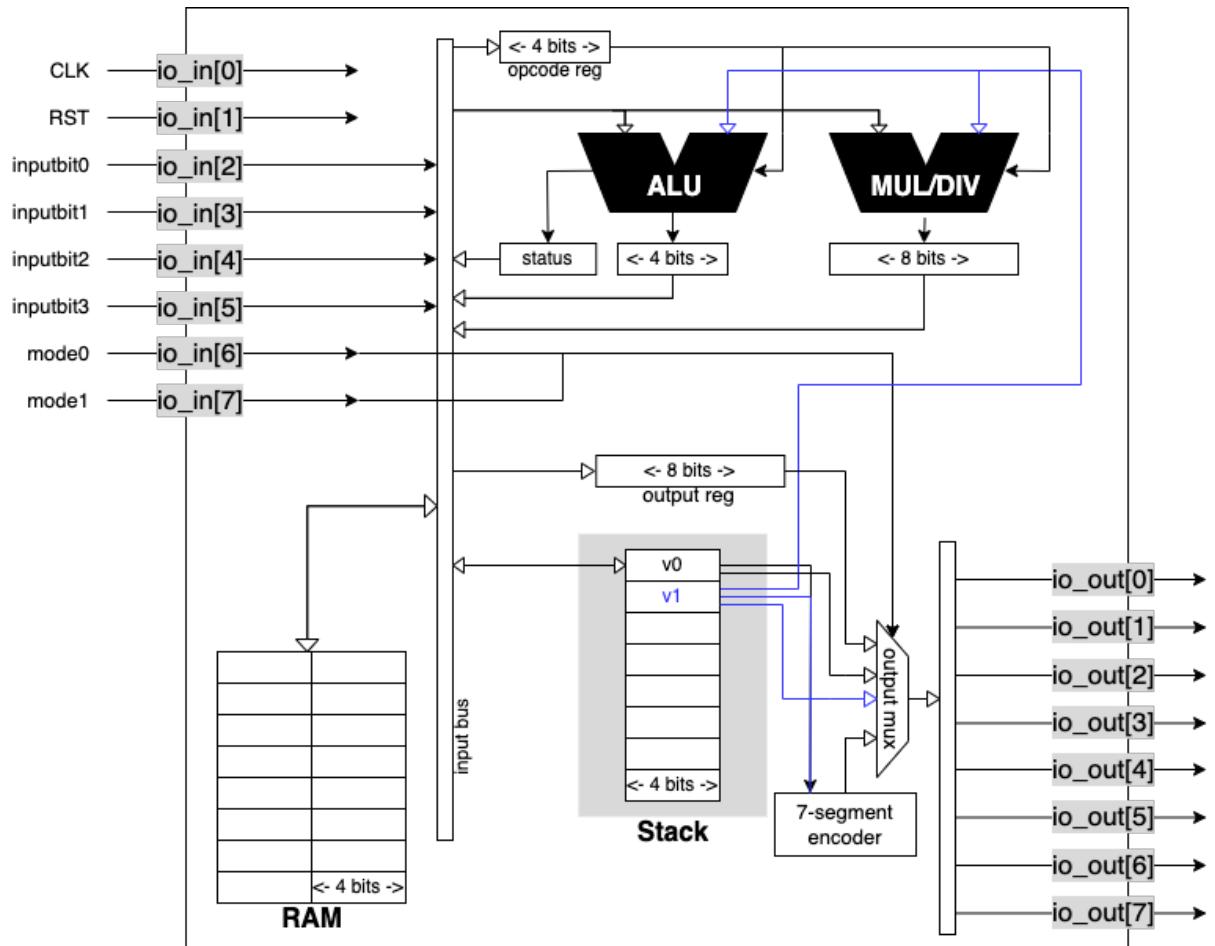


Figure 42: picture

- 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.

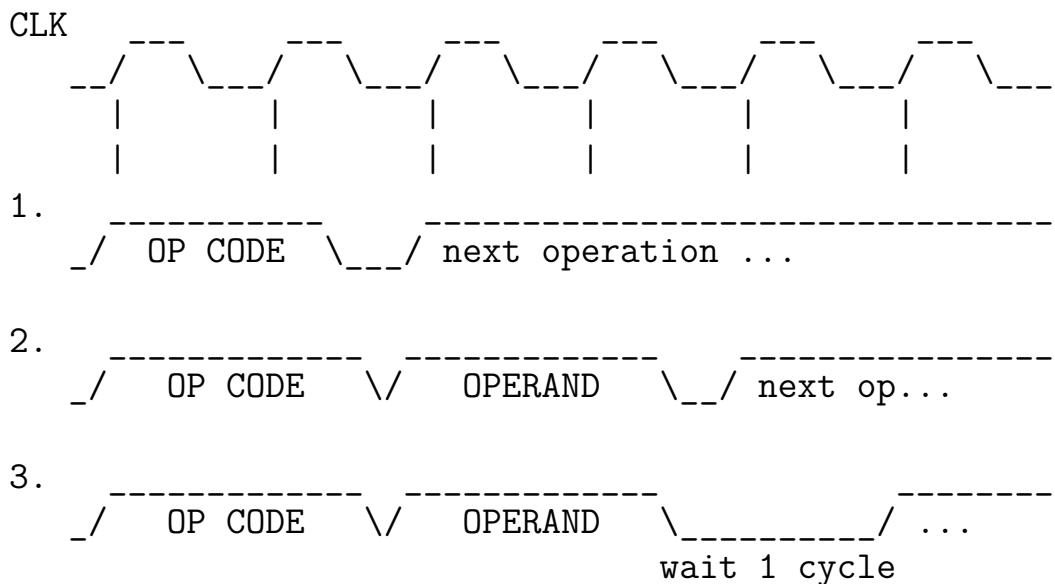
The stack calculator can also save 16 4-bit values using the SAVE and LOAD operations.

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 5 cycles.

Opcodes and operands are always 4-bits wide and are applied to the 4 input pins (pins 2-5) and need to be applied before the clock ticks up. In some cases they need to be held for more than one cycle for them to apply.

All input must be provided in a particular order. Below is a timing diagram that shows how to apply opcodes to the processor

#### Timing diagram



#### LEGEND

1. 2-cycle opcode, no operands
2. 4-cycle opcode, 1 operand
3. 5-cycle opcode, 1 operand (PUSH)

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.

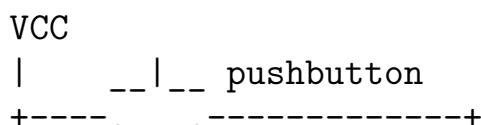
The list opcodes are as follows:

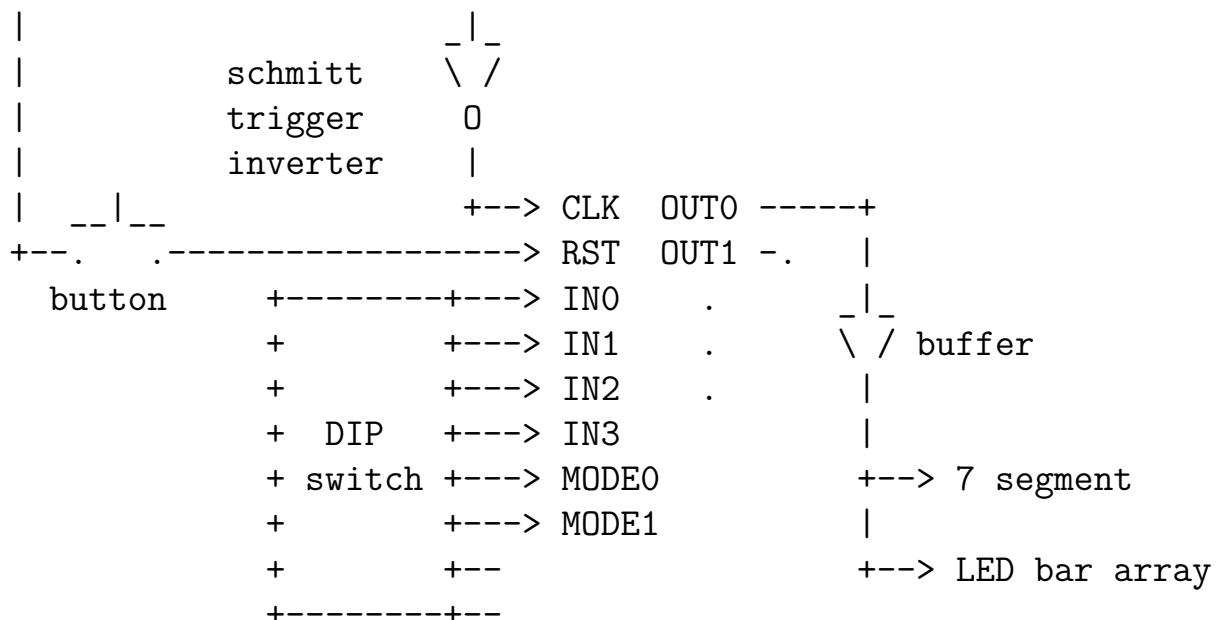
- 0x1 PUSH
  - Pushes a value to the stack. The value must be provided in the following cycle.
  - 5 cycles - push, push, value, value, wait
- 0x2 POP
  - Pops a value from the stack. The value must be provided in the following cycle.
  - 3 cycles - pop, pop, wait
- 0x3 OUTL
  - Copies the value on the top of the stack to the lower 4 bits of the output register.
  - 2 cycles - outl, outl
- 0x4 OUTH
  - Copies the value on the top of the stack to the high 4 bits of the output register.
  - 2 cycles - outh, outh
- 0x5 SWAP
  - Swaps the top two values on the stack.
  - 3 cycles - swap, swap, wait
- 0x6 PUSF
  - Push a value on the stack depending on the operand. The operand determines the values pushed on the stack.
  - 4 cycles - peek/dupl/flag, =, 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
  - 4 cycles - not/neg/incr/decr/shr1/shr2/ror1/rol1, =, 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
  - 4 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 OR - bitwise OR
    - \* 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, mult, 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, idiv, wait, wait
- 0xB CLFL
  - Unset all flags in flag register
  - 4 cycles - clfl, clfl
- 0xC SAVE
  - Writes the value below the top of the stack to the address provided at the top of the stack.
  - 4 cycles - save, save, wait, wait
- 0xD LOAD
  - Loads the value at the address provided at the top of the stack.
  - 4 cycles - load, load, wait, wait

## How to test

The following diagram shows a simple test setup that can be used to test the stack calculator





By using a schmitt trigger for debounce and an inverter, it is possible to perform a tick up with a specific DIP switch setting, allowing us to experiment with different kinds of inputs in sequence.

Using the DIP switch you can also change the MODE pins to debug your stack or display the output register contents.

