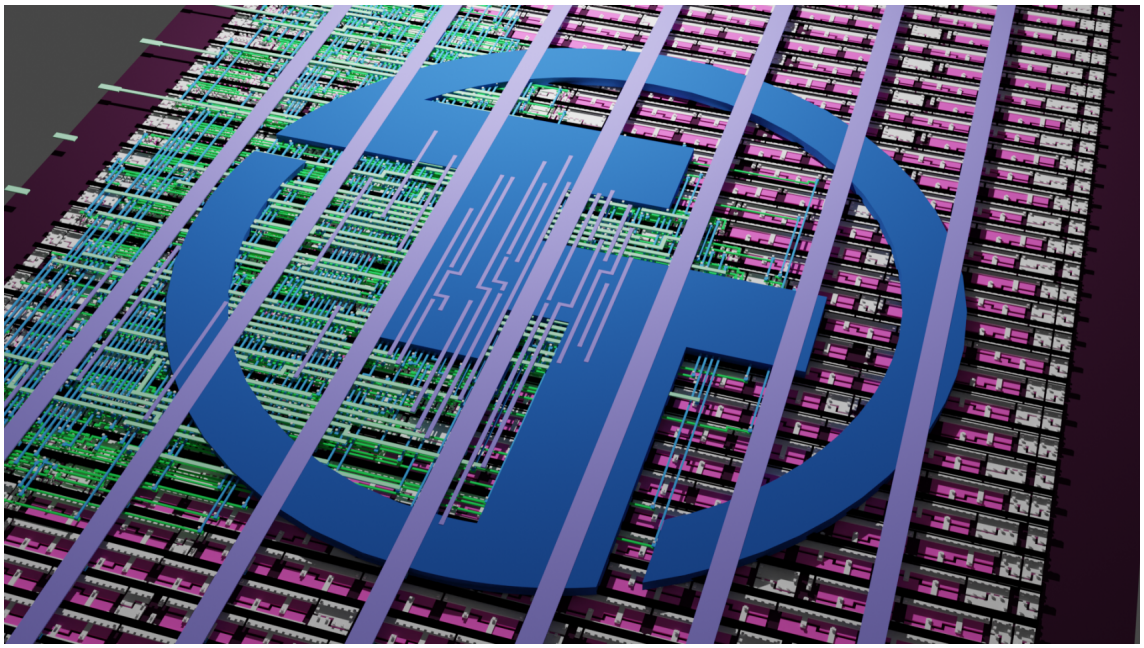


Tiny Tapeout 03 Datasheet

Project Repository
https://github.com/TinyTapeout/tinytapeout-03

April 21, 2023

Contents

Render of whole chip **4**

Projects **5**
0 : Test Inverter Project 5
1 : ChipTune 6
The ChipTune Project 7
2 : 7 Segment Life 12
3 : Another Piece of Pi 14
4 : Wormy 16
5 : Knight Rider Sensor Lights 20
6 : Single digit latch 22

7 : 4x4 Memory . . . 23
8 : KS-Signal . . . 24
9 : Hovalaag CPU . . . 26
10 : SKINNY SBOX . . . 27
11 : Stateful Lock . . . 28
12 : Ascon's 5-bit S-box . . . 29
13 : 8bit configurable galois lfsr . . . 30
14 : Sbox SKINNY 8 Bit . . . 31
15 : BinaryDoorLock . . . 32
16 : bad apple . . . 33
17 : TinyFPGA attempt for TinyTapeout3 . . . 34
18 : 4bit Adder . . . 35
19 : 12-bit PDP8 . . . 36
20 : CTF - Catch the fish . . . 38
21 : Dot operation calculator . . . 39
22 : RiscV Scan Chain based CPU – block 1 – clocking . . . 40
23 : RiscV Scan Chain based CPU – block 2 – instructions . . . 41
24 : RiscV Scan Chain based CPU – block 3 – registers . . . 42
25 : RiscV Scan Chain based CPU – block 4 – ALU . . . 43
26 : 31b-PrimeDetector . . . 44
27 : XOR Stream Cipher . . . 46
28 : LED Panel Driver . . . 49
29 : 6-bit FIFO . . . 51
30 : ezm_cpu . . . 53
31 : tiny logic analyzer . . . 54
32 : 4-bit ALU . . . 56
33 : Pulse-Density Modulators . . . 58
34 : CRC Decelerator . . . 61
35 : 7 segment seconds . . . 65
36 : Binary to DEC and HEX . . . 66
37 : Simon Says . . . 67
38 : Shift Register Ram . . . 70
39 : tinysat . . . 71
41 : POV display . . . 72
42 : Toy CPU . . . 73
43 : Base-10 grey counter counts from zero to a trillion . . . 75
44 : ttFIR filter . . . 79
45 : QTCore-A1 . . . 80
46 : MicroTapeout . . . 91
47 : nipple multiplier . . . 92
48 : Synthesizable Digital Temperature Sensor . . . 94
49 : 4-bits sequential ALU . . . 96

50 : Brightness control of LED with PWM . . . 97

Technical info **98**
Scan chain . . . 98
Clocking . . . 99
Clock divider . . . 100
Wait states . . . 100
Pinout . . . 100
Instructions to build GDS . . . 101
Changing macro block size . . . 102

Verification **103**
Setup . . . 103
Simulations . . . 103
Top level tests setup . . . 104
Formal Verification . . . 105
Timing constraints . . . 105
Physical tests . . . 106

Toplevel STA and Spice analysis for TinyTapeout-02 and 03. **107**
Introduction . . . 107
GitHub action . . . 107
Prerequisites . . . 107
Assumptions . . . 108
Issues . . . 108
Invoking STA analysis . . . 108
How it works . . . 108
Invoking Spice simulation: . . . 109

