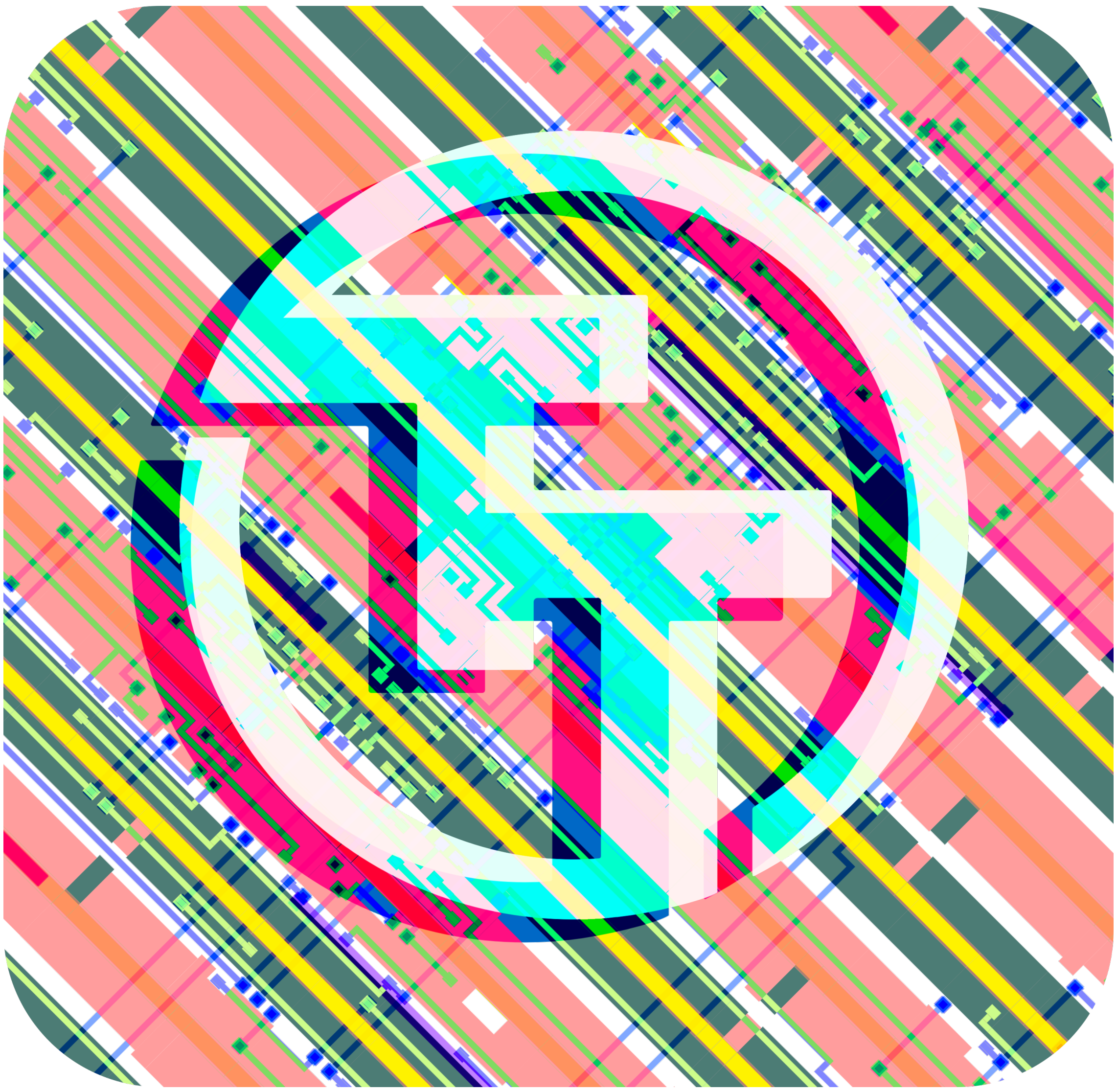


Tiny Tapeout 7 Datasheet

Project Repository

https://github.com/TinyTapeout/tinytapeout-07

May 12, 2025

Contents

Chip map

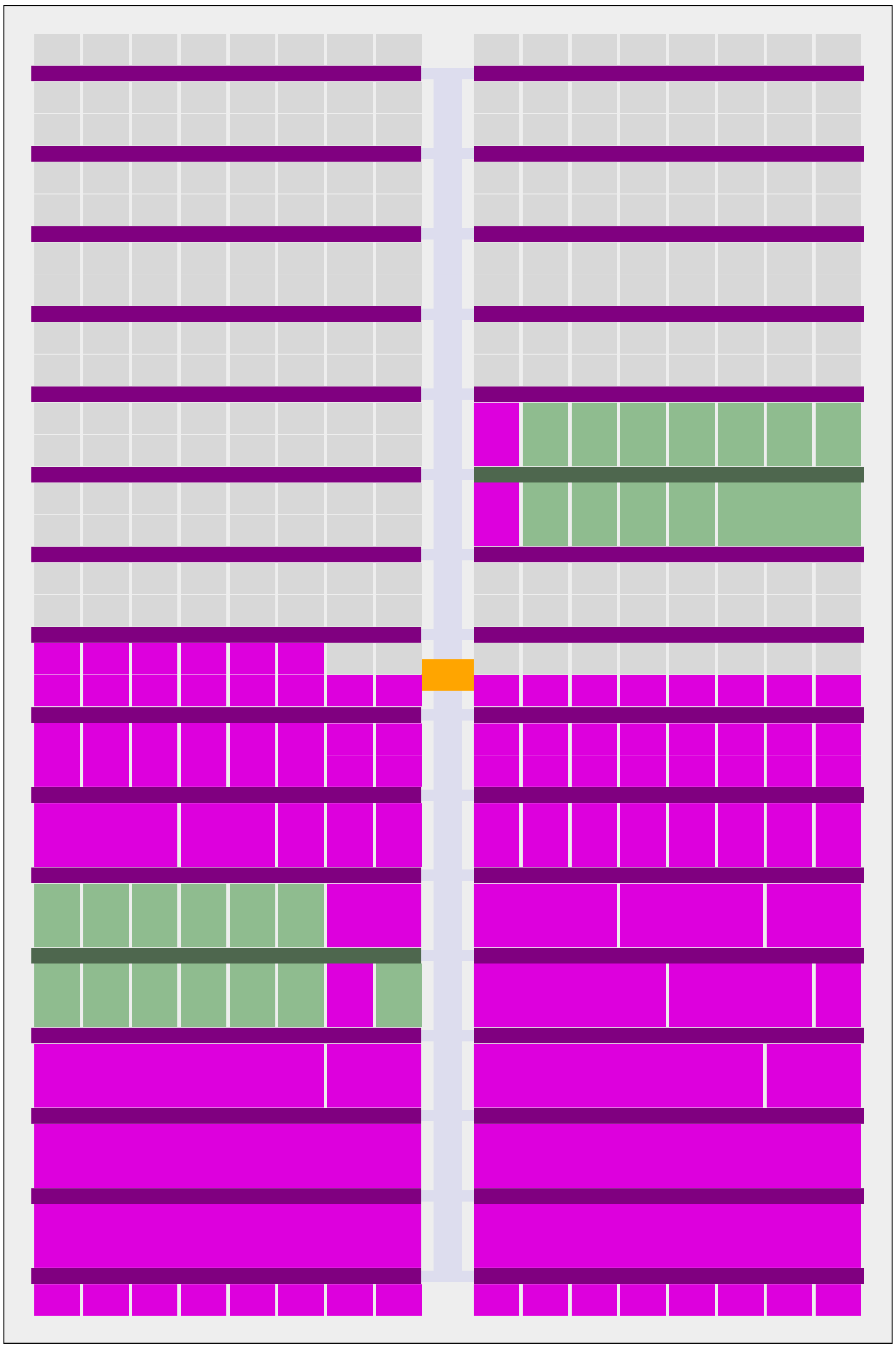


Figure 1: Full chip map

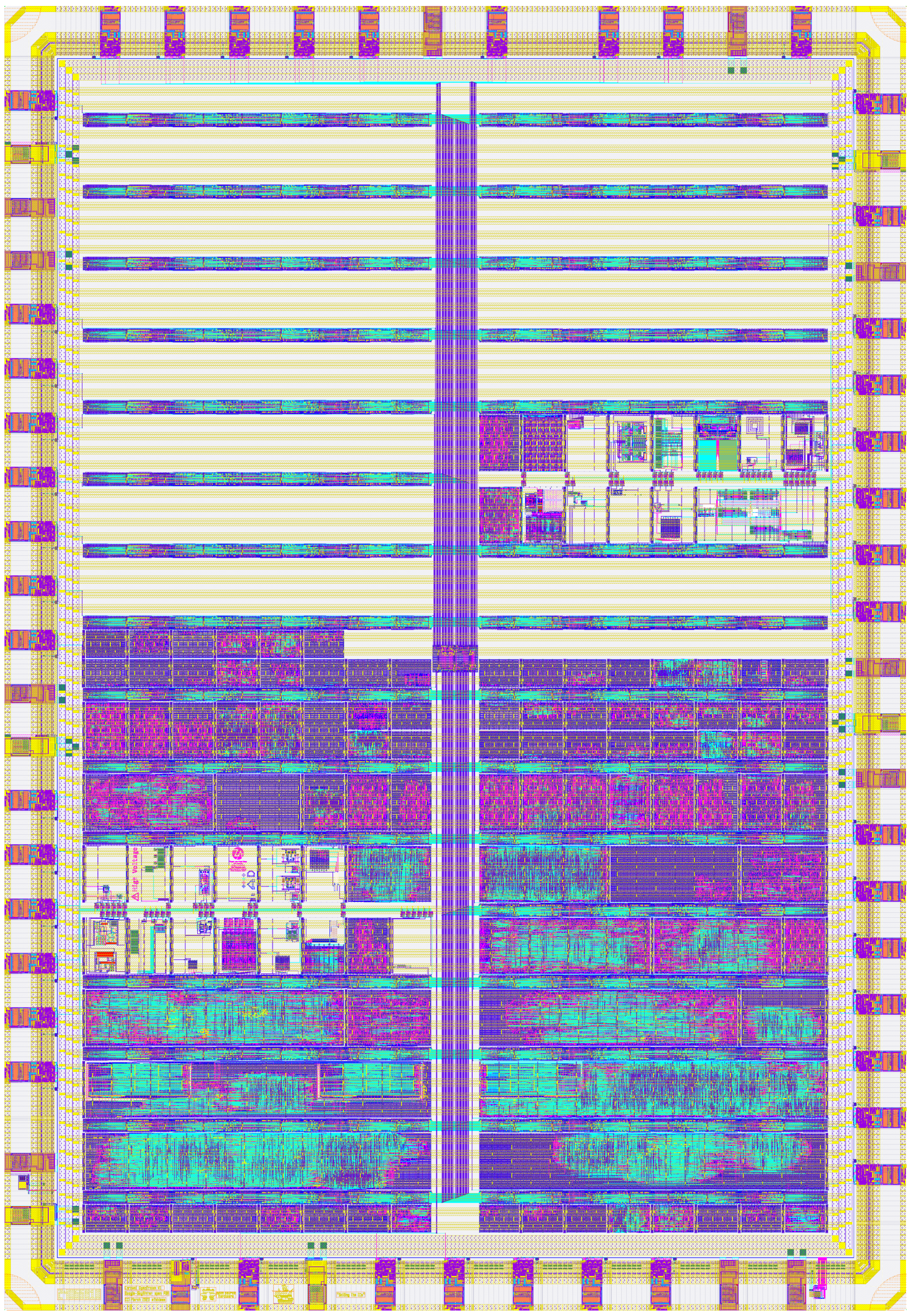


Figure 2: GDS render

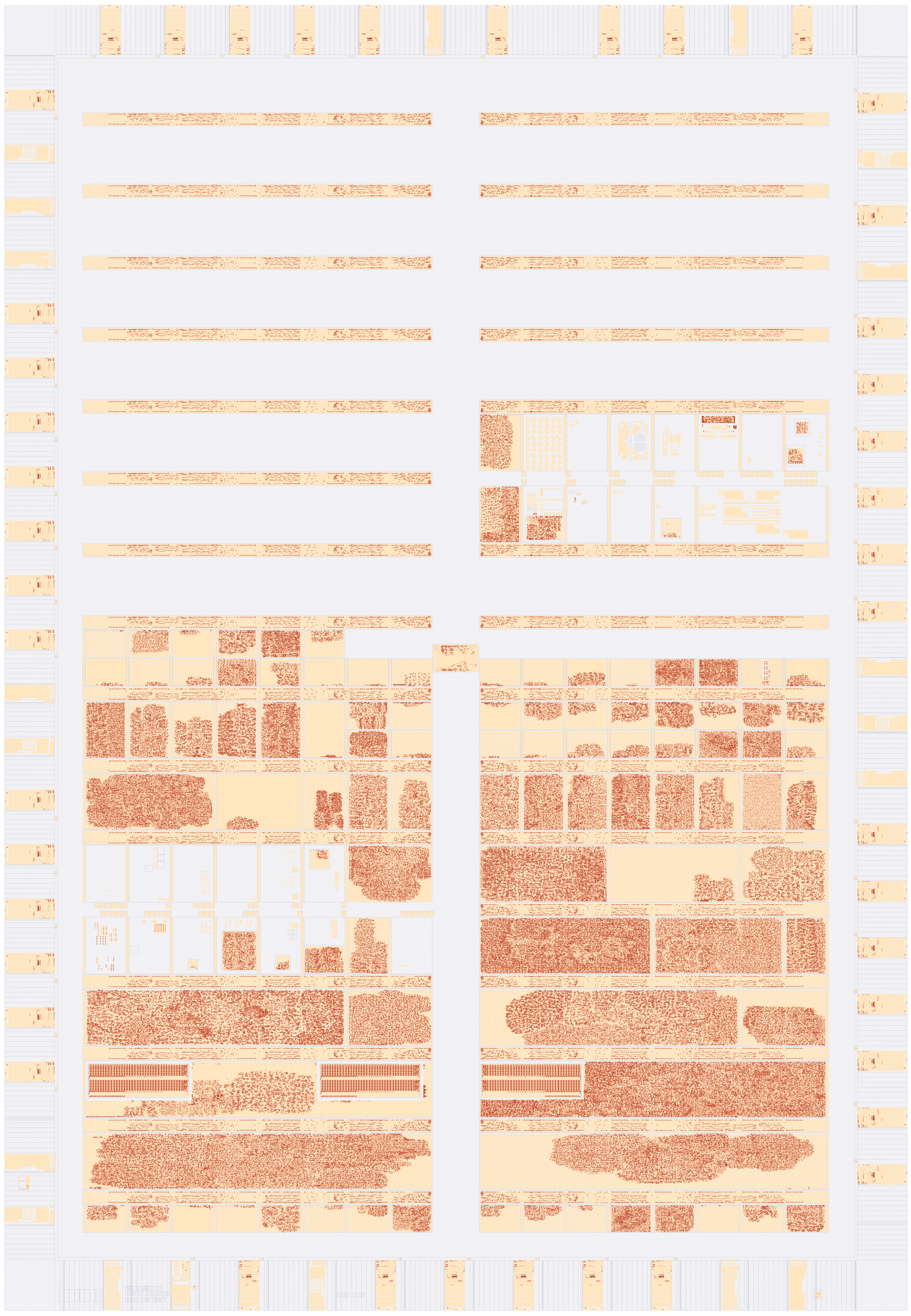


Figure 3: Logic density (local interconnect layer)

Projects

Chip ROM [0]

- Author: Uri Shaked
- Description: ROM with information about the chip
- GitHub repository
- HDL project
- Mux address: 0
- Extra docs
- Clock: 0 Hz

How it works

ROM memory that contains information about the Tiny Tapeout chip. The ROM is 8-bit wide and 256 bytes long.

The ROM layout The ROM layout is as follows:

Address	Length	Encoding	Description
0	8	7-segment	Shuttle name (e.g. "tt07"), null-padded
8	8	7-segment	Git commit hash
32	96	ASCII	Chip descriptor (see below)
248	4	binary	Magic value: `&amp;quot;TT\xFA\xBB&amp;quot;`
252	4	binary	CRC32 of the ROM contents, little-endian

The chip descriptor The chip descriptor is a simple null-terminated string that describes the chip. Each line is a key-value pair, separated by an equals sign. It contains the following keys:

Key	Description	Example value
shuttle	The identifier of the shuttle	tt07
repo	The name of the repository	TinyTapeout/tinytapeout-07
commit	The commit hash *	a1b2c3d4

- The commit hash is only included for Tiny Tapeout 5 and later.

Here is a complete example of a chip descriptor:

```
shuttle=tt07
repo=TinyTapeout/tinytapeout-07
commit=a1b2c3d4
```

How the ROM is generated The ROM is automatically generated by tt-support-tools while building the final GDS file of the chip. Look at the `rom.py` file in the repository for more details.

How to test

Read the ROM contents by setting the address pins and reading the data pins. The first 16 bytes of the ROM are 7-segment encoded and contain the shuttle name and commit hash. You can read them by toggling the first four DIP switches and observing the on-board 7-segment display.

Pinout

#	Input	Output	Bidirectional
0	addr[0]	data[0]	
1	addr[1]	data[1]	
2	addr[2]	data[2]	
3	addr[3]	data[3]	
4	addr[4]	data[4]	
5	addr[5]	data[5]	
6	addr[6]	data[6]	
7	addr[7]	data[7]	

TinyTapeout 7 Factory Test [1]

- Author: Tiny Tapeout
- Description: Factory test module
- GitHub repository
- HDL project
- Mux address: 1
- Extra docs
- Clock: 0 Hz

How it works

The factory test module is a simple module that can be used to test all the I/O pins of the ASIC.

It has three modes of operation:

1. Mirroring the input pins to the output pins (when `rst_n` is low).
2. Mirroring the bidirectional pins to the output pins (when `rst_n` is high `sel` is low).
3. Outputing a counter on the output pins and the bidirectional pins (when `rst_n` is high and `sel` is high).

The following table summarizes the modes:

`rst_n`	`sel`	Mode	ou_out value	uio pins
0	X	Input mirror	ui_in	High-Z
1	0	Bidirectional mirror	uio_in	High-Z
1	1	Counter	counter	counter

The counter is an 8-bit counter that increments on every clock cycle, and resets when `rst_n` is low.

How to test

1. Set `rst_n` low and observe that the input pins (`ui_in`) are output on the output pins (`ou_out`).
2. Set `rst_n` high and `sel` low and observe that the bidirectional pins (`uio_in`) are output on the output pins (`ou_out`).
3. Set `sel` high and observe that the counter is output on both the output pins (`ou_out`) and the bidirectional pins (`uio`).

Pinout

#	Input	Output	Bidirectional
0	sel / in_a[0]	output[0] / counter[0]	in_b[0] / counter[0]
1	in_a[1]	output[1] / counter[1]	in_b[1] / counter[1]
2	in_a[2]	output[2] / counter[2]	in_b[2] / counter[2]
3	in_a[3]	output[3] / counter[3]	in_b[3] / counter[3]
4	in_a[4]	output[4] / counter[4]	in_b[4] / counter[4]
5	in_a[5]	output[5] / counter[5]	in_b[5] / counter[5]
6	in_a[6]	output[6] / counter[6]	in_b[6] / counter[6]
7	in_a[7]	output[7] / counter[7]	in_b[7] / counter[7]

6 bit addr [2]

- Author: Nigel Coburn, Jason Murray
- Description: 6 bit addr, with continuous assignment (no clock)
- GitHub repository
- HDL project
- Mux address: 2
- Extra docs
- Clock: 0 Hz

How it works

6 Bit Addr: Overview

2x6 bit inputs with a carry 6 bit output with a carry

How to test

Addition with no carry

A=0b000001 IC = 0 B=0b000001 O=0b000010 OC = 0

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Pinout

#	Input	Output	Bidirectional
0	0th bit for A input	0th bit for A+B	0th bit for B input
1	1st bit for A input	1st bit for A+B	1st bit for B input
2	2nd bit for A input	2nd bit for A+B	2nd bit for B input
3	3rd bit for A input	3rd bit for A+B	3rd bit for B input
4	4th bit for A input	4th bit for A+B	4th bit for B input
5	5th bit for A input	5th bit for A+B	5th bit for B input
6	Input Carry bit	Output Carry bit	
7			

TTDLL [3]

- Author: Dan Petrisko
- Description: An all-digital DLL
- GitHub repository
- HDL project
- Mux address: 3
- Extra docs
- Clock: 0 Hz

How it works

This is a test design for all-digital tunable, hierarchical and composable oscillation blocks. The base primitives are delay elements, delay rows, delay columns and control registers. Control registers determine which delay elememts are active at a given setting, increasing or decreasing the total period. We deomnstrate a clock generator and 90-degree delay line as potential applications for these elements. This design has been validated in TSMC-28 using commercial tools. This will be the first test of this design on open tools and and open PDK.

For a clock generator there are an odd number of delay elements, which causes the total delay to act as half of a clock oscillation. This is a fairly simple ring oscillator design, although it is designed to be glitch-free upon frequency changes and have even steps between tuning frequencies.

For a 90-degree delay line (useful in DDR controllers among other things), we take two odd delay blocks equal in size. The output of the first block is the generated 90-degree clock. The output of the second block is a generated 180-degree clock. Now, 180-degrees is trivial phase to generate – simply invert the input clock. We can compare the racing 180-degree clocks using a (intentionally) metastable register. If the dly180 > clkinv, then we reduce the dly latency. If the dly180 < clkinv, then we increase the dly latency. This will automatically tune dly90 due to the symmetry of the blocks.

TODO: Images coming soon!

How to test

The main driver for hardware testing is bsg_tag (https://github.com/bespoke-silicon-group/basejump_stl), which has python and C++ drivers available. The exact tag sequence can be found in the test/ directory. Examples of using this testing infrastructure is in ZynqParrot (https://github.com/black-parrot-hdk/zynq-parrot). If the sequence is completed successfully, the IO outputs will increment proporotionally to the current clock frequency!

External hardware

Although the clock and delay period can be approximated by the counter outputs, an external oscilloscope is helpful for visualizing the clock and delay waveforms themselves, including phase and jitter.

Pinout

#	Input	Output	Bidirectional
0	p_tag_clk_i	p_osc_clk_o	p_div_count_0_o
1	p_tag_en_i	p_ds_clk_o	p_div_count_1_o
2	p_tag_data_i	p_gen_clk_o	p_div_count_2_o
3		p_dly_clk_o	p_div_count_3_o
4		p_mon_clk_o	p_div_count_4_o
5		p_div_clk_o	p_div_count_5_o
6		p_ds_reset_o	p_div_count_6_o
7		p_mon_reset_o	p_div_count_7_o

8-Bit Register [4]

- Author: Eric Ulteig
- Description: The 8 inputs are saved to the 8 outputs on a clock pulse
- GitHub repository
- Wokwi project
- Mux address: 4
- Extra docs
- Clock: 1 Hz

How it works

D-type flip-flops are used to store the input bits.

How to test

The 8 inputs are saved to the 8 outputs. Use the push button to save and hold.

Pinout

#	Input	Output	Bidirectional
0	input1	output1	
1	input2	output2	
2	input3	output3	
3	input4	output4	
4	input5	output5	
5	input6	output6	
6	input7	output7	
7	input8	output8	

UACJ_PWM [6]

- Author: Grupo C
- Description: Implementacion de un modulador de ancho de pulso (PWM) mediante Verilog esencial en electronica para regular la energia entregada a un dispositivo, este diseño esta bajo los requisitos del sistema
- GitHub repository
- HDL project
- Mux address: 6
- Extra docs
- Clock: 10000000 Hz

How it works

con una Implementacion de un modulador de ancho de pulso (PWM) mediante Verilog esencial en electronica para regular la energia entregada a un dispositivo, este diseño esta bajo los requisitos del sistema

How to test

con un fpga de xilinx basys 3

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Pinout

#	Input	Output	Bidirectional
0	duty_[0]	pwm_o	
1	duty_[1]		
2	duty_[2]		
3	duty_[3]		
4	duty_[4]		
5	duty_[5]		
6	duty_[6]		
7	duty_[7]		

GPS signal generator [8]

- Author: Grupo de Aplicaciones en Sistemas Embebidos - Universidad Tecnológica Nacional Facultad Regional Haedo
- Description: Generate a GPS IF signals using the following parameters: sat id, code phase, doppler and SNR.
- GitHub repository
- HDL project
- Mux address: 8
- Extra docs
- Clock: 16368000 Hz

How it works

The gps signal generator is a configurable block capable used to test search algorithms for GPS receivers. It is composed by two main blocks:

- Register bank: a set of configuration registers with a uart rx interface for write-only operations. These registers lets the user control: satellite ID, PRN code phase, doppler frequency, noise level, among others.
- Core: the core of the project is composed by a Gold Code generator, an NCO (numerically controlled oscillator) and PRNGs (pseudo random number generators). The core also provides a 1-bit message input to modulate the generated signal with a "navigation message".

Block Diagram

Serial communication

Register bank configuration is performed through the serial interface. The operation to write a single register is divided in two steps:

- Send address byte.
- Send data byte.

Register bank description

This section describes the registers of the device and its functionality.

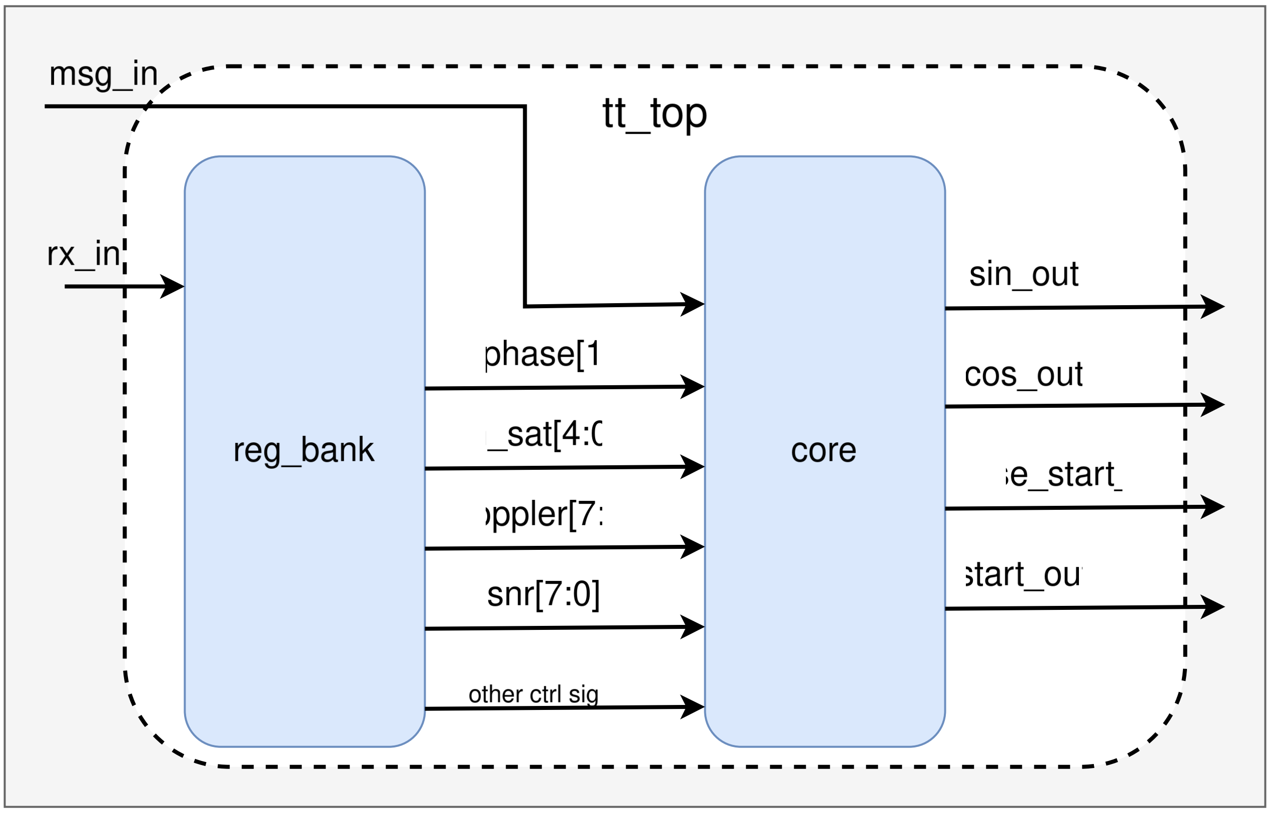


Figure 4: Block Diagram

Control Register:

- Address: 0x0
- Width: 8 bits

b7	b6	b5	b4	b3	b2	b1	b0
X	X	signal off	noise off	X	X	X	general enable

- b0: general enable of the core.
- b4: turn off noise generator.
- b6: turn off signal.

Sat_id register:

- Address: 0x2
- Width: 5 bits

b4	b3	b2	b1	b0
id[4]	id[3]	id[2]	id[1]	id[0]

- id[4:0]: Selects the satellite PRN code. There are up to 32 satellite PRN codes.

Doppler register:

- Address: 0x3
- Width: 8 bits

b7	b6	b5	b4	b3	b2	b1	b0
d[7]	d[6]	d[5]	d[4]	d[3]	d[2]	d[1]	d[0]

- d[7:0]: Doppler selection. The resultant frequency will be equal to:

$$f_{out} = 4092000.0 - 8000.0 + 500.0 * (d - 176) MHz$$

Code phase low register:

- Address: 0x4
- Width: 8 bits

b7	b6	b5	b4	b3	b2	b1	b0
p[7]	p[6]	p[5]	p[4]	p[3]	p[2]	p[1]	p[0]

- p[7:0]: low byte of the PRN code phase.

Code phase high register:

- Address: 0x5
- Width: 8 bits

b7	b6	b5	b4	b3	b2	b1	b0
p[15]	p[14]	p[13]	p[12]	p[11]	p[10]	p[9]	p[8]

- p[15:8]: high byte of the PRN code phase.

SNR register

- Address: 0x6
- Width: 3 bits

b2	b1	b0
snr[2]	snr[1]	snr[0]

- snr[2:0]: This indicates how many right shifts (») will be applied to the generated signal. Applying more shifts reduces the amplitude of the signal with respect to the generated noise, reducing the SNR.

How to test

Clock frequency of the system should be set to 16.368 MHz. The register bank is configured with a uart interface at 115200 bauds. Enable the design by writting the corresponding bit of the control bank.

External hardware A micropocessor or FPGA can be used to modulate the navigation message at the input. The output can be recorded for post-analysis or fed to the digital front end of a GPS receiver. The output is a 1-bit signal.