## IO

#	Input	Output
0	clk	output0
1	rst	output1
2	input0	output2
3	input1	output3
4	input2	output4
5	input3	output5
6	mode0	output6
7	mode1	output7

## 170 : 1-bit ALU

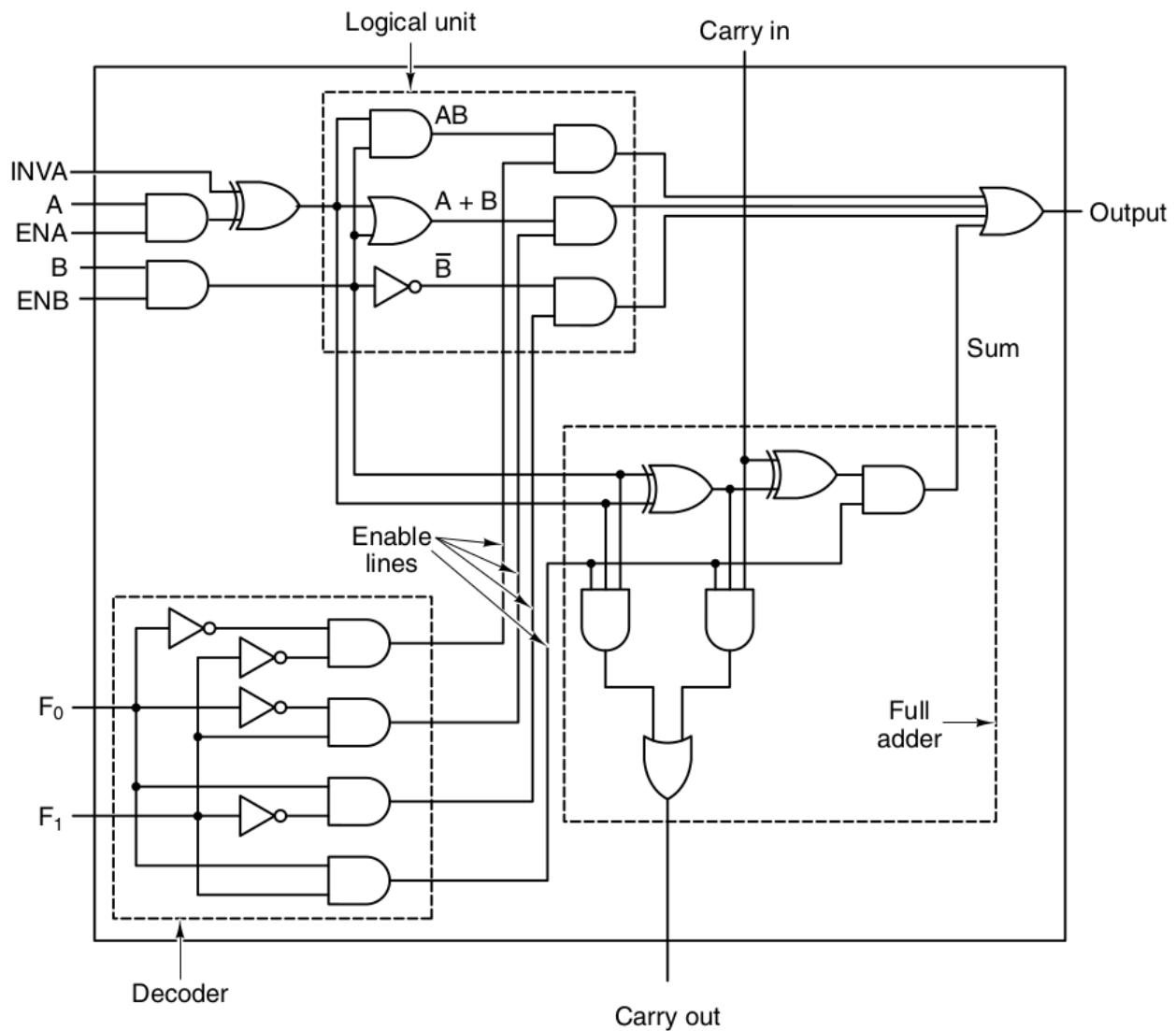


Figure 43: picture

- 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

## 171 : 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

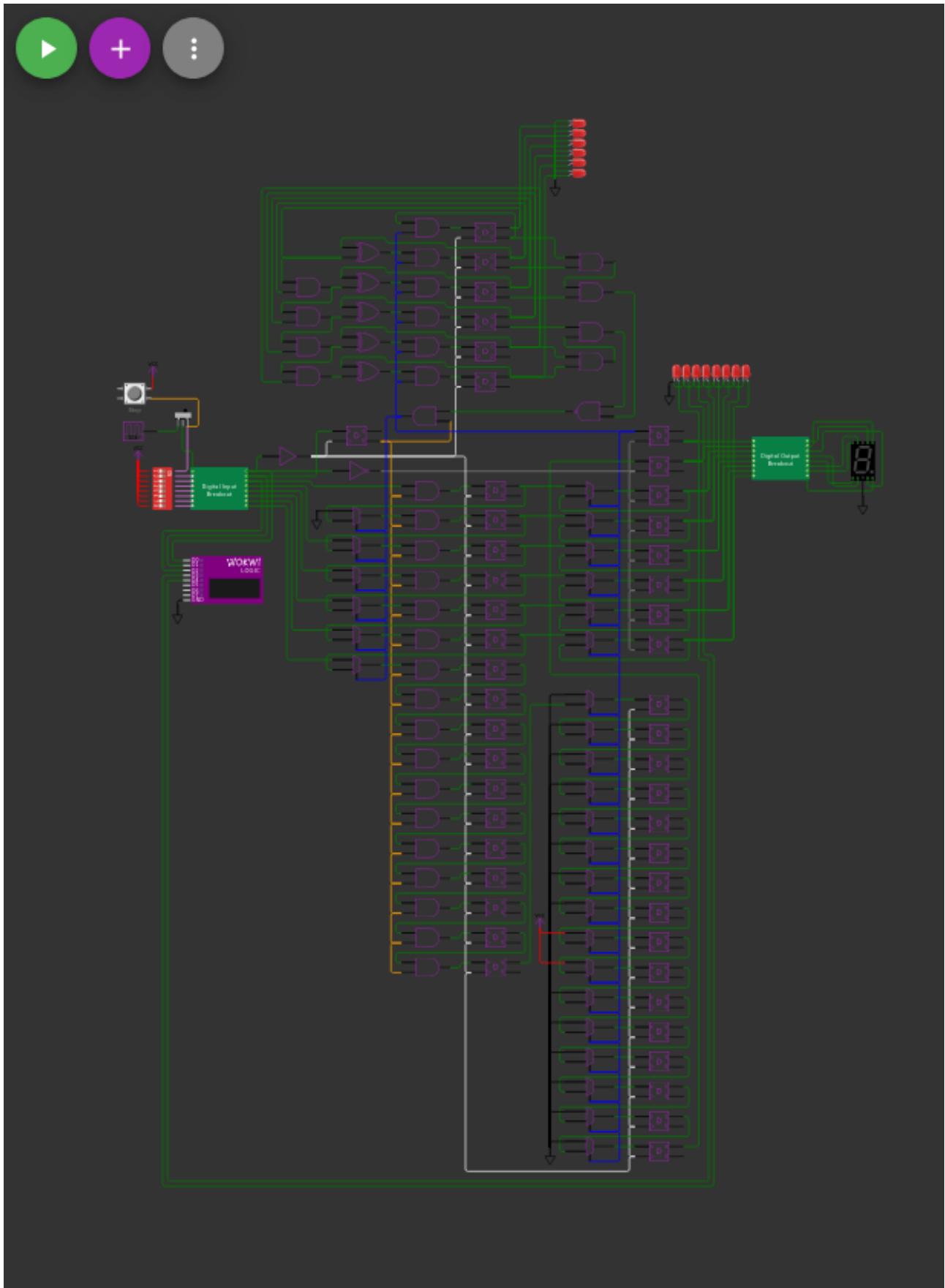


Figure 44: picture

## 172 : 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

## 173 : 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

# 174 : 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

## 175 : 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

# 176 : 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

## 177 : 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

## 178 : 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

## 179 : 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

## 180 : 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

## 181 : 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

## 182 : 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

## 183 : Scalable synchronous 4-bit tri-directional loadable counter

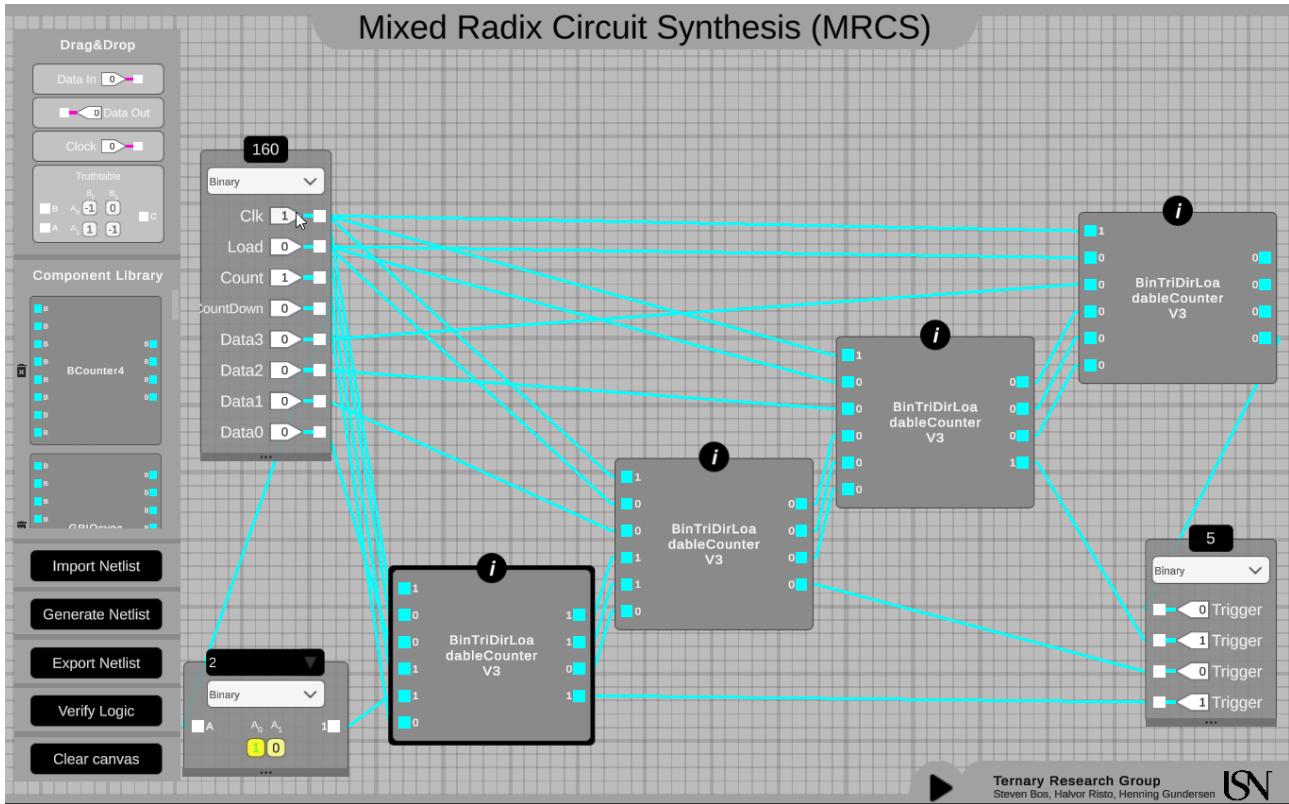


Figure 45: picture

- 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

## 184 : Asynchronous Binary to Ternary Converter and Comparator

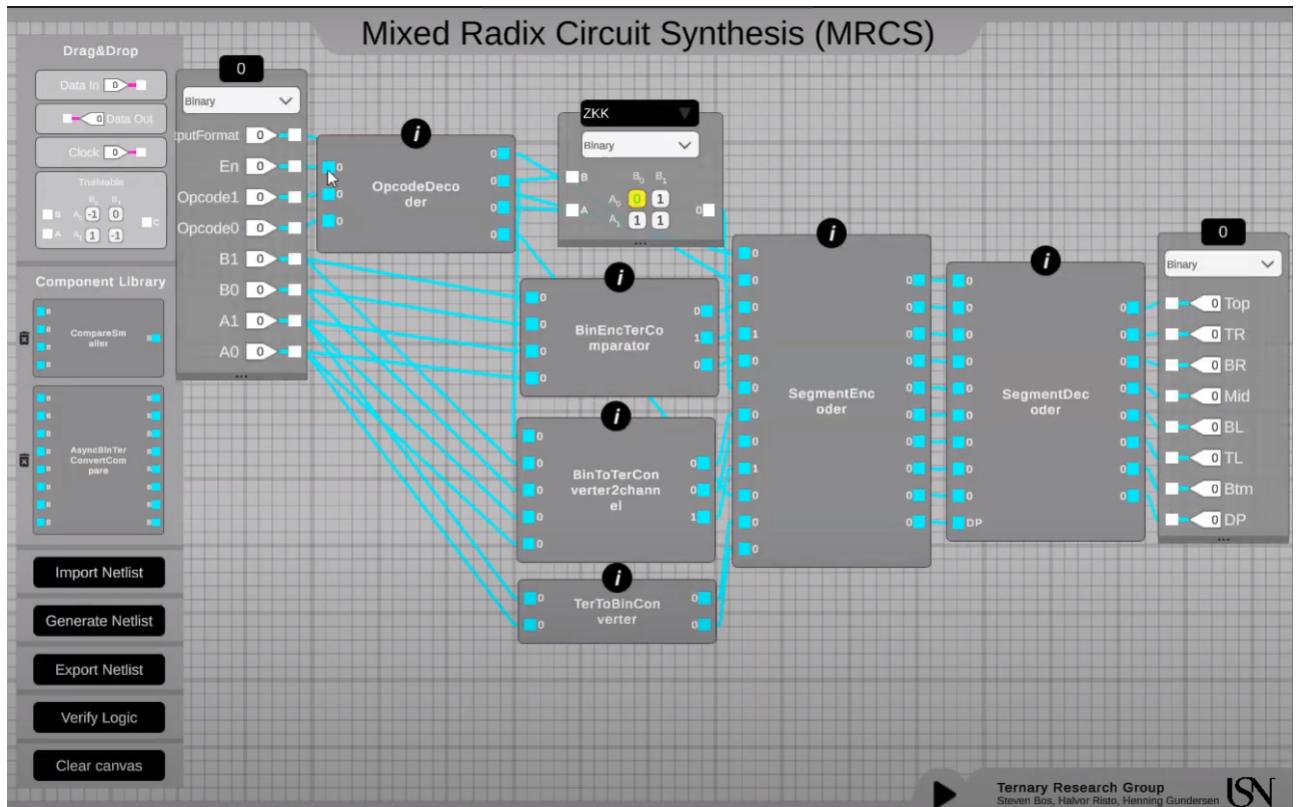


Figure 46: picture

- Author: Steven Bos
- Description: This chip offers various kinds of conversions and comparisons between binary encoded ternary and unary encoded ternary in both machine readable output and human readable (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)

## 185 : 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

$a := \text{in}[0:1]$ ,  $b := \text{in}[2:3]$ ,  $c := \text{in}[4:5]$ ,  $d := \text{in}[6:7]$  -  $[a,b,c,d] \Rightarrow [ac * bd]$

### How to test

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

### 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

## 186 : 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

## 187 : 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

## 188 : 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, 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

## 189 : 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
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

## 190 : 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

## 191 : 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, cycles 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. msbLsb 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	msbLsb	High/low nibble indicator

# 192 : 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

## 193 : 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

## 194 : 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 constraint length 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 IN3 to IN8), 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 12 clock cycles (6 input bits  $\times$  2 symbol bits = 12), after each clock cycle a 0 or 1 is displayed on the 8-segment display. The encoded output for the input 0b100110 is 0b101110010001 (left-to-right == first-to-last-bit displayed).