Sponsored by **111**

Team **111**

Render of whole chip

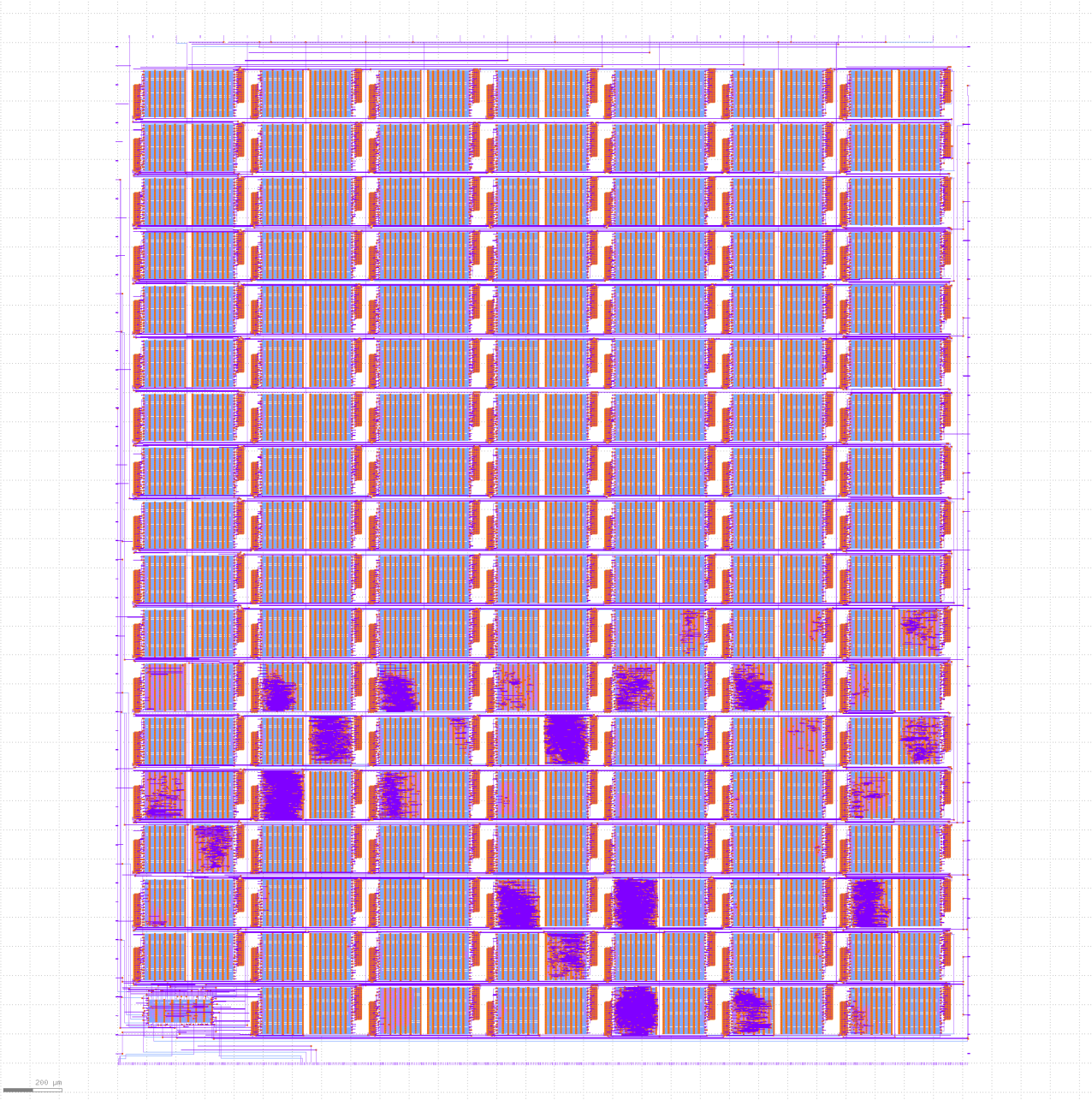


Figure 1: Full GDS

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

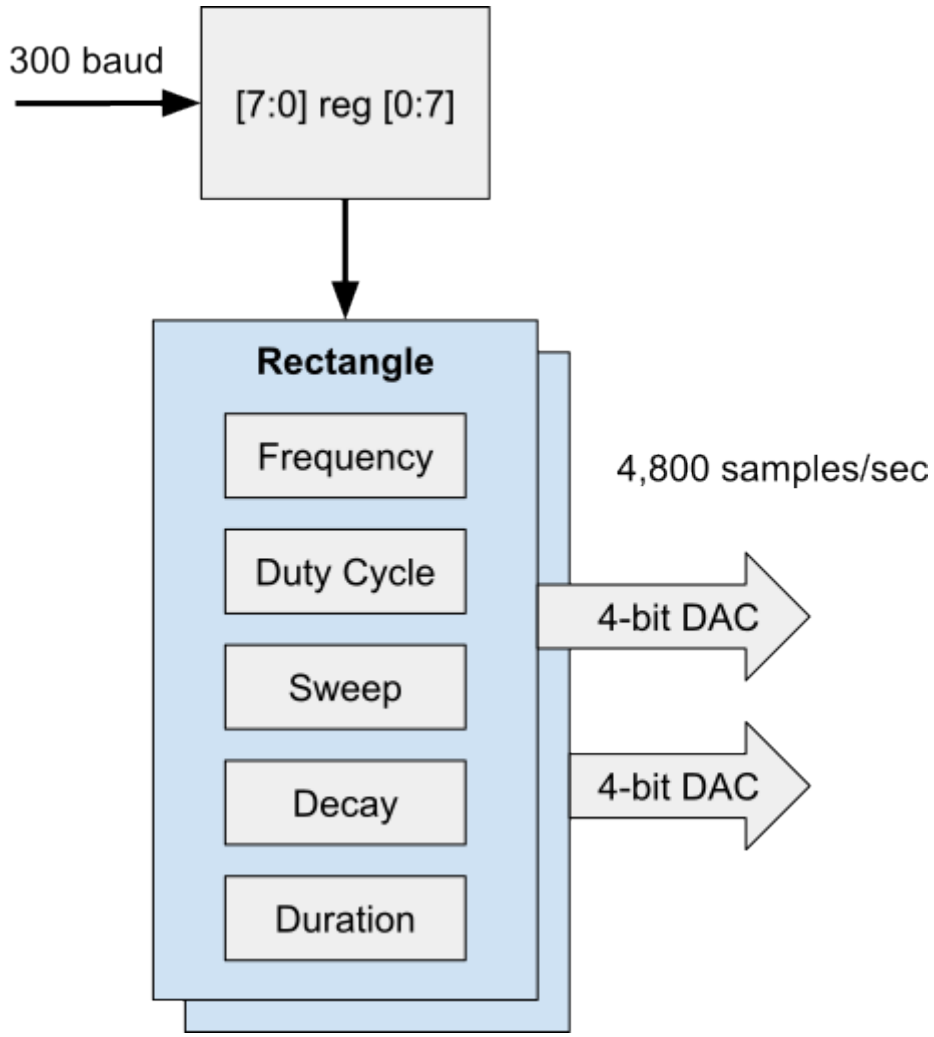


Figure 2: picture

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

Signal	Name	Dir	WCSP	QFN	PCB
mprj_io[15]	ACTIVE_SELECT[3]	In	F1	51	J2.13
mprj_io[16]	ACTIVE_SELECT[4]	In	E1	53	J2.12
mprj_io[17]	ACTIVE_SELECT[5]	In	D2	54	J2.11
mprj_io[18]	ACTIVE_SELECT[6]	In	D1	55	J2.10
mprj_io[19]	ACTIVE_SELECT[7]	In	C10	57	J2.9
mprj_io[20]	ACTIVE_SELECT[8]	In	C2	58	J2.8
mprj_io[21]	IO_IN[0] / EXT_SCAN_CLK_OUT	In	B1	59	J2.7
mprj_io[22]	IO_IN[1] / EXT_SCAN_DATA_OUT	In	B2	60	J2.6
mprj_io[23]	IO_IN[2] / EXT_SCAN_SELECT	In	A1	61	J2.5
mprj_io[24]	IO_IN[3] / EXT_SCAN_LATCH_EN	In	C3	62	J2.4
mprj_io[25]	IO_IN[4]	In	A3	2	J2.3
mprj_io[26]	IO_IN[5]	In	B4	3	J2.2
mprj_io[27]	IO_IN[6]	In	A4	4	J2.1
mprj_io[28]	IO_IN[7]	In	B5	5	J1.14
mprj_io[29]	IO_OUT[0] / EXT_SCAN_DATA_IN	Out	A5	6	J1.13
mprj_io[30]	IO_OUT[1] / EXT_SCAN_DATA_IN	Out	B6	7	J1.12
mprj_io[31]	IO_OUT[2]	Out	A6	8	J1.11
mprj_io[32]	IO_OUT[3]	Out	A7	11	J1.10
mprj_io[33]	IO_OUT[4]	Out	C8	12	J1.9
mprj_io[34]	IO_OUT[5]	Out	B8	13	J1.4
mprj_io[35]	IO_OUT[6]	Out	A8	14	J1.3
mprj_io[36]	IO_OUT[7]	Out	B9	15	J1.2
mprj_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 serpentines 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	io_in[0] (CLOCK)	io_out[0] (DAC_0)
1	io_in[1] (IN_1)	io_out[1] (DAC_1)
2	io_in[2] (IN_2)	io_out[2] (DAC_2)
3	io_in[3] (MODE)	io_out[3] (DAC_3)
4	io_in[4] (BAUD_CLK)	io_out[4] (DAC_4)
5	io_in[5] (TCK)	io_out[5] (REF_CLK)
6	io_in[6] (TMS)	io_out[6] (RTCK)
7	io_in[7] (TDI)	io_out[7] (TDO)

2 : 7 Segment Life

- Author: icegoat9
- Description: Simple 7-segment cellular automaton
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 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

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

- Seven flip-flops hold the cellular automaton's internal state, which is also displayed in the seven-segment display.
- Combinatorial logic generates the next state for each segment based on its neighbors, according to the ruleset...
  - Living segments with exactly one living neighbor (another segment that touches it end to end) survive, and all others die.
  - Dead segments with exactly two living neighbors come to life.
- When either the system clock or a user toggling the clock input go high, this new state is latched into the automaton's state.
- 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 and a few 'exercises for the reader', see the github README doc link. But 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...

- Set all dip switches off and the clock slide switch to the 'manual' clock side.
- Power on the system. An arbitrary state may appear on the 7-segment display.
- Set dip switch 4 on ('run mode').
- 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)...

- 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.
- Set dip switches 5 and 7 on to add a reasonable additional clock divider (see docs for more details on a higher or lower divider).
- Set dip switch 4 on.
- Switch the clock slide switch to the 'system clock' side. The display should advance at roughly 1.5Hz if I've done math correctly.