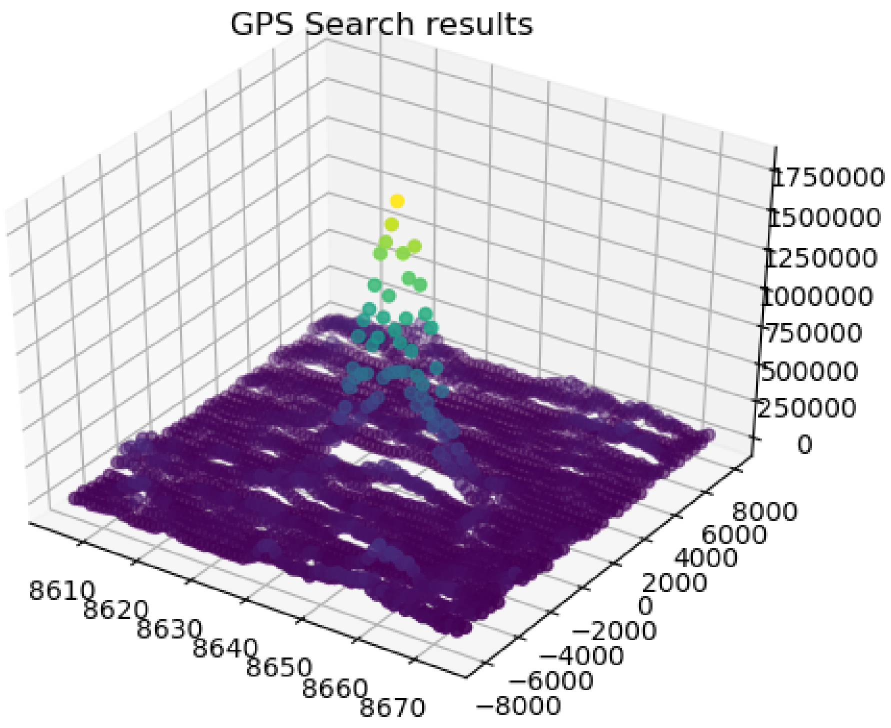


Figure 5: search_example

Example: expected output of a search algorithm

Pinout

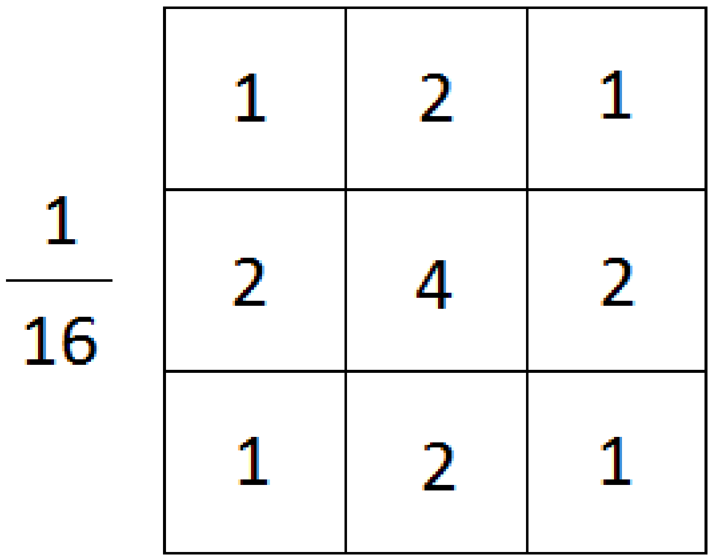
#	Input	Output	Bidirectional
0	msg_in - Used to modulate GPS signal with custom navigation message.	sin_out - Sine output with CA+msg modulation + noise.	Not used
1	Not used	cos_out - Cosine output with CA+msg modulation (no noise).	Not used
2	Not used	noise_start_out - Start of PRNG sequence used as noise.	Not used
3	rx_in - UART rx input. Used to configure the register bank.	start_out - Start of GPS signal. This output goes high when the configured phase matches the actual phase of the output signal.	Not used
4	Not used	clk - Output clock	Not used
5	Not used	Not used	Not used
6	Not used	Not used	Not used
7	Not used	Not used	Not used

Gaussian Blur [10]

- Author: Alfonso Cortes
- Description: 4-bit gaussian blur filter
- GitHub repository
- Wokwi project
- Mux address: 10
- Extra docs
- Clock: 100000 Hz

How it works

This filter receives nine 4-bit pixels in a free-running shift register and performs a gaussian blur, returning the value of the middle pixel. The weights are as shown below.



$$\frac{1}{16}\begin{array}{|c|c|c|}\hline 1 & 2 & 1 \\ \hline 2 & 4 & 2 \\ \hline 1 & 2 & 1 \\ \hline \end{array}$$

Figure 6: 0 _9K4Upm5p0aBIKDS

How to test

Input the pixel value and its neighborhood (nine pixels) from left to right, top to bottom. Once the shift register is full (after nine clock cycles) the output can be sampled. The last stage of the shift register is also available at the output for testing purposes.

External hardware

An FPGA should be useful.

Pinout

#	Input	Output	Bidirectional
0	reg_in[0]	reg_out[0]	
1	reg_in[1]	reg_out[1]	
2	reg_in[2]	reg_out[2]	
3	reg_in[3]	reg_out[3]	
4		blur[0]	
5		blur[1]	
6		blur[2]	
7		blur[3]	

UART [12]

- Author: Elisa Parent
- Description: Connecting two digital devices
- GitHub repository
- Wokwi project
- Mux address: 12
- Extra docs
- Clock: 300 Hz

How it works

Using a connector to send digital signals to different digital parts.

How to test

Switchs are used to pick the ASCII character

External hardware

arduino,7-seg display, 8 switch

Pinout

#	Input	Output	Bidirectional
0	in1	out1	
1	in2	out2	
2	in3	out3	
3	in4		
4	in5		
5	in6		
6	in7		
7	in8		

Digital Timer [14]

- Author: Francisca Donoso
- Description: The circuit acts as a configurable timer displaying the remaining time in binary form as time passes
- GitHub repository
- Wokwi project
- Mux address: 14
- Extra docs
- Clock: 0 Hz

How it works

A value is placed in the inputs from 0 to 7 to define how much it counts. Then, the start input is set to 1 to load that value, and it begins to count. When it reaches zero, the end flag is raised.

How to test

External hardware

It is recommended to use switches for the inputs, a button for the start input, and LEDs for all the outputs.

Pinout

#	Input	Output	Bidirectional
0			in_start
1			
2			
3			
4			out_state
5			out_end_flag
6			
7			

Clock Domain Crossing FIFO [36]

- Author: Pavan Mantri
- Description: This FIFO buffers 4-bits data asynchronously across clock domains
- GitHub repository
- HDL project
- Mux address: 36
- Extra docs
- Clock: 0 Hz

How it works

The cdc_fifo module transfers data between two clock domains: a write clock domain and a read clock domain. The module includes a dual-ported RAM(dpram) for storing data, along with logic for handling read and write operations(cdc_fifo_read_state and cdc_fifo-write-state). synchronizers(synchronizer) and binary/gray converters(binary_to_gray and gray_to_binary) ensure proper synchronization between two clock domains.

How to test

Hold write_reset and read_reset LOW while running the clock for a bit to reset, then raise to initialize the module.

writing to the fifo: Prepare your data on the 4-bit write_data bus, ensure the full state is low and then raise write_increment for 1 cycle of write_clock to write data into the FIFO memory.

Reading from the fifo: The FIFO will present the current output on the read_data bus. If empty is low, this output should be valid and you can acknowledge receive of this vallue by raising read_increment for 1 cycle of read_clock.

External hardware

NO external hardware is used.

Pinout

#	Input	Output	Bidirectional
0	write_clock	empty	write_reset
1	write_increment	full	read_reset
2	read_clock		
3	read_increment		
4	write_data0	read_data0	
5	write_data1	read-data1	
6	write_data2	read_data2	
7	write_data3	read_data3	

Pong-VGA [38]

- Author: Victor Zayakov
- Description: Pong game over a VGA connection
- GitHub repository
- HDL project
- Mux address: 38
- Extra docs
- Clock: 50000000 Hz

How it works

This project is an implementation of the classic Pong video game. The interface is VGA, and the game will be displayed when connected to a monitor using a VGA cable. This includes all elements of the game: the two paddles and the ball, and a scoreboard.

The VGA protocol is implemented in hardware using Counter modules, RangeCheck modules, and an FSM to control them. It follows the VGA specification of 640x480 resolution at 60Hz, and operates using a 50MHz clock signal.

The rest of the hardware for the Pong video game works on top of the VGA module.

How to test

The game can be tested through a direct connection to a monitor over VGA. Four switches and one button are required to interact with the game: An up/down switch and move/no-move switch for each of the two players, and one button for serving the ball.

To test the HDL design, one can compare the timing of all 8 VGA pins to the industry-standard timing defined here: http://www.tinyvga.com/vga-timing/640x480@60Hz

External hardware

The game requires four switches and one button to play, as mentioned above. It also requires an external splitter for each of the three VGA color signals. This is to split each of VGA_R, VGA_G and VGA_B from 1-bit to 8-bit signals. These splitters can be placed on an external PCB.

Pinout

#	Input	Output	Bidirectional
0	R_move_switch	VGA_CLK	
1	R_up_switch	VGA_BLANK_N	
2	L_move_switch	VGA_SYNC_N	
3	L_up_switch	VGA_VS	
4	serve_L_button	VGA_HS	
5		VGA_R	
6		VGA_G	
7		VGA_B	

Iterative MAC [40]

- Author: Raju Machupalli
- Description: Iterative multiply and accumulation unit for ML accelerators
- GitHub repository
- HDL project
- Mux address: 40
- Extra docs
- Clock: 50000000 Hz

How it works

The design contains iterative multiplication and addition unit. The multiplier is a 7x8 bit unit. At reset time, input through ui_in pins stores in a reg and used as operand 1 for multiplier. After reset, operand 2 for multiplier is supplied through ui_in at each clock cycle. The bidirectional pins provide operand 3 which will be added to the multiplier output. the output is read through uo_out pins.

How to test

It's a bit complex, as bias values are supplied in different sequences, and output needs to change or align with the read output. Full instructions will be added here once the design is submitted for fabrication.

External hardware

It does not require any additional hardware supply the input data using CPU.

Pinout

#	Input	Output	Bidirectional
0	ui_in[0]	uo_out[0]	uio_in[0]
1	ui_in[1]	uo_out[1]	uio_in[1]
2	ui_in[2]	uo_out[2]	uio_in[2]
3	ui_in[3]	uo_out[3]	uio_in[3]
4	ui_in[4]	uo_out[4]	uio_in[4]
5	ui_in[5]	uo_out[5]	uio_in[5]
6	ui_in[6]	uo_out[6]	uio_in[6]

#	Input	Output	Bidirectional
7	ui_in[7]	uo_out[7]	uio_in[7]

Adiabatic PSU sequencer test [42]

- Author: Chris Pacejo
- Description: test of a power supply sequencer for adiabatic circuits
- GitHub repository
- HDL project
- Mux address: 42
- Extra docs
- Clock: 32000000 Hz

How it works

This is a simple sequencer for a 9-step 4-phase combined power-clock PSU for adiabatic circuitry. The rate of each phase is 1/32 the input clock rate.

It generates two sets of control signals, for phases 0 and 1. The control signals indicates which step of the charging circuitry should be activated. (Phases 2 and 3 simply invert the meaning of the steps.)

Since there are not enough output pins to represent all steps, pin `ui[0]` selects whether phase 0 or 1 is routed to the output pins. Steps 1 through 7 of the selected phase are routed to `uo[1]` through `uo[7]`. Steps 0 and 8 of both phases 0 and 1 are routed to pins `uio[0]` through `uio[3]` regardless of the setting of `ui[0]` .

Finally, four "digital sync" signals are generated and routed to pins `uio[4]` through `uio[7]`. These mark appropriate clock cycles when data can be latched in to or out from an adiabatic gate on phase 0 or 1.

How to test

This project is free-running. Simply issue a reset, then use `ui[0]` to select which phase you wish to monitor, and monitor it.

External hardware

No external hardware is necessary.

Pinout

#	Input	Output	Bidirectional
0	phase select (X)	selected phase (X)	phase 0 step 0
1		phase X step 1	phase 0 step 8
2		phase X step 2	phase 1 step 0
3		phase X step 3	phase 1 step 8
4		phase X step 4	digital sync phase 0 read
5		phase X step 5	digital sync phase 1 read
6		phase X step 6	digital sync phase 0 write
7		phase X step 7	digital sync phase 1 write

PRBS Generator [44]

- Author: David Parent
- Description: Generates a PRBS signal
- GitHub repository
- HDL project
- Mux address: 44
- Extra docs
- Clock: 0 Hz

How it works

The chip generates a PRBS31 signal using a Fibonacci LFSR and analyzes it with the same structure. The output of the PRBS is taken off the chip and read back in to be analyzed.

Two 7-bit vectors are converted into puedo random signal PSR by comparing the vector to the PRBS. These signals are also output and can be used as an alternative to a PWM DAC. These two PRS are multiplied with an and gate, and the out is sent off-chip. Singal A is squared by delaying it by one clock cycle and anding the signal with the delayed version.

A 131-bit PRBS generator is included as well to fill up the tile as much as possible.

Everything will be documented here:https://docs.google.com/document/d/1nhcHBQsxXUUo1_

How to test

Input Clock and reset

External hardware

ADALM2000

#	Input	Output	Bidirectional

Pinout

#	Input	Output	Bidirectional
0	i1	out1	
1	i2	out2	
2	i3	out3	
3	i4		
4	i5		
5	i6		
6	i7		
7	i8		

Full-adder out of a kmap [46]

- Author: Levi Feldman
- Description: A full-adder made out of a kmap.
- GitHub repository
- Wokwi project
- Mux address: 46
- Extra docs
- Clock: 0 Hz

How it works

It is a straightforward full-adder circuit but it is made directly out of the kmaps for both outputs.

How to test

Standard full-adder usage. But the implementation is different though.

External hardware

None.

Pinout

#	Input	Output	Bidirectional
0	A	SUM	
1	B	C-OUT	
2	C-IN		
3			
4			
5			
6			
7			

dEFAULt 2hAC [64]

- Author: Beau Ambur
- Description: Inverted 7-segment !(tinY tAPeout)
- GitHub repository
- Wokwi project
- Mux address: 64
- Extra docs
- Clock: 1 Hz

How it works

Sequentially spells out a message using the 7-segment display.

How to test

Decipher the hidden message by inverting the segments.

Pinout

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

ECC_test1 [65]

- Author: Dr. Vaibhav Neema
- Description: it is use for detecting single bit error
- GitHub repository
- Wokwi project
- Mux address: 65
- Extra docs
- Clock: 0 Hz

How it works

The objective of this work is to design a chip module which performs 1-bit error detection and correction in the transmitted encoded data. The proposed design includes three sections: a transmitter, built in self-test block (for testing purposes), and a receiver. The transmitter takes an 8-bit input and generates 4 redundant bits, creating a 12-bit encoded data stream for transmission. The second block is used as built in self-test block(BIST). This block is used to intentionally insert an error during transmission of encoded data to properly test our design. If no error is inserted in the transmitted code word, 4 output pins display the redundant bits. However, If 1-bit error is provided, these pins display the error position. The receiver block can correct any 1-bit error in the received 12-bit data and displays the corrected data as provided at the input side.

How to test

This design has 8-input data pins which will use to provide digital input bits, 8 output pins will use to check the output by connectings LED's. Four birectional pins will use as a input ports for BIST and remaining four bidirectional pins will use as output pins to detect the postion of the error.

External hardware

Twelve LED's, LED Display, four bit counter, Eight bit counter and Twelve digital input switch

Pinout

#	Input	Output	Bidirectional
0	IN0	OUT0	ttio0:IN
1	IN1	OUT1	ttio1:IN
2	IN2	OUT2	ttio2:IN
3	IN3	OUT3	ttio3:IN
4	IN4	OUT4	ttio4:OUT
5	IN5	OUT5	ttio5:OUT
6	IN6	OUT6	ttio6:OUT
7	IN7	OUT7	ttio7:OUT

All Digital DAC and Analog Comparators [66]

- Author: Maximiliam Luppe
- Description: Implementation of a DAC and three versions of a analog comparator using only standard cells
- GitHub repository
- HDL project
- Mux address: 66
- Extra docs
- Clock: 0 Hz

How it works

This project implements three different analog comparators based on standard logic cells. They are based on work of Sala et al. [1].

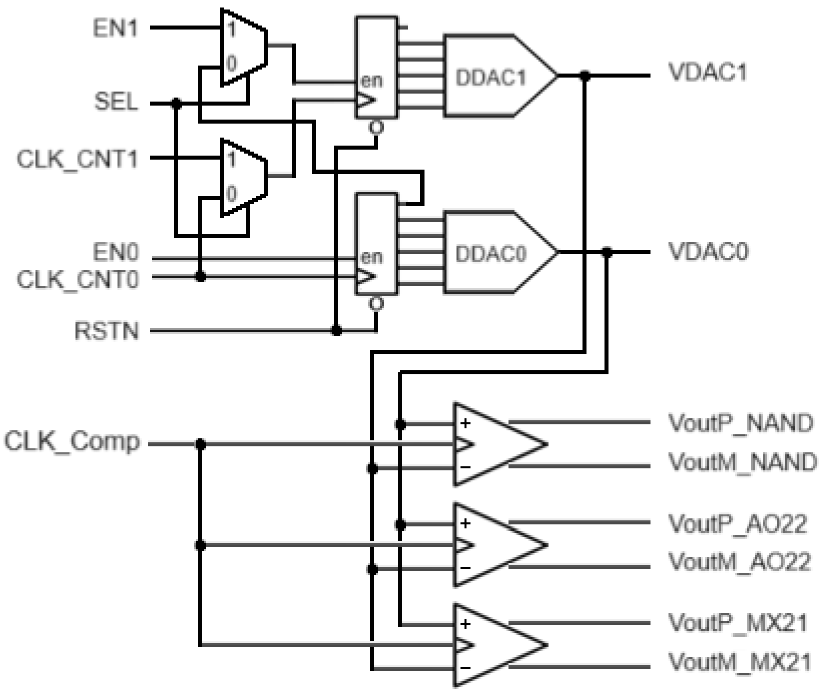


Figure 7: Digital DAC and Comparators diagram

Two DACs, based on the work of Yang et al. [2], also implemented using only standard logic cells, with the aid of two 5-bit counters, generate an analog ramp signal to test the comparators.

1. R. D. Sala, C. Bocciarelli, F. Centurelli, V. Spinogatti and A. Trifiletti, "A Novel Ultra-Low Voltage Fully Synthesizable Comparator exploiting

NAND Gates," 2023 18th Conference on Ph.D Research in Microelectronics and Electronics (PRIME), Valencia, Spain, 2023, pp. 21-24, doi: 10.1109/PRIME58259.2023.10161936

2. D. Yang, T. Ueno, W. Deng, Y. Terashima, K. Nakata, A. T. Narayanan, R. Wu, K. Okada, A. Matsuzawa, "A 0.0055mm2 480µW Fully Synthesizable PLL Using Stochastic TDC in 28nm FDSOI," IEICE Transactions on Electronics, v. E99.C, no. 6, 2016, pp. 632-640, doi: 10.1587/transele.E99.C.632

How to test

The SEL pin alows to select two different test conditions. With SEL=0, both counters work together, generating a 10-bit sequence. For each step in the DAC1, DAC0 generates 32 different voltage levels, from near 0V to near Vcc. With SEL=1, both counters work independently.

External hardware

It's necessary an osciloscope to visualize the outputs from the DACs and the comparators.

Pinout

#	Input	Output	Bidirectional
0	First Counter Clock	VinP	
1	Second Counter Clock	VinM	
2	Comparators Clock	VoutP_NAND	
3	First Counter Enable	VoutM_NAND	
4	Second Counter Enable	VoutP_AO22	
5	Counter Mode Selection	VoutM_AO22	
6		VoutP_MX21	
7		VoutM_MX21	

443MHz Manchester Decoding [67]

- Author: Zachary Kohnen
- Description: A manchester decoder and parser for a 433 mhz transmission
- GitHub repository
- HDL project
- Mux address: 67
- Extra docs
- Clock: 20000 Hz

How it works

There are 2 state machines. One to decode the manchester encoded signal, and the other to parse the data frames. For the manchester decoding aspect, the following state machine is implemented.

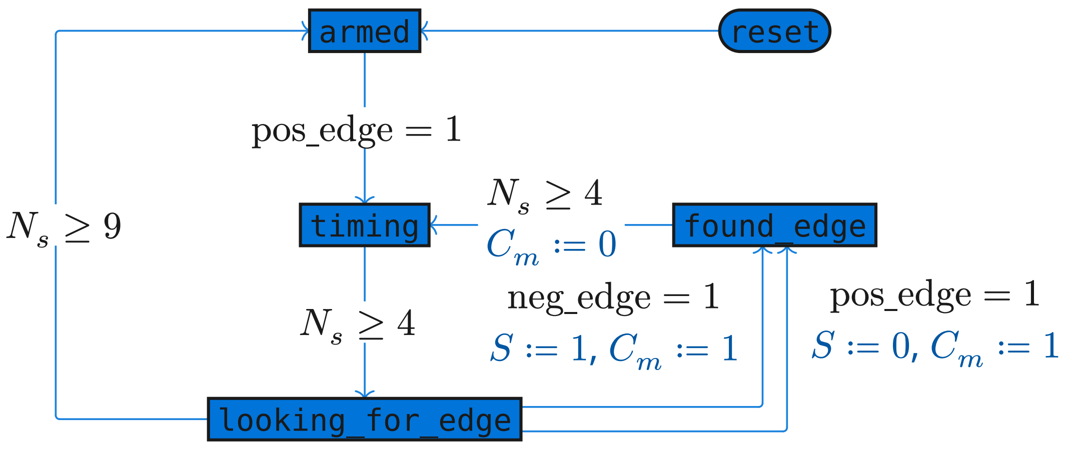
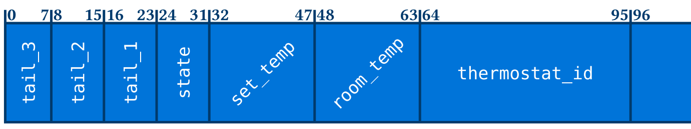


Figure 8: State Diagram

Blue text are side effects of the transitions. N_s is a counter that increments while in each state, and is reset on each state transition.

The state machine makes use of a delay of 5 cycles in order to ignore any non-data carrying transition.

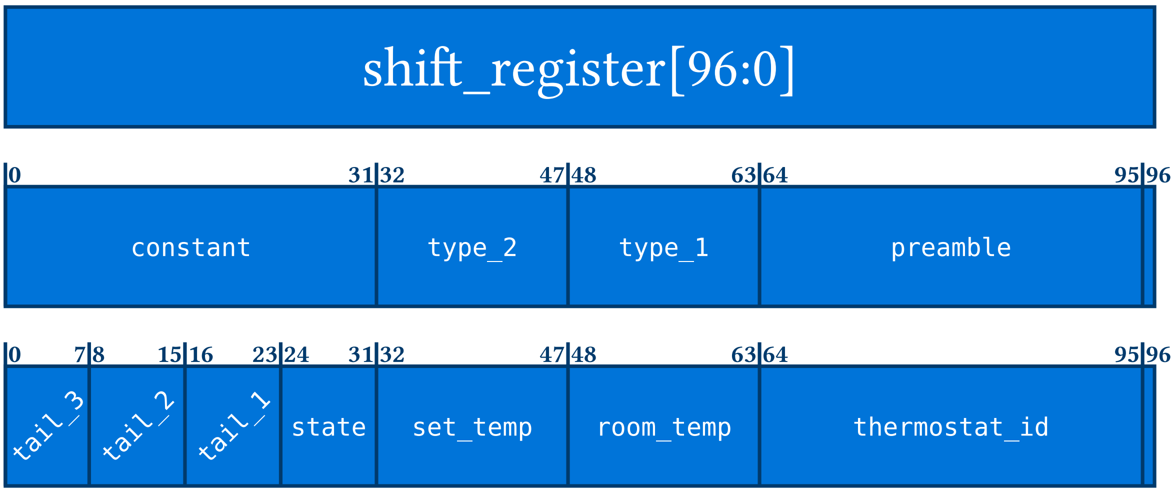
The data sent over the air follows the following format. (bit labels are as the data is



shifted into a shift-register (0 = last bit received))

In order to save space in the design, since the preamble section and data section are of equal length, only half of the message needs to be buffered at a time.

The below diagram shows 2 views into the shift register. The first view is used to validate the existence of a preamble. Once that has been tripped, the system reloads the register with 0x1, and fills the register again until the initial set bit falls into the last (96) place, causing the shift register to halt all loads until it is reset.



How to test

Wire the system up, and read out the internal registers through the `parallel_out` lines

Address	Parallel Out
0000	`thermostat_id[7:0]`
0001	`thermostat_id[15:8]`
0010	`thermostat_id[23:16]`
0011	`thermostat_id[31:24]`
0100	`room_temp[7:0]`
0101	`room_temp[15:8]`
0110	`set_temp[7:0]`
0111	`set_temp[15:8]`
1000	`state`
1001	`tail_1`

Address	Parallel Out
1010	`tail_2`
1011	`tail_3`
1100-1111	00000000

External hardware

Pin `ui_in[0]` (`digital_in`) must be connected to the digital output of a 433MHz receiver

Pinout

#	Input	Output	Bidirectional
0	digital_in	parallel_out[0]	full
1		parallel_out[1]	manchester_clock
2	halt	parallel_out[2]	manchester_data
3		parallel_out[3]	transmission_begin
4	address[0]	parallel_out[4]	neg_edge
5	address[1]	parallel_out[5]	pos_edge
6	address[2]	parallel_out[6]	
7	address[3]	parallel_out[7]	

Dummy Counter [68]

- Author: Chinmay
- Description: A 16-bit counter
- GitHub repository
- HDL project
- Mux address: 68
- Extra docs
- Clock: 0 Hz

How it works

Like a 16-bit counter

How to test

Like a 16-bit counter

External hardware

NA

Pinout

#	Input	Output	Bidirectional
0	count_en	b0	b8
1	mult_en	b1	b9
2	m_a0	b2	b10
3	m_a1	b3	b11
4	m_a2	b4	b12
5	m_b0	b5	b13
6	m_b1	b6	b14
7	m_b2	b7	b15

MicroCode Multiplier [69]

- Author: Neil Powell
- Description: microcode unit for shift and add multiplication
- GitHub repository
- HDL project
- Mux address: 69
- Extra docs
- Clock: 1000 Hz

How it works

Input two 4-bit numbers A and B. 8-bit product is returned.

How to test

Just reset and supply the inputs

External hardware

Switches and LEDs

Pinout

#	Input	Output	Bidirectional
0	inputA[0]	SMP_out[0]	
1	inputA[1]	SMP_out[1]	
2	inputA[2]	SMP_out[2]	
3	inputA[3]	SMP_out[3]	
4	inputB[0]	SMP_out[4]	
5	inputB[1]	SMP_out[5]	
6	inputB[2]	SMP_out[6]	
7	inputB[3]	SMP_out[7]	

My 9-year-old son made an 8-bit counter chip [70]

- Author: Kian Dabui
- Description: My 9-year-old son made an 8-bit counter chip following instructions and a picture with a brief explanation.
- GitHub repository
- Wokwi project
- Mux address: 70
- Extra docs
- Clock: 0 Hz

How it works

My 9-year-old son made an 8-bit counter chip following instructions and a picture with a brief explanation.

How to test

The output from Q0 to Q7 counts up in binary.

External hardware

The output from Q0 to Q7 counts up in binary using an LED Pmod.

Pinout

#	Input	Output	Bidirectional
0		q0	
1		q1	
2		q2	
3		q3	
4		q4	
5		q5	
6		q6	
7		q7	

TT7 Simple Clock [71]

- Author: Joseph Hsu
- Description: This is a very simple, configurable clock. By default it's 20 MHz, but it can be configured to 10/12/14 Mhz.
- GitHub repository
- HDL project
- Mux address: 71
- Extra docs
- Clock: 20000000 Hz

How it works

So this clock is configured off of the FPGA clock at 20 MhZ. However, the clock is configurable to different frequencies and settings.

How to test

See if the clock works. The first, second, and third switches should speed up/slow down the clock, which can be used to set the clock. The fourth switch is used to go to arduino mode. The fifth, sixth switches should allow you to switch between the minutes display and the hours display respectively. The seventh switch should allow you you to run the clock in 4 segment display mode.

External hardware

PMOD 2 segment display, possible 4 segment

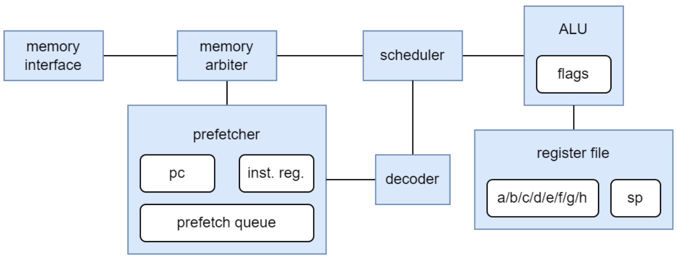
Pinout

#	Input	Output	Bidirectional
0			
1	Clock Option - 10 MHz		
2	Clock Option - 12 MHz		
3	Clock Option - 14 MHz		
4	Arduino Mode		
5	Minute Display		
6	Hour Display		

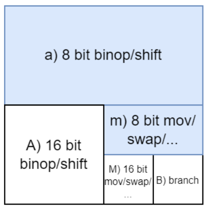
#	Input	Output	Bidirectional
7	4 Digit Mode		

Basilisc-2816 v0.1c CPU (experimental) [72]

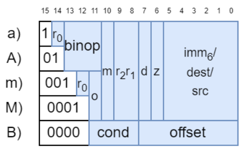
- Author: Toivo Henningsson
- Description: Small 2-bit serial 8/16 bit CPU
- GitHub repository
- HDL project
- Mux address: 72
- Extra docs
- Clock: 50000000 Hz



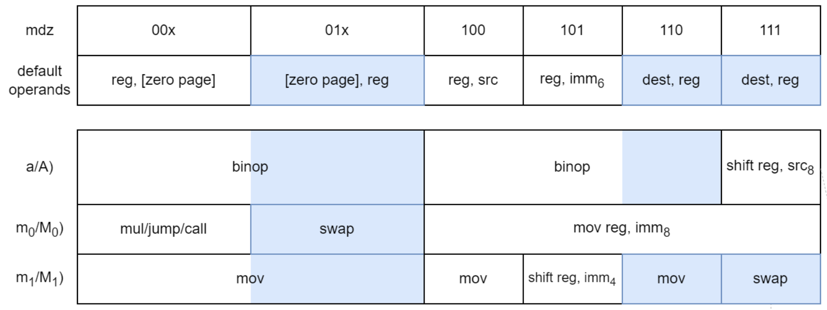
Major parts and interconnections of the processor.



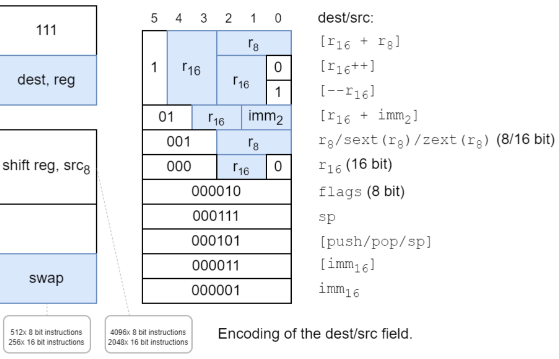
Major division of the instruction encoding space based on the top 4 bits.
8 bit instructions use 3 bits to choose between the 8 registers:a/b/c/d/e/f/g/h,
16 bit instructions use 2 bits to choose between the 4 registers ba/dc/fe/hg.



Bit fields of the major instruction forms.
The o bit selects between m_0/m_1 or M_0/M_1.



Further division of the a/A/m/M) encoding spaces into instruction forms.
Shaded instructions write to memory.



Encoding of the dest/src field.

Overview

Basilisc-2816 v0.1 is a small 2-bit serial 2/8/16 bit processor that fits into one Tiny Tapeout tile. It has been designed around the constraints of

- small area,
- 4 pin serial memory interface to a RAM emulator implemented in an RP2040 microcontroller (which can be supported by the RP2040 microcontroller on the Tiny Tapeout 7 Demo Board),
- to be suitable to be included in in the next version of the AnemoneGrafx-8 retro console https://github.com/toivoh/tt06-retro-console, which motivates the other constraints.

Features:

- 2-bit serial execution:

  - ALU results etc are calculated at 2 bits/cycle
  - 2-bit-serial register file with two read/write ports
  - Addresses and data are sent to/from memory at 2 bits/cycle
    * The processor starts to operate on each bit of incoming read data as it arrives
  - Saves area compared to processing 8/16 bits per cycle / using a parallel access register file
  - No point in calculating faster than the memory interface allows
- 8x 8-bit general purpose registers that can be paired into 4x 16-bit general purpose registers, plus an 8 bit stack register
- 8 bit and 16 bit versions of almost all instructions
- 64 kB address space
- 16 bits/instruction
- Quite regular and orthogonal instruction encoding, most instructions can use most addressing modes
  - `op reg, src` and `op src, reg` instruction forms
- Instructions:
  - `mov`, `swap`
  - `binop`: `add/adc/sub/sbc/and/or/xor/cmp/test`
    * for register-to-register also: `neg/negc/revsub/revsbc/and_not/or_not/xor_not/not`,
  - `shl/shr/sar/rol/ror` with variable or immediate shift count,
  - `mul`: 8x8 and 8x16 bit multiply instructions, producing 2 result bits per cycle like everything else,
  - `branch cc, offset`: relative branch
    * unconditional/call/12 conditions including signed/unsigned comparisons,
  - `jump/call`: absolut direct/indirect jump/call,
  - additional functionality through combination with addressing modes, e g, `ret = jump [pop]`
- Addressing modes:
  - `[imm7] / [imm7*2]`: zero page
  - `[r16 + imm2]`
  - `[r16 + r8]`
  - `[r16]` with postincrement/predecrement
  - `[push] / [pop] / [top-of-stack]` depending on whether the operand is written/read/modified

    – `[imm16]`

- Sign/zero extension of any 8 bit register as source operand to 16 bit instructions
- `imm16` / `[imm16]` operands supported using extra instruction word
- 2-4 word instruction prefetch queue

Basilisc-2816 v0.1 variants

Basilisc-2816 v0.1 has been taped out in three variants for Tiny Tapeout 7:

```
        mul           Prefetch     Hardened      Uses       Mux
        instruction   queue size   with          latches    address
v0.1a           yes            2   OpenLane 1         no        967
v0.1b            no            3   OpenLane 2         no        202
v0.1c           yes            4   OpenLane 2        yes         72
```

successively more experimental. Longer prefetch queue should help contribute to better performance, especially with long memory access latencies.