### 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)

## 195 : 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  $\{(1 \ll \text{address}), \text{data}\}$  (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

## 196 : 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

## 197 : 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

## 198 : 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

## 199 : 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

## 200 : Wolf sheep cabbage river crossing puzzle ASIC design

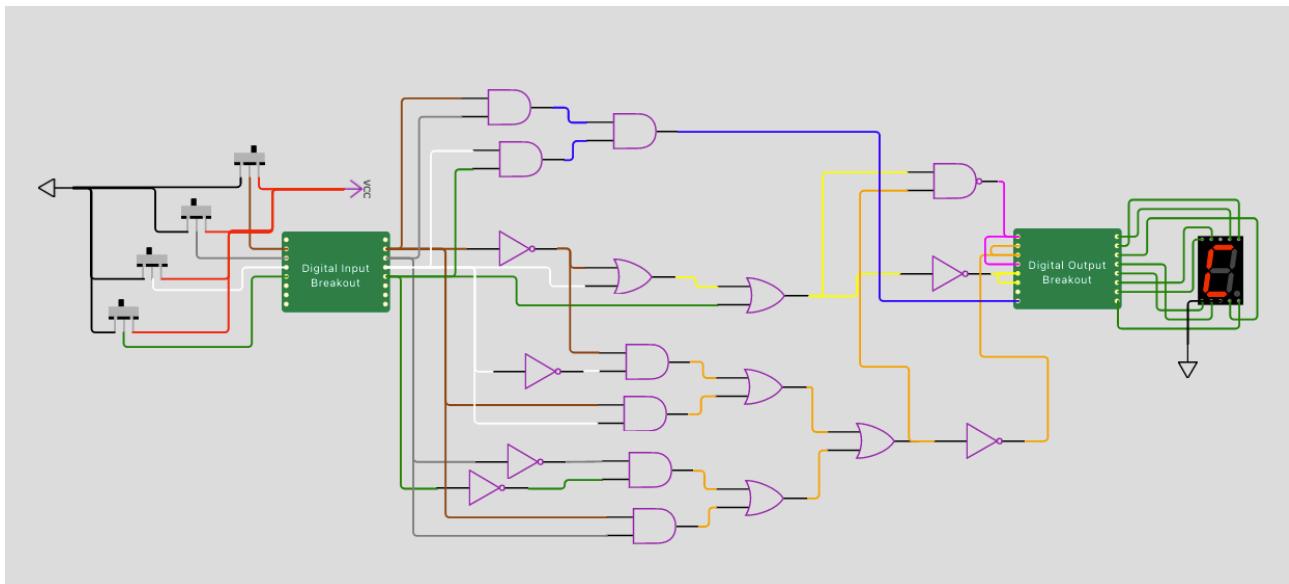


Figure 47: picture

- 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 $\sim E$ , 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 $\sim R$ 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 $\sim R$ 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 $\sim E$ , 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 $\sim L$ 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 $\sim L$ 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!

## 201 : Low-speed UART transmitter with limited character set loading

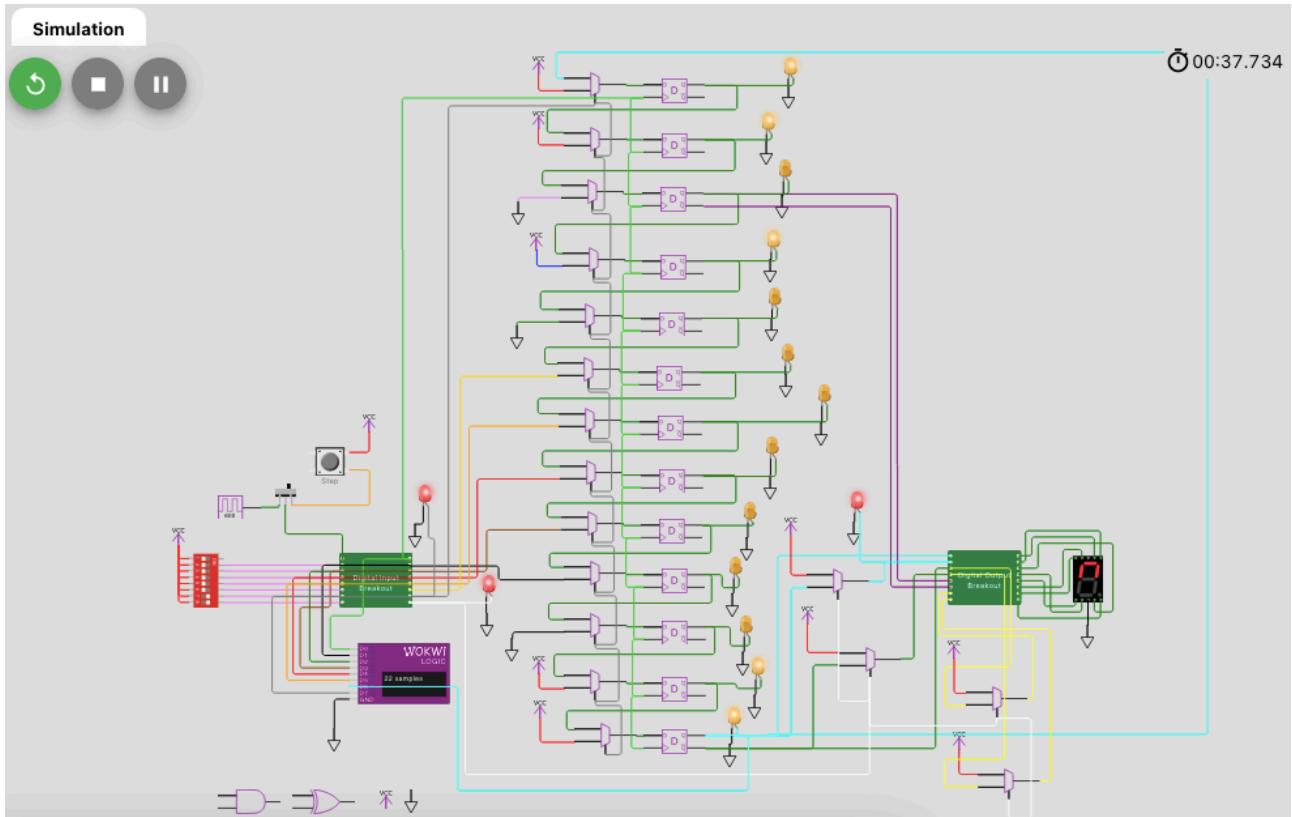


Figure 48: picture

- 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

## 202 : Rotary encoder

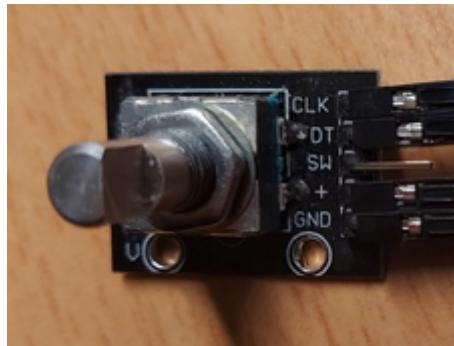


Figure 49: picture

- 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

## 203 : 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

## 204 : 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 as follows (bits between brackets):

- [24:31] 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  $\text{clock\_hz} / (\text{detents} \times \text{transitions} - 16)$  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

# 205 : Baudot Converter

picture

- Author: Arthur Hazleden
- Description: This circuit will convert 5-bit Baudot from a teletype machine to 8-bit ASCII.
- GitHub repository
- HDL project
- Extra docs
- Clock: 9600 Hz
- External hardware: “An optoisolaor is required at the Baudot input (IN1). A USB serial adapter or RS232 converter should be connected at the ASCII output (OUT0). Hopefully the onboard clock can drive the ASCII UART at 9600 and lower baud. Baudot uses 45.5 Hz and a 100x clock divider drives the UART. Since the TTY machines are not always on spec, drive IN1 with an adjustable 4550 Hz source. Baudot Out Ready and baudot[4:0] are available for debugging purposes.”