If you want to load an initial state...

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

IO

#	Input	Output
0	clock	7segmentA
1	load0	7segmentB
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
- 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 tripled 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

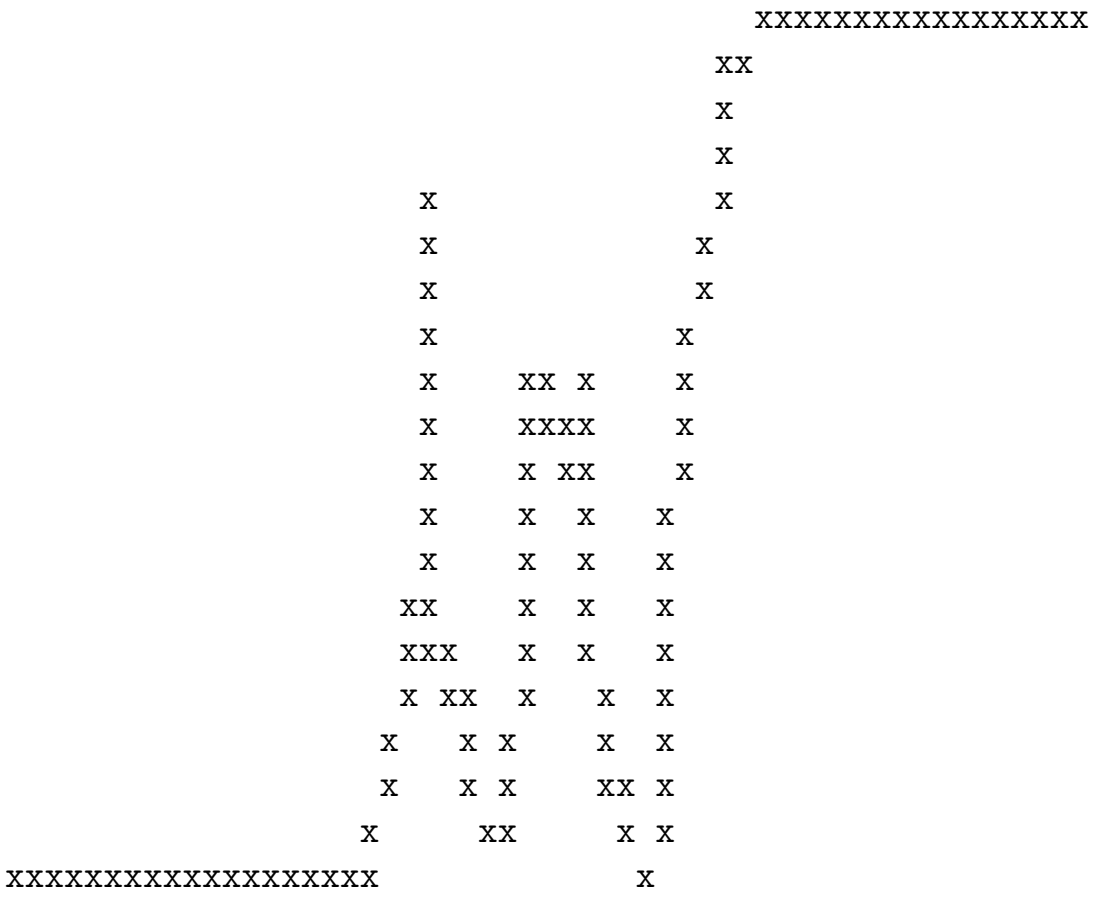
in the next cycle, so be careful if you plan to have Wormy chase its tail! SPLAT!

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|   | O |   |

----------------
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

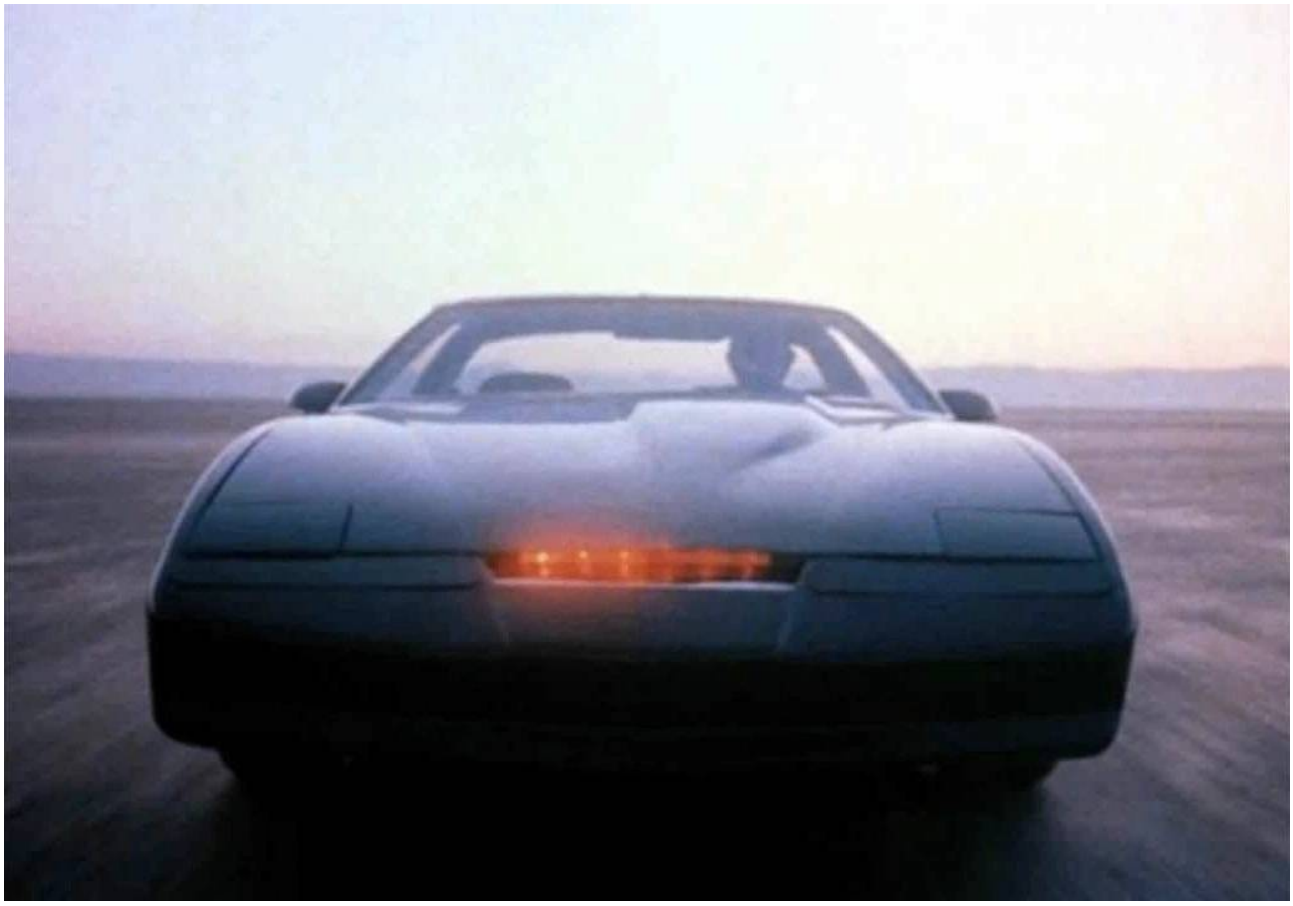


Figure 3: picture

- 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: Conect 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. Conect 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 ordern 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