This is the 0.1c version. For more details, see https://github.com/toivoh/tt07-basilisc-2816-cpu/blob/main/docs/info.md or the documentation for Basilisc-2816 v0.1a CPU [967].

Pinout

#	Input	Output	Bidirectional
0	rx_in[0]	tx_out[0]	
1	rx_in[1]	tx_out[1]	
2		tx_fetch	
3		tx_jump	
4			
5			
6			
7			

SIMON [73]

- Author: seanyen0
- Description: SIMON game
- GitHub repository
- HDL project
- Mux address: 73
- Extra docs
- Clock: 20000000 Hz

How it works

SIMON game. Comes up with sequences of increasing length. User imitates sequence with Digilent PmodBTN. If user loses, 7-segment display shows "L" and user's max number of correct sequences in hex. Upon losing, user can press any button to reset.

How to test

Manual reset can be applied by toggling input "in7" high-low.

User needs PmodBTN to enter the guesses.

External hardware

Digilent PmodBTN plugged into upper row of input header.

Pinout

#	Input	Output	Bidirectional
0	Button 0	sseg0	
1	Button 1	sseg1	
2	Button 2	sseg2	
3	Button 3	sseg3	
4		sseg4	
5		sseg5	
6		sseg6	
7	Reset	sseg7	

VGA Snake Game [74]

- Author: Barak Hoffer
- Description: Snake game with vga output
- GitHub repository
- HDL project
- Mux address: 74
- Extra docs
- Clock: 31500000 Hz

How it works

A simple snake game with vga output and left,right,up,down buttons.

How to test

Connect to a VGA monitor. Change left,right,up,down (ui_in[0:3]) buttons to change movement.

External hardware

TinyVGA PMOD

Pinout

#	Input	Output	Bidirectional
0	left	R1	
1	right	G1	
2	up	B1	
3	down	VSync	
4		R0	
5		G0	
6		B0	
7		HSync	

Modified Booth Multiplier [75]

- Author: Varun Chandra Pendyala
- Description: The proposed design multiplies two 8bit signed numbers using modified booth algorithm
- GitHub repository
- HDL project
- Mux address: 75
- Extra docs
- Clock: 100000000 Hz

How it works

The project multiplies two 8 bit signed numbers and generates a 16 bit product using Modified Booth Multiplier Algorithm

How to test

Two 8-bit signed numbers are fed as input to the multiplier, the multiplier bits are recoded and fed to a dadda multiplier design and the corresponding outputs are added in a carry lookahead adder to get the final 32 bit product

External hardware

N.A

Pinout

#	Input	Output	Bidirectional
0	multiplicand[0]	product[0]	product[8]
1	multiplicand[1]	product[1]	product[9]
2	multiplicand[2]	product[2]	product[10]
3	multiplicand[3]	product[3]	product[11]
4	multiplicand[4]	product[4]	product[12]
5	multiplicand[5]	product[5]	product[13]
6	multiplicand[6]	product[6]	product[14]
7	multiplicand[7]	product[7]	product[15]

GDS counter-measures experiment 1 [76]

- Author: Aurélien Hernandez
- Description: Experiment with GDS-level open-source countermeasure implementation
- GitHub repository
- HDL project
- Mux address: 76
- Extra docs
- Clock: 0 Hz

How it works

PoC of custom hardened macro for fab testing and research purposes. It simply expose a set of customized matrices hardened.

How to test

Select one of the two sets of matrices using the MAT_SEL input. Compare the observed IN –> OUT mapping using the reference model (GDS).

External hardware

Nothing required.

Pinout

#	Input	Output	Bidirectional
0	in_mat_0	out_mat_0	mat_sel
1	in_mat_1	out_mat_1	
2	in_mat_2	out_mat_2	
3	in_mat_3	out_mat_3	
4	in_mat_4	out_mat_4	
5	in_mat_5	out_mat_5	
6	in_mat_6	out_mat_6	
7	in_mat_7	out_mat_7	

unisnano [77]

- Author: Maria, Diego, Victor
- Description: UART menu with text output and hardware actions
- GitHub repository
- HDL project
- Mux address: 77
- Extra docs
- Clock: 50000 Hz

How it works

A basic UART driver TX and RX is created and waits for the user input as characters (From 1 to 7 with no line ending). If options from 1 to 5 are selected, an informative text is displayed. If options from 6 to 7 are selected, an output bit is toggled

How to test

An UART transciever is requiered, using the recommended UART pins RX on #3 and TX on #34. A 50MHZ clock is needed at 115200bps

External hardware

Two LED's with a MOSFET driver

Pinout

#	Input	Output	Bidirectional
0			
1		output1	
2		output2	
3	rx		
4		tx	
5			
6			
7			

fractran-tt [78]

- Author: Jack Leightcap
- Description: Hardware implementation of John Conway's estoeric turing-complete lanugage Fractran
- GitHub repository
- HDL project
- Mux address: 78
- Extra docs
- Clock: 1 Hz

How it works

Fractran is an esoteric programming language built around prime factorization.

Fractran programs are lists of positive fractions: e.g.,

$$\frac{17}{91}, \frac{78}{85}, \frac{19}{51}, \frac{23}{38}, \frac{29}{33}, \frac{77}{29}, \ldots$$

Execution follows 3 rules:

1. The program is given an initial input integer $n \in \mathbb{N}$. This is the "accumulator" value.
2. To compute the next state, $n \leftarrow qn$ where $q \in \mathbb{Q}$ is the first fraction in the program where $qn \in \mathbb{N}$.
3. Repeat (2) until no such q exists, then halt with output n.

Depending on how terms are represented, (2) is a very simple operating to implement in hardware. The "cheat" is to operate on pre-factored values: for example, the first few fractions of the above example:

$$2^0 3^0 5^0 7^{-1} 11^0 13^{-1} 17^1, 2^1 3^1 5^{-1} 7^0 11^0 13^1 17^{-1}, 2^0 3^{-1} 5^0 7^0 11^0 13^0 17^{-1} 19^1, 2^{-1} 3^0 5^0 7^0 11^0 13^0$$

For $n = 825 = 3^1 5^2 11^1$, $nq \in \mathbb{N}$ if all pairwise added prime factor degrees are positive: testing $825 \times \frac{17}{91}$:

$$3^1 5^2 11^1 \times 7^{-1} 13^{-1} = 3^1 5^2 7^{-1} 11^1 13^{-1}$$

The negative degrees are not cancelled by the terms of n: testing $825 \times \frac{29}{33}$,

$$3^1 5^2 11^1 \times 3^{-1} 11^{-1} 29^1 = 5^2 29^1$$

All negative degrees cancel, and the result is written as the new accumulator.

How to test

See port mapping in info.yaml.

Encodings:

- accumulator: 8-bit unsigned integer degrees. Value `0b11111111` reserved as sentinel "STOP" value.
- fraction: 8-bit signed (one's complement) degrees. Value `0b11111111` (the "second zero") reserved as sentinel "STOP" value.

Apply to these two inputs pair of streams of prime factor degrees. When the each stream is exhausted, apply the "STOP" value.

For each input, there is output:

- resultant degree, or "STOP" when both input streams exhausted, indicating a positive result and accumulator writeback.
- HALT, when a negative degree is calculuated, indicating the start of the next fraction.

External hardware

The logic implemented internally is quite small, requiring support circuitry. This might include:

1. fraction counter: program counter
2. degree pointer: counter for current prime term
3. fraction ROM: storing prime degrees, punctuaed by "STOP"s
4. banked accumulator RAM: two banks of memory to store current accumulator, and calculated value. on an integral result, the 'scratch' bank is switched to accumulator, old accumulator becomes 'scratch'.

Pinout

#	Input	Output	Bidirectional
0	accumulator stream [0]	factorized stream [0]	fraction stream [0]
1	accumulator stream [1]	factorized stream [1]	fraction stream [1]
2	accumulator stream [2]	factorized stream [2]	fraction stream [2]
3	accumulator stream [3]	factorized stream [3]	fraction stream [3]
4	accumulator stream [4]	factorized stream [4]	fraction stream [4]
5	accumulator stream [5]	factorized stream [5]	fraction stream [5]
6	accumulator stream [6]	factorized stream [6]	fraction stream [6]
7	accumulator stream [7]	factorized stream [7]	fraction stream [7]

Phase Shifted PWM Modulator [79]

- Author: Nelson Salvador & Francisca Donoso
- Description: Phase-Shifted Pulse Width Modulation (PS-PWM) that generates the switching signals for 2 PMOS and 2 NMOS from a duty cycle (d1 and d2)
- GitHub repository
- HDL project
- Mux address: 79
- Extra docs
- Clock: 1000 Hz

How it works

The Phase Shifted PWM (PS-PWM) system generates phase-shifted PWM signals used for controlling power converters. The main module orchestrates the process by integrating various submodules. It starts by receiving and assigning inputs, then uses a shift register to process serial data, which determines control signals for selecting clock sources and phase-shifted triangular waveforms. These waveforms are generated by dedicated modules for different phases (0, 90, 180, and 270 degrees). The system selects the appropriate phase for two channels and compares these waveforms with input data to produce raw PWM signals. Dead time generators add configurable delays to these signals to prevent transistor cross-conduction. Finally, an output multiplexer and enable control ensure the PWM signals are correctly outputted based on enable signals, producing the desired PS-PWM output.

How to test

1. Initial Setup

- **Connect Power Supply:**
  - Ensure the module is powered correctly.
- **Clock Signal:**
  - Connect a function generator to the `clk` input.
- **Control Signals:**
  - Connect switches or signal sources for `rst_n`, `CLK_SR`, and `data_SR`.
- **Inputs:**
  - Connect `ui_in` and `uio_in` to signal sources like DIP switches or a microcontroller.

2. **Reset the Module**
   - **Procedure:**
     - Set `rst_n` to low to reset the module.
     - Observe the module's outputs to confirm they reset.
     - Set `rst_n` to high to release the reset.

3. **Shift Data into the Shift Register**
   - **Procedure:**
     - Set `data_SR` to the first bit of your 11-bit data (`1` or `0`).
     - Pulse `CLK_SR` high, then low to clock in the bit.
     - Repeat for each bit in your data sequence (e.g., `11'b00011001101`. Sequentially input each bit representing dt[0] to dt[4], SELECTOR_SIGNAL_GENERATOR_1[0], SELECTOR_SIGNAL_GENERATOR_1[1], SELECTOR_SIGNAL_GENERATOR_2[0], SELECTOR_SIGNAL_GENERATOR_2 OUTPUT_SELECTOR_EXTERNAL[0], and OUTPUT_SELECTOR_EXTERNAL[1 into the data_SR input. For each bit, you set data_SR to the corresponding value (1 or 0) and toggle CLK_SR high, then low, to clock in the bit. This sequential shifting ensures that each data_out corresponds to the specified comment name within the Shift_Register module.).

4. **Configure `ui_in` and `uio_in` (example, 20% duty cycle)**
   - **Procedure:**
     - Set `ui_in` to `11010000` to set `d1 = 13` (d1 and d2 are 6 bit length, so 13/64 is about 20%).
     - Set `uio_in[3:0]` to `1101` to set part of `d2 = 13`.

5. **Monitor Outputs**
   - **Procedure:**
     - Use an oscilloscope or logic analyzer to check `uo_out` signals.
     - Verify the PWM signals on `uo_out[0]` (PMOS1), `uo_out[1]` (NMOS2), `uo_out[2]` (PMOS2), `uo_out[3]` (NMOS1), and the clock signal on `uo_out[4]`.
     - Confirm the PWMs duty cycle matches the expected 20%.

External hardware

There is no need of external hardware.

Pinout

#	Input	Output	Bidirectional
0	d1[0]	PMOS1	d2[2]
1	d1[1]	NMOS2	d2[3]
2	d1[2]	PMOS2	d2[4]
3	d1[3]	NMOS1	d2[5]
4	d1[4]	clk_in	CLK_SR
5	d1[5]		Data_SR
6	d2[0]		CLK_EXT
7	d2[1]		

4bit_CPU_td4 [128]

- Author: Ko Kosugi
- Description: 4bit_CPU
- GitHub repository
- HDL project
- Mux address: 128
- Extra docs
- Clock: 10 Hz

How it works

4-bit CPU with 4 input ports and 4 output ports.

How to test

Set the inputs and check the outputs.

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Pinout

#	Input	Output	Bidirectional
0	data_0	addres_0	IO_0
1	data_1	addres_1	IO_1
2	data_2	addres_2	IO_2
3	data_3	addres_3	IO_3
4	data_4	cf	0
5	data_5	ALU_to_reg_0	ALU_to_reg_3
6	data_6	ALU_to_reg_1	select_0
7	data_7	ALU_to_reg_2	select_1

DVD Screensaver with Tiny Tapeout Logo (Tiny VGA) [130]

- Author: Uri Shaked
- Description: Tiny Tapeout Logo bouncing around the screen (640x480 VGA)
- GitHub repository
- HDL project
- Mux address: 130
- Extra docs
- Clock: 25175000 Hz

How it works

Displays a bouncing Tiny Tapeout logo on the screen.

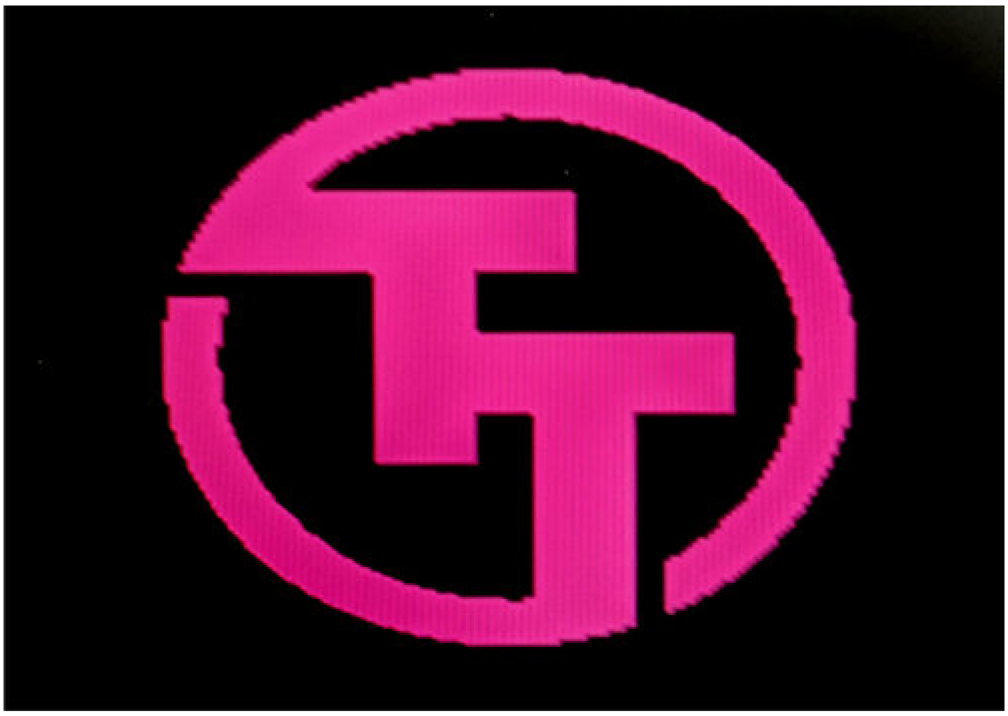


Figure 9: Tiny Tapeout screensaver

How to test

Connect to a VGA monitor. Set the following inputs to configure the design:

- `tile` (ui_in[0]) to repeat the logo and tile it across the screen,
- `color` (ui_in[1]) to enable color output.

External hardware

TinyVGA PMOD

Pinout

#	Input	Output	Bidirectional
0	tile	R1	
1	color	G1	
2		B1	
3		VSync	
4		R0	
5		G0	
6		B0	
7		HSync	

Ripple Carry Adder 8 bit [132]

- Author: Jason Kaufmann
- Description: Adds two 8 bit numbers together
- GitHub repository
- HDL project
- Mux address: 132
- Extra docs
- Clock: 0 Hz

How it works

The "Ripple Carry Adder 8 bit" project adds two 8-bit numbers together using a ripple carry adder (RCA) architecture. This is implemented using a series of full adders, each of which adds two bits along with a carry-in from the previous less significant bit. The result is an 8-bit sum and a carry-out bit for the most significant bit.

The project consists of the following modules:

1. `halfadder`: A simple module that computes the sum and carry-out of two 1-bit inputs.
2. `fulladder`: A module that uses two `halfadder` instances to add three 1-bit inputs (two bits and a carry-in) and produces a sum and carry-out.
3. `rca8`: An 8-bit ripple carry adder module that cascades eight `fulladder` instances to add two 8-bit numbers.
4. `tt_um_example`: The top module that connects the inputs and outputs to the `rca8` module, performing the addition operation and providing the result through the output pins.

The `tt_um_example` module takes two 8-bit inputs (`ui_in` and `uio_in`), adds them using the `rca8` module, and outputs the 8-bit sum (`uo_out`).

How to test

To test the "Ripple Carry Adder 8 bit" project:

1. Provide two 8-bit numbers as inputs via the `ui_in` and `uio_in` pins.
2. Observe the 8-bit sum output on the `uo_out` pins.
3. Ensure that all connections are made correctly as per the pinout configuration.

For example, if you input the binary numbers 00001101 (13 in decimal) and 00000111 (7 in decimal) on `ui_in` and `uio_in` respectively, the output on `uo_out` should be 00010010 (20 in decimal).

External hardware

No external hardware is required for this project. It operates purely based on the digital inputs provided and generates a digital output. However, for testing and demonstration purposes, you may use input switches and output LEDs or a similar setup to visualize the input and output binary numbers.

Pinout

#	Input	Output	Bidirectional
0	A0	OUT0	B0
1	A1	OUT1	B1
2	A2	OUT2	B2
3	A3	OUT3	B3
4	A4	OUT4	B4
5	A5	OUT5	B5
6	A6	OUT6	B6
7	A7	OUT7	B7

DuckCPU [134]

- Author: Alex Studer
- Description: Small System-on-Chip based around a custom 8-bit CPU.
- GitHub repository
- HDL project
- Mux address: 134
- Extra docs
- Clock: 50000000 Hz

How it works

This is a small System-of-Chip (SoC) built around the DuckCPU, an 8-bit CPU that implements a custom architecture based on the Zilog Z80 and Sharp LR35902. It was designed primarily for learning purposes.

The following peripherals are provided:

- RSPI (reserved SPI, for flash/RAM acccess)
- UART0
- SPI0

More detailed documentation will be added to the project page after the tapeout.

How to test

Connect a SPI flash IC to the RSPI pins, with rspi_flash_ce_n as its chip enable. Similarly, connect a SPI RAM IC, with rspi_ram_ce_n as its chip enable.

Pull the bootsel pin low and reset the chip. This puts it into its bootloader mode. You can then use ducktool to download code onto the flash/RAM and reset the CPU.

External hardware

Minimum requirement:

- SPI flash IC (tested with ISSI IS25LP020E)
- SPI RAM IC (tested with ISSI IS66WVS1M8BLL)
- USB to UART circuit (included on demo board)

While we do follow the standard Tiny Tapeout pinout for UART, we do not for SPI. This is because we want to save the bidirectional pins for use by the GPIO peripheral.

Pinout

#	Input	Output	Bidirectional
0		rspi_clk	gpio0_data[0]
1		rspi_mosi	gpio0_data[1]
2		rspi_flash_ce_n	gpio0_data[2]
3		rspi_ram_ce_n	gpio0_data[3]
4	bootsel	uart0_tx	gpio0_data[4]
5	spi0_miso	spi0_clk	gpio0_data[5]
6	rspi_miso	spi0_mosi	gpio0_data[6]
7	uart0_rx	spi0_ce_n	gpio0_data[7]

John Pong The Second [136]

- Author: Sophia Rustfield (Representing HSWAW)
- Description: a hyper simple pong game with the polish pope taking the role of the ball outputted over vga
- GitHub repository
- HDL project
- Mux address: 136
- Extra docs
- Clock: 25175000 Hz

How it works

it outputs VGA, using all of the dedicated output pins and three of the bidirectional pins, and takes player input on 5 of the dedicated input pins, it works by having a counter that counts clock cycles since boot, and outputs one pixel per clock cycle, on the first clock cycle of vsync, all game logic happens.

How to test

you're gonna need to play the game, and hook it up to a monitor with a DAC

External hardware

- a circuit that can convert the digital output to analog VGA
- a monitor that supports VGA
- a circuit for some buttons for the player input there are no part numbers here because we built everything outselves except for the monitor, and monitors are ubiquotus

Pinout

#	Input	Output	Bidirectional
0	move paddle up	red channel bit 0	blue channel bit 0
1	move paddle down	red channel bit 1	blue channel bit 1
2	move player 2 paddle up	red channel bit 2	blue channel bit 2
3	move player 2 paddle down	green channel bit 0	
4	high voltage to activate player 2, low for ai	green channel bit 1	

#	Input	Output	Bidirectional
5		green channel bit 2	
6		horizontal sync signal	
7		vertical sync signal	

Four NIST SP 800-22 tests implementation [138]

- Author: Maximiliam Luppe
- Description: Implementation of the first four NIST statistic tests
- GitHub repository
- HDL project
- Mux address: 138
- Extra docs
- Clock: 0 Hz

How it works

This project implements a LFSR where the D-type FFs (DFFs) are replaceded by configurable delay lines.

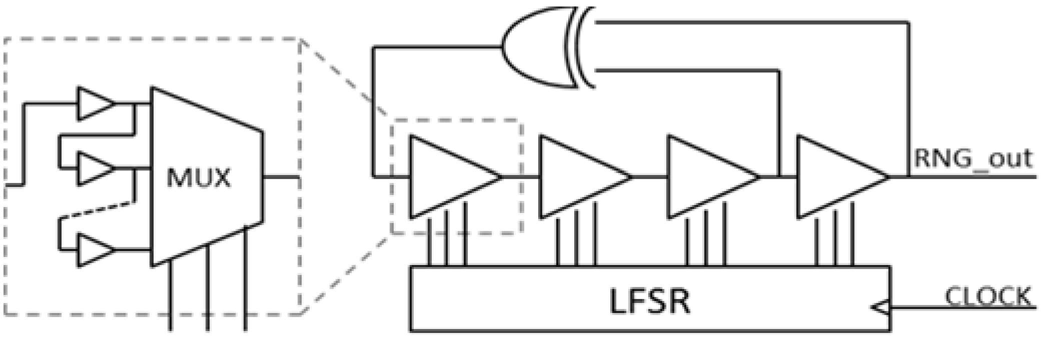


Figure 10: ALFSR diagram

The configurable delay lines are implement using multiplexers and buffers. The buffers generate a delay line path and the multiplexer select differents paths according to its selection control signals. The selection control signals are generated by a conventional LFSR.

It's expected that this circuit act as either as a convencional LFSR, or as an oscillator, or as a chaotic oscilator, according to the path generated by each delay line.

To verify its functionality, four statistical tests from the NIST statistical suite are also implemented:

1. The Frequency (Monobit) Test
2. Frequency Test within a Block
3. The Runs Test
4. Tests for the Longest-Run-of-Ones in a Block

They are choosen because they require at least 100 bits sequence to test.

The implementations are based on the following works:

- L. B. Carreira, P. Danielson, A. A. Rahimi, M. Luppe and S. Gupta, "Low-Latency Reconfigurable Entropy Digital True Random Number Generator With Bias Detection and Correction," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 67, no. 5, pp. 1562-1575, May 2020, doi: 10.1109/TCSI.2019.2960694
- V. B. Suresh, D. Antonioli and W. P. Burleson, "On-chip lightweight implementation of reduced NIST randomness test suite," 2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), Austin, TX, USA, 2013, pp. 93-98, doi: 10.1109/HST.2013.6581572.
- F. Veljković, V. Rožić and I. Verbauwhede, "Low-cost implementations of on-the-fly tests for random number generators," 2012 Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, Germany, 2012, pp. 959-964, doi: 10.1109/DATE.2012.6176635.

How to use

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Pinout

#	Input	Output	Bidirectional
0	LFSR Configurator clock	NIST 01 test output	ALFSR *analog* output 0
1	ALFSR reset	NIST 02 test output	ALFSR *analog* output 1
2	NIST random bits input	NIST 03 test output	ALFSR *analog* output 2
3	Operation mode	NIST 04 test output	ALFSR *analog* output 3
4		NIST Global error output	
5		LFSR Configurator output	
6		ALFSR *digitalized* output 3	
7		ALFSR *analog* output 3	

Subleq CPU with FRAM and UART [140]

- Author: Philip Mohr
- Description: Stupid slow Subleq CPU using an external SPI FRAM
- GitHub repository
- HDL project
- Mux address: 140
- Extra docs
- Clock: 10000000 Hz

How it works

Subleq refers to a kind of OISC where the one instruction is "SUBtract and branch if Less-than or EQual to zero", conventionally abbreviated to subleq.

Subleq is a simple one instruction language. Each Subleq instruction has 3 memory address operands. Since Subleq has only one instruction, the opcode itself is conventionally omitted, so each instruction is three addresses long.

You can easily output data in the C Code. Look in the examples how it's done. In Subleq it's implemented like this: If B is -1 (negative unity), then the number contained in the address given by A is interpreted as a character and written to the machine's output. C is unused.

The Baud is 115200, when a 10MHz clock is used.

There is a C Compiler for Subleq. It only supports a typeless simplified subset of C, but most simple things can be done with it. It is written in C++ and doesn't depend on libraries or external tools. The original website for it is offline, but infos about the compiler still exist on the Esolang Wiki (->Higher Subleq) with a Download Link at the web archive on the bottom of the page. But i will also include the compiler code in the repo.

How to test

Use an Arduino compatible Microcontroller with at least 16KB Memory. Convert the output from the compiler with the given converter to an array, import it to the given Arduino Sketch, flash it and run it. The cpu is designed for 10MHz, but it needs to be tested how fast or slow it can go. The SPI clock is half the input clock. Keep in mind that when changing the clock speed the uart baud will also change. I will make a kinda userfriendly toolchain and publish it on my github. It will include a Subleq to hex converter (for Quartus etc.), a Subleq to C string converter, an Arduino sketch for flashing to FRAM and example C/Subleq codes.

External hardware

SPI FRAM

As RAM an 8KB FRAM is used with a 20MHZ SPI interface (MB85RS64V). The advantage of SPI RAM in comparison to SPI Flash is the access time. Every byte can be accessed directly without having bank switching, which leads to different access when randomly accessing data. But in comparison to usual RAM, FRAM is non-volatile. So it has the advantages of Flash and RAM memory (but costs much more).

Pinout

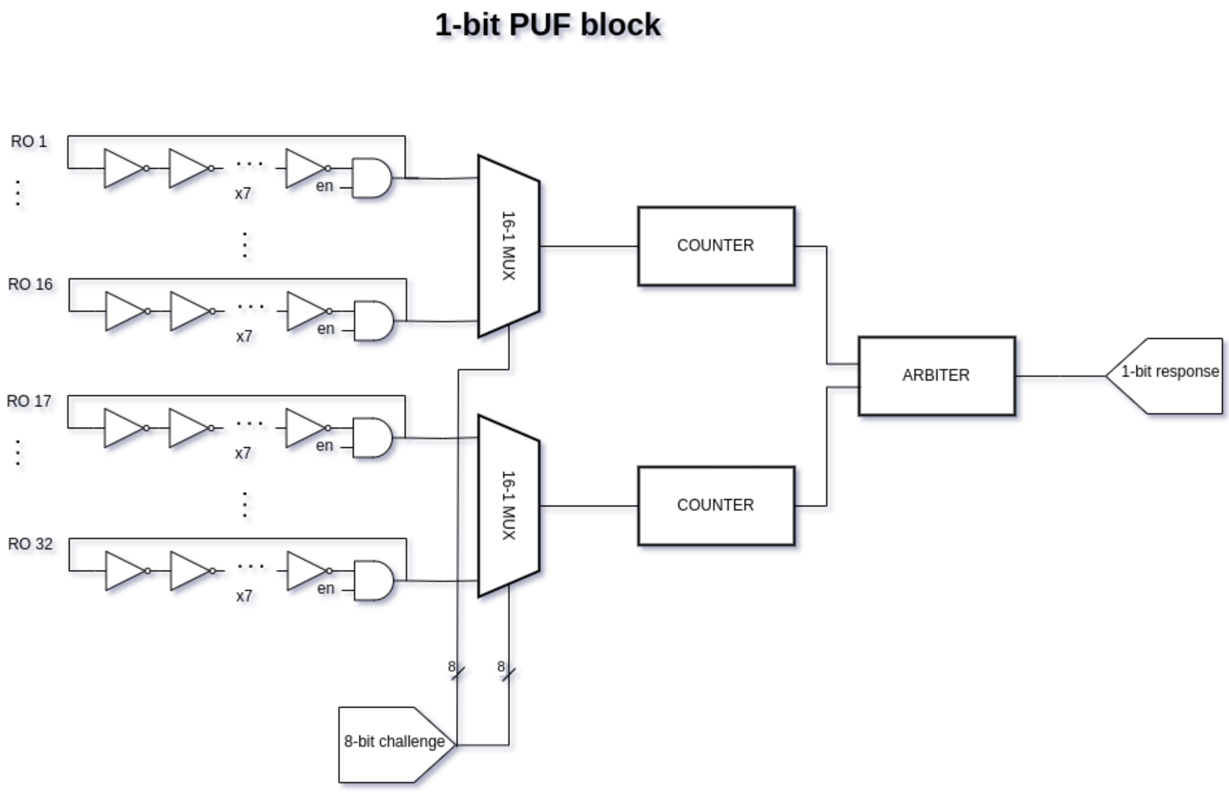
#	Input	Output	Bidirectional
0	in_miso	out_mosi	data_0
1		out_sck	data_1
2		out_cs	data_2
3		tx	data_3
4		tx_credits	data_4
5			data_5
6			data_6
7			data_7

RO-based Physically Unclonable Function (PUF) [142]

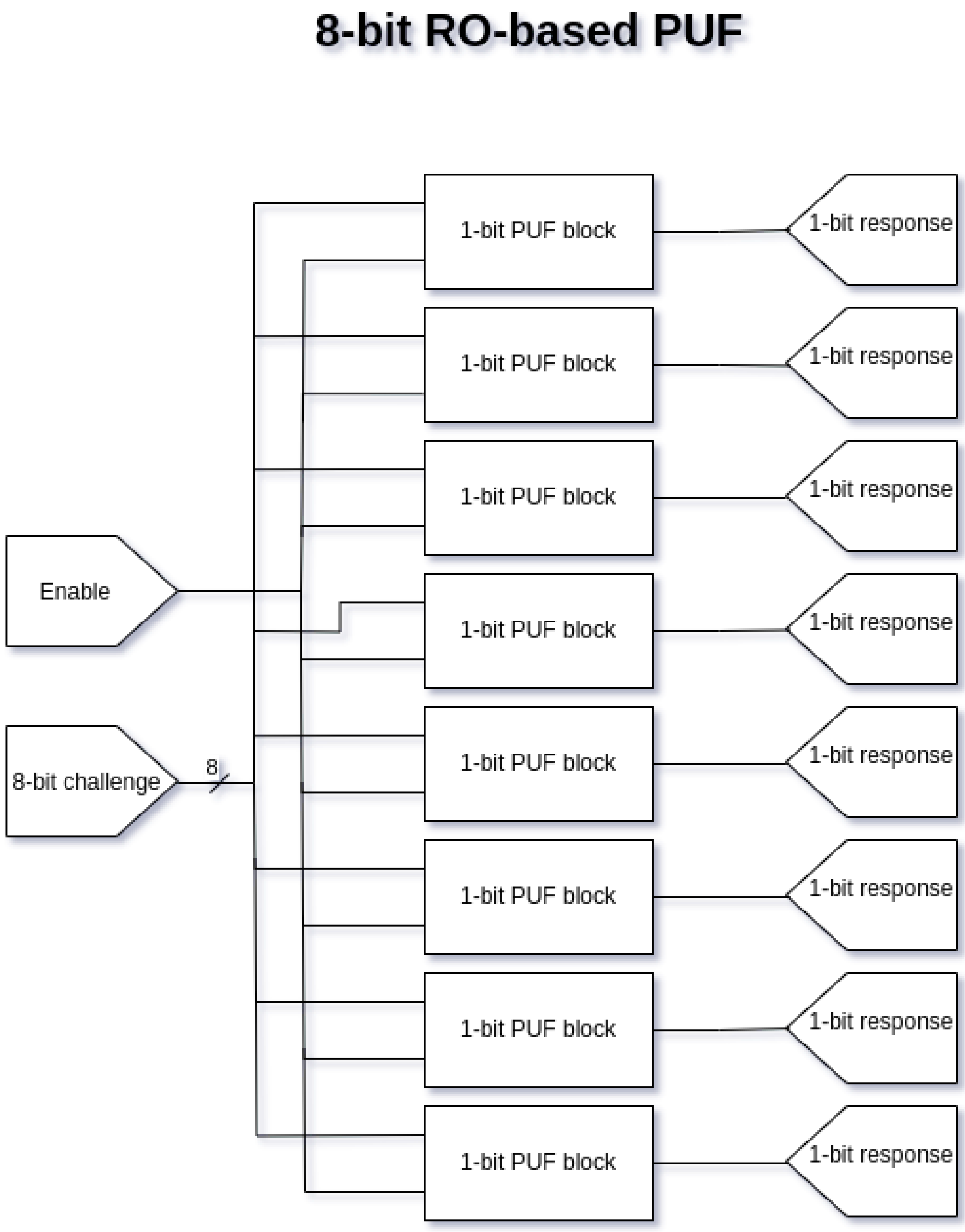
- Author: Pablo Aravena
- Description: Implementation of a Ring Oscillator-based Physically Unclonable Function (PUF) in Sky130, with 8 bits of Challenge-Response Pairs (CRP)
- GitHub repository
- HDL project
- Mux address: 142
- Extra docs
- Clock: 10000000 Hz

How it works

A physical unclonable function (PUF) is a hardware security primitive that maps an input (called a challenge) to an output (called a response) in a similar fashion to a hash function. The goal of a PUF can be many: uniquely identifying an integrated circuit (IC) from another while still keeping deterministic outputs for the same IC, using a set of challenge-response pairs (CRP); generating random-enough nonces; or even authenticating an IC (stronger version of identification). In order to evaluate its performance for those goals, relevant metrics such as **uniqueness**, **reliability**, **uniformity** and **entropy** of CRPs over many PUF ICs are commonly employed. The PUF implementation for this project uses many identical, 7-inverter ring oscillators (RO) which introduce randomness or variation in their operating frequencies at the time of the fabrication process itself.