## How it works

Two UARTs, a clock divider and a conversion ROM

## How to test

“Provide 9600Hz at IN0 and 5000Hz at IN1. This sets up a 50 baud input and 9600 baud output. Use a PC set for 50 5n1 to drive the Baudot input. Check the baudot[4:0] pins and baudot\_ready(OUT1) if the ASCII output isn’t making sense.”

## IO

#	Input	Output
0	ascii clock at 8x desired baudrate	ASCII serial output at 9600
1	baudot clock at 100x desired baudrate	Baudot UART output b
2	baudot input, should be held high when line is idle but connected	none
3	none	Baudot bit 0
4	none	Baudot bit 1
5	none	Baudot bit 2
6	none	Baudot bit 3
7	none	Baudot bit 4

## 206 : 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-flops 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

## 207 : 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

## 208 : 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. Simultaneously, 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

## 209 : Adder with 7-segment decoder

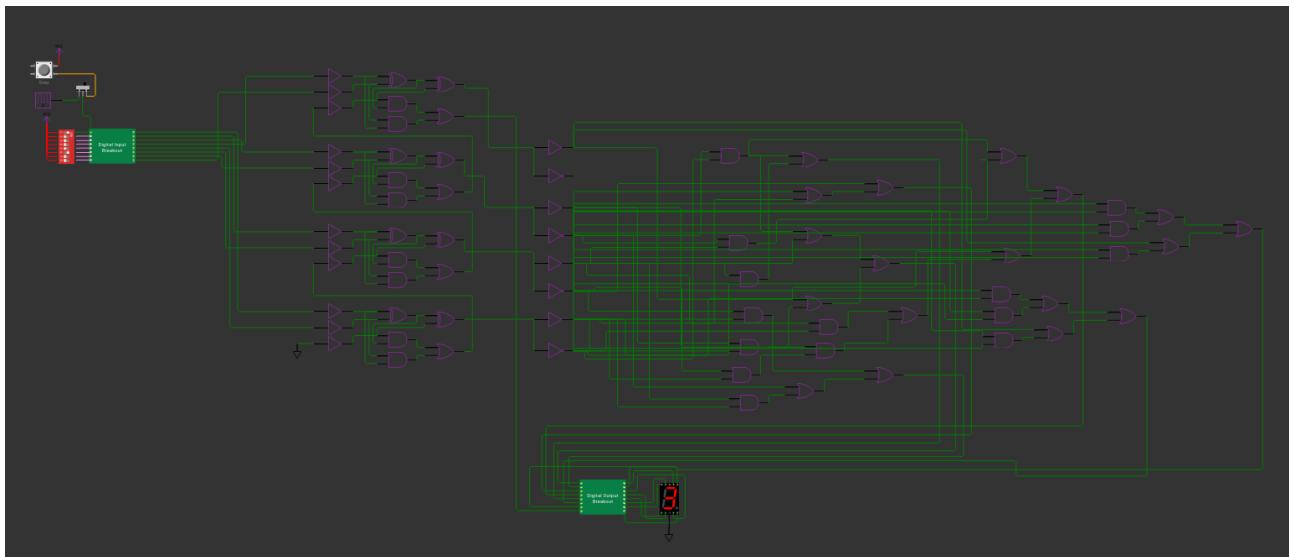


Figure 50: picture

- 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

#	Input	Output
2	first number bit 2	segment c
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)

## 210 : 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

## 211 : 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

## 212 : 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

## 213 : 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

## 214 : 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]

## 215 : Test3

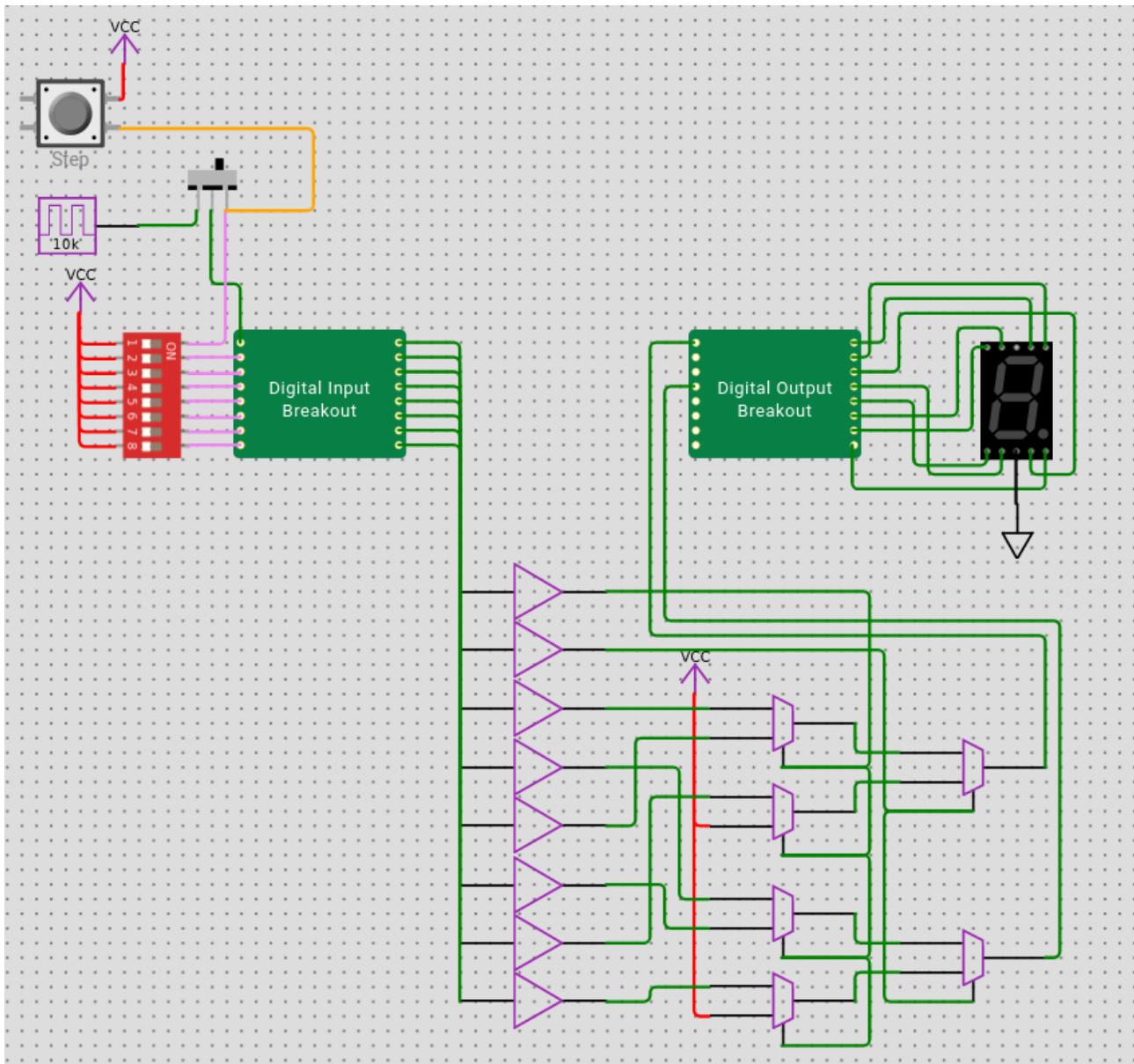


Figure 51: picture

- Author: Shaos
- Description: Binary Coded Ternary Test
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

Should work as ternary selector

## How to test

Set inputs, read outputs

## IO

#	Input	Output
0	C+	segment a
1	C-	segment b
2	N+	segment c
3	N-	segment d
4	O+	segment e
5	O-	segment f
6	P+	segment g
7	P-	dot

## 216 : Seven Segment Clock

picture

- Author: Greg Davill
- Description: Logic to drive 6 external 74hct595's that in turn drive 7 segment displays. The displays form a digital clock.
- GitHub repository
- HDL project
- Extra docs
- Clock: 128Hz Hz
- External hardware: 6x 74hct595's, 6x 7segment

### How it works

TBD

### How to test

TBD

### IO

#	Input	Output
0	clock	sclk
1	reset	latch
2	minute_up	data
3	hour_up	none
4	none	none
5	none	none
6	none	none
7	none	none

## 217 : serv - Serial RISCV CPU

picture

- Author: Greg Davill
- Description: An award winning RISCV CPU!
- GitHub repository
- HDL project
- Extra docs
- Clock: 0 Hz
- External hardware: tbd

### How it works

This project contains a 96bit serial scanchain, and the core of the serv CPU. Signals present on the scanchain are a wishbone bus and the native registerfile interface. As there is not enough room inside the TinyTapeout project area to fit RAM/registerfiles these have to be implemented externally. In theory just a bit of custom code running on caravel will be enough to get the serv core running.

### How to test

tbd

### 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

## 218 : 4:2 Compressor

- Author: saicharan0112
- Description: A Basic 4:2 compressor which contains 4 inputs and 1 carry\_in bit which compresses to 2 outputs and 1 carry\_out bit
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

It compresses 4 inputs to 2 inputs. This is used to compress partial products inside multipliers

### How to test

Follow the truth table of 4:2 compressor online with 4 inputs and carry\_in bit and observe the 2 outputs and carry\_out bit

### IO

#	Input	Output
0	a1 is one of the 4 main input bits	o1 is the one of the 3 compressed output bits
1	a2 is one of the 4 main input bits	o2 is the one of the 3 compressed output bits
2	a3 is one of the 4 main input bits	cout is the carry_out bit
3	a4 is one of the 4 main input bits	none
4	cin is the carry_in input bit	none
5	none	none
6	none	none
7	none	none

## 219 : PS2 keyboard Interface

- Author: Tanish Khanchandani
- Description: PS2 keyboard interface to enter characters into a computer. Use the PS2 hex scan codes (<https://techdocs.altium.com/display/FPGA/PS2+Keyboard+Scan+Codes>) to enter hex codes and it will send the letter to your computer.
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 6000 Hz
- External hardware: 3.3V to 5V logic level converter

### How it works

Most likely does not work. Takes in keyboard hex scan codes and sends data to your PC. The chip emulates a key being pressed down and released. (link to protocol - [https://techdocs.altium.com/display/FPGA/PS2\\_W--Transmission+Protocols](https://techdocs.altium.com/display/FPGA/PS2_W--Transmission+Protocols)). The logic puts data into a parallel to serial interface and sends the data with some of the other protocol necessities to simulate a key being pressed and released.

### How to test

Select clock with input 1. Set the first hex character using inputs 2-5. Set input 6 to 1. Set input 6 back to 0. Set the second hex character using inputs 2-5 and set input 7 to 1 and then back to 0. Set input 8 to 1 to send the data.

### IO

#	Input	Output
0	clock	NC
1	hex Bit 1	NC
2	hex Bit 2	NC
3	hex Bit 3	NC
4	hex Bit 4	NC
5	Set 1st hex	NC
6	set 2nd hex	Clock
7	Enable to send	Data

## 220 : Hello Generator

- Author: Skyler Saleh
- Description: Flashes 'H-E-L-L-O' on the 7 segment display
- GitHub repository
- HDL project
- Extra docs
- Clock: 2048 Hz
- External hardware: None

### How it works

An input clock signal is fed into a configurable clock divider which generates a slower clock every 1 to  $2^{15}$  cycles (depending on configuration). The rate of the clock divider is configured using the dipswitches under the equation of  $\text{output\_clock\_hz} = \text{input\_clock\_hz}/(2^{\text{clock\_divider\_ratio}[3:0]})$ . This slow clock increments a 3 bit counter which is used to index a built in character generator ROM, whose outputs will be used to drive the segment a,b,c,d,e,f,g on the 7 seg display. The character rom contains bits to light up the segments as 'H-E-L-L-O- - -'. The outputs of the character rom are anded with ( $\text{slow\_clock} \& \text{flash\_enable}$ ) to cause the display to blank between letters when flashing is enabled. A debug harness (accessed by setting the `debug_mode` dip switch to 1) allows the character generator rom to be indexed using dip switch settings, and for the slow clock to be source from dip switches instead of the clock divider.