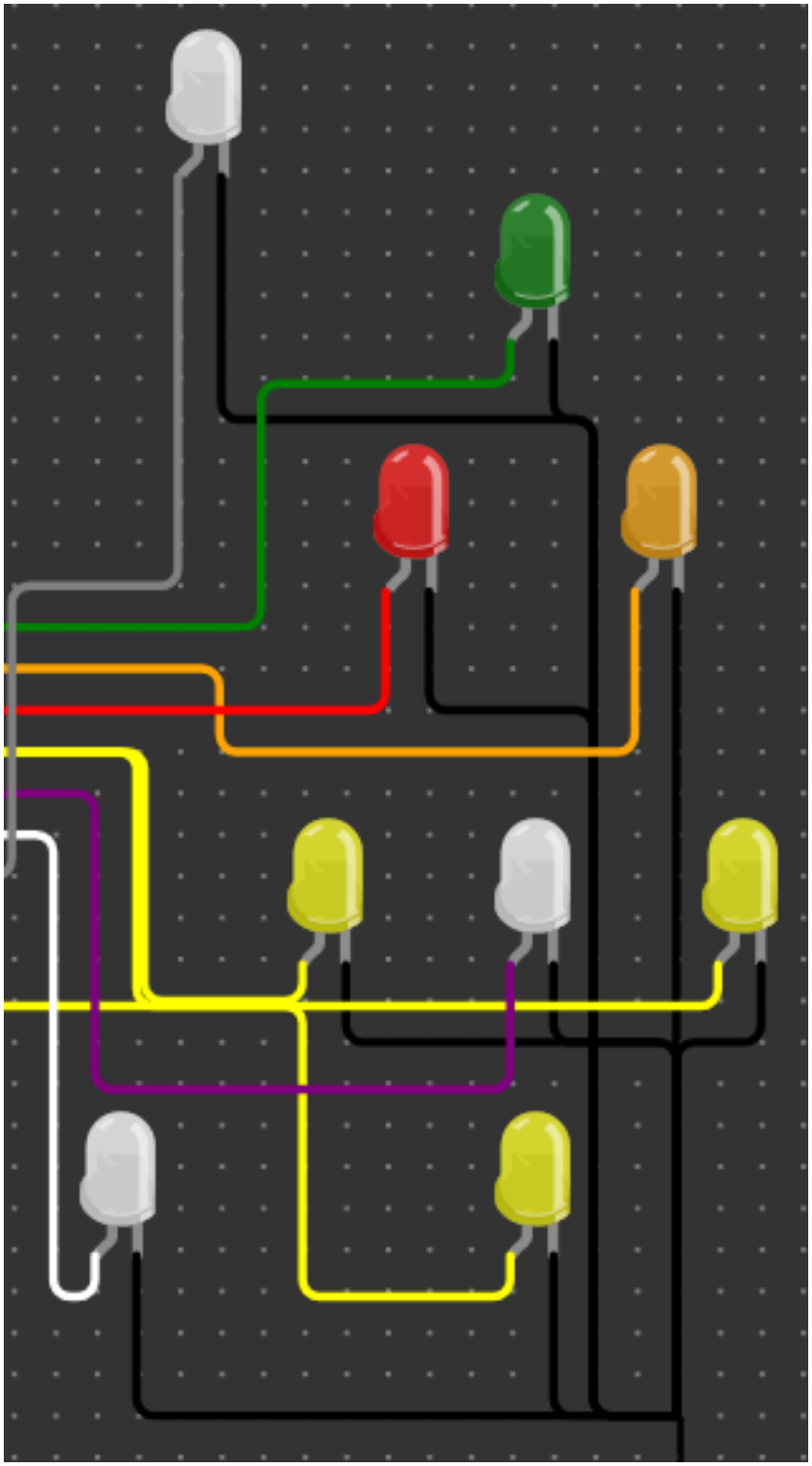


Figure 4: picture

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 lfsr

- Author: Alexander Schönborn
- Description: A 8bit configurable galois lfsr.
- 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 lfsr 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 cyle 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

- 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

Explain how your project works

How to test

Explain how to test your project

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 numberbs 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     IO 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 reminder.

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/reminder
4	operand1_3	product/reminder
5	operand2_1	product/reminder
6	operand2_2	none
7	operand2_3	none