In this case, an 8-bit parallel architecture for each bit of a CR P is adopted. The 8-bit challenge is shared along 8 independent blocks in order to derive only 1-bit of the response per block. One PUF block contains 32 ROs, where one RO is selected between the top half of the ROs over a challenge-dependent (4 MSB) 16-bit mux, while other RO is selected from the remaining bottom half over the second chall-dependent (4 LSB) 16-bit mux. Then, both muxes connect directly to 1 counter of its own. Both of the counters then race each other until a given threshold (65535 in this case) is reached, and an arbiter module that's connected to both counters declares the winner in a 1-bit response.



How to test

Start by feeding an 8-bit challenge to the 8 input pins in the Tiny Tapeout board before enabling or selecting this module. This will ensure that the appropiate input data is sampled while initiating the ring oscillators. After some milliseconds, a result should appear at the 8 output pins as individual response bits. To generate another CRP without powering-off the board, first start sending the new challenge on the input pins continously. After it, drive the reset pin high and then immediately low in order to sample the newest input values. Note that the output or responses **may** change as the device reaches operational temperature, after which they will become consistent. This instability is due to the ROs' sensitivity to temperature, which will slightly change frequency operation.

EXTRA: In order to accurately estimate entropy, uniqueness, reliability and uniformity for this PUF architecture in Sky130, **a lot** of measurements must be taken to ensure unbiased representation of data and validity of the metric's results. That's why the module's author needs you, a Tiny Tapeout board owner, to join in this open-source effort of characterizing this manufacturing process and PUF architecture. The required data to collect would be the set of all possible 8-bit Challenge-Response Pairs (CRP) generated by your specific device. For more information on how to help and where to register your CRPs, check the GitHub repo related to this project. The plan is to make this data and the metrics transparent and public, updating them in real time.

External hardware

No external hardware is required for this project.

Pinout

#	Input	Output	Bidirectional
0	challenge bit 1	response bit 1	
1	challenge bit 2	response bit 2	
2	challenge bit 3	response bit 3	
3	challenge bit 4	response bit 4	
4	challenge bit 5	response bit 5	
5	challenge bit 6	response bit 6	
6	challenge bit 7	response bit 7	
7	challenge bit 8	response bit 8	

FastMagnitudeComparator [192]

- Author: Daniel Burke
- Description: Digital neuron threshold detector
- GitHub repository
- HDL project
- Mux address: 192
- Extra docs
- Clock: 0 Hz

How it works

For neuron thereshold evaluation in digital approaches, a fast magnitude determination is often necessary.

This component is based upon well-documented Clint Cole (Digilent) bit-sliced expandable, structural code re-expressed in AND-INV format to target optimized ABC9 AIG graph synthesis in OpenLane. https://www.realdigital.org/doc/a39d855f71772426c968c0151112

It is intentionally unclocked for measurements, and can be easily modified as a windowing-comparator for inference field requirements.

The fast magnitude comparitor is second of a series of common, scaleable library of elements intended to support CMOS digital neuron biomemetic building blocks, the first being a scaleable fast accumulator for vector evaluation and integration based upon a generated Sklansky adder/subtractor.

Each component is intended to be fashioned in structural Verilog for future optimization, scale well so as to support varying bit-width large vector resolutions, and whenever possible described in AND-INV form to leverage the OpenLane ABC9 logic optimizer which uses AIG graphs.

How to test

The user will supply two numbers of appropriate length (A, B, each 8 bits in this case) with returning signals indicating "A less than B", "A equal to B", or "A greater than B" as LT_out, EQ_out, and GT_out respectively.

External hardware

Means to supply appropiate width words (in this instance one 8b byte) and read back GT, EQ, LT signals.

Pinout

#	Input	Output	Bidirectional
0	ui_in[0]	ui_out[0]	uio[0]
1	ui_in[1]	ui_out[1]	uio[1]
2	ui_in[2]	ui_out[2]	uio[2]
3	ui_in[3]	ui_out[3]	uio[3]
4	ui_in[4]	ui_out[4]	uio[4]
5	ui_in[5]	ui_out[5]	uio[5]
6	ui_in[6]	ui_out[6]	uio[6]
7	ui_in[7]	ui_out[7]	uio[7]

8 bit PRNG [194]

- Author: Jorge Garcia Martinez
- Description: 8 bit PRNG based on Xorshift
- GitHub repository
- HDL project
- Mux address: 194
- Extra docs
- Clock: 0 Hz

How it works

PRNG: Xorshift Overview A Xorshift is a type of pseudorandom number generator (PRNG) that utilizes shift and XOR operations to generate a sequence of pseudorandom numbers. Introduced by George Marsaglia in 2003, the Xorshift is renowned for its speed and simplicity of implementation in both hardware and software platforms.

Theoretical Basis for an 8-bit Xorshift

State Register The Xorshift algorithm employs a state register that, in the case of an 8-bit Xorshift, consists of an 8-bit integer. This register maintains the current state of the PRNG, which is updated with each iteration to produce the subsequent pseudorandom number.

Fundamental Operations

- **XOR**: The XOR ( ) operation is essential for bit scrambling in the state. It takes two bits and returns 1 if the bits are different and 0 if they are the same.
- **Shift**: Shift operations (either left or right) move bits by a specified number of positions, introducing zeros into the vacated positions.

Sequence of Operations At each step, the current state is altered using a series of XOR and shift operations. For instance, an 8-bit Xorshift might proceed as follows:

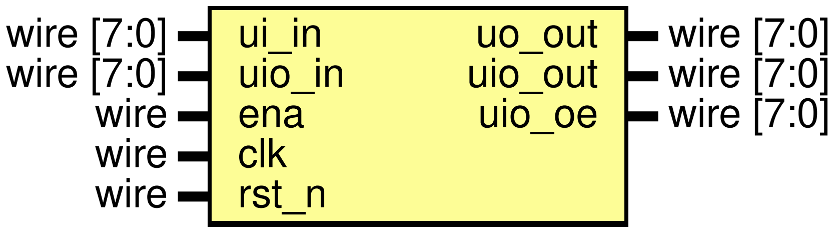
1. `x = x   (x &amp;lt;&amp;lt; a)`
2. `x = x   (x &amp;gt;&amp;gt; b)`
3. `x = x   (x &amp;lt;&amp;lt; c)`

Here, `x` denotes the current state, while `&amp;lt;&amp;lt;` and `&amp;gt;&amp;gt;` signify left and right shifts, respectively. The constants `a`, `b`, and `c` determine the magnitude of these shifts.

Period The period of a Xorshift depends on both the number of bits in the state register and the choice of shift parameters `a`, `b`, and `c`. For an 8-bit register, the maximum achievable period is `2^8 - 1 = 255`, assuming the parameters are selected wisely to prevent short cycling.

How to test

To test this random number generator you must place an 8-bit seed as input (ui_in) and after 2 cycles of latency you will obtain a random number at each clock cycle at the out-



put (uo_out).

For more information about rtl: <tt_um_jorga20j_prng.md>

Pinout

#	Input	Output	Bidirectional
0	Seed bit 0	Pseudo-Random number bit 0	
1	Seed bit 1	Pseudo-Random number bit 1	
2	Seed bit 2	Pseudo-Random number bit 2	
3	Seed bit 3	Pseudo-Random number bit 3	
4	Seed bit 4	Pseudo-Random number bit 4	
5	Seed bit 5	Pseudo-Random number bit 5	
6	Seed bit 6	Pseudo-Random number bit 6	
7	Seed bit 7	Pseudo-Random number bit 7	

8-bit DEM R2R DAC [196]

- Author: Eric Fogleman
- Description: 8-bit segmented mismatch-shaping R2R DAC
- GitHub repository
- HDL project
- Mux address: 196
- Extra docs
- Clock: 10000000 Hz

Operation

This design implements a linear 8-bit DAC suitable for dc and low-frequency inputs. The encoder quantizes the 8-bit input to a 55-level signal. An analog voltage is produced by connecting the encoder's outputs to a modified R-2R ladder on the PCB (see External Hardware). Quantization noise is shaped in the frequency domain with a 1st order highpass shaped. The residual high frequency noise is suppressed using an analog lowpass filter. With a clock frequency of 6.144 MHz and a lowpass filter corner of 24 kHz, the oversampling ratio (OSR) is 256.

This encoder provides quantization noise shaping similar to that of a multibit delta-sigma modulator, but it is a purely feedforward network with no quantization error feedback. The theory behind this encoder is described in: A. Fishov, E. Fogleman, E. Siragusa, I. Galton, "Segmented Mismatch-Shaping D/A Conversion", IEEE International Symposium on Circuits and Systems (ISCAS), 2002

This design is a revision to that on TT06 that implements element mismatch shaping as well as quantization noise shaping.

External hardware

Ideally, this encoder would be buffered through a clean analog supply and retimed at the output with a clean clock to align the bit transitions. However, reasonable performance should be possible driving the resistor ladder directly from the encoder through the IO supply. When used n this way, the IO supply acts as the DAC's reference voltage.

DAC input data is provided through `ui_in[7:0]`, and the encoder updates using the project clock. Clock frequencies in the range of 1-10 MHz are reasonable. Higher clock frequency increases the OSR. The encoder output is `uo_out[7:0]`, and it can be reconstructed by summing the bits with the following weights:

```
v_out = 0.5*vdd_io*(1 + (2**-4)*(8*uo_out[7]+uo_out[6]) + 4*(uo_out[5]+uo
(uo_out[1]+uo_out[0])
 - 15)
```

The DAC's output swing ranges from `0.25*vdd_io` to `0.75*vdd_io`, where `vdd_io` is the IO supply voltage.

An external resistor ladder is required to create the analog output voltage, and a capacitor is required to filter high-frequency noise. The termination resistors are placed at the ends of the ladder to ensure that each section has nominally identical load resistance.

The resistor ladder shown below sums the outputs with this weighting. Any output network that can create this weighting will work.

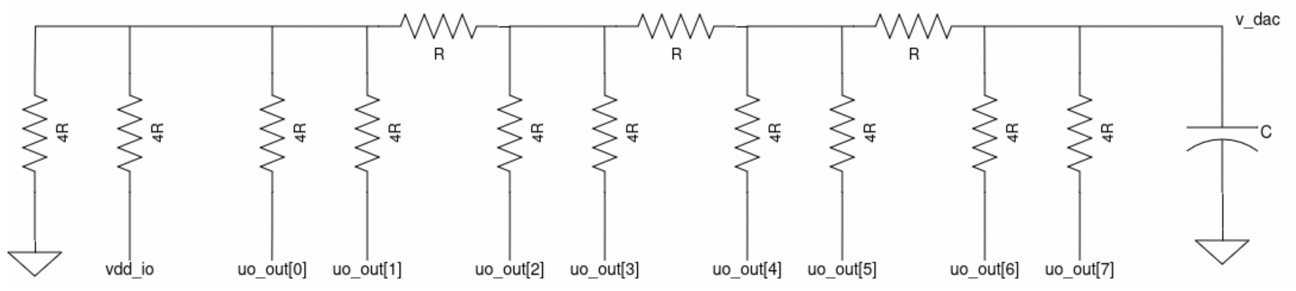


Figure 11: DAC resistor network

The suggested unit R value is 10 kOhm, thus all elements marked R should be 10 kOhm, and all 4R elements should be 40 kOhm. This gives an equivalent output resistance at **v_dac** of 10 kOhm. A 680 pF output capacitor provides a 23 kHz lowpass corner. With this choice of R, each IO driver will sink/source a maximum of 55 uA with `vdd_io` at 3.3 V.

Precise resistor matching is not required to obtain 8-bit linearity. Resistor mismatch error appears as first-order shaped noise in the output and can be removed by analog lowpass filtering.

Testing

The DAC is free-running off the project clock, and inputs appear at the output immediately after clock synchronization. A simple dc test can be performed using the input DIP switches and the resistor ladder.

The encoder has four modes of operation determined by `uio_in[1:0]`:

- 3: 1st order shaping with dither

- 2: randomization (flat spectral shaping)
- 1: 1st order shaping, no dither
- 0: static encoding (no shaping)

The dc value DAC output can be measured with a DMM. The DAC's output swing ranges from `0.25*vdd_io` to `0.75*vdd_io`, where `vdd_io` is the IO supply voltage. The DAC's LSB voltage is `vdd_io/32`.

Pinout

#	Input	Output	Bidirectional
0	d_in[0]	y8[0]	en_enc
1	d_in[1]	y8[1]	en_dith
2	d_in[2]	y16[0]	y1[0]
3	d_in[3]	y16[1]	y1[1]
4	d_in[4]	y32[0]	y2[0]
5	d_in[5]	y32[1]	y2[1]
6	d_in[6]	y64[0]	y4[0]
7	d_in[7]	y64[1]	y4[1]

FP-8 MAC Module [198]

- Author: Wilfred Kisku
- Description: A simple MAC implementation using Chisel for two FP8 numbers, incorporates both addition and multiplication for FP8 numbers. Partial register at 0x00 which is the start value. Result is valid after 3 clock cycles.
- GitHub repository
- HDL project
- Mux address: 198
- Extra docs
- Clock: 0 Hz

Design

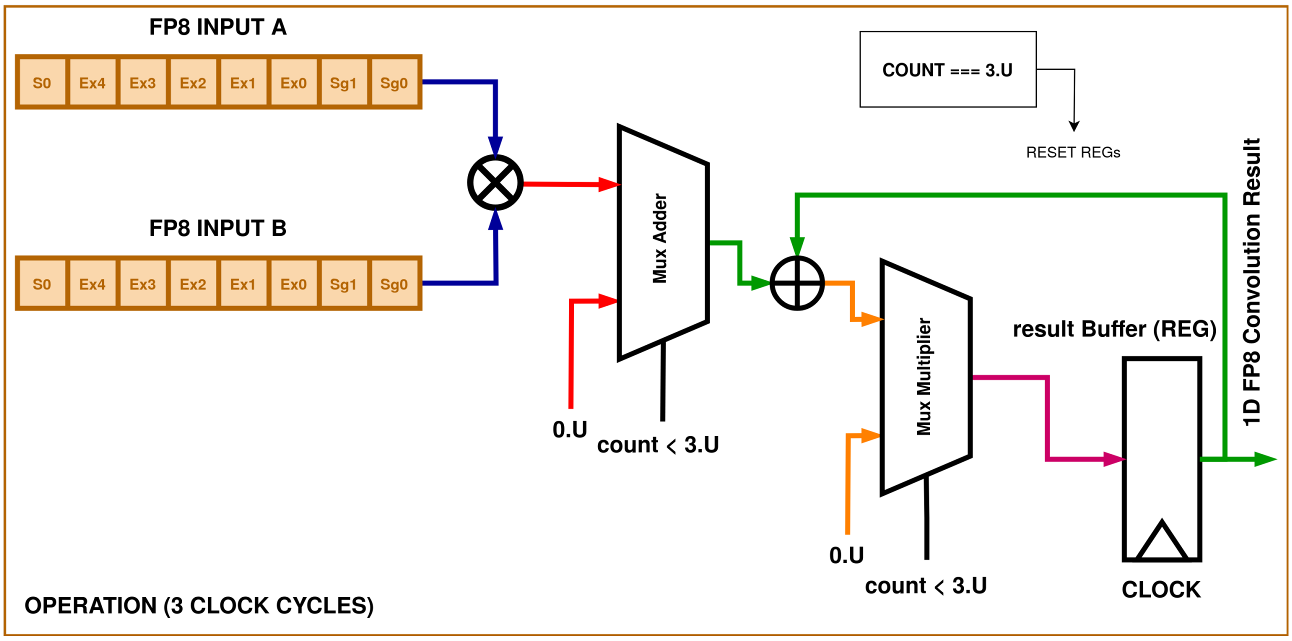


Figure 12: block diagram

The digital block comprises of two sub blocks and a top module that incorporates a MAC (multiply-and-accumulate) operation.

This IEEE 754 format for a 8-bit FP precision for addition and multiplication is implemented. The operations incorporate intricacies and corner cases for handling +/- inf, NaN, Zeros and a full 8-bit precision range.

Details of FP8	Binary Formats
Exponent Bias	15
Infinites	$S.11111.00_2$

Details of FP8	Binary Formats
NaN	$S.11111.XX_2$
Zero	$S.00000.00$
Max Normal	$S.11110.11_2$
Min Normal	$S.00001.00_2$
Max Subnormal	$S.00000.11_2$
Min Subnormal	$S.00000.01_2$

Though this format is highly limited in precision and range compared to standard floating-point formats like IEEE 754 single-precision (32-bit) or double-precision (64-bit). It would likely be used in specialized scenarios where memory is at a premium or where precision beyond this level is unnecessary. The MAC operations can be verified on the 8-bit FP data, with both addition and multiplication.

The following are the highlights and the pin descriptions:

- There are two inputs that take in FP8 data in the format S.EEEEE.SS
- The first stage of operation is a multiplication between the two inputs.
- The second stage is the addition on the multiplication result and the partial residing in the output register.
- At each clock cycle the partial result in latched to the output register, taking 3 cycles for giving the result.

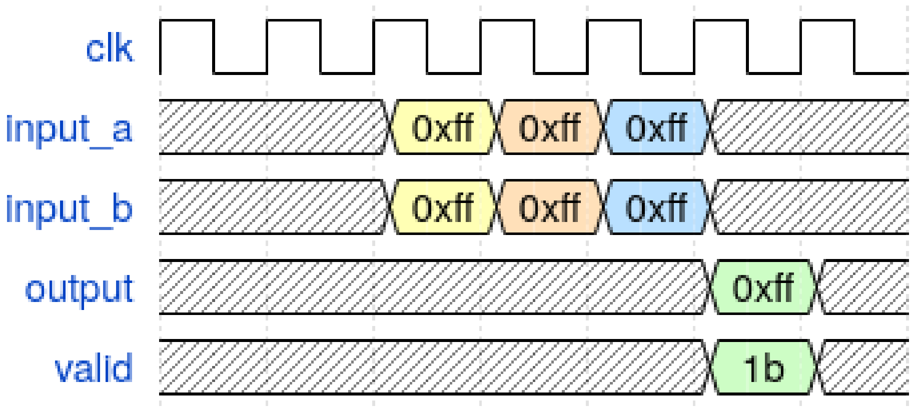


Figure 13: timing diagram

How to test

Run `make` in the `/test` directory.

External hardware

The design is self sustaining sequential, requiring only an output buffer to store the current partial products and the MAC resultant after 3 clock cycles. This is supposed to be a preliminary result block that can make up systolic arrays, hardware accelerators and many other.

Pinout

#	Input	Output	Bidirectional
0	FP8 input A pin 0.	FP8 output pin 0.	FP8 input B pin 0.
1	FP8 input A pin 1.	FP8 output pin 1.	FP8 input B pin 1.
2	FP8 input A pin 2.	FP8 output pin 2.	FP8 input B pin 2.
3	FP8 input A pin 3.	FP8 output pin 3.	FP8 input B pin 3.
4	FP8 input A pin 4.	FP8 output pin 4.	FP8 input B pin 4.
5	FP8 input A pin 5.	FP8 output pin 5.	FP8 input B pin 5.
6	FP8 input A pin 6.	FP8 output pin 6.	FP8 input B pin 6.
7	FP8 input A pin 7.	FP8 output pin 7.	FP8 input B pin 7.

SerDes [200]

- Author: Mahaa Santeep G
- Description: The project implements a Serializer and Deserializer (SerDes) module for converting data between parallel and serial formats in digital systems.
- GitHub repository
- HDL project
- Mux address: 200
- Extra docs
- Clock: 100 Hz

How it works

The SerDes project operates by efficiently converting data between parallel and serial formats, enabling seamless communication between systems with different data transfer requirements. Here's how it works:

1. Serialization Path:

   - Parallel data is received through the `data_8b_in` input.
   - Upon the rising edge of the clock (`clk`) and when the data enable signal (`data_en`) is active, the input data is latched to ensure synchronization.
   - The latched parallel data undergoes encoding into a 10-bit format using an 8b/10b encoding scheme. This encoding scheme provides sufficient data integrity and clock recovery capabilities for serial transmission.
   - The encoded data is further latched and transmitted serially through the `ser_out` output when the serializer enable signal (`ser_en`) is asserted.

2. Deserialization Path:

   - Serial data is received through the `ser_in` input.
   - The serial data is shifted in parallel using a Serial-In-Parallel-Out (SIPO) shift register when the parallel enable signal (`par_en`) is active.
   - The parallel data is latched upon the rising edge of the clock (`clk`) and when the data enable signal (`data_en`) is active to ensure synchronization.
   - The latched parallel data undergoes decoding from the 10-bit format back to an 8-bit format, effectively reversing the serialization process.
   - The decoded parallel data is output through the `data_out` output.

3. Synchronization and Reset:

   - Synchronous reset (`rst`) ensures proper initialization of internal state variables and output data.

- Latch modules are employed throughout the process to synchronize data with the clock and ensure stable output.

4. Integration and Flexibility:

   - The project provides a modular and flexible design, facilitating easy integration into larger systems.
   - Users can adjust control signals (`data_en`, `ser_en`, `par_en`) to tailor the operation to specific requirements.
   - Proper clock domain crossing techniques are employed to ensure reliable data transfer between parallel and serial domains.

Overall, the SerDes project offers an efficient and reliable solution for bidirectional data conversion, enabling seamless communication between systems with disparate data transfer interfaces.

How to test

The provided RTL project implements a Serializer and Deserializer (SerDes) module capable of converting data between parallel and serial formats. The module, encapsulated within `serdes_top`, features inputs for clock signal (`clk`), active high reset (`rst`), data enable signal (`data_en`), and control signals for serialization (`ser_en`) and deserialization (`par_en`). Parallel data is input through `data_8b_in`, undergoing latching, encoding into 10-bit data, and serial output (`ser_out`) generation during serialization, while serial input (`ser_in`) undergoes deserialization, parallelization, and output (`data_out`) generation. Synchronous reset ensures proper initialization, and latch modules synchronize data. The project facilitates seamless integration into larger systems, offering flexibility and reliability for applications requiring efficient data transmission across parallel and serial interfaces.

External hardware

There are No External Hardware Used in the Project

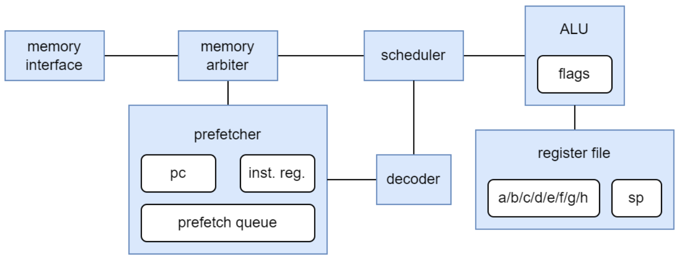
Pinout

#	Input	Output	Bidirectional
0	data_8b_in[0]	data_out[0]	ser_in
1	data_8b_in[1]	data_out[1]	ser_out
2	data_8b_in[2]	data_out[2]	data_en

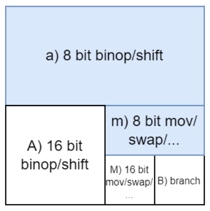
#	Input	Output	Bidirectional
3	data_8b_in[3]	data_out[3]	par_en
4	data_8b_in[4]	data_out[4]	ser_en
5	data_8b_in[5]	data_out[5]	
6	data_8b_in[6]	data_out[6]	
7	data_8b_in[7]	data_out[7]	

Basilisc-2816 v0.1b CPU [202]

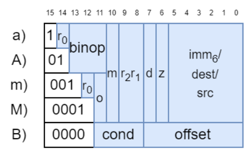
- Author: Toivo Henningsson
- Description: Small 2-bit serial 8/16 bit CPU
- GitHub repository
- HDL project
- Mux address: 202
- Extra docs
- Clock: 50000000 Hz



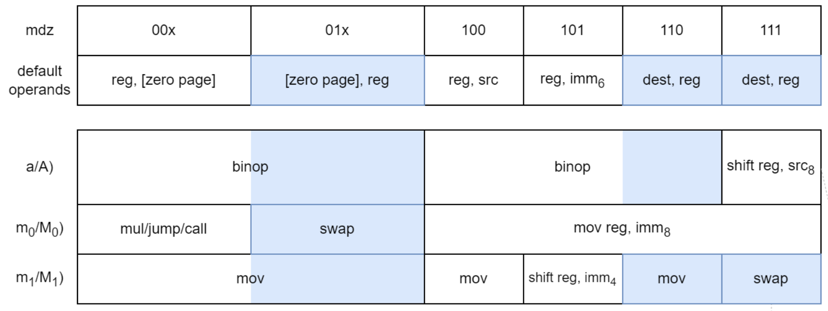
Major parts and interconnections of the processor.



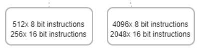
Major division of the instruction encoding space based on the top 4 bits.
8 bit instructions use 3 bits to choose between the 8 registers:a/b/c/d/e/f/g/h,
16 bit instructions use 2 bits to choose between the 4 registers ba/dc/fe/hg.

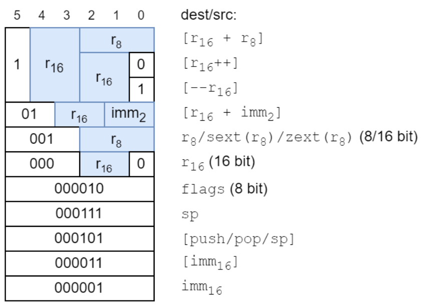


Bit fields of the major instruction forms.
The o bit selects between m_0/m_1 or M_0/M_1.



Further division of the a/A/m/M) encoding spaces into instruction forms.
Shaded instructions write to memory.





Encoding of the dest/src field.

Overview

Basilisc-2816 v0.1 is a small 2-bit serial 2/8/16 bit processor that fits into one Tiny Tapeout tile. It has been designed around the constraints of

- small area,
- 4 pin serial memory interface to a RAM emulator implemented in an RP2040 microcontroller (which can be supported by the RP2040 microcontroller on the Tiny Tapeout 7 Demo Board),
- to be suitable to be included in in the next version of the AnemoneGrafx-8 retro console https://github.com/toivoh/tt06-retro-console, which motivates the other constraints.

Features:

- 2-bit serial execution:

- ALU results etc are calculated at 2 bits/cycle
- 2-bit-serial register file with two read/write ports
- Addresses and data are sent to/from memory at 2 bits/cycle
  - The processor starts to operate on each bit of incoming read data as it arrives
- Saves area compared to processing 8/16 bits per cycle / using a parallel access register file
- No point in calculating faster than the memory interface allows

- 8x 8-bit general purpose registers that can be paired into 4x 16-bit general purpose registers, plus an 8 bit stack register
- 8 bit and 16 bit versions of almost all instructions
- 64 kB address space
- 16 bits/instruction
- Quite regular and orthogonal instruction encoding, most instructions can use most addressing modes
  - `op reg, src` and `op src, reg` instruction forms
- Instructions:
  - `mov`, `swap`
  - `binop`: `add/adc/sub/sbc/and/or/xor/cmp/test`
    - for register-to-register also: `neg/negc/revsub/revsbc/and_not/or_not/xor_not/not`,
  - `shl/shr/sar/rol/ror` with variable or immediate shift count,
  - `mul`: 8x8 and 8x16 bit multiply instructions, producing 2 result bits per cycle like everything else,
  - `branch cc, offset`: relative branch
    - unconditional/call/12 conditions including signed/unsigned comparisons,
  - `jump/call`: absolut direct/indirect jump/call,
  - additional functionality through combination with addressing modes, e g, `ret = jump [pop]`
- Addressing modes:
  - `[imm7]` / `[imm7*2]`: zero page
  - `[r16 + imm2]`
  - `[r16 + r8]`
  - `[r16]` with postincrement/predecrement
  - `[push]` / `[pop]` / `[top-of-stack]` depending on whether the operand is written/read/modified

    - `[imm16]`

- Sign/zero extension of any 8 bit register as source operand to 16 bit instructions
- `imm16` / `[imm16]` operands supported using extra instruction word
- 2-4 word instruction prefetch queue

Basilisc-2816 v0.1 variants

Basilisc-2816 v0.1 has been taped out in three variants for Tiny Tapeout 7:

	mul instruction	Prefetch queue size	Hardened with	Uses latches	Mux address
v0.1a	yes	2	OpenLane 1	no	967
v0.1b	no	3	OpenLane 2	no	202
v0.1c	yes	4	OpenLane 2	yes	72

successively more experimental. Longer prefetch queue should help contribute to better performance, especially with long memory access latencies.

This is the 0.1b version. For more details, see https://github.com/toivoh/tt07-basilisc-2816-cpu/blob/main/docs/info.md or the documentation for Basilisc-2816 v0.1a CPU [967].

Pinout

#	Input	Output	Bidirectional
0	rx_in[0]	tx_out[0]	
1	rx_in[1]	tx_out[1]	
2		tx_fetch	
3		tx_jump	
4			
5			
6			
7			

integer to posit converter and adder [204]

- Author: A. Fasolino, G.D. Licciardo, A. Torino, F. Del Prete, C. Parrella
- Description: Our module executes a fixed to posit conversion and an addition
- GitHub repository
- HDL project
- Mux address: 204
- Extra docs
- Clock: 30000000 Hz

How it works

The module (Fig. 1) is fed by two fixed-point numbers, namely af and bf, coverts them into the posit arithmetic [1] format (ap and bp) and sums them to produce a posit output (sp).