### How to test

Configure input clock rate as 2048hz on the first input. Set `dip_switch[1]` to 1 Set `dip_switch[2]` to 1 Set `dip_switch[3]` to 0 Set `dip_switch[4]` to 1 This will configure the input clock divider to generate a 1Hz slow clock from the 2048hz input clock. Set `dip_switch[5]` to 0 Set `dip_switch[6]` to 0 Set `dip_switch[7]` to 0 This will disable the test harness and setup normal operation. Connect a 7 segment display to outputs, and the device should flash 'H-E-L-L-O' followed by 3 letters worth of blank display.

### IO

#	Input	Output
0	clock	segment a
1	if <code>debug_mode == 0:</code> <code>clock_divider_ratio[0]</code> elif <code>debug_mode == 1:</code> <code>character_rom_index[0]</code>	segment b

#	Input	Output
2	if debug_mode == 0: clock_divider_ratio[1] elif debug_mode == 1: character_rom_index[1]	segment c
3	if debug_mode == 0: clock_divider_ratio[2] elif debug_mode == 1: character_rom_index[2]	segment d
4	if debug_mode == 0: clock_divider_ratio[3] elif debug_mode == 1: slow_clock_output (used for flash generator)	segment e
5	flash enable: 0 = Flash display between each output letter. 1 = Do not flash display.	segment f
6	must be zero: 0 = Prints 'H-E-L-L-O' 1 = Implementation defined behavior	segment g
7	debug_mode: 0 = normal operation, 1 = debug mode	segment decimal

## 221 : MicroASIC VI

- Author: Mikhail Svarichevsky
- Description: Free-running oscillators to verify simulation vs reality + TRNG
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware: You might benefit from frequency counter than can do averaging across extended period of time.

### How it works

Combinational loops with dividers to bring output frequency to <50kHz range

### How to test

Select oscillator (pins 4-6) and measure frequency on one of output pins. Observe true random numbers at pin 7.

### IO

#	Input	Output
0	clock in (for debugging)	clock divided by $2^{10}$
1	reset	clock divided by $2^{14}$
2	shift register clk	clock divided by $2^{18}$
3	shift register data	clock divided by $2^{22}$
4	clock source id_0	clock divided by $2^{26}$
5	clock source id_1	clock divided by $2^{30}$
6	clock source id_2	clock divided by $2^{32}$
7	unused	Bit 11 of shift register (to ensure it's not optimized away)

## 222 : Optimised Euclidean Algorithm

- Author: Recep Said Dulger
- Description: Finding gcd of 2 4-bit number
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware: None

### How it works

This circuit finds the gcd (greatest common divisor) of 2 4-bit numbers which are entered by dip switch and it uses the Euclidean algorithm. Result displays by seven segment display. The algorithm has been optimized by designing the control unit and datapath.

### How to test

Enter 4-bit 1st number by dip switches and set num\_okey switch to 1. By doing that 1st number saved in register. Set num\_okey switch to 0 and enter 2nd 4-bit number. Set num\_okey switch to 0 and after that gcd result will appear in seven segment display.

### IO

#	Input	Output
0	clock	ssd_out[0]
1	number[0]	ssd_out[1]
2	number[1]	ssd_out[2]
3	number[2]	ssd_out[3]
4	number[3]	ssd_out[4]
5	none	ssd_out[5]
6	rst	ssd_out[6]
7	num_okey	none

## 223 : CRC-16 and Parity calculator

- Author: Chris Burton
- Description: CRC-16/XModem and Even Parity calculator based on Ben Eater error detection videos.
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware: Can be used with any microcontroller, or by toggling switches.

### How it works

Two banks of CRC calculators (A and B) each with inputs for the shift register data and clock, active low reset (when high toggle shiftClk to reset) and read back mode which disables the feedback XOR to allow reading data back out.

### How to test

Connect Pico as shown in Wokwi and run test code to send a string, read back calculated CRC/parity and compare.

### IO

#	Input	Output
0	nRst_A	crcOutput_A
1	shiftData_A	parity_A
2	shiftClk_A	none
3	nRead_A	none
4	nRst_B	crcOutput_B
5	shiftData_B	parity_B
6	shiftClk_B	none
7	nRead_B	none

## 224 : SevSegFX

- Author: Mazen Saghir, ECE Department, American University of Beirut (mazen@aub.edu.lb)
- Description: Seven segment display effect generator
- GitHub repository
- HDL project
- Extra docs
- Clock: 12500 Hz
- External hardware:

### How it works

Generates hexadecimal digits and 16 dynamic patterns on the seven segment display.

### How to test

Use input[7] to display digits (=0) or effects (=1). Use input[6] to blink displayed digits (=1) or not (=0). Only digits can be blinked. Use inputs [5:2] to select digit or effect pattern to display.

### IO

#	Input	Output
0	clock	segment a
1	reset	segment b
2	sel0/d0	segment c
3	sel1/d1	segment d
4	sel2/d2	segment e
5	sel3/d2	segment f
6	blink	segment g
7	fx	none

## 225 : LAB11

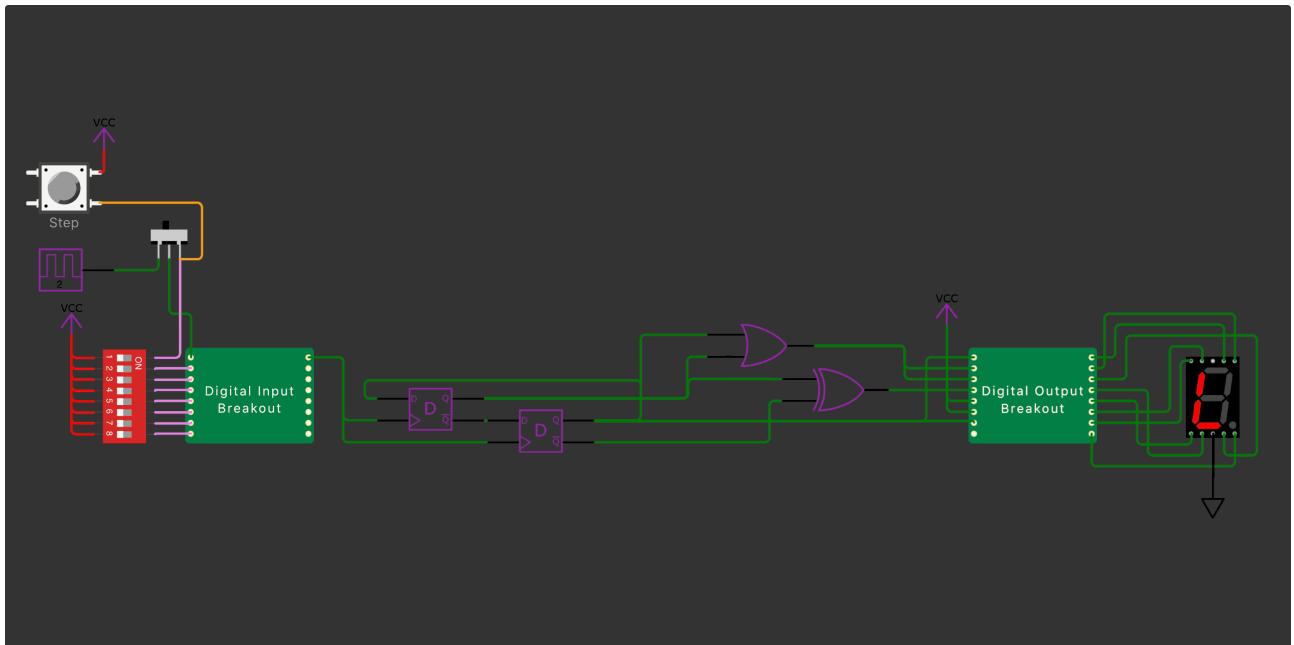


Figure 52: picture

- Author: Thomas Zachariah
- Description: Cycles through the characters of LAB11
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 1 Hz
- External hardware: None

### How it works

Gates & flip-flops connected to the 7-segment display change the state of corresponding LED segments to form the next character, each cycle

### How to test

Set to desired clock speed — characters are most readable at the lowest speed

### IO

#	Input	Output
0	clock	segment a
1	reset	segment b

#	Input	Output
2	none	segment c
3	none	segment d
4	none	segment e
5	none	segment f
6	none	segment g
7	none	none

## 226 : Option23 Serial

- Author: bitluni
- Description: Character ROM and bitmap shifter for POV displays
- GitHub repository
- HDL project
- Extra docs
- Clock: 0 Hz
- External hardware: Pink LEDs

### How it works

clock: clocks out character columns or shifts in data. Data is 8bits LSB first. Highest bit is ignored. Data = b01xxxxx ASCII character no x + 32. data = b00xxxxxx bitmap column. under/over is underline and overline for all bitmap columns

### How to test

Shift in some data and set din = 1111111 to clock out characters and graphics

### IO

#	Input	Output
0	clock	led 0
1	reset	led 1
2	write	led 2
3	din	led 3
4	under	led 4
5	over	led 5
6	none	led 6
7	none	led 7

## 227 : Option23

- Author: bitluni
- Description: Character ROM and bitmap shifter for POV displays
- GitHub repository
- HDL project
- Extra docs
- Clock: 0 Hz
- External hardware: Pink LEDs

### How it works

clock: clocks out character columns or shifts in data. din = 1111111: show characters and bitmaps column by column. din = 1xxxxxx shift in ASCII character x. din = b10xxxxx : shift in bitmap column xxxx

### How to test

Shift in some data and set din = 1111111 to clock out characters and graphics

### IO

#	Input	Output
0	clock	led 0
1	din 0	led 1
2	din 1	led 2
3	din 2	led 3
4	din 3	led 4
5	din 4	led 5
6	din 5	led 6
7	din 6	led 7

## 228 : Option22

- Author: bitluni
- Description: Looong shift register. 22x8 bit
- GitHub repository
- HDL project
- Extra docs
- Clock: 0 Hz
- External hardware: Pink LEDs

### How it works

write=high: Data is shifted-in on clock positive edge. Each 8 clocks a full byte is buffered at the output. It rotates all 22 words. Reset only resets internal counter for the bit index.

### How to test

Keep write high and push 22x8 bits in. Keep clock with write low to receive the bytes at the output

### IO

#	Input	Output
0	clock	led 0
1	reset	led 1
2	write	led 2
3	data	led 3
4	none	led 4
5	none	led 5
6	none	led 6
7	none	led 7

## 229 : 4x4 RAM

- Author: Michael Bartholic
- Description: 4 word, 4 bit read/write RAM
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware: None.

### How it works

Set write enable to write to the given address. Read given address on output.

### How to test

Set a word on data lines, set address on addr lines. Cycle write enable. Try reading value on rdata.