22 : 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

23 : 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

24 : 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

25 : 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

26 : 31b-PrimeDetector

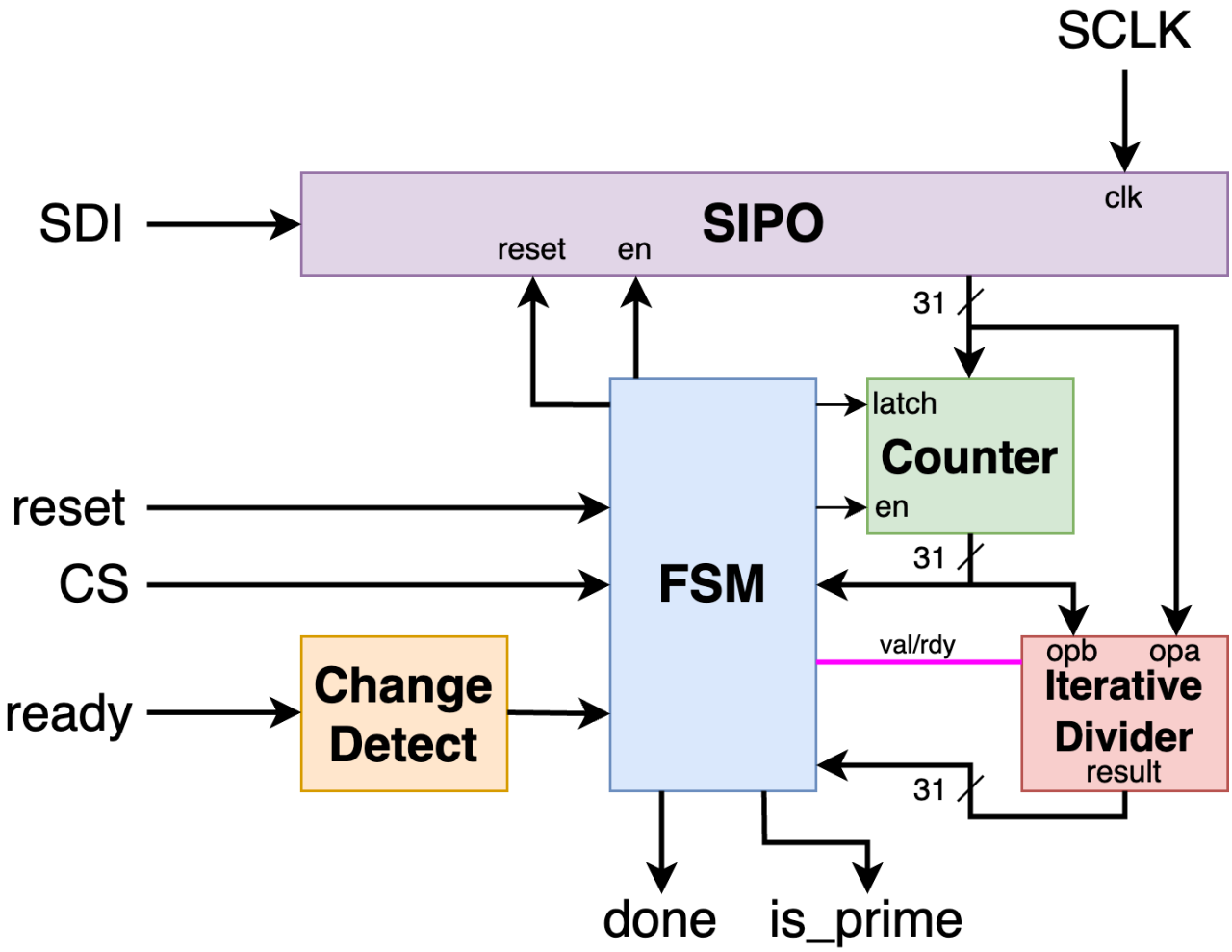


Figure 5: picture

- 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

27 : XOR Stream Cipher

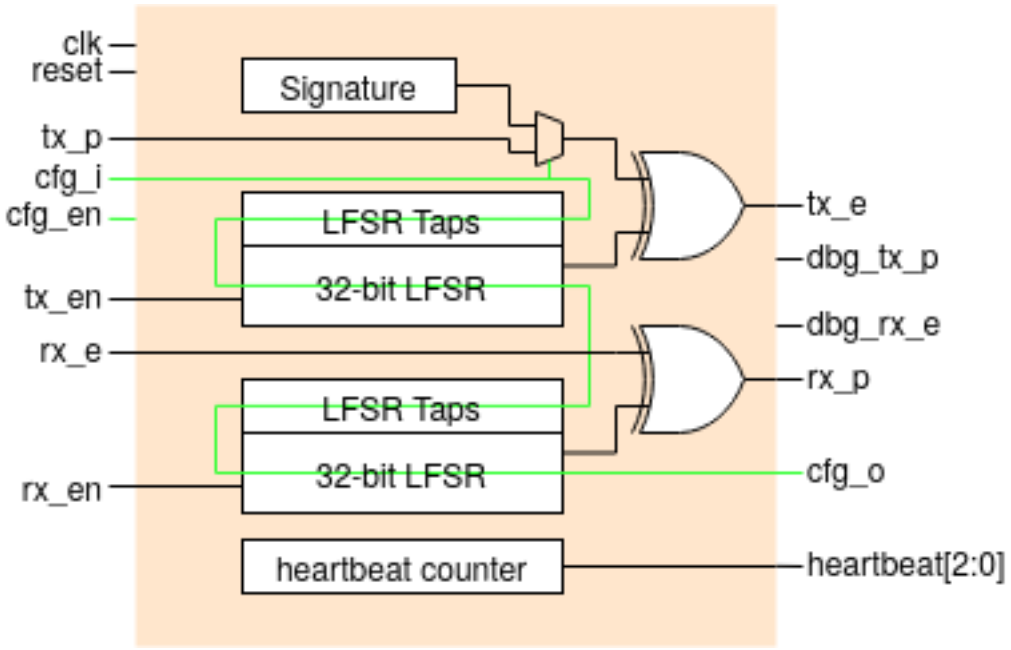


Figure 6: picture

- 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 receiption bitstream can be encrypted and decrypted simultaneously. When the system is reset, a default confgiruation is applied that will encrypt the bitsream on tx_p and decrypt the bistream 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 receiption)
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 receiption)	heartbeat[8] (bit from heartbeat counter)
7	rx_en (Receive channel enable)	heartbeat[9] (bit from heartbeat counter)

28 : LED Panel Driver

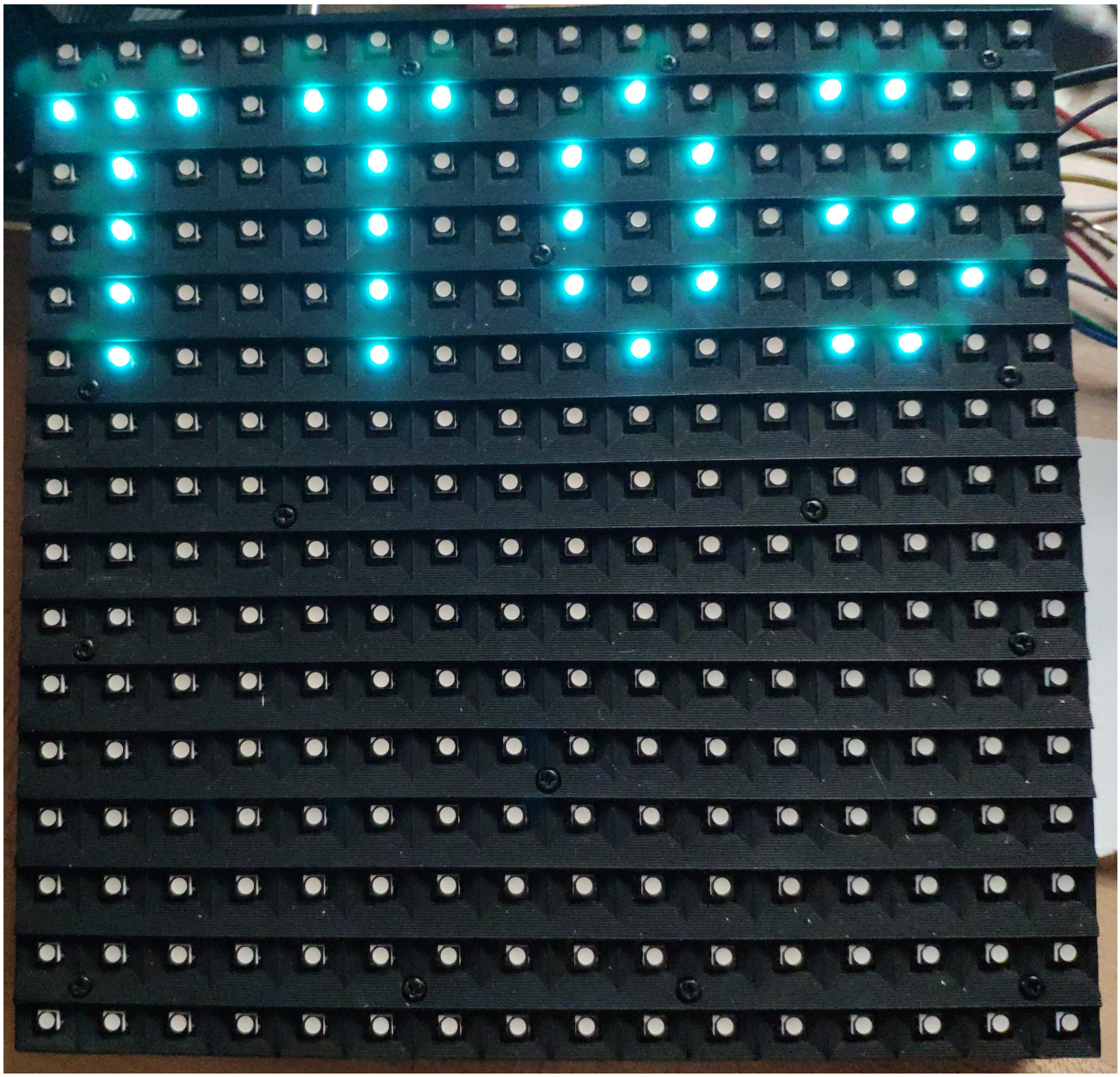


Figure 7: picture

- 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 : 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

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

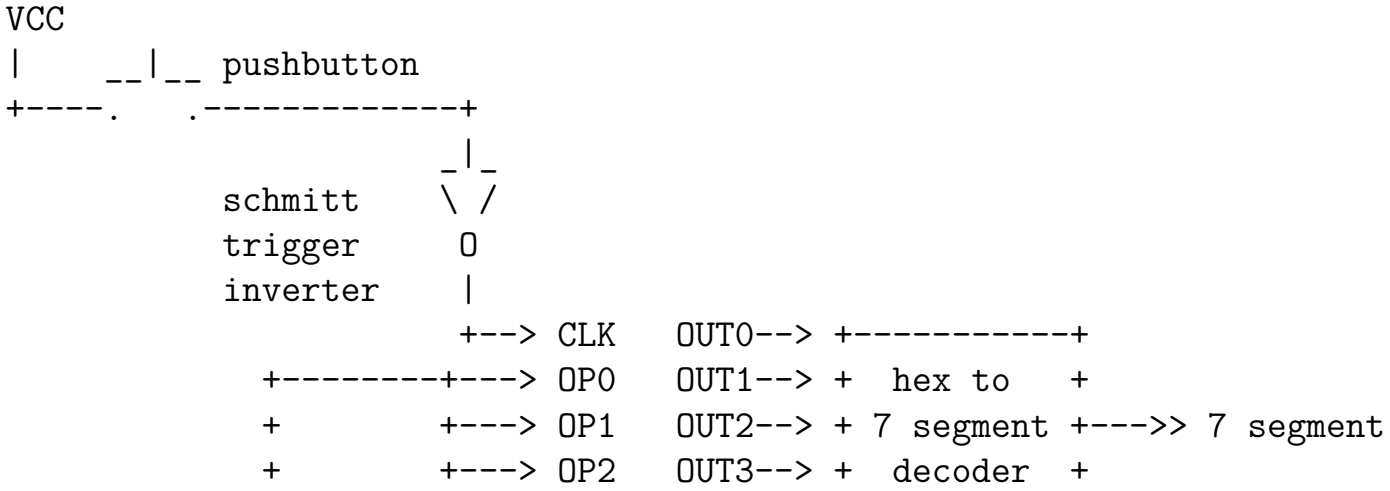
accumulator [4 bit] = accumulator [4 bit] (operation) operand [4 bit]