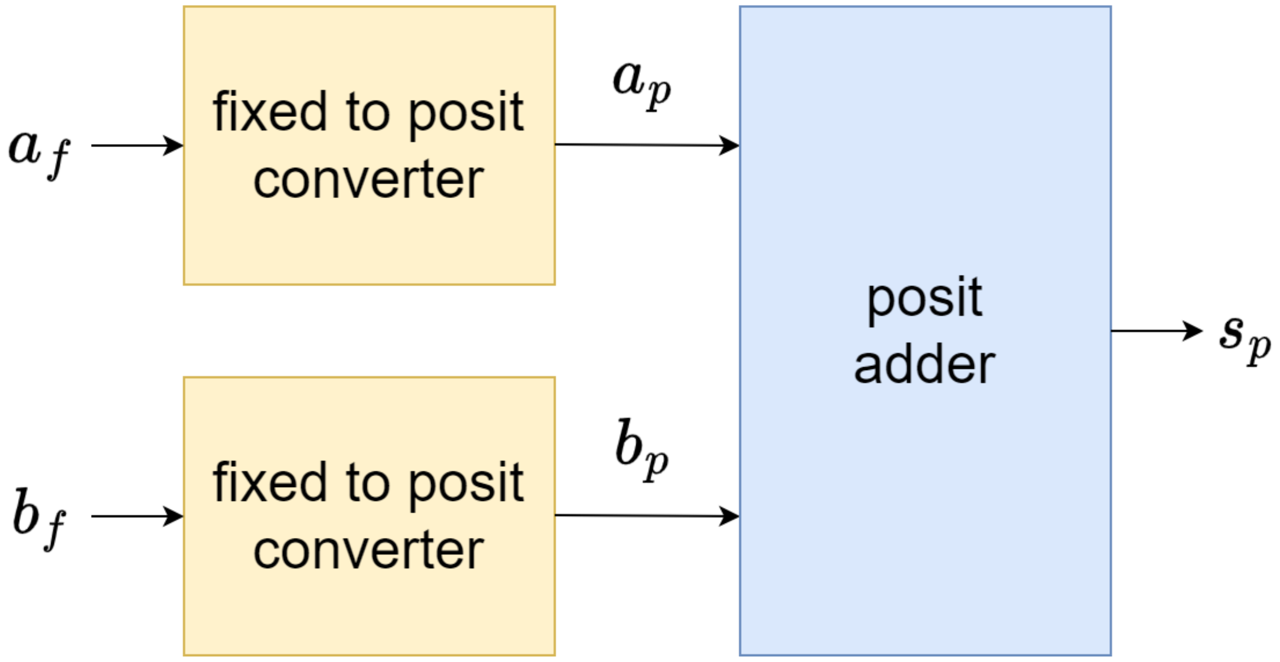


Figure 14: fixed2posit-module drawio

It is made of two units: 1.

1) 16-bit 2's complement fixed-point 0.15 coded to 16-bit standard posit (16,1) converter, namely fixed to posit converter,
2) posit adder, that executes the addition of posit numbers according to the posit standard.

The conversion is operated as described in [1], leveraging a leading zero counter [2] and some glue logic.

2. The addition leverages the architecture presented in [3].

References

[1] J. Gustafson. "Posit arithmetic." Mathematica Notebook describing the posit number system, 2017.

[2] Milenković, Nebojša & Stankovic, Vladimir & Milić, Miljana. (2015). Modular Design Of Fast Leading Zeros Counting Circuit. Journal of Electrical Engineering. 66. 329-333. 10.2478/jee-2015-0054.

[3] R. Murillo, A. A. Del Barrio and G. Botella, "Customized Posit Adders and Multipliers using the FloPoCo Core Generator," 2020 IEEE International Symposium on Circuits and Systems (ISCAS), 2020, pp. 1-5, doi: 10.1109/ISCAS45731.2020.9180771.

How to test

Provide two fixed-point input data and they will be added in posit arithmetic.

Pinout

#	Input	Output	Bidirectional
0	data input	addition result	data valid
1	data input	addition result	alu valid
2	data input	addition result	read data valid
3	data input	addition result	read data ready
4	data input	addition result	
5	data input	addition result	
6	data input	addition result	
7	data input	addition result	

LFSR [206]

- Author: James Meech and Werner Florian
- Description: Linear feedback shift register random number generator
- GitHub repository
- HDL project
- Mux address: 206
- Extra docs
- Clock: 0 Hz

How it works

It is a linear feedback shift register random number generator connected to a wishbone bus to allow it to fit within the pin constraints of Tiny Tapeout.

How to test

Please see the cocotb testbench in the test.py in the test directory for the startup procedure for loading a seed and starting the linear feedback shift register output.

External hardware

Use the microcontroller on the development board to apply the correct startup signals to the board.

Pinout

#	Input	Output	Bidirectional
0	Wishbone data input bit 0	Output bit to indicate whether or not the wishbone has stalled (o_wb_stall)	Wishbone input bit to indicate that a valid bus cycle is in progress (i_wb_cyc, hardcoded as an input)

#	Input	Output	Bidirectional
1	Wishbone data input bit 1	LFSR output bit (o_wb_data)	Wishbone chipselect input bit to indicate a valid seed data transfer cycle (i_wb_stb, hardcoded as an input)
2	Wishbone data input bit 2	Output bit for the wishbone to acknowledge the successful end of writing part of the LFSR seed (o_wb_ack)	Wishbone input bit to indicate a read or a write cycle read = 0, write = 1 (i_wb_we, hardcoded as an input)
3	Wishbone data input bit 3	Not used in this design	Wishbone input address bit zero to select which eight bit byte of the seed to write (i_wb_addr[0])
4	Wishbone data input bit 4	Not used in this design	Wishbone input address bit one to select which eight bit byte of the seed to write (i_wb_addr[1])
5	Wishbone data input bit 5	Not used in this design	Wishbone input address bit two to select which eight bit byte of the seed to write (i_wb_addr[2])
6	Wishbone data input bit 6	Not used in this design	Not used in this design
7	Wishbone data input bit 7	Not used in this design	Not used in this design

Serial Character LED Matrix [225]

- Author: Ciro Cattuto
- Description: LED matrix character display controlled via UART
- GitHub repository
- HDL project
- Mux address: 225
- Extra docs
- Clock: 20000000 Hz

How it works

This project drives an LED-matrix character display composed of one or more Pixie Chroma chainable devices, each one featuring two 5x7 LED matrices based on the WS2812B RGB LEDs. Each 5x7 LED matrix displays one character. The display shows characters received over a serial port.

Up to 4 chained devices are supported for a maximum of 8 characters. The displayed characters are received from a serial port using the UART protocol (9600 baud, 8N1). The project includes a simple UART implementation. Every time a character is received over UART, the displayed characters shift left, and the new character appears to the right. 5x7 matrix representations for printable ASCII characters are supported using the font from Arduino Microview Library encoded in a character ROM. Non-printable ASCII characters are shown as an empty rectangle. Each new character appears with a color randomly chosen among a palette of 16 colors contained in a color ROM. A pseudo-random number generator based on a linear-feedback shift register is used for color selection.

The project is designed to demonstrate components that should be easily re-usable:

- WS2812B LED strip driver
- UART receiver and transmitter (8N1 only, no flow control)
- linear-feedback pseudo-random number generator
- character ROM
- cocotb tests for UART and WS2812B protocol

How to test

Basic setup:

- Connect `uo[0]` to the input pin (`DATA_IN`) of a Pixie Chroma LED-matrix display (two 5x7 WS2812B LED matrices). Ensure the `VCC` and `GND` pins are connected to an adequate power source.
- Configure the input (e.g., using the DIP switches of the PCB) as follows: set `ui[0]` and `ui[1]` to `0` to use one Pixie Chrome (i.e., two 5x7 LED matrices); set `ui[2]` to `1` to enable UART echo; set `ui[4]` and `ui[5]` to `0` to disable LED dimming, set `ui[6]` to `0` to select internal display refresh, and `ui[7]` to `0` to select random color selection.
- Connect the UART interface of the project (RX is `ui[3]`, TX is `uo[4]`) to a serial terminal or a UART-to-USB PMOD or adapter (e.g., the one provided by the onboard RP2040 of the Tiny Tapeout PCB). Configure the serial interface for 9600 baud, 8 bits, 1 start bit, no parity bit, and 1 stop bit (8N1), with no hardware or software flow control.
- Open the terminal and type any characters: printable ASCII characters will appear from the right-hand side on the LED matrix and shift left as more characters are typed. Each character will appear with a different random color. Non-printable ASCII characters are shown as an empty rectangle. When `ui[2]` is set to `1`, received characters are echoed on the serial connection.

To use **more than one Pixie Chroma**, chain additional displays after the first one. This project supports up to 4 displays (e.g., 8 5x7 LED matrices). Set `ui[1]` and `ui[0]` (e.g., by using the DIP switches of the PCB) to configure the number of displays you are using: `00` for 1 display, `01` for 2 displays, `10` for 3 displays, and `11` for 4 displays (8 5x7 characters).

Dimming of the LED matrix is controlled by the `ui[5]` and `ui[4]` signals: `00` for no dimming, `01`, `10`, and `11` for increasing dimming.

The **color of characters** is randomly chosen when `ui[6]` is low, and fixed when ‘`ui[6]` is high. The fixed color can be changed by sending over UART a non-printable byte >= 128: the lower 4 bits of such value are stored and used as an index in the color ROM (16 different colors).

Two **display refresh modes** are available, controlled by `ui[6]`. When `ui[6]` is low, refresh is internally triggered at a frequency of about 75 Hz. When `ui[6]` is high, refresh is triggered via UART, whenever a CR or LF character is received. For both refresh modes, `uo[1]` is pulsed high on LED matrix refresh.

External hardware

- 1 to 4 Pixie Chroma WS2812B LED-matrix displays (chained if more than one is used). An external power source for the LED matrix is recommended.

- UART terminal or UART-to-USB adapter (PMOD or on-board via RP2040). TX-only is sufficient.

Pinout

#	Input	Output	Bidirectional
0	num chars selector 0	LED strip	
1	num chars selector 1	LED strip latch	
2	UART loopback option		
3	UART RX		
4	dimmer selector 0	UART TX	
5	dimmer selector 1		
6	internal/external refresh selector		
7	random/fixed color selector	UART RX valid	

DDS and DAC [227]

- Author: Meonwara
- Description: DDS with DAC and analogue output pin
- GitHub repository
- Analog project
- Mux address: 227
- Extra docs
- Clock: 40000000 Hz

How it works

Simplistic DDS (accumulator + sinewave lookup table) with resistive DAC to provide analogue output. Board switches control the output frequency.

How to test With a 40MHz clock selected, change the input switches to some binary value 1-255. Observe a rail to rail sinewave at the analogue pin ua[0].

External hardware DAC output resistance about 10kOhm. Could add an external capacitor to ground to smooth / filter the waveform.

Pinout

#	Input	Output	Bidirectional
0	ui[0]	uo[0]	uio[0]
1	ui[1]	uo[1]	uio[1]
2	ui[2]	uo[2]	uio[2]
3	ui[3]	uo[3]	uio[3]
4	ui[4]	uo[4]	uio[4]
5	ui[5]	uo[5]	uio[5]
6	ui[6]	uo[6]	uio[6]
7	ui[7]	uo[7]	uio[7]

Analog pins

ua#	analog#	Description
0	4	ua[0]

PLL Playground [229]

- Author: Sean Patrick O'Brien
- Description: Phase-Locked Loop and its parts
- GitHub repository
- Analog project
- Mux address: 229
- Extra docs
- Clock: 0 Hz

How it works

VCO The VCO is a current-starved ring oscillator with a variable number of stages. The digital inputs `s0` and `s1` determine the number of stages. The analog `vcont` input controls the frequency by adjusting the amount of current received by the inverters.

PFD The PFD takes two input signals and uses the difference in phase+frequency to drive a charge pump.

How to test

VCO Apply a voltage between 1.0V and 1.8V to `vcont` (`ua0`) and observe the oscillator output on `uo0`.

PFD Apply two reference signals to the inputs `ui2` and `ui3` and observe the voltage change on `vout` (`ua1`).

External hardware

An osilloscope, function generator, and a benchtop power supply would be handy but are not required.

#	Input	Output	Bidirectional

Pinout

#	Input	Output	Bidirectional
0	vco0_s0	vco0_out	
1	vco0_s1		
2	pfd0_clk		
3	pfd0_ref		
4			
5			
6			
7			

Analog pins

ua#	analog#	Description
0	0	vco0_vcont
1	5	pfd0_vout

Analog comparator [231]

- Author: Diego Carrera and Leonel Miranda
- Description: Analog comparator made by the MNEL team from USFQ.
- GitHub repository
- Analog project
- Mux address: 231
- Extra docs
- Clock: 0 Hz

How it works

Circuit that compares two voltages or currents and produces a binary output indicating which input is greater.

How to test

To test a comparator, apply known input voltages and check if the output logic level matches the expected result based on the input conditions. Vary the input voltages across the entire range and verify the output transitions occur at the specified threshold voltage.

External hardware

none

Pinout

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

Analog pins

ua#	analog#	Description
0	1	analog input
1	3	analog reference
2	2	analog output

Gilbert Mixer [233]

- Author: Kolos Koblasz
- Description: Gilbert Cell mixer for up and down conversion
- GitHub repository
- Analog project
- Mux address: 233
- Extra docs
- Clock: 100000000 Hz

How it works

This mixed signal circuit is a mixer. It can up or down convert a signal. The digital control part can generate internal LO signal with different frequencies, but also can pass an external Lo signal to the analog mixer.

How to test

Send analog signal to the input ports and measure the output ports. Also apply external LO or use internal LO source.

External hardware

Since the user have to bias the in_n and in_p ports with 900-950mV and apply a differential signal in the range of 20mV peak-to-peak I would say a custom interface PCB is needed to use the mixer. TO DO later.

Pinout

#	Input	Output	Bidirectional
0	ext_lo_en		
1	ext_lo_n		
2	ext_lo_p		
3	int_lo_settings[0]		
4	int_lo_settings[1]		
5	int_lo_settings[2]		
6			
7			

Analog pins

ua#	analog#	Description
0	4	in_n
1	1	in_p
2	3	out_p
3	2	out_n

Instrumentation Amplifier for Electrocardiogram Signal Adquisition [239]

- Author: Rocha Judith Rocha-Torres
- Description: The amplifier gatters the electrical signal coming from the skin/muscles over the heart
- GitHub repository
- Analog project
- Mux address: 239
- Extra docs
- Clock: 0 Hz

How it works

The designed instrumentation amplifier extracts very low differencial mode signal from very noisy common mode signals by means of a three opamp array. The differential gain can be changed by two digital bits called "Sel_26dB" and "Sel_42dB", when Sel_26dB = 0 Sel_42dB = 0 the differential gain is 0 dB, when Sel_26dB = 1 and Sel_42dB = 0 the differential gain is 26 dB, when Sel_26dB = 1 and Sel_42dB = 1 the differential gain is 46 dB. In all cases the CMRR is 120 dB. The designed instrumentation amplifier also included a sense circuits that helps to stabilize the analog ground.

How to test

1. Apply a DC level equals to vdd = 1.8V and vss = 0V.
2. Set the following bits to Sel_26dB = 0 and Sel_42dB = 0.
3. Apply a 1 mV differential signal into Vip and Vin analog ports using a balum.
4. Measure the differential responds at node Vo using Keysight-E5061B ENA vector network analyzer, a 0 dB gain shall be expected.
5. Apply a 1 mV common signal into Vip and Vin analog ports.
6. Measure the common responds at node Vo using Keysight-E5061B ENA vector network analyzer.
7. Performs the substraction of the differential gain and the common gain, the CMRR responds should be obtined.
8. Set the following bits to Sel_26dB = 1 and Sel_42dB = 0.
9. Apply a 1 mV differential signal into Vip and Vin analog ports using a balum.
10. Measure the differential responds at node Vo using Keysight-E5061B ENA vector network analyzer, a 26 dB gain shall be expected.
11. Apply a 1 mV common signal into Vip and Vin analog ports.

12. Measure the common responds at node Vo using Keysight-E5061B ENA vector network analyzer.
13. Performs the substraction of the differential gain and the common gain, the CMRR responds should be obtined.
14. Set the following bits to Sel_26dB = 1 and Sel_42dB = 1.
15. Apply a 1 mV differential signal into Vip and Vin analog ports using a balum.
16. Measure the differential responds at node Vo using Keysight-E5061B ENA vector network analyzer, a 42 dB gain shall be expected.
17. Apply a 1 mV common signal into Vip and Vin analog ports.
18. Measure the common responds at node Vo using Keysight-E5061B ENA vector network analyzer.
19. Performs the substraction of the differential gain and the common gain, the CMRR responds should be obtined.

External hardware

- 1 PCB
- 1 Balum
- 1 Keysight-E5061B ENA vector network analyzer
- 3 Female-Female BNC cable.
- 1 Female-Female-Female BNC T-adapter
- 1 Keithly 2231A power supply
- 1 Agilent 34401A digital multimeter

Pinout

#	Input	Output	Bidirectional
0	Sel_42dB		
1	Sel_26dB		
2			
3			
4			
5			
6			
7			

Analog pins

ua#	analog#	Description
0	5	vo
1	0	AGND
2	4	VirGND_FB
3	1	VirGND_Out
4	3	Vin
5	2	Vip

router [256]

- Author: Leeja & Saranya
- Description: 1*3 router
- GitHub repository
- HDL project
- Mux address: 256
- Extra docs
- Clock: 10000000 Hz

How it works

routes the data to 3 directions

How to test

give input to data_in

External hardware

Pinout

#	Input	Output	Bidirectional
0	data_in[0]	data_out_0[0]	data_out_2[0]
1	data_in[1]	data_out_0[1]	data_out_2[1]
2	data_in[2]	data_out_0[2]	data_out_2[2]
3	pkt_valid	data_out_1[0]	vld_out_0
4	read_enb_0	data_out_1[1]	vld_out_1
5	read_enb_1	data_out_1[2]	vld_out_2
6	read_enb_2	err	
7		busy	

Conway's Terminal [258]

- Author: Ciro Cattuto
- Description: A simulation of Conways' Game of Life visualized to an ANSI terminal over UART
- GitHub repository
- HDL project
- Mux address: 258
- Extra docs
- Clock: 24000000 Hz

How it works

This projects simulates Conway's Game of Life in hardware on a small (32x16) grid with periodic boundary conditions. At each time step, the output of the simulation is printed to an ANSI serial terminal over a serial (UART) interface. The initial state of the board is pseudo-random, generated using a linear-feedback shift register. Single characters received over the serial interface are used to control the simulation, according to the following table:

- `&amp;lt;space&amp;gt;`: start/stop simulation
- `0`: randomize state
- `1`: single-step the simulation

The UART interface of the project is exposed according the the Tiny Tapeout recommended pinout, with `ui_in[3]` used for RX signal and `uo_out[4]` for TX. The UART is configured as 8N1 at 115200 baud, with no flow control.

VGA output of the simulation state is also exposed on the bidirectional pins, that are all configured as outputs and wired to work with a TinyVGA PMOD.

How to test

Connected the UART interface of the project to any UART terminal, or to an UART-to-USB PMOD or adapter, e.g., the one provided by the onboard RP2040 of the PCB. Configure the serial interface for 8 bits, 1 start bit, no parity bit, 1 stop bit (8N1), with no hardware or software flow control. Open the terminal and type any character: this will bring up a welcome message explaining how to control the simulation.

External hardware

UART terminal, or UART-to-USB adapter (PMOD or on-board via RP2040). Optionally, TinyVGA PMOD for VGA output.

Pinout

#	Input	Output	Bidirectional
0			hsync
1			B[0]
2			G[0]
3	UART RX		R[0]
4		UART TX	vsync
5			B[1]
6			G[1]
7			R[1]

Zilog Z80 [259]

- Author: ReJ aka Renaldas Zioma
- Description: Z80 open-source silicon. Goal is to become a silicon proven, pin compatible, open-source replacement for classic Z80.
- GitHub repository
- HDL project
- Mux address: 259
- Extra docs
- Clock: 16000000 Hz

How it works

On April 15 of 2024 Zilog has announced End-of-Life for Z80, one of the most famous 8-bit CPUs of all time. It is a time for open-source and hardware preservation community to step in with a Free and Open Source Silicon (FOSS) replacement for Zilog Z80.

The implementation is based around Guy Hutchison's TV80 Verilog core.

The future work

- Add thorough instruction (including 'illegal') execution tests ZEXALL to test-bench
- Compare different implementations: Verilog core A-Z80, Netlist based Z80Explorer
- Create gate-level layouts that would resemble the original Z80 layout. Zilog designed Z80 by manually placing each transistor by hand.
- Tapeout QFN44 package
- Tapeout DIP40 package

Z80 technical capabilities

- nMOS original frequency 4MHz. CMOS frequency up to 20 MHz. This tapeout on 130 nm is expected to support frequency up to 50 MHz.
- 158 instructions including support for Intel 8080A instruction set as a subset.
- Two sets of 6 general-purpose reigsters which may be used as either 8-bit or 16-bit register pairs.
- One maskable and one non-maskable interrupt.
- Instruction set derived from Datapoint 2200, Intel 8008 and Intel 8080A.

Z80 registers

- `AF`: 8-bit accumulator (A) and flag bits (F)
- `BC`: 16-bit data/address register or two 8-bit registers

- `DE`: 16-bit data/address register or two 8-bit registers
- `HL`: 16-bit accumulator/address register or two 8-bit registers
- `SP`: stack pointer, 16 bits
- `PC`: program counter, 16 bits
- `IX`: 16-bit index or base register for 8-bit immediate offsets
- `IY`: 16-bit index or base register for 8-bit immediate offsets
- `I`: interrupt vector base register, 8 bits
- `R`: DRAM refresh counter, 8 bits (msb does not count)
- `AF'`: alternate (or shadow) accumulator and flags (toggled in and out with `EX AF, AF'`)
- `BC'`, `DE'` and `HL'`: alternate (or shadow) registers (toggled in and out with `EXX`)

Z80 Pinout

```
                 ,---------.__.---------.
<--     A11 |1                    40| A10     -->
<--     A12 |2                    39| A9      -->
<--     A13 |3       Z80 CPU      38| A8      -->
<--     A14 |4                    37| A7      -->
<--     A15 |5                    36| A6      -->
-->     CLK |6                    35| A5      -->
<->      D4 |7                    34| A4      -->
<->      D3 |8                    33| A3      -->
<->      D5 |9                    32| A2      -->
<->      D6 |10                   31| A1      -->
        VCC |11                   30| A0      -->
<->      D2 |12                   29| GND
<->      D7 |13                   28| /RFSH   -->
<->      D0 |14                   27| /M1     -->
<->      D1 |15                   26| /RESET <--
-->    /INT |16                   25| /BUSRQ <--
-->    /NMI |17                   24| /WAIT   <--
<--   /HALT |18                   23| /BUSAK -->
<--   /MREQ |19                   22| /WR     -->
<--   /IORQ |20                   21| /RD     -->
                 `----------------------'
```

How to test

Hold all `bidirectional` pins (**Data bus**) low to make CPU execute **NOP** instruction. **NOP** instruction opcode is 0. Hold all `input` pins high to disable interrupts and signal that data bus is ready.

Every 4th cycle 8-bit value on output pins (**Address bus low 8-bit**) should monotonously increase.

```
Timing diagram, input pins

 Z80CLK____      ____      ____      ____      ____      ____
    __/    \____/    \____/    \____/    \____/    \____/     `____  .
      |        |         |         |         |         |
      |        |         |         |         |         |

 /RESET____________________________________________________________
    __/
 /WAIT ____________________________________________________________
    __/
 /INT  ____________________________________________________________
    __/
 /NMI  ____________________________________________________________
    __/
 /BUSRQ____________________________________________________________
    __/

 D7..D0             NOP       NOP       NOP       NOP       NOP
    __ XXXXXXXXX ___#00___ ___#00___ ___#00___ ___#00___ ___#00___

 Expected signals on output pins
 /M1   _________                    ____________________
                \__________________/                    \__________
 /MREQ ___________________          ______________________________
                          \________/
 /RD   ___________________          ______________________________
                          \________/
 A0..A7
    __ XXXXXXXXX ___#00___ ___#00___ XXXXXXXXX XXXXXXXXX ___#01___
```

External hardware

Bus de-multiplexor, external memory, 8-bit computer such as ZX Spectrum.

Alternatively the RP2040 on the TinyTapeout test PCB can be used to simulate RAM and I/O.

Pinout

#	Input	Output	Bidirectional
0	/WAIT	/M1, A0, A8	D0
1	/INT	/MREQ, A1, A9	D1
2	/NMI	/IORQ, A2, A10	D2
3	/BUSRQ	/RD, A3, A11	D3
4		/WR, A4, A12	D4
5		/RFSH, A5, A13	D5
6		/HALT, A6, A14	D6
7		/BUSAK, A7, A15	D7

VGA Perlin Noise [260]

- Author: Uri Shaked
- Description: Simple animated perlin noise for TinyVGA PMod
- GitHub repository
- HDL project
- Mux address: 260
- Extra docs
- Clock: 31500000 Hz

How it works

Generates an animated perlin noise pattern on the screen. The perlin noise code was created with the help of the Tiny Tapeout AI Assist GPT.

How to test

Connect to a VGA monitor. Change pattern_sel (ui_in[0]) to choose between two different patterns.

External hardware

TinyVGA PMOD

Pinout

#	Input	Output	Bidirectional
0	pattern_sel	R1	
1		G1	
2		B1	
3		VSync	
4		R0	
5		G0	
6		B0	
7		HSync	

4-bit R2R DAC [261]

- Author: Vishal Bingi
- Description: It is a 4-bit R2R DAC with a sawtooth waveform driver
- GitHub repository
- Analog project
- Mux address: 261
- Extra docs
- Clock: 0 Hz

How it works

It is a simple 4-bit R2R DAC, which is driven externally by an openlane generated sawtooth waveform generator.

How to test

Drive externally: Set the "external data" input high to provide the DAC with external data. Then drive the 4 inputs and observe the analog output.

Drive with internal sawtooth wave generator: Set the "external data" input low to enable the sawtooth generator. A sawtooth wave should be seen on the analog output. To change the frequency of the sawtooth wave, set the inputs and then raise the "load divider" input.

External hardware

A multimeter to measure the output voltage on analog pin 0.

Pinout

#	Input	Output	Bidirectional
0	bit 0		external data
1	bit 1		load divider
2	bit 2		
3	bit 3		
4			
5			
6			

#	Input	Output	Bidirectional
7			

Analog pins

ua#	analog#	Description
0	7	DAC output

MOS Bandgap [263]

- Author: Matt Venn
- Description: bandgap using only MOSFETs
- GitHub repository
- Analog project
- Mux address: 263
- Extra docs
- Clock: 0 Hz

How it works

Read the paper here and see the DTMOS variant below.

Circuit

DTMOS variant connects the lower PMOS body to VSS instead of VDD.

Simulation

MOS

- Simulated output is 0.714 to 0.716v across 10 to 120 degrees C.
- Simulated output is 0.6 to 0.8v across 1.6 to 2v VDD.

DTMOS

- Simulated output is 0.515 to 0.510v across 10 to 120 degrees C.
- Simulated output is 0.3 to 0.6v across 1.6 to 2v VDD.

How to test

MOS Connect a multimeter to analog output 0. It should measure around 0.7v and remain constant with temperature.

DTMOS Connect a multimeter to analog output 0. It should measure around 0.5v and remain constant with temperature.