### IO

#	Input	Output
0	clock	rdata[0]
1	data[0]	rdata[1]
2	data[1]	rdata[2]
3	data[2]	rdata[3]
4	data[3]	addr[0]
5	addr[0]	addr[1]
6	addr[1]	clock
7	write_enable	write_enable

## 230 : Digital padlock

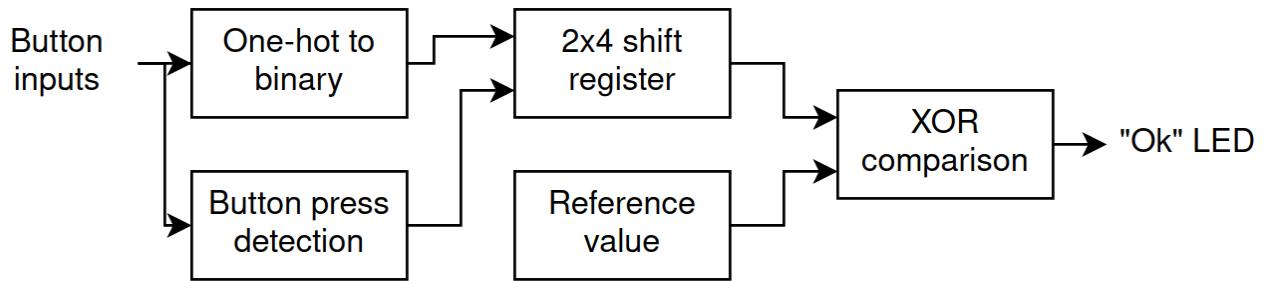


Figure 53: picture

- Author: Jean THOMAS
- Description: A 4-digit electronic padlock
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 100 Hz
- External hardware: Four push buttons, cabled as active high with hardware debounce circuitry.

### How it works

Each button's press is detected by a rising edge detector, and each button press is decoded into a binary code. That binary code is stored in a shift-register which is continuously checked against a reference value ('the padlock code').

### How to test

Connect a clock generator to the clock input, connect all four buttons with a debounce circuit - the buttons should act as active high.

### IO

#	Input	Output
0	clock	none
1	Button A	none
2	Button B	none
3	Button C	none
4	Button D	none

#	Input	Output
5	none	none
6	none	Button press detected
7	none	Code valid

## 231 : FFT Butterfly in Wokwi

- Author: James R
- Description: Single FFT butterfly with 2-bit resolution
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware: None

### How it works

Calculates low-resolution FFT of two 2-bit inputs

### How to test

Change the inputs

### IO

#	Input	Output
0	xr0.0	xr0.0
1	xr0.1	Xr0.1
2	xj0.0	Xj0.0
3	xj0.1	Xj0.1
4	xr1.0	Xr1.0
5	xr1.1	Xr1.1
6	xj1.0	Xj1.0
7	xj1.1	Xj1.1

## 232 : Femto 4-bit CPU

- Author: Majdi Abdul Samad, ECE Dept., American University of Beirut (mia42@mail.aub.edu)
- Description: Design of a small single-cycle CPU with simple RISC/Accumulator ISA
- GitHub repository
- HDL project
- Extra docs
- Clock: 5 Hz
- External hardware: None

### How it works

NOTE: ISA is included in the ReadMe. Contains a register file, ALU, and 7 segment decoder. Instructions are sent in from inputs 7 downto 1 (0 reserved for clk), the register source and destination are sent to the register file (synch write/asynch read). Opcode and register read data are sent to the ALU for the operation. The output data could be stored in the ALU, the register file, or sent to the 7 segment decoder to power the LED output. See the ReadMe for more details.

### How to test

Design was tested with a ModelSim TCL script, provided here and should be compatible with other TCL accepting simulators. A cocotb testbench will also be made available.

### IO

#	Input	Output
0	clock	segment a
1	opcode[0]	segment b
2	opcode[1]	segment c
3	opcode[2]	segment d
4	reg_dest[0]	segment e
5	reg_dest[1]	segment f
6	reg_src[0]	segment g
7	reg_src[1]	none

## 233 : Logisim demo - LED blinker

- Author: Tholin
- Description: Example of how to use Logisim Evolution generated Verilog for TinyTapeout.
- GitHub repository
- HDL project
- Extra docs
- Clock: 2 Hz
- External hardware: A button for reset, some way to display the output (LEDs)

### How it works

Its a 4-bit ring-shift register with a single '1' cycling through it after reset.

### How to test

After starting the clock, the 4 outputs will remain off or in a random state until the reset input is activated. Then it should work as described.

### IO

#	Input	Output
0	CLK	O_0
1	RST	O_1
2	none	O_2
3	none	O_3
4	none	none
5	none	none
6	none	none
7	none	none

## 234 : Secret File

- Author: bitluni
- Description: Nothing to see here
- GitHub repository
- HDL project
- Extra docs
- Clock: 0 Hz
- External hardware: Shredder

### How it works

Leving it alone works

### How to test

Don't test me

### IO

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

## 235 : Basic 4 bit cpu

- Author: Noah Gaertner
- Description: 4-bit CPU that does add, subtract, multiply, left and right shifts, conditional jump based on external signal, logical and bitwise AND and OR, equality and inequality checking, bitwise inversion, and logical NOT
- GitHub repository
- HDL project
- Extra docs
- Clock: 50K (or lower, whatever) Hz
- External hardware: test pattern generator, output reader (will probably work with just an arduino for both)

### How it works

Implements a highly reduced ISA that fits on the limited allowed space, and uses a 4-bit bus to get the program and data values in and out of the chip, in addition to a two bit bus to tell it what to do at any given time, as well as a clock and reset signal

### How to test

Write a program for the ISA and try to run it! Remember you need to synchronously RESET and then SETRUNPT to the proper value before you try to do anything!

### IO

#	Input	Output
0	clock	program counter
1	reset	program counter
2	instruction	program counter
3	instruction	program counter
4	data	output data
5	data	output data
6	data	output data
7	data	output data

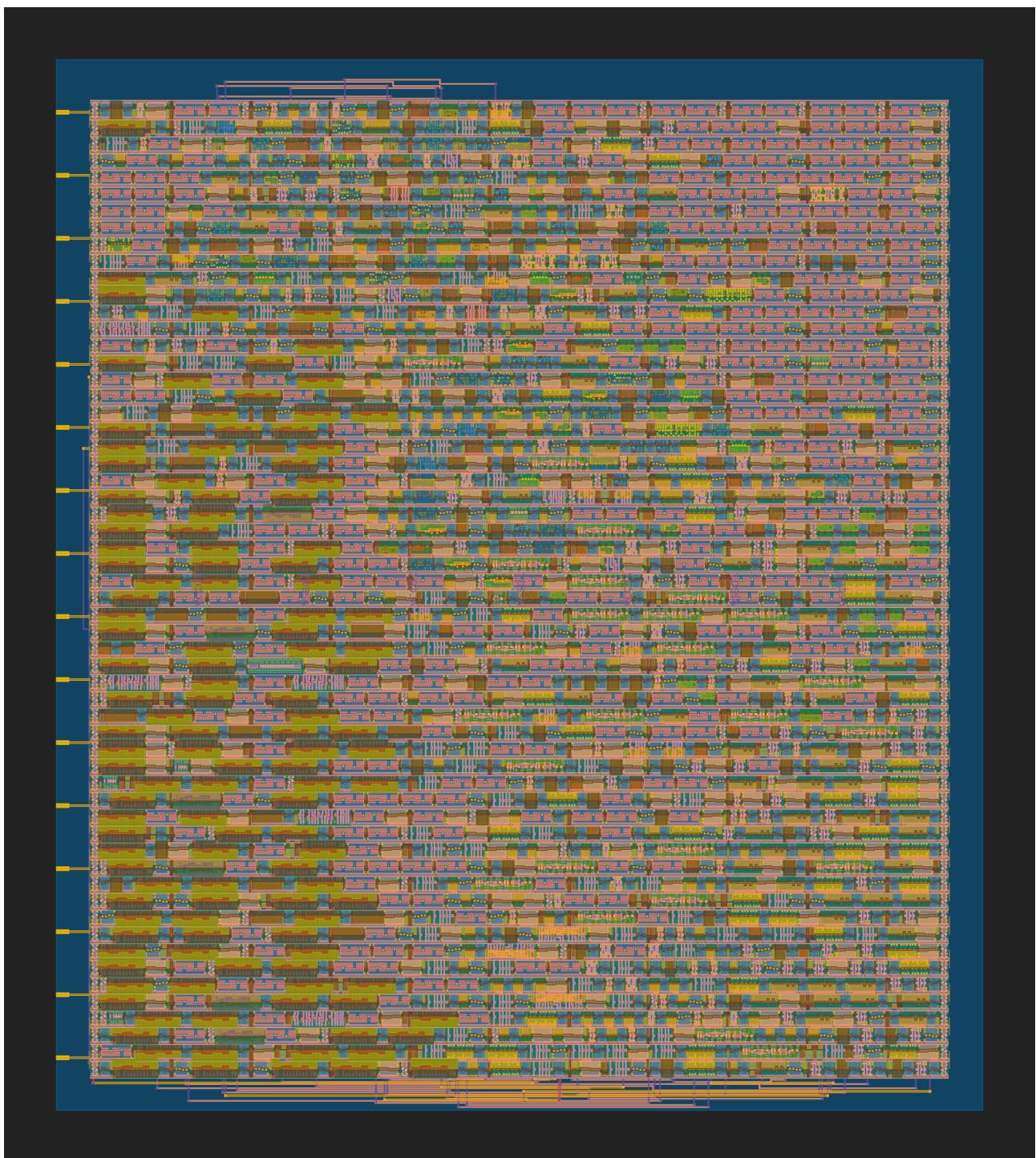


Figure 54: picture

## 236 : Adi counter

- Author: Prabal Dutta
- Description: Test FSM
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 1 Hz
- External hardware: Just PB and 7-seg

### How it works

Clocks FSM on button push

### How to test

Hook up to 7-deg display, push button, and see A-d-i cycle on LEDs

### 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

## 237 : Clock divider ASIC

- Author: Sad Electronics
- Description: Uses a series of flip flops to divide the clock
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

todo

### How to test

todo

### 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

## 238 : Amaranth 6 Bits Gray counter

picture

- Author: Camilo Soto
- Description: Amaranth Gray 6 Bits gray counter
- GitHub repository
- HDL project
- Extra docs
- Clock: 3000 Hz
- External hardware: None

### How it works

The reflected binary code (RBC), also known as reflected binary (RB) or Gray code after Frank Gray, is an ordering of the binary numeral system such that two successive values differ in only one bit (binary digit). For example, the representation of the decimal value “1” in binary would normally be “001” and “2” would be “010”. In Gray code, these values are represented as “001” and “011”. That way, incrementing a value from 1 to 2 requires only one bit to change, instead of two (Wikipedia [https://en.wikipedia.org/wiki/Gray\\_code](https://en.wikipedia.org/wiki/Gray_code))

### How to test

Apply clk to the in[0], rst on in[1]

### IO

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

## 239 : Laura's L

- Author: Laura
- Description: Makes an L on the 7 segment when you press buttons 1 & 2
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

just an and gate

### How to test

press buttons 1 & 2 to see the L

### IO

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

## 240 : Two Bit Universal FSM

- Author: Emilio Alvarez
- Description: A two bit FSM inspired by Rue Mohr's 1Bit CPU
- GitHub repository
- HDL project
- Extra docs
- Clock: 1-1000 Hz
- External hardware: A microcomputer to load the RAM instructions (can be an Arduino), and a clock to run the FSM

### How it works

It's a Moore Finite State Machine where the “next step logic” is synthesized by a Dual-port RAM and latched in a register. “Instructions” are loaded into the ram using port A through a 16 bit shift register, clocked in A7-A0-D7-D0 format; once loaded they are written into RAM gating the RW pin. The FSM itself uses port B of ram. Two inputs are concatenated with the address, forming the “next state”, and the two outputs are taken from the Data bus memory. The CLK runs the machine.