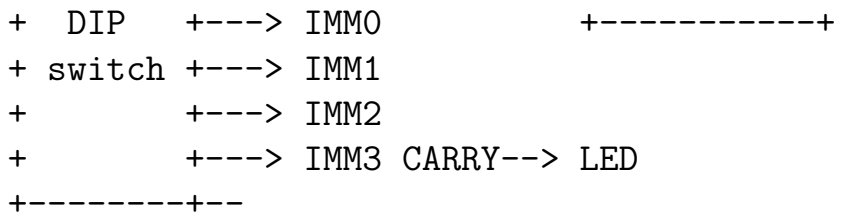
Supported operations: lda imm :: imm -> accumulator neg imm :: 0x0F - imm -> accumulator shr :: accumulator / 2 -> accumulator sub imm :: accumulator - imm -> accumulator and imm :: accumulator & imm -> accumulator xor imm :: accumulator ^ imm -> accumulator or imm :: accumulator | imm -> accumulator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





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

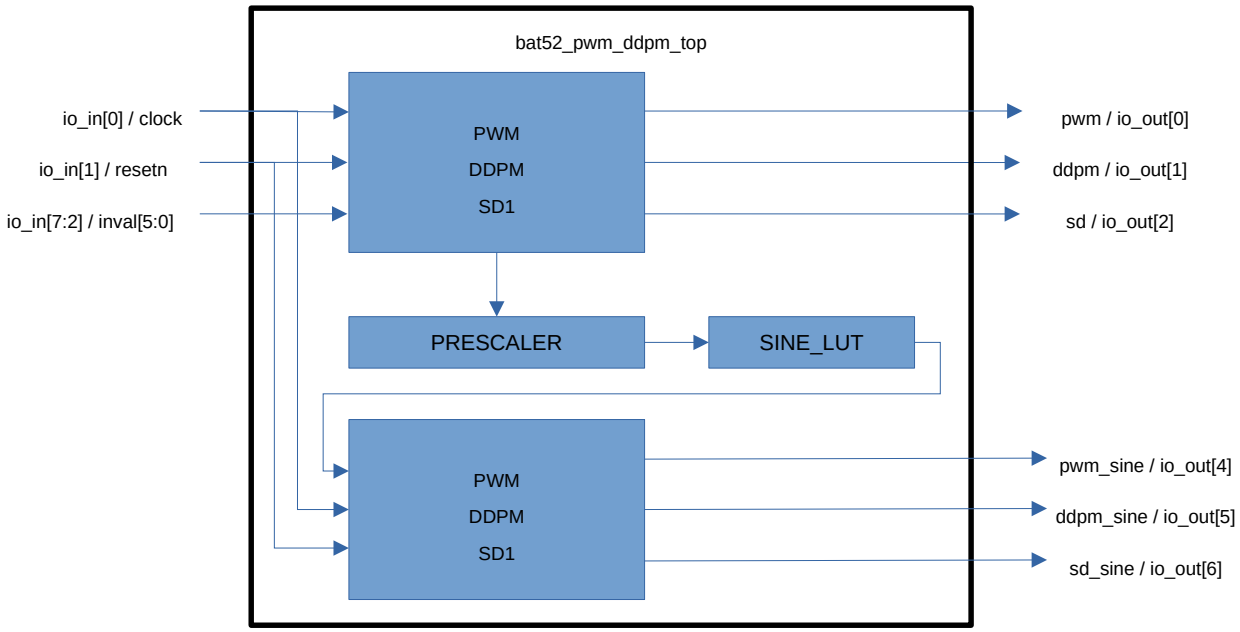


Figure 8: picture

- 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 Moulation (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 Moduldation (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 performaces, 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 section: 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 (fsin = fclock / 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

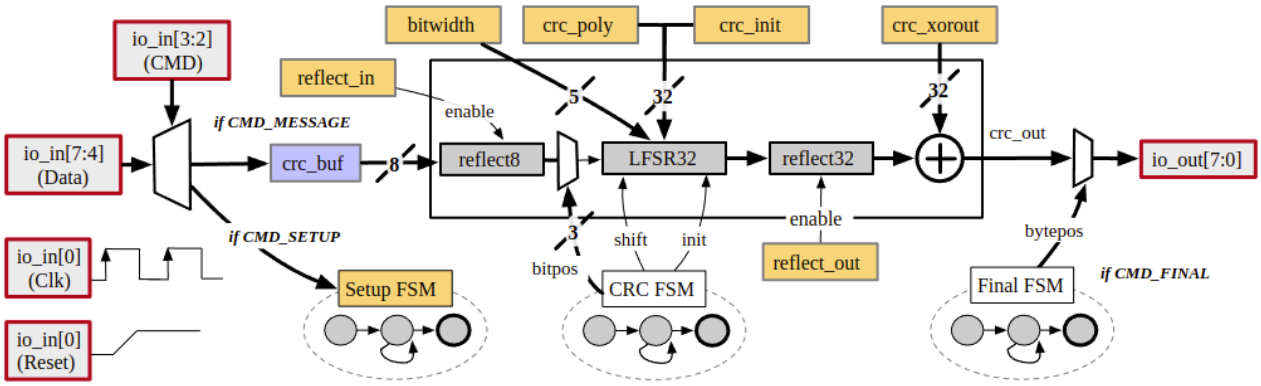


Figure 9: picture

- Author: Grant Hernandez (@grant-h)
- Description: A reconfigurable CRC engine
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware:

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 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 `f35008ffffffff`. 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	`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]`

35 : 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

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 forbiden 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

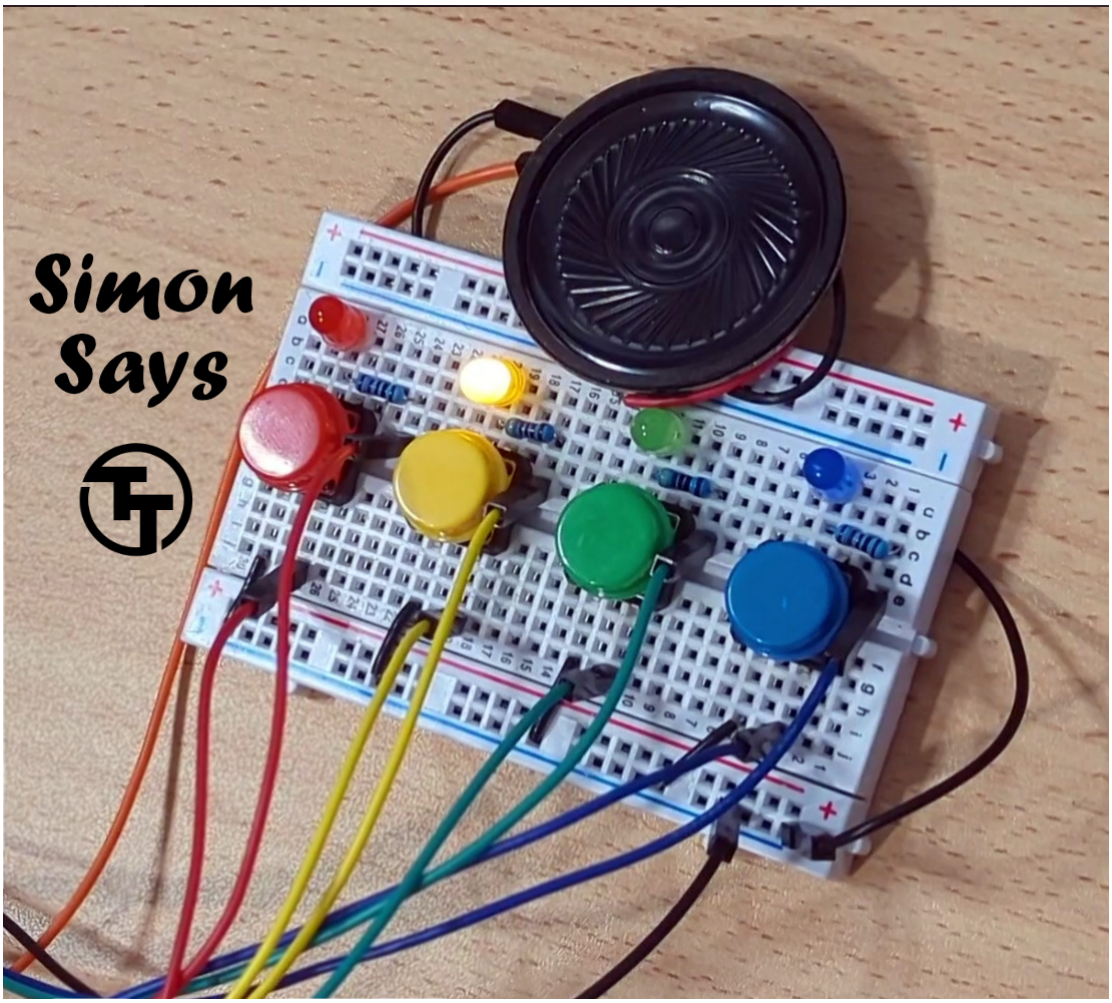


Figure 10: picture

- 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 assignements 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