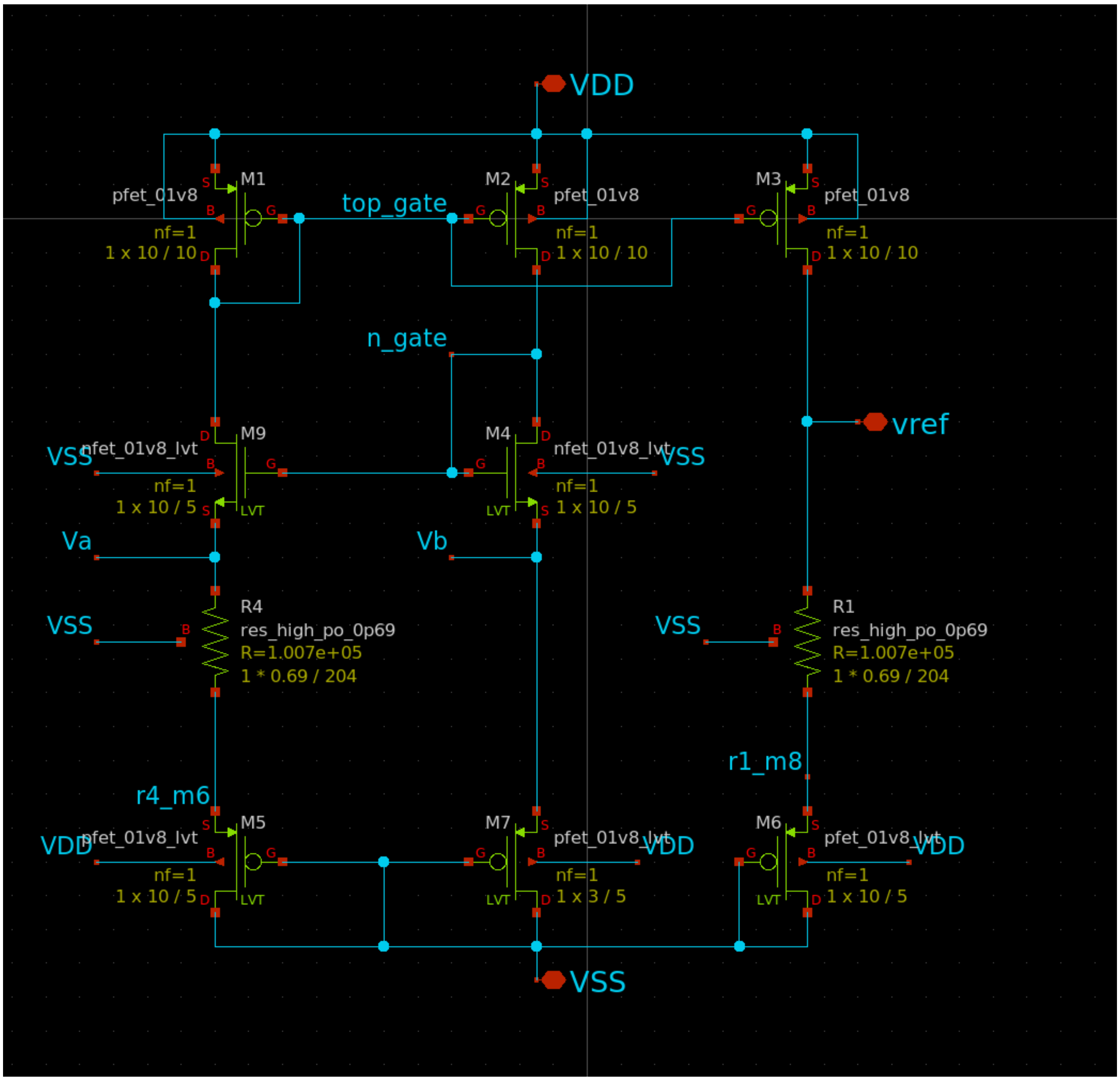


Figure 15: bandgap circuit

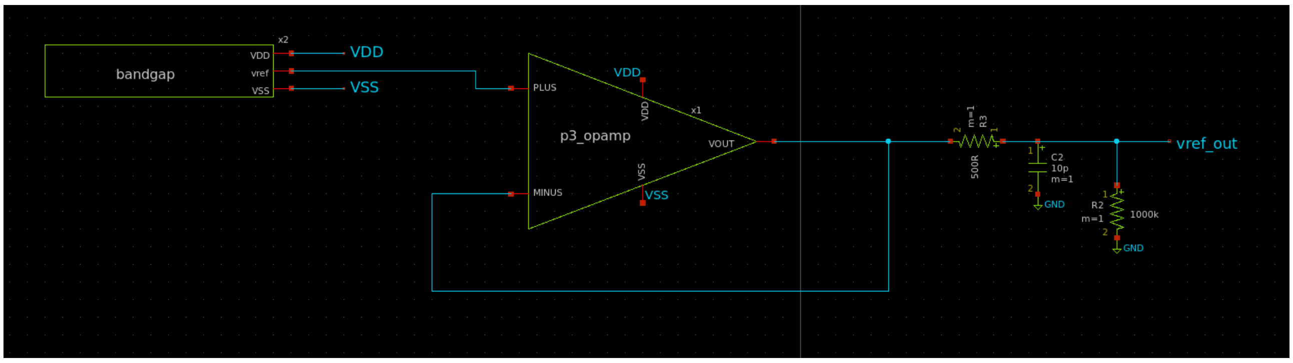


Figure 16: opamp buffer

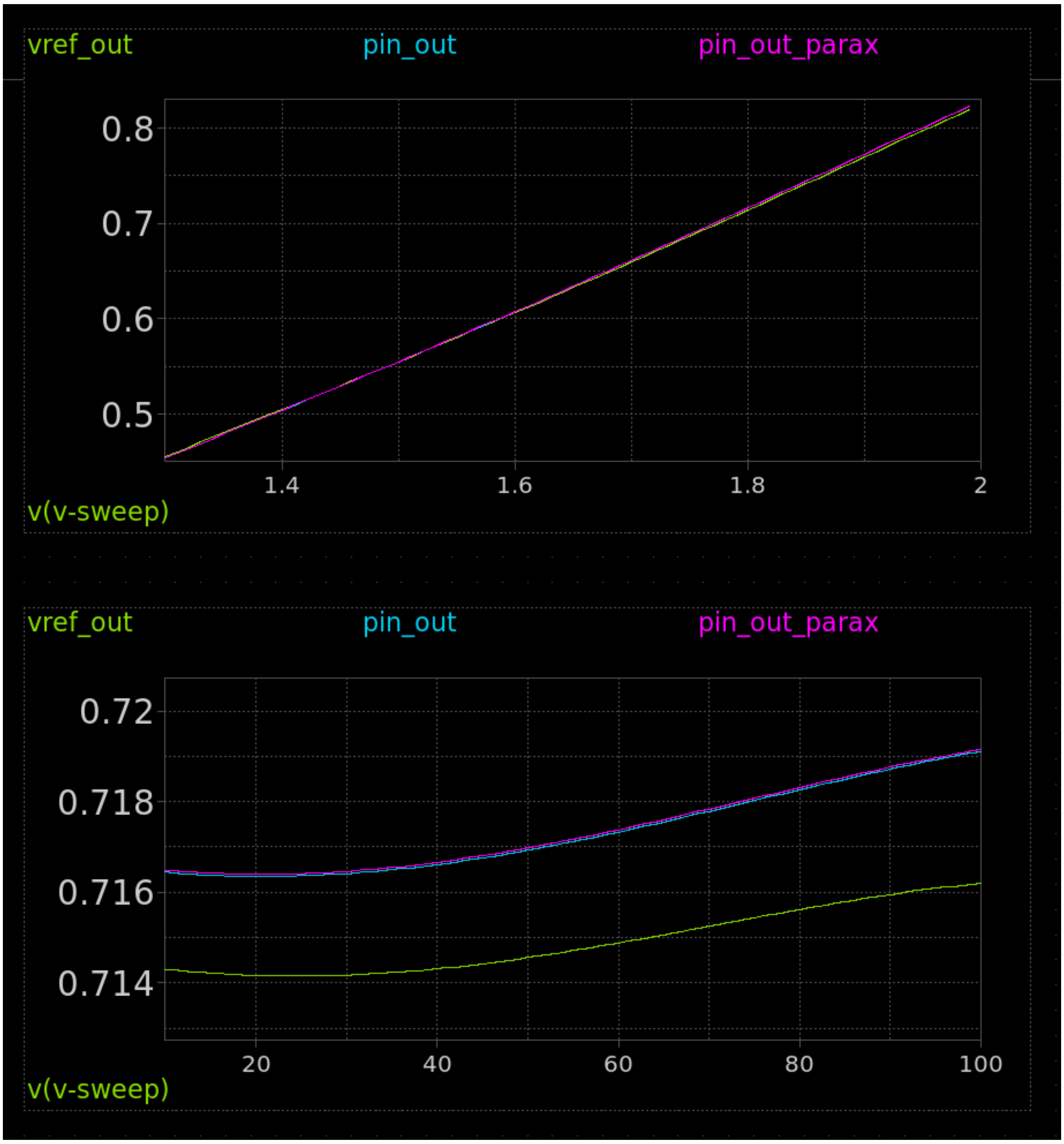


Figure 17: sim

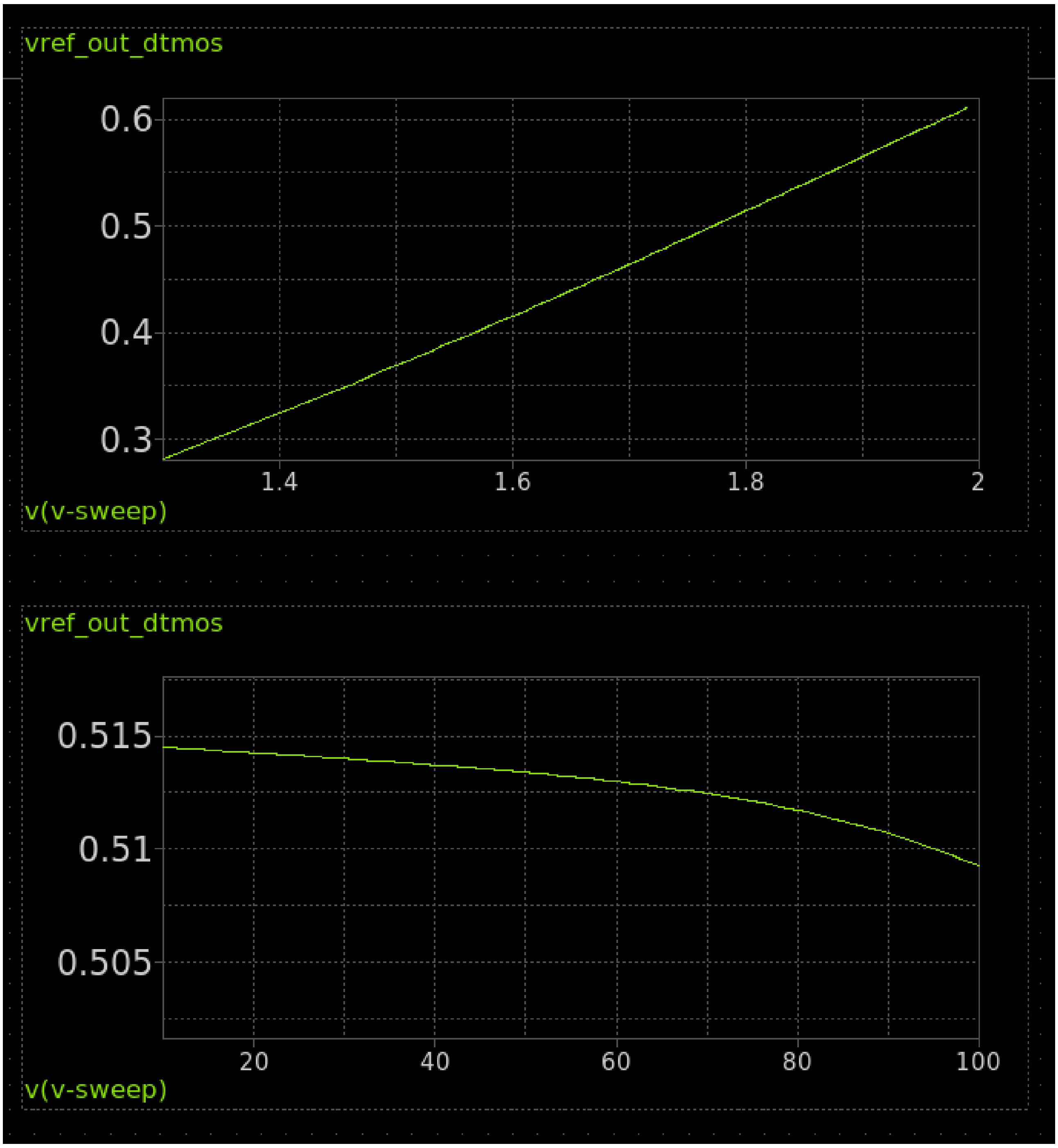


Figure 18: sim

External hardware

Multimeter, hot air gun to heat the chip

References

- A sky130 reference bandgap with results
- DTMOS varient

Pinout

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

Analog pins

ua#	analog#	Description
0	9	bandgap
1	8	bandgap_dtmos

VGA Checkers [264]

- Author: ReJ aka Renaldas Zioma
- Description: VGA
- GitHub repository
- HDL project
- Mux address: 264
- Extra docs
- Clock: 25200000 Hz

How it works

It generates patterns on VGA screen.

How to test

Connect to VGA monitor.

External hardware

TinyTapeout VGA PMOD, VGA monitor

Pinout

#	Input	Output	Bidirectional
0		R	
1		G	
2		B	
3		vsync	
4		R	
5		G	
6		B	
7		hsync	

Analog buffer test [265]

- Author: Aron Dennen
- Description: Double inverter project from Matt's analog course
- GitHub repository
- Analog project
- Mux address: 265
- Extra docs
- Clock: 0 Hz

How it works

This is the double inverter project from Matt's analog course, it's two inverters (one big and one small) connected to form a buffer

How to test

Analog input is ua[1], output is ua[0]

External hardware

none

Pinout

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

Analog pins

ua#	analog#	Description
0	7	analog output
1	9	analog input

Current Mode Trigger [267]

- Author: Alfiero Leoni
- Description: Hysteresis Trigger working with input currents
- GitHub repository
- Analog project
- Mux address: 267
- Extra docs
- Clock: 0 Hz

How it works

The current mode trigger is a Schmitt trigger, therefore with hysteresis, that takes currents as input instead of voltages and produces a digital output that rises from 0 to 1 when the input current overcomes the first threshold and falls from 1 to 0 when the current falls below the second threshold. The hysteresis thresholds are internally set with reference currents. In particular, the rising edge current threshold is fixed, while the falling edge trigger can be adjusted by means of an external voltage reference. The current mode trigger is useful to detect signals coming from devices that work in current mode, such as photodiodes, SPADs, and Silicon photomultipliers. The latter represents an emerging technology for LIDAR systems, medical diagnosis, and particle physics detection systems, where even a single photon can be detected and converted into a photocurrent. In this sense, the current mode trigger can quickly detect a rare event of specific particle detection, as in the experiments for Dark Matter research.

How to test

The trigger can be tested with current input signal (such as a triangular wave) and the output should be monitored to look for digital transitions. A reference voltage of 0.9 V should be applied to teh vref pin, while the falling threshould can be set by applying a voltage between 1.45 and 1.55V to the vgf pin.

External hardware

A Current signal generator is needed, such as the keithley 2450 Sourcemeter or any equivalent SMU, to produce the input signal. As an alternative, a current could be generated by means of a voltage source with a series resistance. In any case, the input current range should be around tens of microAmperes. A tunable Voltage supply is needed to produce the voltage references and a fast oscilloscope can be used to monitor the trigger output

Pinout

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

Analog pins

`ua#`	`analog#`	Description
0	6	input
1	7	input
2	9	output
3	8	input

Dickson Charge Pump [269]

- Author: Uri Shaked
- Description: Pumps the input voltage up to ~5.4V
- GitHub repository
- Analog project
- Mux address: 269
- Extra docs
- Clock: 2000000 Hz

How it works

A 3-stage dickson charge pump. The output voltage is `Vout = 4*(VPWR - Vths) = ~5.44 V` where `VPWR` is the digital input voltage (1.8 V), and Vths is the threshold voltage of the LVS NMOS (nominal 0.44 V when width=7, length=8).

How to test

Apply a clock signal of 2 MHz to the `clk` input. In TT07, the analog pin voltage is limited to VDDIO/VDDA (usually 3.3 V), so the output voltage will be divided by two. You can measure the divided output voltage at the `ua[0]` (vout_div) pin.

Simulation results

Post layout simulation showing the output voltage `x1.vout` and the divided output voltage on ta `ua[0]` pin. The output voltage stabilizes at ~5.0 V, and the divided output voltage at ~2.5 V. The current draw is about 357 nA.

The following graph shows the input clock, the intermediate voltages at the output of each stage, the output voltage, and the divided voltage as they rise during the first 10 us of operation.

Silicon measurements

The output voltage on `ua[0]` was measured with a digital multimeter that has a 7.8MΩ input impedance, at various clock frequencies. The following table summarizes the results:

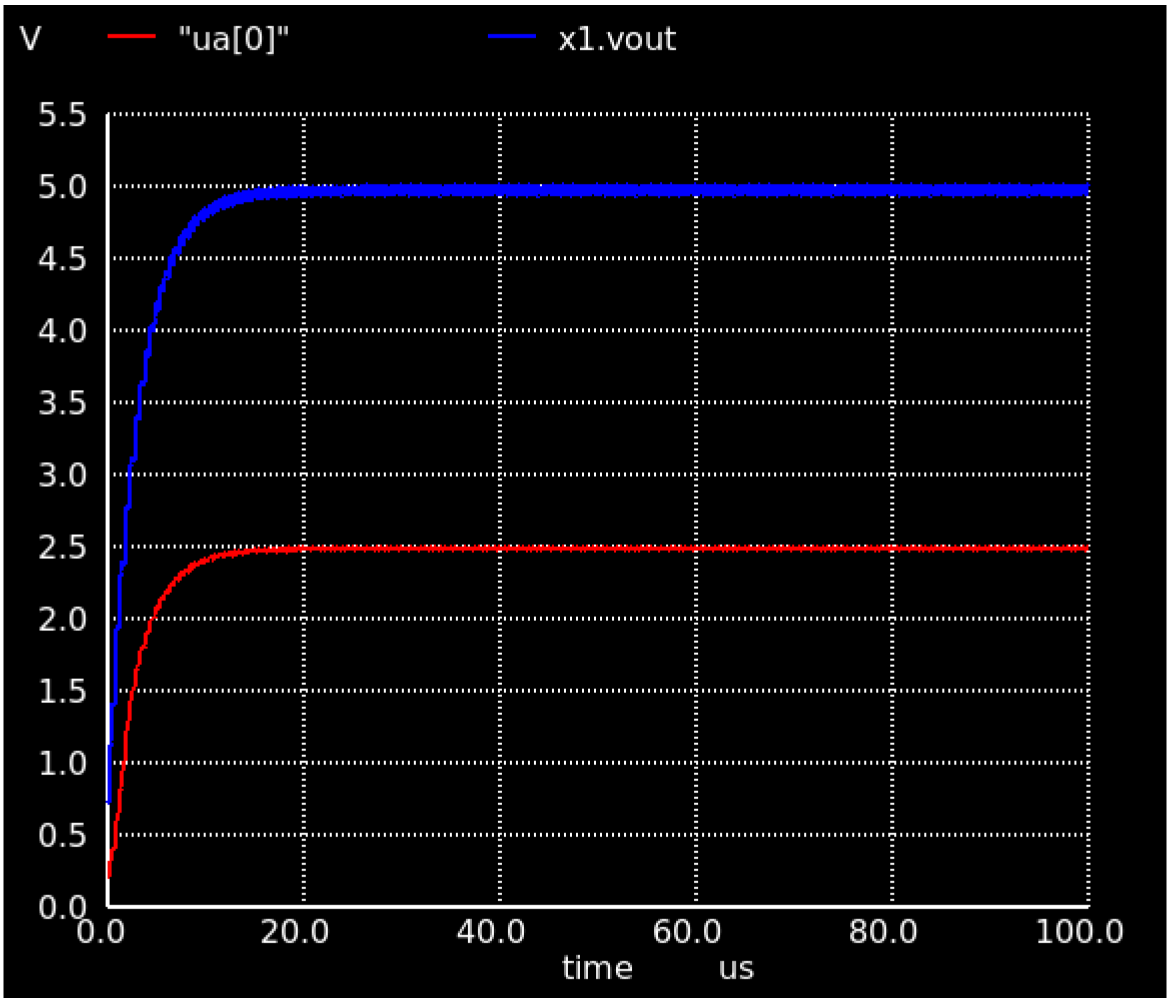


Figure 19: output voltage and divided voltage

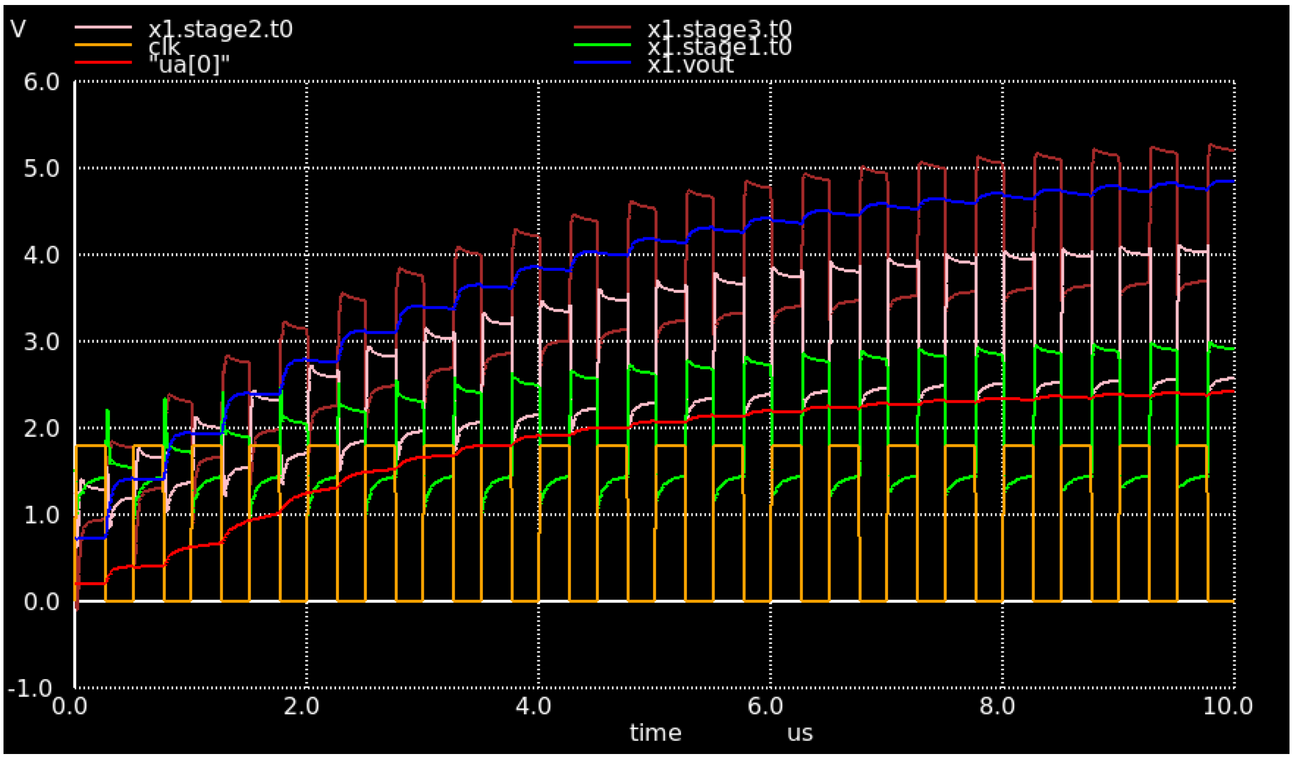


Figure 20: output voltage and intermediate voltages

Input Frequency (KHz)	ua[0] Voltage	Charge Pump Voltage *
0	0.065	0.130
10	0.236	0.472
50	0.643	1.286
100	1.006	2.012
250	1.524	3.048
500	1.862	3.724
1000	2.091	4.182
2000	2.213	4.426
5000	2.271	4.542
7500	2.276	4.552
10000	2.274	4.548
15000	2.265	4.530
20000	2.254	4.508
40000	2.190	4.380
62000	2.086	4.172
100000	1.768	3.536

- The charge pump voltage is the ua[0] voltage measurement multiplied by 2. This is because the analog pin voltage is limited to 3.3 V, so the charge pump voltage is divided by 2.

The following graph shows the output voltage as a function of the input frequency:

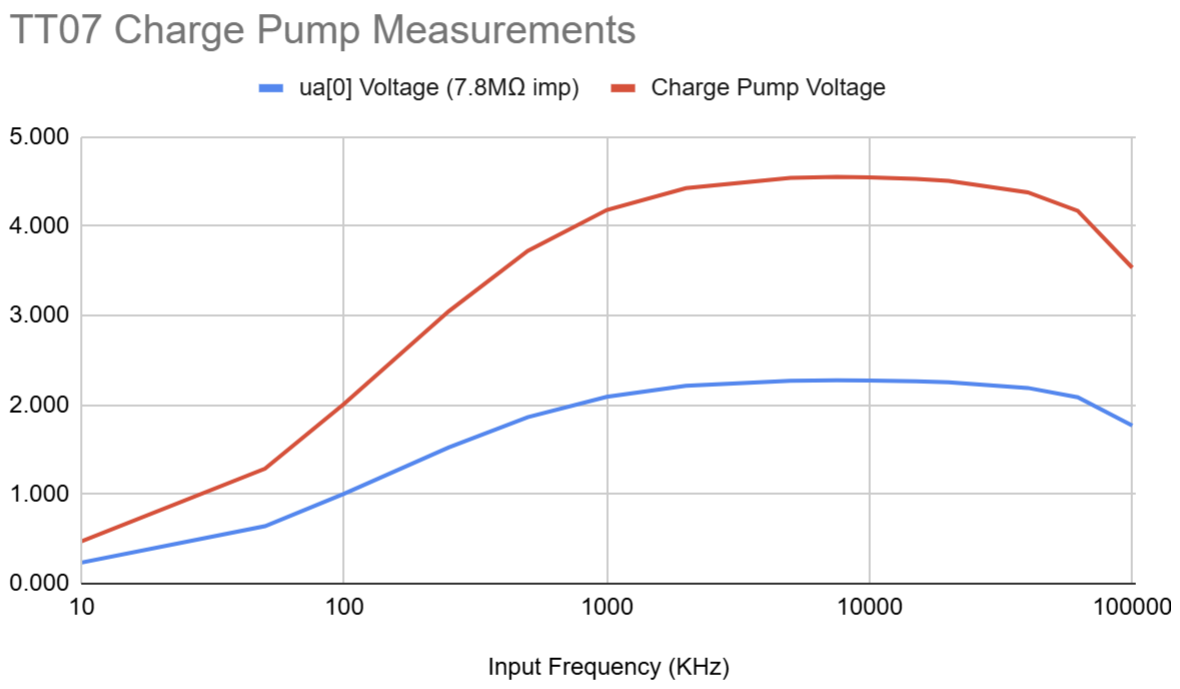


Figure 21: output voltage vs frequency

Overall, it seems that the charge pump works as expected, with the output voltage peaking at around 4.55 V when the input frequency is in the 5-10 MHz range.

Project layout

Pinout

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

Analog pins

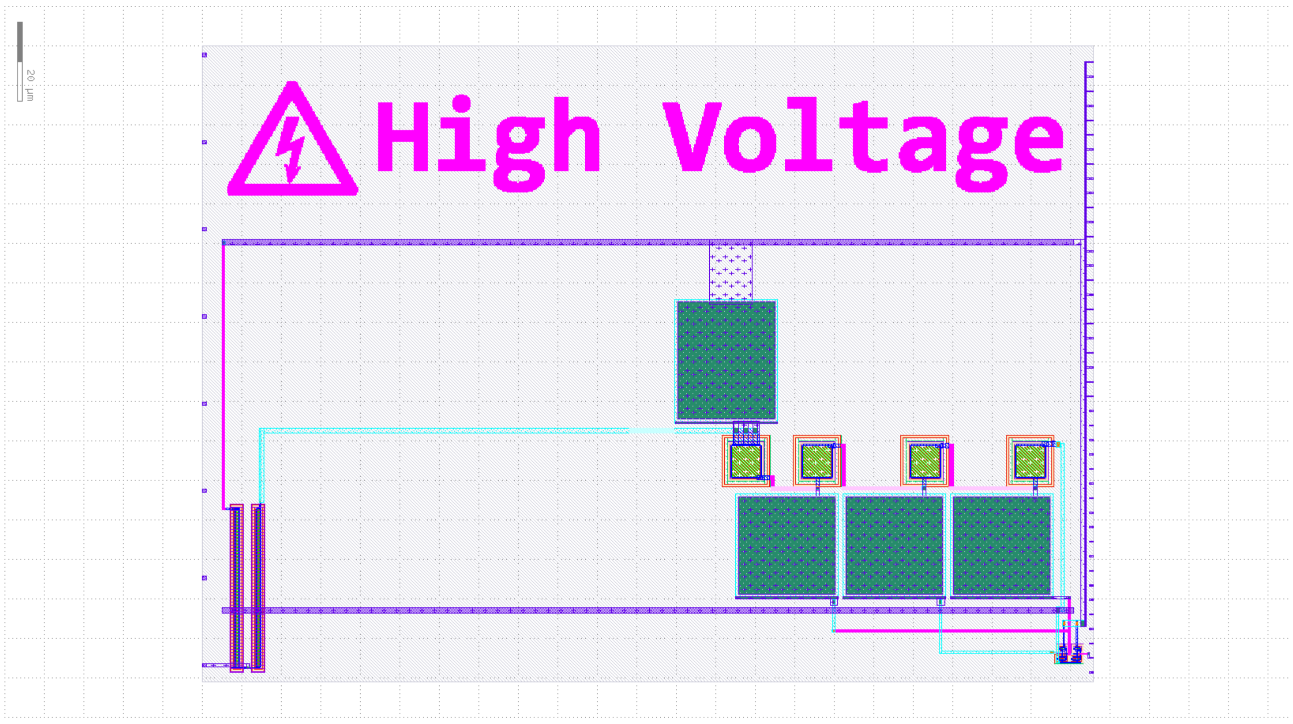


Figure 22: Project layout

ua#	analog#	Description
0	8	vout_div

DDA solver for van der Pol oscillator [270]

- Author: Adonai Cruz
- Description: Digital differential analyzer (DDA) solver for the van der Pol oscillator using posit (16,1)
- GitHub repository
- HDL project
- Mux address: 270
- Extra docs
- Clock: 0 Hz

How it works

The DDA core expects to receive via SPI port the parameter for the van der Pol oscillator encoded in posit (16,1) padded with 2 zero bytes to compose a 32-bit word. When an SPI message is started by the master (SPI CS pin low) the integrators are clocked and solutions for both state variables, X and Y, are transmitted back serially via SPI as a single 32-bit word for each time step with the 16 bits MSB encoding X and the 16 bits 16 bits LSB encoding Y. Simulation can be stopped by stopping communication via SPI.

How to test

In order to test chip reset the chip (RST_N active low) and start a duplex SPI communication transmitting 32-bit word with the van der Pol parameter μ encoded in posit (16,1) using the 16 bits LSB of the 32-bit word (padded with zeros). A controller software to interface with the chip via FTDI FT232H using SPI is available at https://github.com/adonairc/tt07-dda-van-der-pol

External hardware

This projects uses the top row pins of PMOD for SPI:

uio[0] - CS
uio[1] - MOSI
uio[2] - MISO
uio[3] - SCK

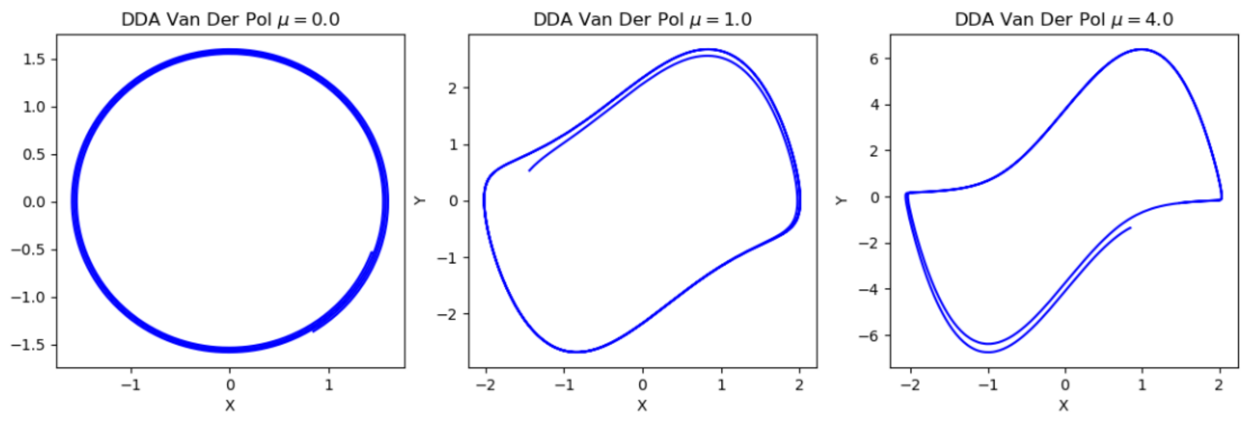


Figure 23: DDA solutions of the van der Pol oscillator showing the evolution of the limit cycle for different values of μ

Pinout

#	Input	Output	Bidirectional
0			SPI CS
1			SPI MOSI
2			SPI MISO
3			SPI CLK
4			
5			
6			
7			

Analog Test Circuit ITS 2 [271]

- Author: A. N Irfansyah, Raditya Eka, Yohanes Stefanus
- Description: PLL parts (VCO and phase detector)
- GitHub repository
- Analog project
- Mux address: 271
- Extra docs
- Clock: 0 Hz

How it works

This tinytapeout submission consists of:

1. A VCO based on transmission gates with additional on-chip capacitors to further linearize the response.
2. A phase detector and VCO to form parts of a PLL.

How to test

Pinouts:

Analog pins:

ua[0] - VCO #1 output

ua[1] - VCO#1 VCONT- / Phase Detector (PLL) ref

ua[2] - VCO#1 VCONT+ / Phase Detector (PLL) input / PLL feedback

ua[3] - PLL VCO output

ua[4] - PLL Filter (n)

ua[5] - PLL Filter (p)

External hardware

To test, typical experimental setup to test pll, or vco, would be required.

Pinout

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

Analog pins

ua#	analog#	Description
0	11	vco1_out
1	6	vco1_in-
2	10	vco1_in+
3	7	pllvco_out
4	9	pllfilter_n
5	8	pllfilter_p

VGA player [320]

- Author: shadow1229
- Description: 80 x 60 binary pixel video player with PCM/PWM audio playing feature
- GitHub repository
- HDL project
- Mux address: 320
- Extra docs
- Clock: 31500000 Hz

How it works

This project plays binary-colored 80px x 60px @ 24fps video recorded in SPI NOR flash, playing with 640px x 480px @72Hz VGA. Additionally, the project plays PCM or PWM Audio recorded in same flash chip. The input chooses type of audio(PCM/PWM), type of VGA PMOD, and the color of the video. Also, uio[7:2] is used for SPI communication and uio[1:0] is used for audio output. Finally, output is used as video output.

How to test

Hook up a VGA monitor to the outputs and provide a clock at 31.5 MHz. choose the type of audio output with input[0], and choose the type of VGA PMOD with the input[1]. color of pixels which turned off (which data of the pixel is 0) is selected with inputs[4:2] (2:R, 3:G, 4:B), and color of pixels which turned on (which data of the pixel is 1) is selected with inputs[7:5] (5:R, 6:G, 7:B).

Data structure of SPI flash chip:

1. Data address starts with 0x000000.
2. Each frame takes 65 x 128 bit, where each 128 bits are used for video/audio data for each line(640px x 8px). 2.1. Since the video uses 24fps, while the VGA uses 72Hz, each frame is shown three times in the VGA. 2.2. Thus, each line(=128 bits) are uses as following way: line[127:0] = {audio_0[15:0], audio_1[15:0], audio_2[15:0], video[79:0]}, where audio_i is audio data for the line in the (i+1)th iteration. 2.3 The first 5 lines in the frame are used for porch and vsync, which means video data in the line is ignored. However, audio data in the line still valid. Also, this is why each frame uses 65 x 128 bit and not 60 x 128 bit.

3. Due to limitation of data, maximum amount of 16131 frames will be supported. reaching 16132th frame will restart the project. (check overflow in tt_vga_player.v) For more infomation, check bit_dump_8060_wav.py in bad_apple folder.