### How to test

- Initial conditions: clk\_shft=0, clk\_data=0, wr\_shft=0, reset=0; clk\_cpu=0, entradas=00 (sorry for the spanish)
- Pulse reset line (0->1->0)
- Gate 16 bits using clk\_shft and data\_shft inputs, knowing that (MSB)(LSB) are (A7 downto A0)(D7 downto D0) “instructions”, rising edge clock active.
- Once 16 clocks are gated, pulse wr\_shft to 1 (0->1->0). Now, contents of shift register are written into RAM
- Once the 256 (or desired) instructions are written in ram, start clk\_cpu.
- Output salidas(Q1,Q0) should start responding to input entradas(D1,D0) in a finite state machine fashion, in sync with cl.

### IO

#	Input	Output
0	{‘clk_shft’: ‘clock to the 16 bit shift register, rising clock active.’}	{‘salidaQ0’: ‘bit Q0’}
1	{‘data_shft’: ‘data to be gated into the 16 bit shift register’}	{‘salidaQ1’: ‘bit Q1’}

#	Input	Output
2	{'wr_shft': 'a pulse here transfers the contents from shift register to RAM'}	{'none': 'unused'}
3	{'reset': 'active high, resets the 16bit shift reg and the Next State reg'}	{'none': 'unused'}
4	{'clk_cpu': 'clock to run the FSM'}	{'none': 'unused'}
5	{'entradaD0': 'input bit D0'}	{'none': 'unused'}
6	{'entradaD1': 'input bit D1'}	{'none': 'unused'}
7	{'none': 'unused'}	{'none': 'unused'}

## 241 : Super Mario Tune on A Piezo Speaker

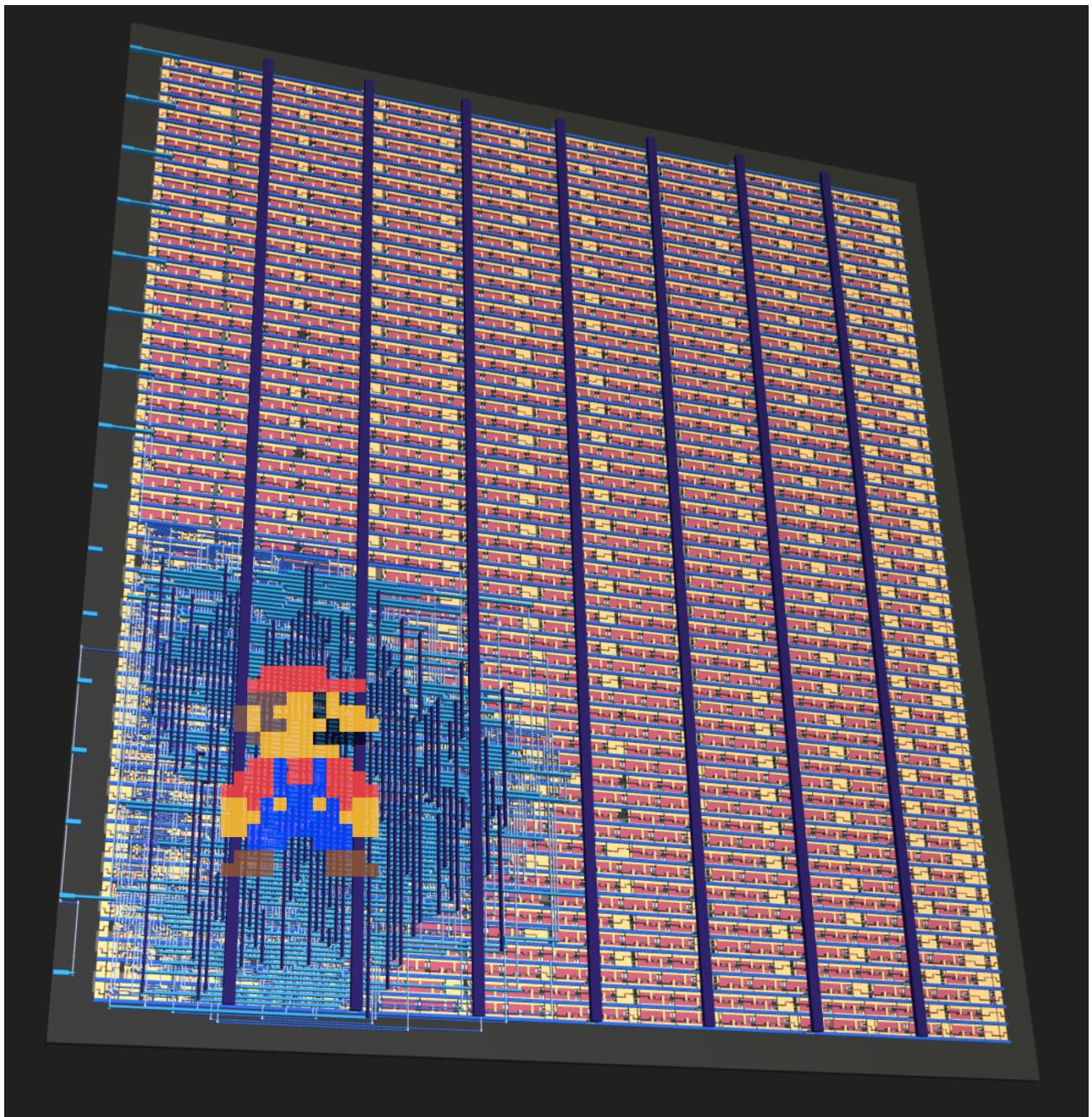


Figure 55: picture

- 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: 6250 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 6.25kHz 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

## 242 : PSRANDOM

- Author: CMUA F.Segura-Quijano, J.S.Moya
- Description: Pseudo Random generator.
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware:

### How it works

Uses a General Register controled with a State Machine with a Pseudo Random Generator Circuit.

Puts a pseudo random data un wave output bus.

### How to test

After reset, push load o rand button. Load change one time data out bus. Rand change continius data out bus.

### IO

#	Input	Output
0	BB_SYSTEM_CLOCK_50	BB_SYSTEM_data_OutBUS[7]
1	BB_SYSTEM_RESET_InHigh	BB_SYSTEM_data_OutBUS[6]
2	BB_SYSTEM_loadseed_InLow	BB_SYSTEM_data_OutBUS[5]
3	BB_SYSTEM_loaddata_InLow	BB_SYSTEM_data_OutBUS[4]
4	BB_SYSTEM_rand_InLow	BB_SYSTEM_data_OutBUS[3]
5	BB_SYSTEM_data_InBUS[0]	BB_SYSTEM_data_OutBUS[2]
6	BB_SYSTEM_data_InBUS[1]	BB_SYSTEM_data_OutBUS[1]
7	BB_SYSTEM_data_InBUS[2]	BB_SYSTEM_data_OutBUS[0]

## 243 : Gameshow Buzzer

- Author: Christopher Haddad, Jenna Nandlall, Matthew Nunez, Farhan Kobir
- Description: Jeopardy gameshow type of buzzer.
- GitHub repository
- HDL project
- Extra docs
- Clock: 0 Hz
- External hardware:

### How it works

Explain how your project works

### How to test

Explain how to test your project

### 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	dot

## 244 : Balanced Ternary Calculator

- Author: Steven Bos
- Description: A balanced ternary calculator allowing multiplication, addition and subtraction with negative numbers in binary encoded ternary
- GitHub repository
- HDL project
- Extra docs
- Clock: 0 Hz
- External hardware: none

### How it works

2DO

### How to test

2DO

### IO

#	Input	Output
0	x1H	s3H
1	x1L	s3L
2	x0H	s2H
3	x0L	s2L
4	Y1H	s1H
5	y1L	s1L
6	y0H	s0H
7	y0L (reused as function selector, eg add/subtract or multiply)	s0L

## 245 : RiscV Scan Chain based CPU – block 1 – clocking

- Author: Emilian Miron
- Description: RiscV Scan Chain based CPU – block 1 – clocking
- GitHub repository
- HDL project
- Extra docs
- Clock: 20000 Hz
- External hardware:

### How it works

TODO

### How to test

After reset, the counter should increase by one every second.

### 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	slow clock output

## 246 : RiscV Scan Chain based CPU – block 2 – instructions

- Author: Emilian Miron
- Description: RiscV Scan Chain based CPU – block 2 – instructions
- GitHub repository
- HDL project
- Extra docs
- Clock: 20000 Hz
- External hardware:

### How it works

TODO

### How to test

After reset, the counter should increase by one every second.

### 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	slow clock output

## 247 : RiscV Scan Chain based CPU – block 3 – registers

- Author: Emilian Miron
- Description: RiscV Scan Chain based CPU – block 3 – registers
- GitHub repository
- HDL project
- Extra docs
- Clock: 20000 Hz
- External hardware:

### How it works

TODO

### How to test

After reset, the counter should increase by one every second.

### 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	slow clock output

## 248 : RiscV Scan Chain based CPU – block 4 – ALU

- Author: Emilian Miron
- Description: RiscV Scan Chain based CPU – block 4 – ALU
- GitHub repository
- HDL project
- Extra docs
- Clock: 20000 Hz
- External hardware:

### How it works

TODO

### How to test

After reset, the counter should increase by one every second.

### 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	slow clock output

# Technical info

## Scan chain

All 250 designs are joined together in a long chain similar 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.