41 : 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

42 : 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	data_out = reg[src_a]
001	ADD	Add	reg[dest] = reg[src_a] + reg[src_b]
010	ADD_I	Add (immediate)	reg[dest] = reg[src_a] + imm
011	AND	And	reg[dest] = reg[src_a] & reg[src_b]
100	AND_I	And (immediate)	reg[dest] = reg[src_a] & imm
101	OR	Or	reg[dest] = reg[src_a] | reg[src_b]
110	OR_I	Or (immediate)	reg[dest] = reg[src_a] | imm
111	STRE	Store	reg[dest] = 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]

43 : 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 exampe of how this grey counter works, this is the progression of grey bits when counting from 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]

44 : ttFIR filter

- Author: Georg Boecherer
- Description: 6 tap Finite Impulse Response (FIR) filter with 6 bit input signal, 6 bit filter coefficients, and 8 bit output signal.
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware:

How it works

ttFIR implements a digital 6 tap Finite Impulse Response (FIR) filter. Inputs, coefficients, and outputs are 6bit, 6bit, and 8bit values, respectively, in 2's complement format.

How to test

- reset high: shift registers for coefficient and output are set to zero.
- reset low:
  - 6 clock cycles: 6bit coefficients in 2's complement format are loaded into registers.
  - 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.

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.

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

Control and Branching Instructions

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

    - PC Behavior: PC <- 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 <- PC + 1, PC <- 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 <- 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 <- 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 <- 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 <- 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 <- ACC << 1
- SHR: Shift Accumulator right
    - Opcode (8 bits): 11110111
    - Register Effects: ACC <- ACC >> 1
- SHL4: Shift Accumulator left by 4 bits
    - Opcode (8 bits): 11111000
    - Register Effects: ACC <- ACC << 4
- ROL: Rotate Accumulator left
    - Opcode (8 bits): 11111001
    - Register Effects: ACC <- (ACC << 1) OR (ACC >> 7)
- ROR: Rotate Accumulator right
    - Opcode (8 bits): 11111010

    - Register Effects: ACC <- (ACC >> 1) OR (ACC << 7)
- LDAR: Load Accumulator via indirect memory access (ACC as ptr)
    - Opcode (8 bits): 11111011
    - Register Effects: ACC <- M[ACC]
- DEC: Decrement Accumulator
    - Opcode (8 bits): 11111100
    - Register Effects: ACC <- ACC - 1
- CLR: Clear (Zero) Accumulator
    - Opcode (8 bits): 11111101
    - Register Effects: ACC <- 0
- INV: Invert (NOT) Accumulator
    - Opcode (8 bits): 11111110
    - Register Effects: ACC <- ~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 <- M[PC]
2. PC <- PC + 1

EXECUTE cycle

For **Immediate Data Manipulation Instructions**:

- ADDI: ACC <- ACC + IMM

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

- LDA: ACC <- M[Address]
- STA: M[Address] <- ACC
- ADD: ACC <- ACC + M[Address]
- SUB: ACC <- ACC - M[Address]
- AND: ACC <- ACC & M[Address]
- OR: ACC <- ACC | M[Address]
- XOR: ACC <- ACC ^ M[Address]

For **Control and Branching Instructions**:

- JMP: PC <- ACC
- JSR: ACC <- PC, PC <- ACC
- BEQ_FWD: If ACC == 0, then PC <- PC + 2
- BEQ_BWD: If ACC == 0, then PC <- PC - 3
- BNE_FWD: If ACC != 0, then PC <- PC + 2
- BNE_BWD: If ACC != 0, then PC <- PC - 3
- HLT: (No operation, processor halted)

For **Data Manipulation Instructions**:

- SHL: ACC <- ACC << 1
- SHR: ACC <- ACC >> 1
- SHL4: ACC <- ACC << 4
- ROL: ACC <- (ACC << 1) OR (ACC >> 7)

- ROR: ACC <- (ACC >> 1) OR (ACC << 7)
- LDAR: ACC <- M[ACC]
- DEC: ACC <- ACC - 1
- CLR: ACC <- 0
- INV: ACC <- ~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.

46 : MicroTapeout

- Author: htfab
- Description: 427 single-cell 'projects' with a mux to select between them. Sounds familiar from somewhere.
- GitHub repository
- HDL project
- Extra docs
- Clock: 1000 Hz
- External hardware:

How it works

TODO

How to test

TODO

IO

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

47 : nipple 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

48 : Synthesizable Digital Temperature Sensor

- 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)

49 : 4-bits sequential ALU

- Author: Diego Satizabal
- Description: A 4-bits sequential ALU that takes operands and opcode seqentially 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, the 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

50 : 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

Technical info

Scan chain

All 250 designs are joined together in a long chain similiar to JTAG. We provide the inputs and outputs of that chain (see pinout below) externally, to the Caravel logic analyser, and to an internal scan chain driver.