External hardware

Audio - For PCM, using piezo on uio[1:0] would work. For PWM, external DAC like LTC2644 chip is needed (not tested though) Set input[0] low to use 74880Hz 1-bit PCM mode and high to 9360Hz 8-bit PWM mode.

VGA PMOD - you can use one of these VGA PMODs:

- https://github.com/mole99/tiny-vga
- https://github.com/TinyTapeout/tt-vga-clock-pmod Set input[1] low to use tiny-vga and high to use vga-clock

SPI flash (W25Q128JVSSIQ)

- https://www.adafruit.com/product/5634

Pinout

#	Input	Output	Bidirectional
0	74880Hz 1-bit PCM(0) / 9360Hz 8-bit PWM(1)	hsync / R1	sound n
1	Tiny VGA(0) / VGA clock PMOD(1)	vsync / G1	sound p
2	color_off(0) - R	B0 / B1	spi I/O 0 (W25Q128JVSSI
3	color_off(0) - G	B1 / VS	spi I/O 1
4	color_off(0) - B	G0 / R0	spi I/O 2
5	color_on(1) - R	G1 / G0	spi I/O 3
6	color_on(1) - G	R0 / B0	spi flash clock
7	color_on(1) - B	R1 / HS	spi chip select

Explorer [322]

- Author: sylefeb
- Description: none
- GitHub repository
- HDL project
- Mux address: 322
- Extra docs
- Clock: 33000000 Hz

How it works

This design performs a 2D 'voxel' raycasting of a terrain, implementing in actual hardware the 1992 Voxel Space algorithm used in the Comanche game.

The chip is designed in Silice, the source code is in the main repo.

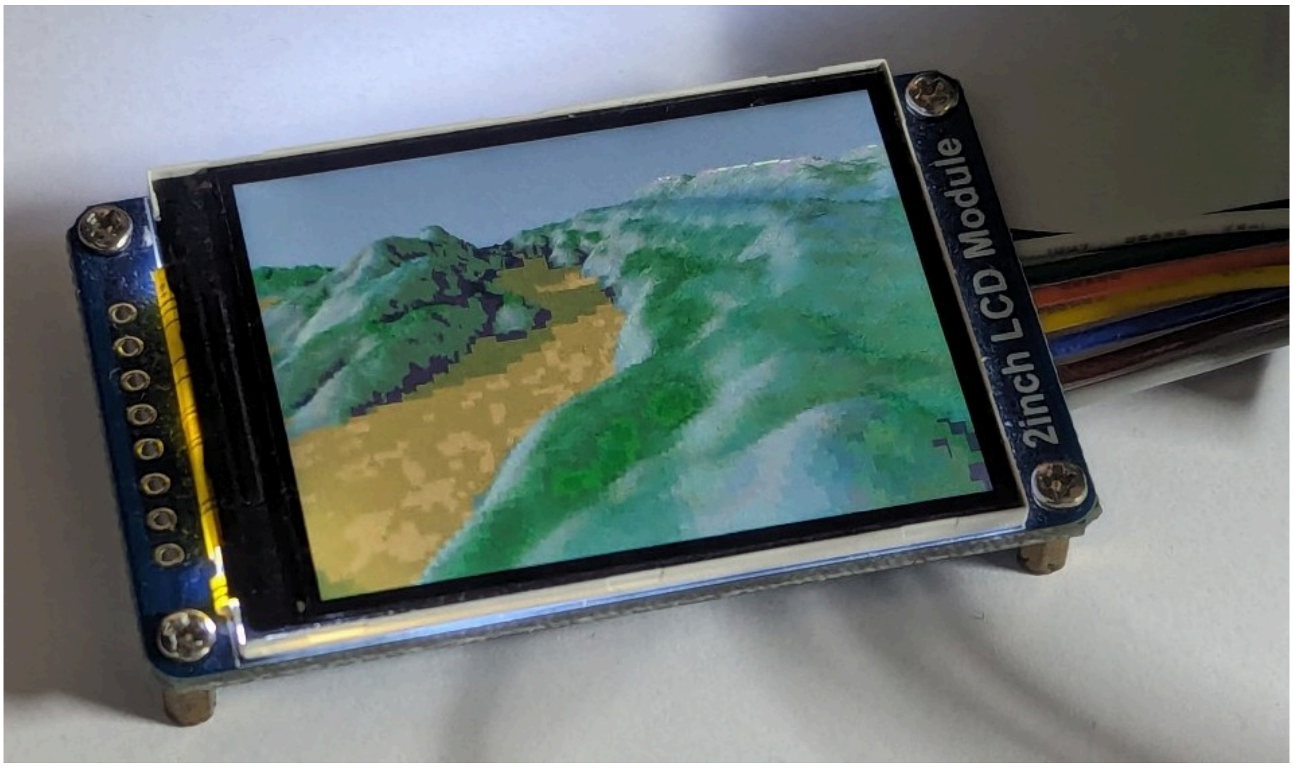


Figure 24: A terrain

How to test

A specific data file containing terrain data has to be uploaded to SPI-ram before this can run. The plan is to do that from the RP2040 of the PCB.

Another way to test is on a IceStick HX1K. Instructions coming soon!

External hardware

- QSPI PSRAM PMOD from machdyne
- 240x320 ST7789V screen

Pinout

#	Input	Output	Bidirectional
0	Button 0		QSPI ram, csn (output)
1	Button 1	SPI screen, clock	QSPI ram, io0 (bidir)
2	Button 2	SPI screen, csn	QSPI ram, io1 (bidir)
3	Button 3	SPI screen, dc	QSPI ram, clock (output)
4		SPI screen, mosi	QSPI ram, io2 (bidir)
5		SPI screen, resn	QSPI ram, io3 (bidir)
6			QSPI ram, bank select 0 (output)
7			QSPI ram, bank select 1 (output)

Real Time Motor Controller [324]

- Author: J. R. Petrus
- Description: Controls a stepper motor with precise timing between steps.
- GitHub repository
- HDL project
- Mux address: 324
- Extra docs
- Clock: 50000000 Hz

Introduction

The Real Time Motor Controller (RTMC) is designed to control a stepper motor such as the SEQ_28BYJ_48. The step_table is programmed with the coil positions for the motor, and each step advances the table_idx to the next coil position. The size of the step is programmable via step_size. The step_size may be positive or negative. The example motor supports step_size in [-2, -1, 1, 2]. The table_last set the limit for the table_idx before resetting to 0. It would be set to 7 for the example motor if abs(step_size) == 1 or 6 if abs(step_size) == 2.

The step control increments the table_idx + step_size. The run control continously increments table_idx, with a pause of step_delay cycles between increments.

SPI Peripheral Protocol

Control of the RTMC is accomplished via its SPI peripheral interface and simple read/write protocol.

SPI must be byte-oriented and send most significant bit first.

SPI mode must be CPOL=0, CPHA=0.

1-byte OPCODE: NOP=0, RD=1, WR=2.

1-byte ADDRESS: Select the register

2-byte DATA: Transfer in two bytes, MSB first.

1-byte RESULT: BUSY=0, ACK=1, ACK_DATA=2

The SPI peripheral *should* operate at up to 1/2 the core clock frequency. My target is 50 MHz core and 25 MHz SCK.

Write Operation Tx: WR, ADDR, DATA0, DATA1

Rx: Loop reading 1-byte RESULT until ACK seen.

Read Operation Tx: RD, ADDR

Rx: Loop reading 1-byte RESULT until ACK_DATA seen.

Rx: DATA0, DATA1

Memory Map

16-bit Register	Offset	Access	Description
id	0x00	R	ID: {version, idcode}
gpio	0x01	RW	GPIO: {mc_oe[7:0], gpo[3:0], gpi[3:0]}
step_ctrl	0x02	RW	Step Control: {run, step, reserved[4:0], table_last[3:0], step_size[4:0]}
step_stat	0x03	R	Step Status: {reserved[7"0], state[3:0], table_idx[3:0]}
step_delay0	0x04	RW	Step Delay: Most significant 16 bits, unsigned.
step_delay1	0x05	RW	Step Delay: Least significant 16 bits, unsigned.
step_count0	0x06	R, WC	Step Count: Most significant 16 bits, signed. WC = write-to-clear
step_count1	0x07	R, WC	Step Count: Least significant 16 bits, signed.
delay_count0	0x08	R, WC	Delay Count: Most significant 16 bits, unsigned.
delay_count1	0x09	R, WC	Delay Count: Least significant 16 bits, unsigned.
step_table[0]	0x10	RW	Motor State 0: 8-bits mapped to uio[7:0].
…			…
step_table[15]	0x1F	RW	Motor State 15: 8-bits mapped to uio[7:0].

Pinout

Inputs ui[0]: General Purpose Input gpi[0]

ui[1]: General Purpose Input gpi[1]

ui[2]: General Purpose Input gpi[2]

ui[3]: General Purpose Input gpi[3]

ui[4]: SPI0.cs

ui[5]: SPI0.sck

ui[6]: SPI0.tx

ui[7]: Connected to uo[6]

Outputs uo[0]: General Purpose Output gpo[0]

uo[1]: General Purpose Output gpo[1]

uo[2]: General Purpose Output gpo[2]

uo[3]: General Purpose Output gpo[3]

uo[4]: Connected to ⁀uio_in

uo[5]: Connected to ui[7]

uo[6]: Connected to ena

uo[7]: SPI0.rx

Bidirectional pins uio[0]: Motor Control mc[0]

uio[1]: Motor Control mc[1]

uio[2]: Motor Control mc[2]

uio[3]: Motor Control mc[3]

uio[4]: Motor Control mc[4]

uio[5]: Motor Control mc[5]

uio[6]: Motor Control mc[6]

uio[7]: Motor Control mc[7]

How to test

Connect up the external hardware, program the registers, and write a 1 to the run bit. MicroPython code in the works, but not until the chips come back.

External hardware

Assuming use of the TT Demo board.

Utilize the RP2040 SPI0 in controller mode to communicate with the RTMC.

Connect uio[3:0] to a SEQ_28BYJ_48 motor + ULN2003 Driver.

uio[7:4] could optionally be connected to a second motor driven in tandem.

Pinout

#	Input	Output	Bidirectional
0	General Purpose Input gpi[0]	General Purpose Output gpo[0]	Motor Control mc[0]
1	General Purpose Input gpi[1]	General Purpose Output gpo[1]	Motor Control mc[1]
2	General Purpose Input gpi[2]	General Purpose Output gpo[2]	Motor Control mc[2]
3	General Purpose Input gpi[3]	General Purpose Output gpo[3]	Motor Control mc[3]
4	SPI0.cs	Connected to ^uio_in	Motor Control mc[4]
5	SPI0.sck	Connected to ui[7]	Motor Control mc[5]
6	SPI0.tx	Connected to ena	Motor Control mc[6]
7	Connected to uo[6]	SPI0.rx	Motor Control mc[7]

QOA Decoder [326]

- Author: Nicholas West
- Description: A decoder for the QOA audio format.
- GitHub repository
- HDL project
- Mux address: 326
- Extra docs
- Clock: 30303030 Hz

How it works

This chip is for decoding the QOA audio format, which is designed to be a simple, fast format for 16 bit PCM audio data. The specification is one page, and is availible at qoaformat.org. The chip communicates through an SPI slave mode 0 interface to a controller chip, which handles the file interface and all adjecent functions. The chip only handles decoding samples into their 16 bit uncompressed versions.

Block diagram The chip itself consists of two main parts, an SPI interface for communication, and the decoder itself. The decoder contains a parser for the SPI data, the LMS predictor/updater at the heart of the QOA format, and the history/weights for the LMS predictor. For die area savings, we use a sequential multiplier in the LMS predictor, and save on the expensive dequantizing computations by using a precalculated table from the reference code on Github. We save space further by only saving half the values, since every odd index is just the negative counterpart of the previous value, and we can just flip the sign.

The decoder itself The decoder has three main parts, registers for the LMS history and weights, a parser for handling SPI data, and the QOA decoder in the parser.

First off, whenever `data_rdy` is pulsed, the main state machine in the parser decodes `spi_in` into either a hist/weights fill instruction, a sample decode instruction, or a sample send instruction.

- If the instruction is a hist/weights fill, it takes the next 2 bytes (i.e. 2 `data_rdy` pulses) and puts them into the history or weights registers specified by the index in the instruction.
- If the instruction is a sample send request, the parser will set the `spi_out` register to the upper bit of sample, wait for it to transmit (8 SPI clock cycles, so a `data_rdy` pulse), then set it to the low byte and finish the transmission.

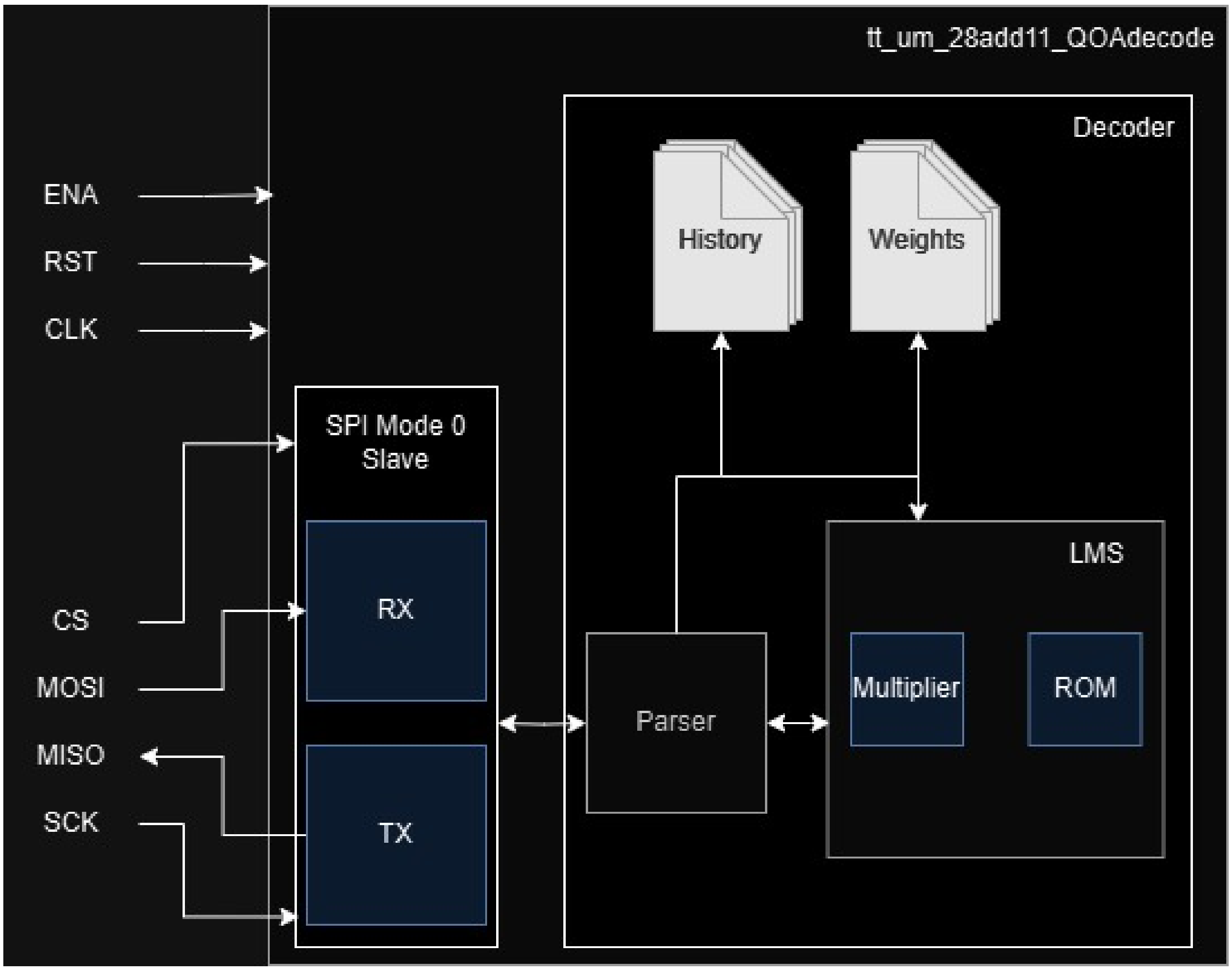


Figure 25: A diagram showing the internal structure of the chip

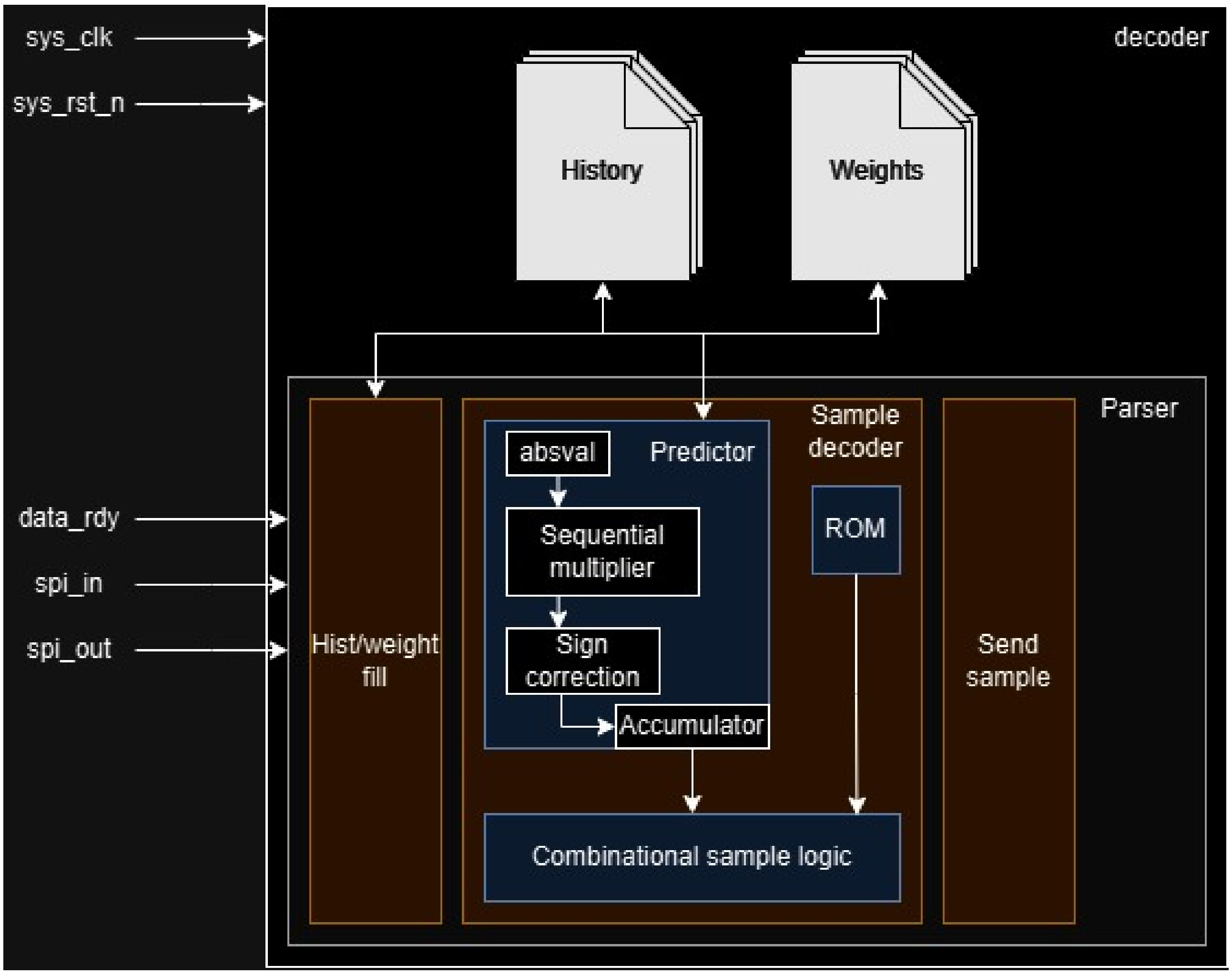


Figure 26: A block diagram of the decoder itself, showing the components and how they are linked

- Finally, if it is a sample decode instruction, it will iteratively multiply the history and weights values using a sequential multiplier, adding them to an accumulator. It then uses combinational logic and the ROM to calculate the final sample. This is then used to update history, weights, and is sent if a sample send request is recived.

How to test

Connect the chip to a mode 0 SPI master, with a clock rate at least 6x slower than the chip clock. Then, fill the LMS history and weights, by using the following instruction: | bit[7] | bit[6] | bit[5] | bit[4] | bit[3] | bit[2] | bit[1] | bit[0] | | —— | —— | —— | —— | —— | —— | —— | —— | | 0 | | | | Adress[1] | Adress[0] | BankSel | 0 |

BankSel chooses between history and weights, 1 for weights and 0 for history. Adress is just which of the 4 values to fill, as specified by QOA. The next two bytes are the data to fill the history or weights with, MSB first. If you want to then send a sample, the following instruction is used: | bit[7] | bit[6] | bit[5] | bit[4] | bit[3] | bit[2] | bit[1] | bit[0] | | —— | —— | —— | —— | —— | —— | —— | —— | | sf_quant[3] | sf_quant[2] | sf_quant[1] | sf_quant[0] | qr[2] | qr[1] | qr[0] | 1 |

qr and sf_quant are exactly as they are in the QOA specification, with this chip decoding sample by sample.

After sending the sample, wait 40 chip clock cycles, then request the sample with the following instruction: | bit[7] | bit[6] | bit[5] | bit[4] | bit[3] | bit[2] | bit[1] | bit[0] | | —— | —— | —— | —— | —— | —— | —— | —— | | 1 | | | | | | | 0 |

Once you send that instruction, the next two bytes sent by the chip will be the decoded sample, MSB first. While you are reciving the sample, you can send any data, but it will be ignored. The chip will send unknown data when the instruction is not used.

On the testbench, it can calculate one sample every 5680ns, SPI transfer included, at a clock speed of 50MHz and an SPI frequency of 8MHz, which should be achivable on hardware. This can likely be improved by using a custom/different approach to SPI, since my test bench leaves relatively long periods of inactivity. The current speed results in a max of 176,056 samples per second, more than enough for real time audio streaming.

Eventually I will get arould to writing code for the interface on my Github, please look back there for updates.

External hardware

Since this is a co-processor for the QOA format, a seperate microcontroller is required to interface with it. Since I am used to the RP2040 and it is included on the Tiny Tapeout PCB, I will likely provide software for it on my Github in the future. I plan to take a streaming approach with this software, so a PC supporting USB will also be needed to send, store, and convert the files.

Pinout

#	Input	Output	Bidirectional
0			CS
1			MOSI
2			MISO
3			SCK
4			
5			
6			
7			

underserved [328]

- Author: Olof Kindgren
- Description: The award-winning SERV, the world's smallest RISC-V CPU. Now on Tiny Tapeout!
- GitHub repository
- HDL project
- Mux address: 328
- Extra docs
- Clock: 20000000 Hz

How it works

When the system boots up, it will start accessing the SPI bus to set up a connected SPI Flash memory in XIP mode and start executing instructions from there. The GPIO can be used to output data, e.g. as a bitbanged UART.

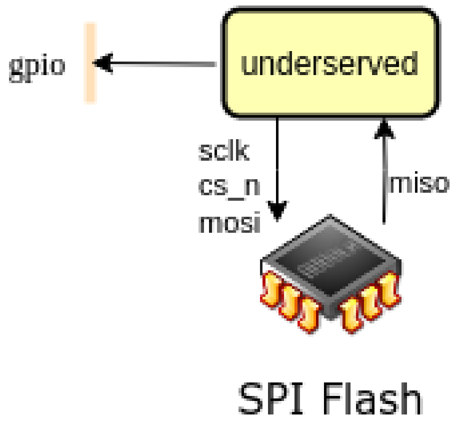


Figure 27: Environment

How to test

The testbench contains a model of an SPI Flash. A program in Verilog Hex format can be preloaded into the Flash model.

Underserved can easiest be run locally using FuseSoC.

Install FuseSoC

```
pip install fusesoc
```

Create and enter a new workspace

```
mkdir workspace && cd workspace
```

Register underserved as a library in the workspace

```
fusesoc library add underserved /path/to/prince
```

...if repo is available locally or... ...to get the upstream repo

```
fusesoc library add underserved https://github.com/olofk/underserved
```

Show available cores in workspace (probally just underserved for now if you haven't added other libraries)

```
fusesoc core list
```

Show info about underserved

```
fusesoc core show underserved
```

Run linting (static code checks) using Verilator

```
fusesoc run --target=lint underserved
```

Run underserved testbench

```
fusesoc run --target=sim underserved
```

Run with modelsim instead of default tool (icarus)

```
fusesoc run --target=sim underserved --tool=modelsim
```

External hardware

Expects a compatible SPI Flash. The XIP controller was stolen from PicoSoC which also contains some info about compatible SPI Flash components.

Pinout

#	Input	Output	Bidirectional
0		gpio0	
1		gpio1	
2		gpio2	
3		gpio3	
4		gpio4	
5		sclk	
6		cs_n	
7		mosi	miso

co processor for precision farming [330]

- Author: MITS ECE
- Description: The processor will detect the deviation in sensor data and the sensor fault
- GitHub repository
- HDL project
- Mux address: 330
- Extra docs
- Clock: 0 Hz

How it works

The processor will read the datas from the four sensors sequentially and analyse whether any deviation has been occoured with respect to the previous data and provide a warning signal also it continuously checks the senor datas and identify any fault has been occured and provides another warning signal with a signal providing the sensor identification.

How to test

If the sensor identifier data is 00 which means it is sensor1 and input data is 10000001 and this compared with the previously stored data which may be 10000100 for example ,then there is a deviation and the processor will provide output as 1 and the bidirectional as 00.

External hardware

8 bit ADC is needed to convert the sensor data

Pinout

#	Input	Output	Bidirectional
0	Input data from the sensors	Deviation detector	Sensor identifier
1	Input data from the sensors	Falut warning	Sensor identifier
2	Input data from the sensors	Falut warning	
3	Input data from the sensors	Falut warning	
4	Input data from the sensors	Sensor identifier	

#	Input	Output	Bidirectional
5	Input data from the sensors	Sensor identifier	
6	Input data from the sensors	Sensor identifier	
7	Input data from the sensors	Sensor identifier	

Delay Line Time Multiplexed NAND Gate [332]

- Author: Frans Skarman (TheZoq2)
- Description: A time multiplexed nand gate powered by a giant shift reigster
- GitHub repository
- HDL project
- Mux address: 332
- Extra docs
- Clock: 0 Hz

How it works

This is a single NAND gate that is fed its inputs from, and writes its results to a giant shift register. With this, we can achieve the ultimate time/space tradeoff, a single nand gate able to emulate quite complex logic.

How to test

Undecided, we are working on a yosys backend to generate "programs" for this which you can then run by driving the inputs

External hardware

You need an FPGA or similar to drive the inputs at high enough precision to feed "instructions" to the device.

Pinout

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

Neural Network dinamic [334]

- Author: Kevin Gajardo, David Tapia
- Description: One line description
- GitHub repository
- HDL project
- Mux address: 334
- Extra docs
- Clock: 66000000 Hz

How it works

The project consists of a neural network of 4 (parameterizable and reusable) neurons, thanks to control signals.

From an 8-bit input, the inputs will be introduced into a reusable neural network of 4 neurons. Through a shift register, 4 different inputs are captured. Furthermore, thanks to a state machine, the parameters associated with each neuron are obtained: 4 weights and 1 bias, in total 20 parameters per network layer.

State changes are made using a binary signal, where the input data and neuron parameters are received and then the neurons are fed back their outputs to the next layer. To observe the network's output one need to bring the state machine to the first state using the pin "Finished", then, in the next 4 clock cycles the outputs of the neurons 3 to 0 will be shown on the output at the same time some new external inputs can be introduced to a new neural network without the need for a reset

State	Description
State_IN	The inputs of the neurons are found entering and the outputs are shown.
State_BUFF	Neuron inputs are maintained while network parameters are entered
State_OUT	Feedback of neurons with their previous result

Below is the structure for inputting the neurons entries:

Node A corresponds to the output of the state machine, and node B to the parameters for each neuron.

Regarding the parameters, they are fed in sequence, from neuron 3 to neuron 0. And the weights correspond to powers of 2, so, if w00=3, the input0 to the neuron0 will be multiplied by 2^3.

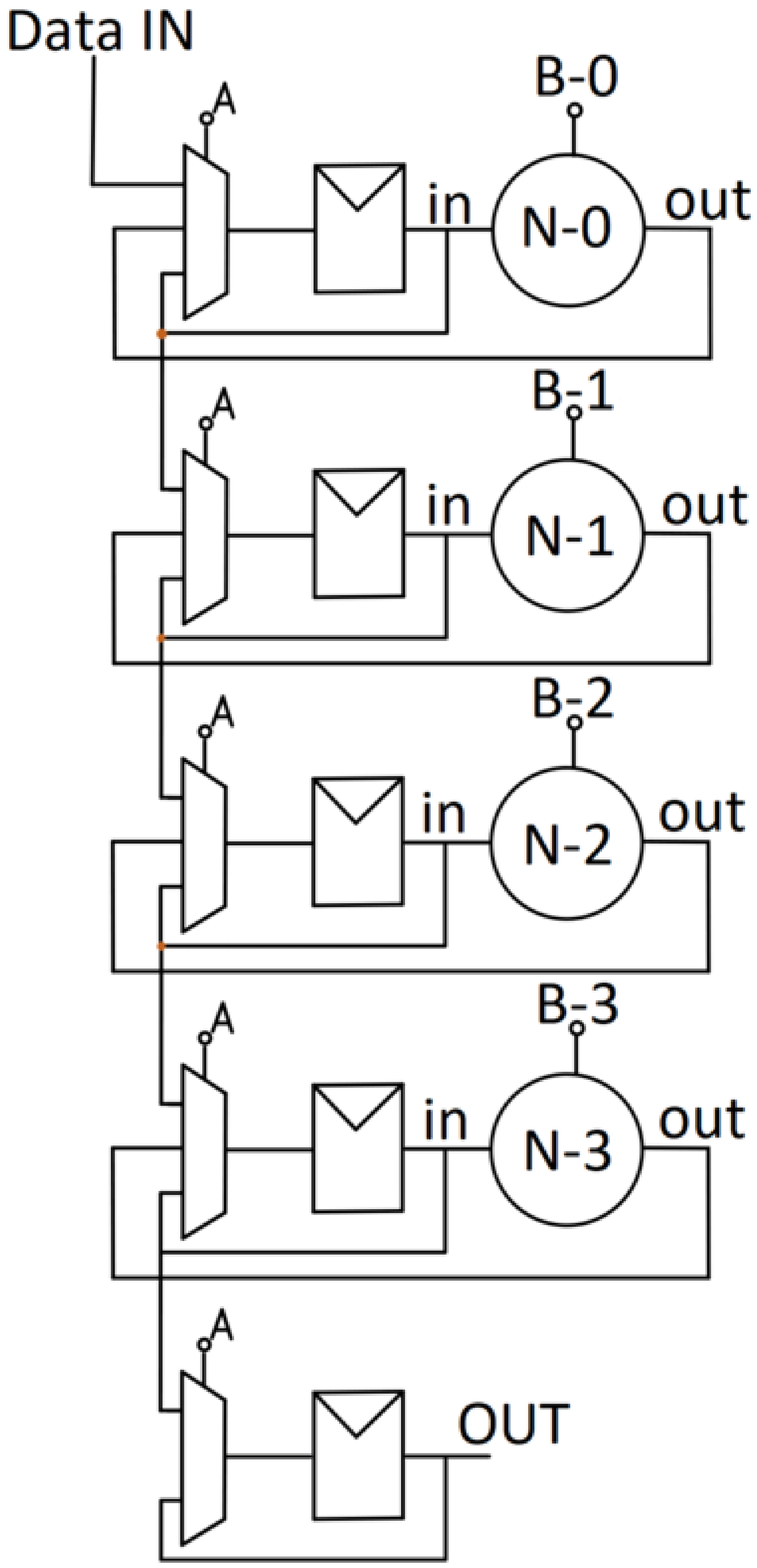


Figure 28: ChipUSM1 (1)

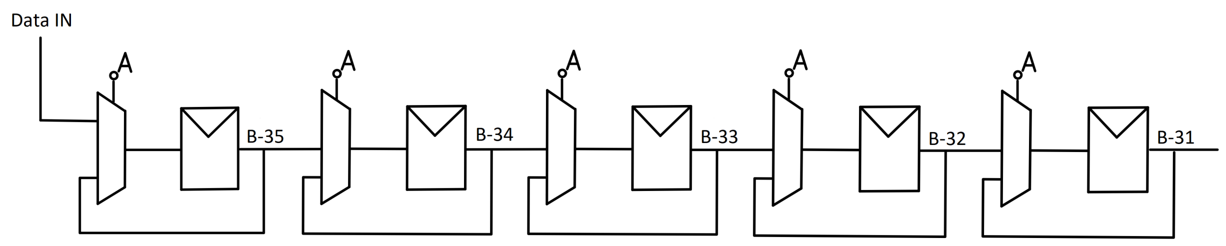


Figure 29: ChipUSM2

How to test

The input signals must be coordinated to achieve correct testing, where the following order must be followed to enter the inputs and parameters considering multiple layers.

in_3 > in_2 > in_1 > in_0 > b3 > w33 > w32 > w31 > w30 > b2 > w23 > w22 > w21 > w20 > b1 > w13 > w12 > w11 > w10 > b0 > w03 > w02 > w01 > w00

Each entry must be maintained for 2 clk, to be captured on the rising edge and thus there is displacement with the shift registers.