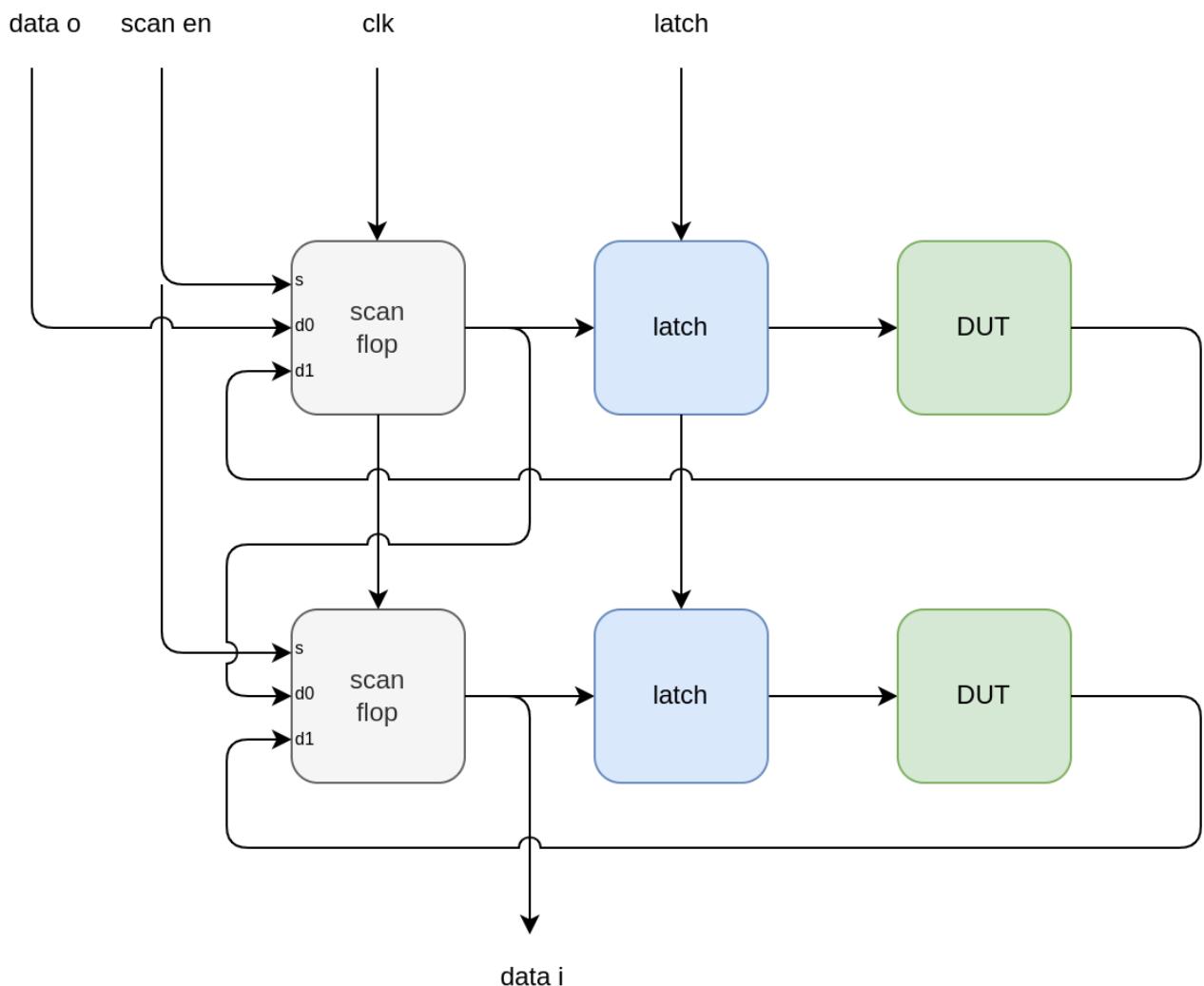


Figure 56: block diagram

## 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

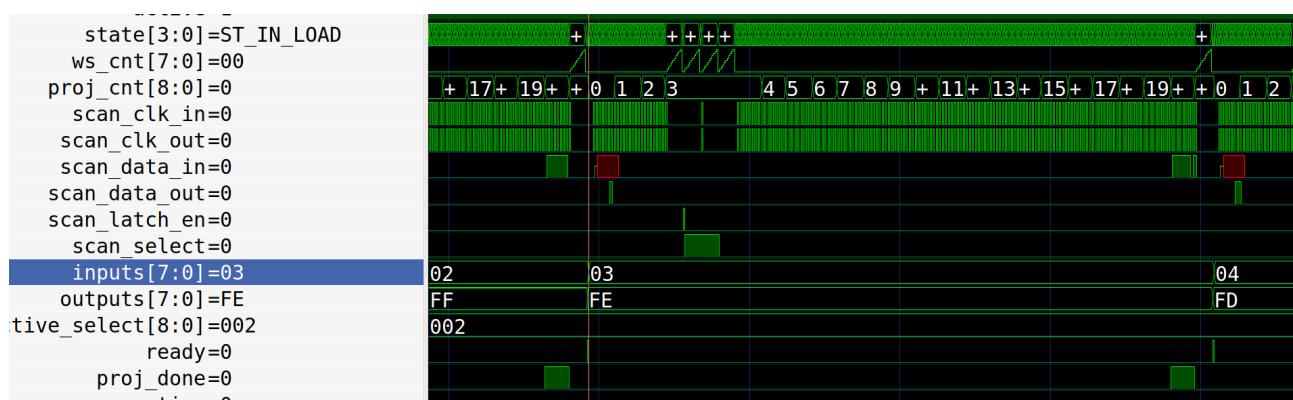


Figure 57: update cycle

#### *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  $100\text{MHz} / (8 * 2 * 250) = 25000\text{Hz}$ .

## 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  $\text{scan\_rate} / (2 * (N+1))$ .

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	<code>active_select</code>	9 bit input to set which design is active
28:21	<code>inputs</code>	8 inputs
36:29	<code>outputs</code>	8 outputs
37	<code>ready</code>	goes high for one cycle everytime the scanchain
10	<code>slow_clk</code>	slow clock from internal clock divider
11	<code>set_clk_div</code>	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 --fetch-gds
```

## 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 easiest 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 analysis 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>

# **Toplevel STA and Spice analysis for TinyTapeout-02 and 03.**

Contributed by J. Birch 22 March 2023

## **Introduction**

TinyTapeout is built using the OpenLane flow and consists of three major components:

- 1) up to 250 user designs with a common interface, each of which has been composed using the OpenLane flow,
- 2) a scanchain block that is instanced for each user design to provide input data and to sink output data,
- 3) a scan chain controller.

The OpenLane flow allows for timing analysis of each of these individual blocks but does not provide for analysis of the assembly of them, which leaves potential holes in timing that could cause failures.

This work allows a single design to be assembled out of the sub-blocks and this can then be submitted to STA for analysis.

In addition, an example critical path (for the clock that is passed along the scanchain) is created using a Python script so that it can be run through SPICE.

## **GitHub action**

The STA github action automatically runs the STA when the repository is updated.

## **Prerequisites**

To run STA, the following files need to be available:

- 1) a gate level verilog netlist of the top level design (`verilog/rtl/user_project_wrapper.v`)
- 2) a gate level verilog netlist for each sub-block (`verilog/rtl/`)
- 3) a SPEF format parasitics file for the top level wiring (`spef/`)
- 4) a SPEF format parasitics file for each sub-block (`spef/`)
- 5) the relevant SkyWater libraries for STA analysis - installed with the PDK
- 6) OpenSTA 2.4.0 (or later) needs to be on the PATH. `rundocker.sh` shows how to use STA included in the OpenLane docker
- 7) python3.9 or later
- 8) verilog parser (`pip3 install verilog-parser`)

- 9) a constraints file (sta\_top/top.sdc)

## Assumptions

- all of the Verilog files are in a single directory
- all of the SPEF files are in a single directory

## Issues

The Verilog parser has two bugs: it does not recognise ‘inout’ and it does not cope with escaped names (starting with a ”). Locally this has been fixed but to use the off-the-shelf parser we preprocess the file to change inout to input and remove the escapes and substitute the [nnn] with *nnn* in the names

## Invoking STA analysis

```
sta_top/toplevel_sta.py <path to verilog> <path to spef> <sdc>
```

eg:

```
sta_top/toplevel_sta.py ./verilog/gl/user_project_wrapper.v  
./spef/user_project_wrapper.spef ./sta_top/top.sdc > sta.log
```

Alternatively you can enter interactive mode after analysis:

```
sta_top/toplevel_sta.py ./verilog/gl/user_project_wrapper.v  
./spef/user_project_wrapper.spef ./sta_top/top.sdc -i
```

## How it works

- preprocesses the main Verilog
- parses the main Verilog to find which modules are used
- creates a new merged Verilog containing all the module netlists and the main netlist
- creates a tcl script in the spef directory to load the relevant spefs for each instanced module in the main verilog
- creates all\_sta.tcl in the verilog/gl directory that does the main STA steps (loading design, loading constraints, running analyses) - this mimics what OpenLane does
- creates a script sta.sh in the verilog/gl directory which sets up the environment and invokes STA to run all\_sta.tcl
- runs sta.sh

## **Invoking Spice simulation:**

### **Important notes**

- This uses at least ngspice-34, tested on ngspice-39, which has features to increase speed and reduce memory image when running with the SkyWater spice models.  
In addition the following needs to be set in `~/spice.rc` and/or `~/.spiceinit`:  
`set ngbehavior=hs`
- Note the spelling of `ngbehavior` (no u). If this is not set then the simulation will run out of memory]
- If the critical path changes then the `spice_path.py` script will need to be adapted to follow suit

### **Instructions**

- go to where the included spice files will be found  
`cd $PDK_ROOT/sky130A/libs.tech/ngspice`
- run the script to get the critical path spice circuit - 250 stages of the clock  
`/sta_top/spice_path.py path.spice`
- invoke spice (takes about 2 minutes)  
`ngspice path.spice`
- run the sim (takes about 2 minutes)  
`tran 1n 70n`
- show the results  
`plot i0 i250`

This chart shows the input clock and how it changes after 250 blocks. It was simulated for 200n to capture a clean pair of clock pulses.

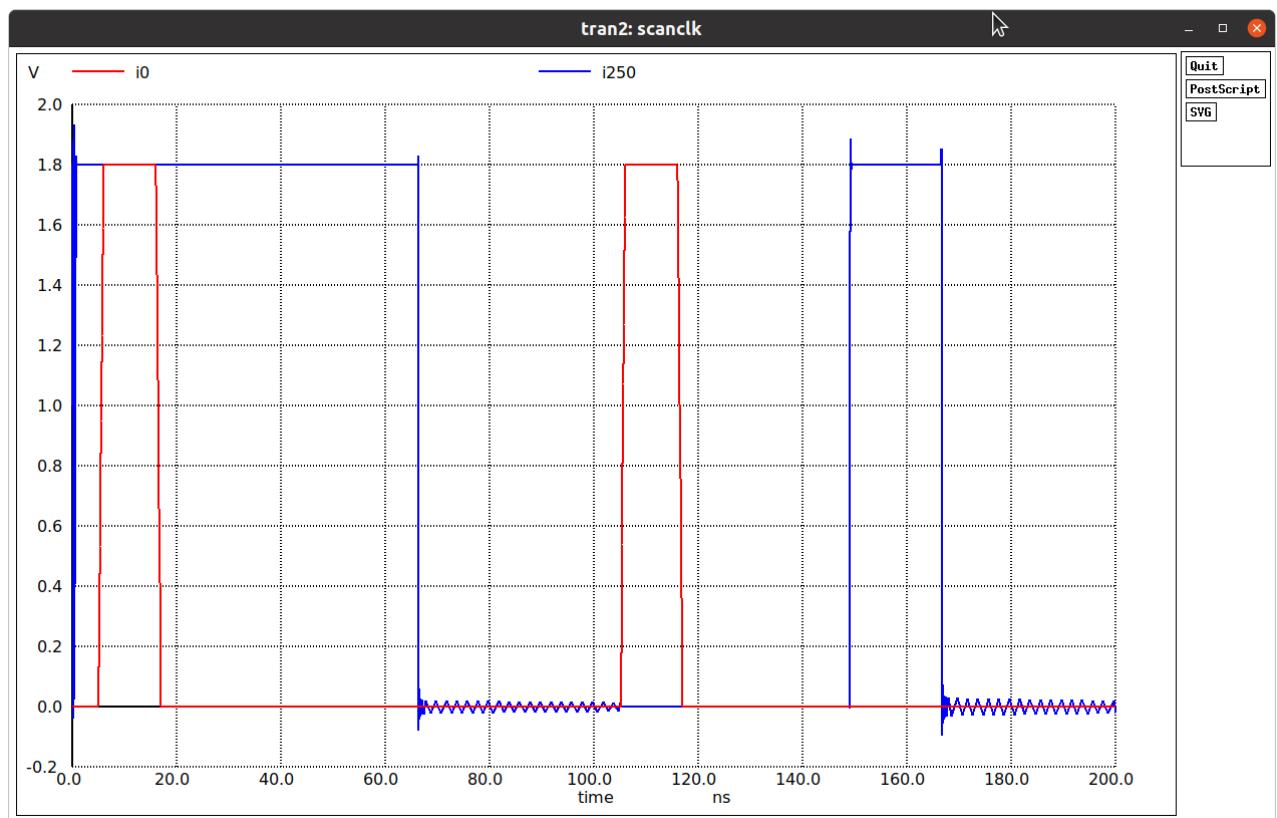


Figure 58: clock\_spice.png

## 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
- Jeremy Birch for help with STA
- Aisler for sponsoring PCB development