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

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

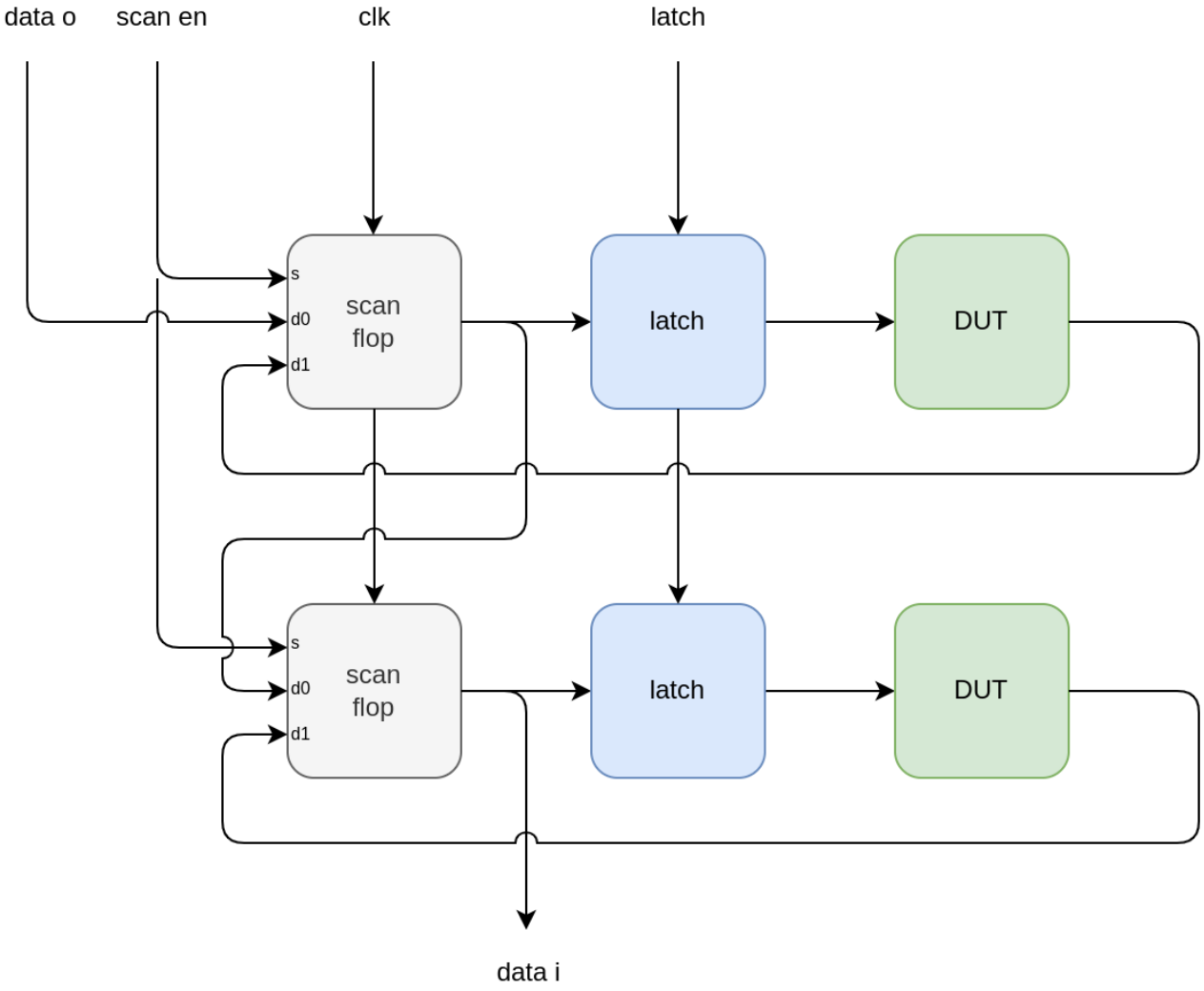


Figure 11: block diagram

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

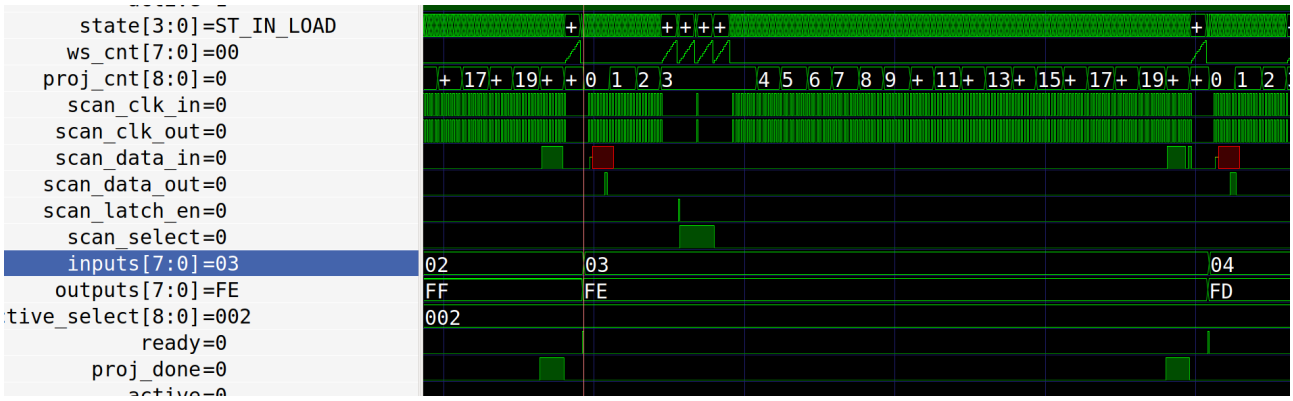


Figure 12: update cycle

Notes on understanding the trace

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

Clocking

Assuming:

- 100MHz input clock
- 8 ins & 8 outs
- 2 clock cycles to push one bit through the scan chain (scan clock is half input clock rate)
- 250 designs
- scan controller can do a read/write cycle in one refresh

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

Clock divider

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

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

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

See the test_clock_div test in the scan chain verification.

Wait states

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

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

See the test_wait_state test in the scan chain verification.

Pinout

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

```
9:8     driver_sel          which scan chain driver: 00 = external, 01 =

21      ext_scan_clk_out    for external driver, clk input
22      ext_scan_data_out   data input
23      ext_scan_select     scan select
24      ext_scan_latch_en   latch
29      ext_scan_clk_in     clk output from end of chain
30      ext_scan_data_in    data output from end of chain
```

Instructions to build GDS

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

GH_USERNAME should be set to your GitHub username.

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

To run locally, make a file like this:

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

And then source it before running the tool.

Fetch all the projects

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

```
./configure.py --clone-all --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 easist way to install these are to download and install the oss-cad-suite.

Simulations

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

Scan controller

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

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

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

- Edit openlane/scan_controller/config.tcl and change NUM_DESIGNS=250 to NUM_DESIGNS=20.
- Then from the top level directory:

  make scan_controller make user_project_wrapper
- Then run the GL test

  cd verilog/dv/scan_controller make test_scan_controller_gl

single

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

```
make test_single
```

custom wait state

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

```
make test_wait_state
```

clock divider

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

```
make test_clock_div
```

Top level tests setup

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

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

Top level test: internal control

Uses the scan controller, instantiated inside the whole chip.

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

Top level test: external control

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

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

Top level test: logic analyser control

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

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

Formal Verification

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

Scan chain

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

Scan controller MUX

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

Timing constraints

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

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

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

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

Physical tests

- LVS
- DRC
- CVC

LVS

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

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

DRC

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

CVC

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

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

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.

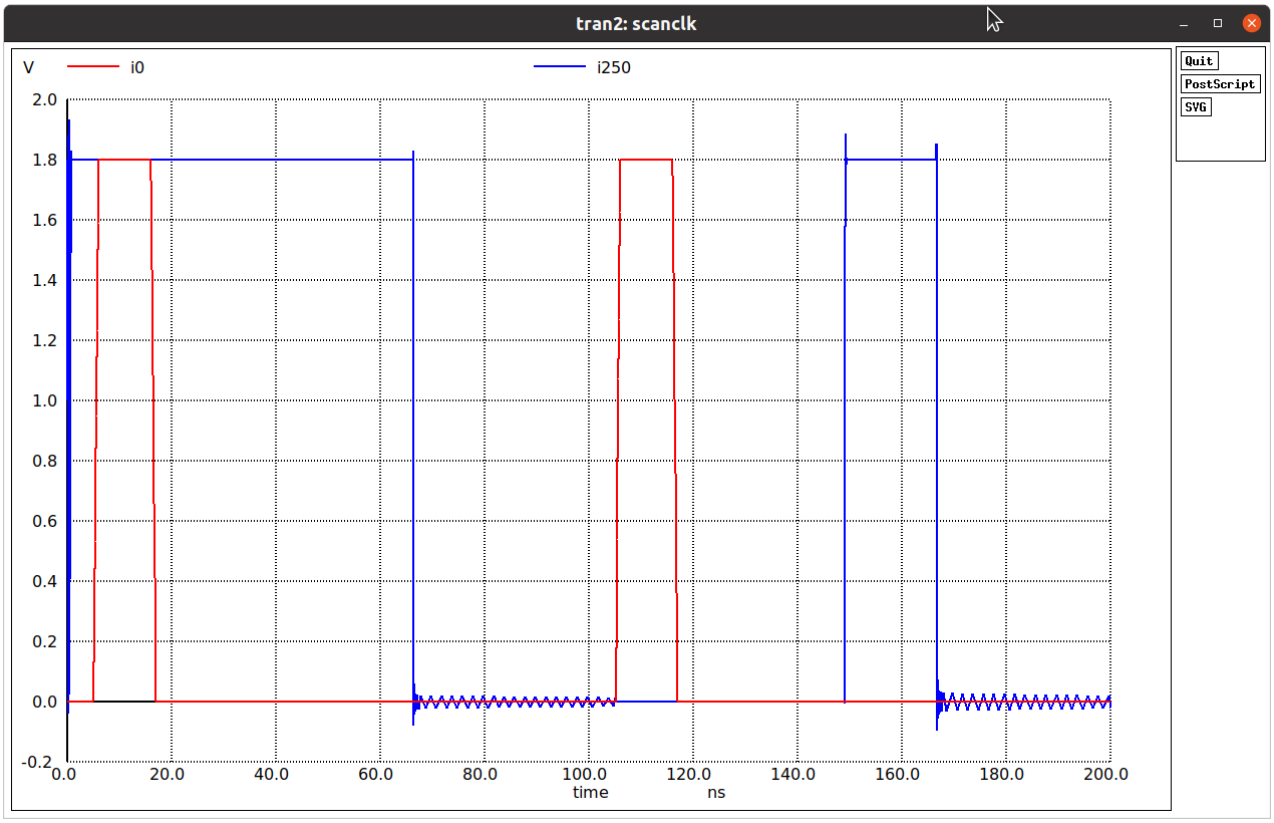


Figure 13: clock_spice.png

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