External hardware

An FPGA is recomended in to perform the tests and feed the weights correctly. Also, the weights must be trained first in some other system, e.g. in a pc using Python or Matlab.

Pinout

#	Input	Output	Bidirectional
0	data_in[0]	data_out[0]	selector[0]
1	data_in[1]	data_out[1]	selector[1]
2	data_in[2]	data_out[2]	selector_out[0]
3	data_in[3]	data_out[3]	selector_out[1]
4	data_in[4]	data_out[4]	
5	data_in[5]	data_out[5]	
6	data_in[6]	data_out[6]	
7	data_in[7]	data_out[7]	

Chess [452]

- Author: Hannah Ravensloft
- Description: chess move generator
- GitHub repository
- HDL project
- Mux address: 452
- Extra docs
- Clock: 0 Hz

A Reimplementation of Belle's Move Generator

In honour of about 30 years since the creation of Deep Blue, I decided to recreate the move generation system that it uses, dating back to Belle from 1983.

How it works Internally, there is a 256-bit chessboard (4 bits per square), along with a 64-bit square-enable mask.

Each square "transmits" attacks to its neighbour squares, which either propagate attacks along empty squares, or generate their own. These attacks are processed by "receivers", which produce a priority level based on opcode and the piece on that square. The priority levels go through an arbitration network, which chooses the most promising square, which gets output from the chip.

Due to the space limitations present on Tiny Tapeout, though, there are some very notable design differences. The original design calculates all 8x8 squares in a single cycle, handling both positive and negative directions.

Opcodes

> To be finalised.

The chip has 16 input bits and 8 output bits.

bit pattern	command	description
`1111 __ss ssss ____`	`FIND-SRC`	output the least-valuable enabled attacker of square `s`.

bit pattern	command	description
`1110 ____ ____ ____`	`FIND-DST`	output the most-valuable enabled piece on the board.
`1101 __ss ssss ___v`	`ENABLE-SET`	set the square-enable bit of square `s` to `v`.
`1100 ____ ____ ____`	`ENABLE-ALL`	set all square-enable bits.
`1011 __ss ssss vvvv`	`SQUARE-SET`	set the chessboard on square `s` to have value `v`.
`1010 ____ ____ ____`	`ROTATE`	rotate the chessboard 180 degrees.
`1001 ____ ____ ____`	`FLIP-COLOR`	flip the colours of all pieces on the chessboard, so that friendly becomes enemy and vice versa.
`1000 ____ ____ ____`	`ENABLE-US`	set the square-enable bits of all friendly pieces.

How to test Use the test suite.

External hardware The RP2040 microprocessor in the dev board is intended to be used to drive the move generator, as there isn't enough room in the chip to do it itself.

Pinout

#	Input	Output	Bidirectional
0	Address bit 0	Square out bit 0	Data in bit 0
1	Address bit 1	Square out bit 1	Data in bit 1
2	Address bit 2	Square out bit 2	Data in bit 2

#	Input	Output	Bidirectional
3	Address bit 3	Square out bit 3	Data in bit 3
4	Address bit 4	Square out bit 4	Data in bit 4
5	Address bit 5	Square out bit 5	Data in bit 5
6	Address bit 6	End iteration bit	Data in bit 6
7	Address bit 7 (valid)	Illegal position bit	Data in bit 7

Mastermind [458]

- Author: Tom Gurrieri and Anthony Gurrieri
- Description: Play the game: Mastermind
- GitHub repository
- HDL project
- Mux address: 458
- Extra docs
- Clock: 20000000 Hz

How it works

The goal of Mastermind is to guess the correct color combination with the minimal number of inputs. Guesses are made using specific inputs which represent different colors.

How to test

To randomly pick the hidden color answer, press any push button once.

After the answer is chosen, the user can start to make guesses. No color can be repeated in a guess (and it won't be repeated in the answer either). A guess consists of 4 button pushes to select 4 colors.

Once a guess is selected, the LEDs will show the guide code. The four LEDs closest to the left will show how many colors are correct (but not necessarily in the correct place), and the four LEDs closes to the right will show how many colors are in the correct position. Using this hint, the user can make an educated guess as to what their next guess will be.

If 10 rounds go by and the user doesn't get the correct combination, the LEDs will flash on and off to indicate a loss. If the user guesses the correct color combination, the light is shifted to the left by a clock to indicate a win.

External hardware

The external hardware that may be useful in playing the game are two Pmod BTNs (410-077) for a total of 8 push buttons (for color inputs), a Pmod8LD (410-163) for 8 output LEDs, and a 2x6 pin to dual 6-pin Pmod splitter cable (240-110) to use both push button Pmods.. All can be purchased on digilent.com

Pinout

#	Input	Output	Bidirectional
0	Red	Correct Position 1	unused
1	Yellow	Correct Position 2	unused
2	Green	Correct Position 3	unused
3	Blue	Correct Position 4	unused
4	Orange	Correct Color 1	unused
5	Black	Correct Color 2	unused
6	White	Correct Color 3	unused
7	Purple	Correct Color 4	unused

TinyWSPR [462]

- Author: asinghani
- Description: Tiny WSPR Beacon
- GitHub repository
- HDL project
- Mux address: 462
- Extra docs
- Clock: 0 Hz

How it works

Tiny WSPR Beacon for transmitting Weak Signal Propagation Reporter radio signals. Recommended for 20m and 40m bands.

How to test

Use the "config in" pins to input one byte of config data at a time. Input the 6-letter callsign as ASCII characters, padded as is standard for WSPR. Input the 4-letter grid square with the second and third character swapped (i.e. FN01 -> F0N1). Input the power level as a single byte in dBm. It must end in a 0, 3, or 7 for proper decoding. Input the symbol time (the number of clock cycles in 0.6827 seconds) as a 4-byte, big-endian value. Input the divisor (`2^32 * (transmit_freq / clk_freq)`) as a 4-byte, big-endian value. Input the deviation (`2^32 * (1.4648 / clk_freq)`) as a 2-byte, big-endian value. Wait until exactly one second after an even-numbered UTC minute, then raise the RF start pin. It is recommended to do this using a GPS-based timekeeping source.

External hardware

An external lowpass filter to remove harmonics is absolutely required, as this design uses a square-wave transmitter that generates harmonics well outside the legal limit. An appropriately tuned antenna for the band being used is also necessary in order to transmit effectively.

Pinout

#	Input	Output	Bidirectional
0	Config valid	RF out	config[0]
1	Config start	RF out (mirror)	config[1]
2	RF start transmit	unused	config[2]
3	unused	unused	config[3]
4	unused	unused	config[4]
5	unused	unused	config[5]
6	unused	Debug out	config[6]
7	unused	Debug out	config[7]

Mini AIE: 2x2 CGRA with Ring-NoC [480]

- Author: Lyte Venn
- Description: A mini aie/coarse-grained reconfigurable array
- GitHub repository
- HDL project
- Mux address: 480
- Extra docs
- Clock: 10000000 Hz

How it works

This is a minimalistic coarse-grained reconfigurable array inspired by AMD AI engine architecture. The hardware design consists of

- A 2x2 array of compute tiles
- A simplified packet-switched network-on-chip (NoC) to connect the compute tiles
- Two interface tiles to connect the array to external memory and host

The packets loaded by interface tiles are routed through the NoC to the compute tiles. The compute tiles process the packets and send to next compute tile or interface tile. The packets are processed in a pipelined manner.

How to test

TBA

Pinout

#	Input	Output	Bidirectional
0	data_0	out_0	debug_out0
1	data_1	out_1	debug_out1
2	data_2	out_2	debug_out2
3	data_3	out_3	debug_out3
4	data_4	out_4	debug_out4
5	data_5	out_5	debug_out5
6	data_6	out_6	debug_out6
7	data_7	out_7	debug_out7

Field Programmable Resistor Network [482]

- Author: htfab
- Description: A few resistors and switches wired up in a matrix pattern.
- GitHub repository
- Analog project
- Mux address: 482
- Extra docs
- Clock: 0 Hz

How it works

A few resistors and switches are wired up in a matrix pattern. Switches are implemented as pass gates controlled by latches that keep the configuration. The network can be used as a makeshift DAC by controlling the "bitstream".

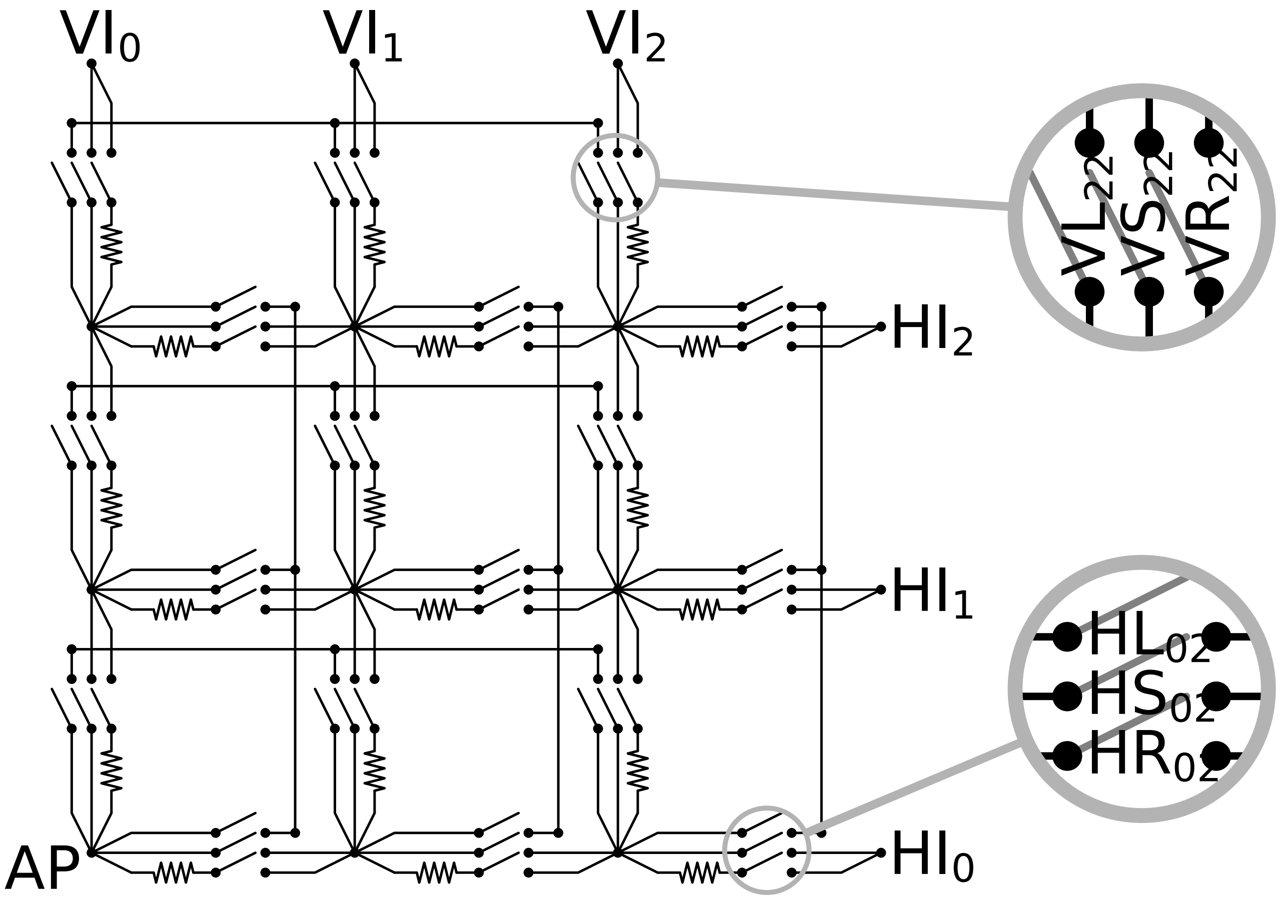


Figure 30: Circuit diagram

Matrix cells can be selected using the `H_GATE_i` and `V_GATE_j` inputs:

- `H_GATE_0` = `uio_in[5]`

- H_GATE_1 = uio_in[2]
- H_GATE_2 = ui_in[1]
- V_GATE_0 = uio_in[6]
- V_GATE_1 = uio_in[3]
- V_GATE_2 = uio_in[0]

When the inputs H_GATE_i and V_GATE_j are on, the latches in the cell *ij* become transparent and configure the pass gates as follows:

- HR_ij ← HD_RES = ui_in[4]
- HS_ij ← HD_SHORT = ui_in[3]
- HL_ij ← HD_LINE = ui_in[2]
- VR_ij ← VD_RES = ui_in[7]
- VS_ij ← VD_SHORT = ui_in[6]
- VL_ij ← VD_LINE = ui_in[5]

Once H_GATE_i or V_GATE_j is off again, the latches close and the pass gates keep their configuration. Thus a new cell with different *i* or *j* can be configured using the same inputs.

How to test

After the network is configured as above, manipulate the digital inputs H_INPUT_i and V_INPUT_j to apply 0 V or 1.8 V at the respective nodes of the network:

- HI_0 ← H_INPUT_0 = ui_in[0]
- HI_1 ← H_INPUT_1 = rst_n
- HI_2 ← H_INPUT_2 = clk
- VI_0 ← V_INPUT_0 = uio_in[7]
- VI_1 ← V_INPUT_1 = uio_in[4]
- VI_2 ← V_INPUT_2 = uio_in[1]

The voltage can be measured externally at the analog pin AP = ua[0].

External hardware

Multimeter (or microcontroller with ADC) to measure the output voltage.

Pinout

#	Input	Output	Bidirectional
0	H_INPUT_0		V_GATE_2
1	H_GATE_2		V_INPUT_2
2	HD_LINE		H_GATE_1
3	HD_SHORT		V_GATE_1
4	HD_RES		V_INPUT_1
5	VD_LINE		H_GATE_0
6	VD_SHORT		V_GATE_0
7	VD_RES		V_INPUT_0

Analog pins

ua#	analog#	Description
0	0	ANALOG_PIN

AY-8193 single channel DAC [484]

- Author: ReJ aka Renaldas Zioma
- Description: Logarithmic 4-bit DAC for AY-8193 sound generator
- GitHub repository
- Analog project
- Mux address: 484
- Extra docs
- Clock: 0 Hz

How it works

Current steering DAC.

How to test

Set one of the input / bidir pins to regulate current.

External hardware

Measure with osciloscope or use amplifier and speakers.

Pinout

#	Input	Output	Bidirectional
0			db
1	db		db
2	db		db
3	db		db
4	db		db
5	db		db
6	db		db
7	db		db

Analog pins

ua#	analog#	Description
0	5	output

dual oscillator [486]

- Author: Devin Atkin
- Description: 20 Mhz and 21Mhz Output Sine Waves
- GitHub repository
- Analog project
- Mux address: 486
- Extra docs
- Clock: 0 Hz

How it works

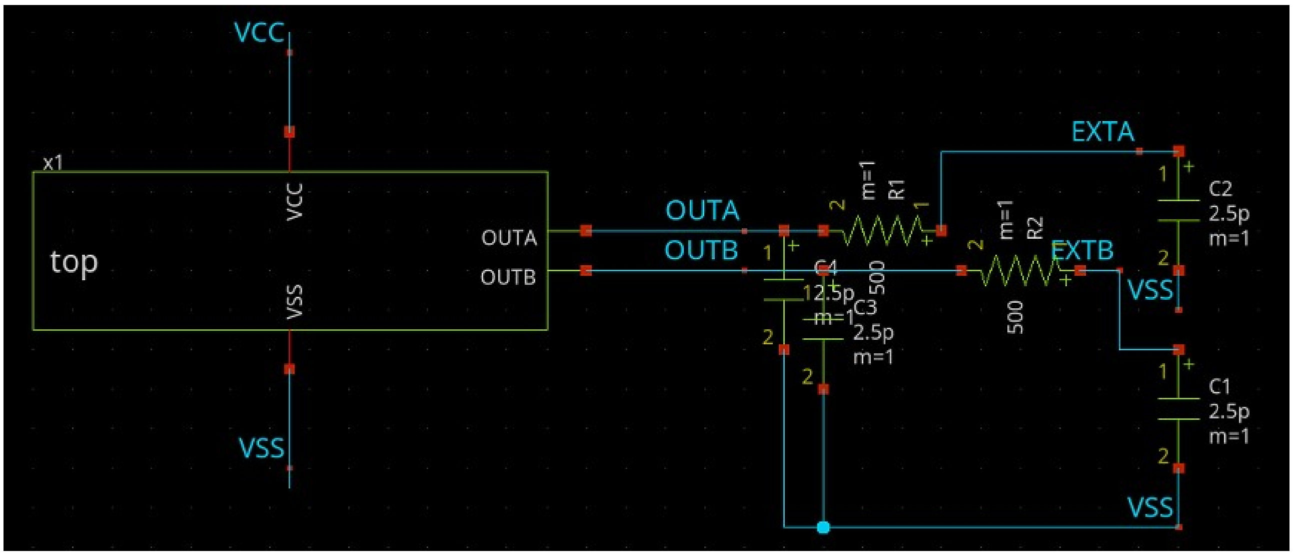


Figure 31: Top Level TB Schematic

This project should generate 2 sinusoidal oscillators centered around ~0.9V relative to the chips ground. The oscillators are based on a pair of ring oscillators so they should oscillate regardless of the chips corner; however, their frequency will vary together dramatically. Their final simulated output frequencies are slightly above their respective targets of 20MHz and 21Mhz with a slightly greater seperatation between them.

Inverter The inverter has dummy transistors added on both sides of it. This is less for some practical matching purpose and more as a way to deliberately add some additional capacitance into the circuit to slow the inverters down. A good potential improvement would be to add switches to the dummy transistors here to enable and disable the oscillators on command.

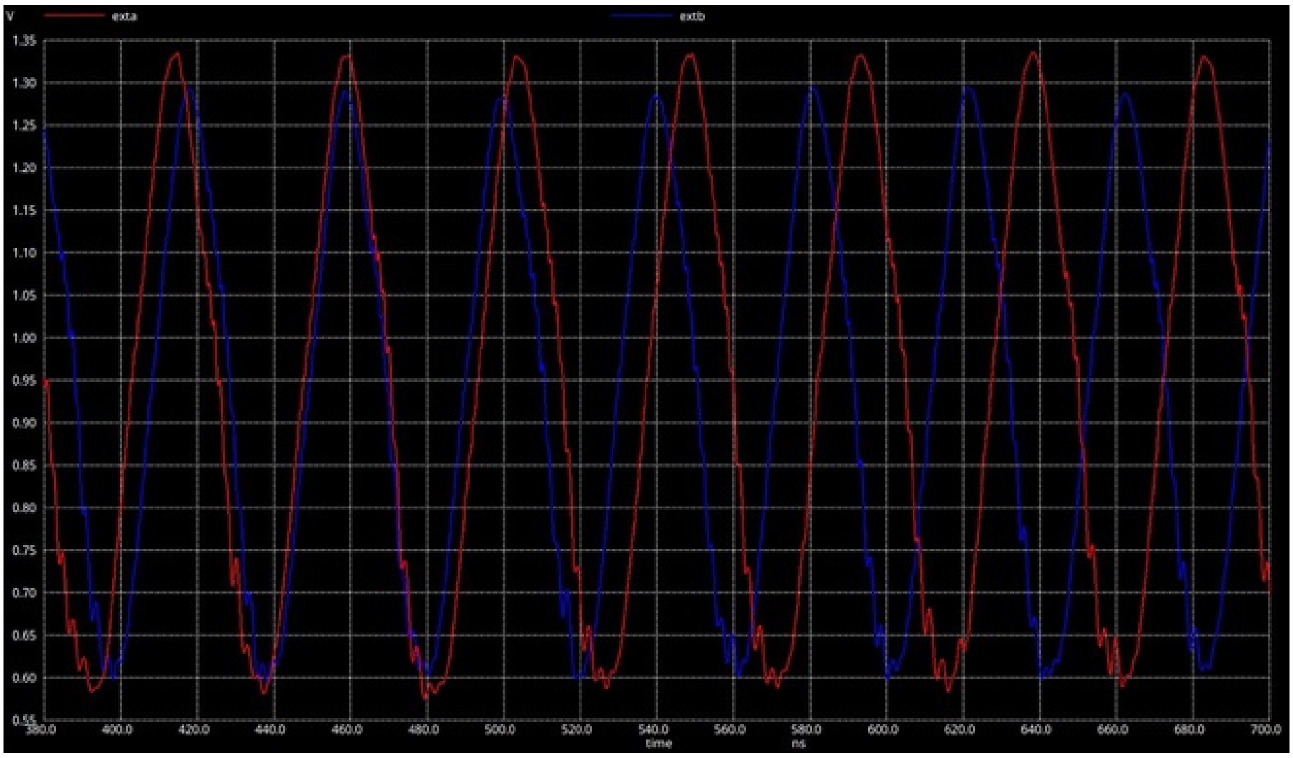


Figure 32: Sine Output

Op Amp The Op Amp is designed based on the work in opamp_1_design.md. It is a basic 2 stage voltage feedback op-amp. The final transistor sizings have been adjusted post design to help with simplifying layout. The designed frequency for the Op Amp is a unity gain frequency of ~50Mhz, with a load capacitance of 5pF to allow it to effectively drive the chip outputs.

Analog Buffer The analog buffer component is simply an op-amp configured as a voltage follower. This is mostly made into a seperate module to maintain consistency across followers as well as to minimize the amount of layout work required as the component gets reused multiple times.

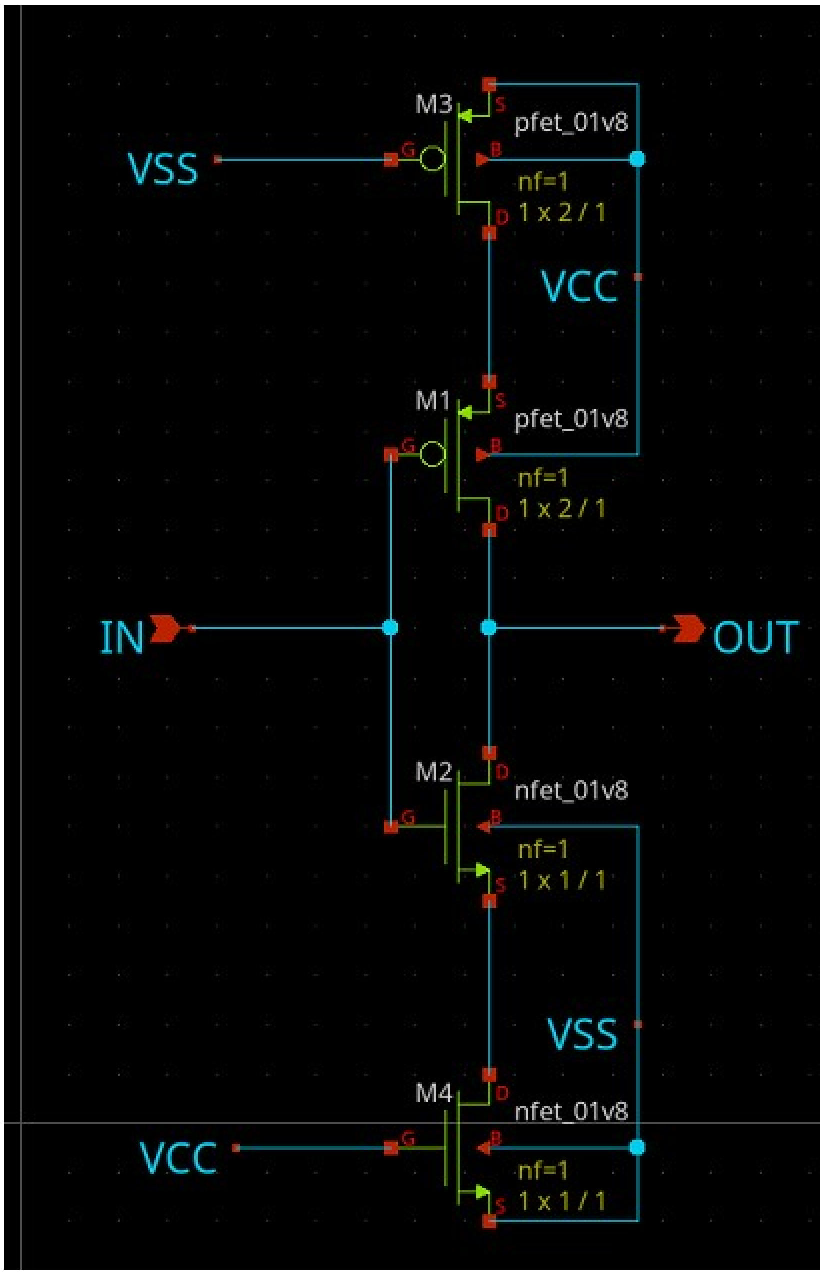


Figure 33: Inverter Schematic

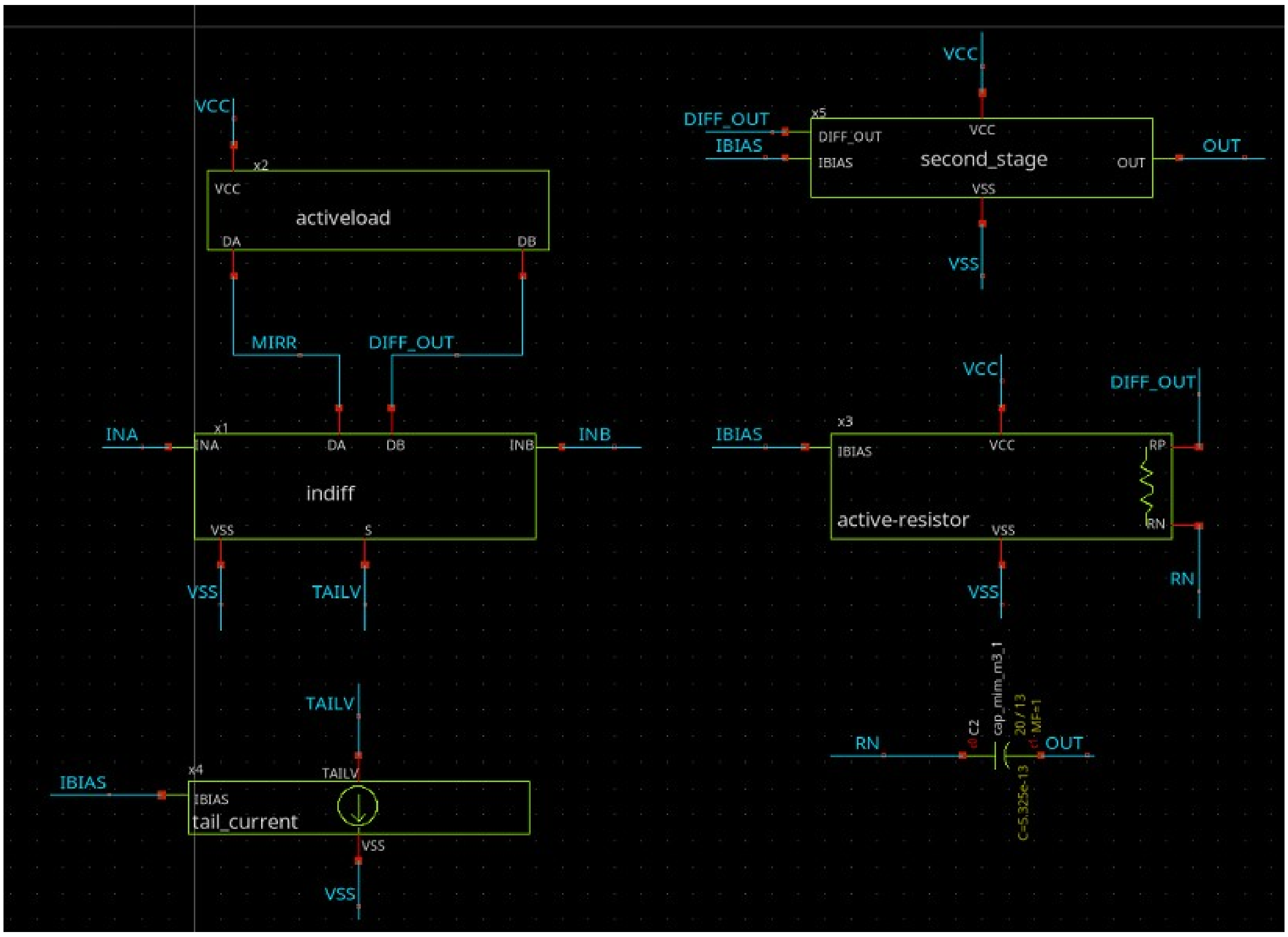
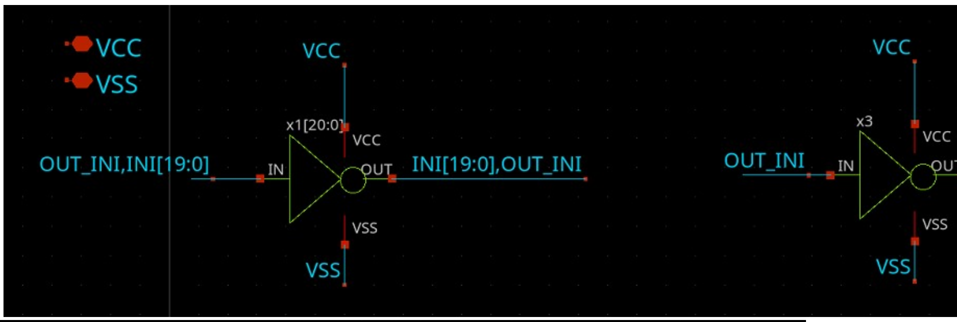
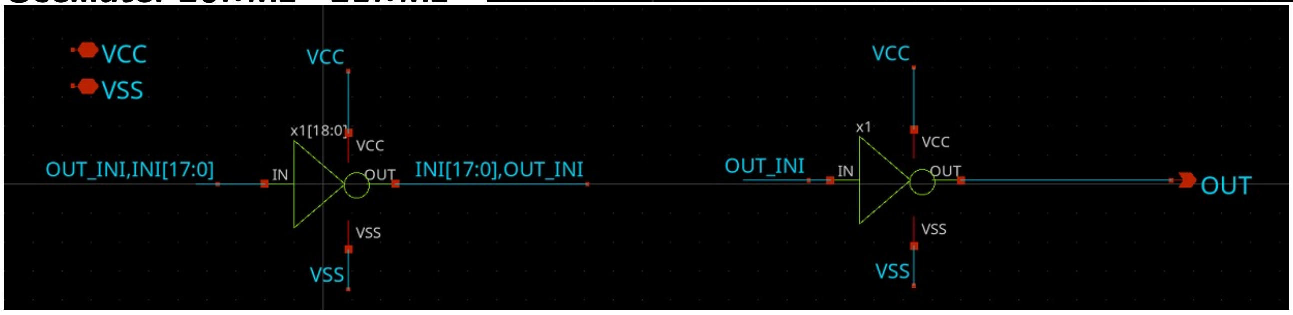


Figure 34: Op Amp Schematic



Oscillator 20Mhz - 21Mhz



The oscillators are basic ring oscillators with a single op-amp on the output. They are 21 and 19 inverters long respectively. These produce close to the desired output frequencies.

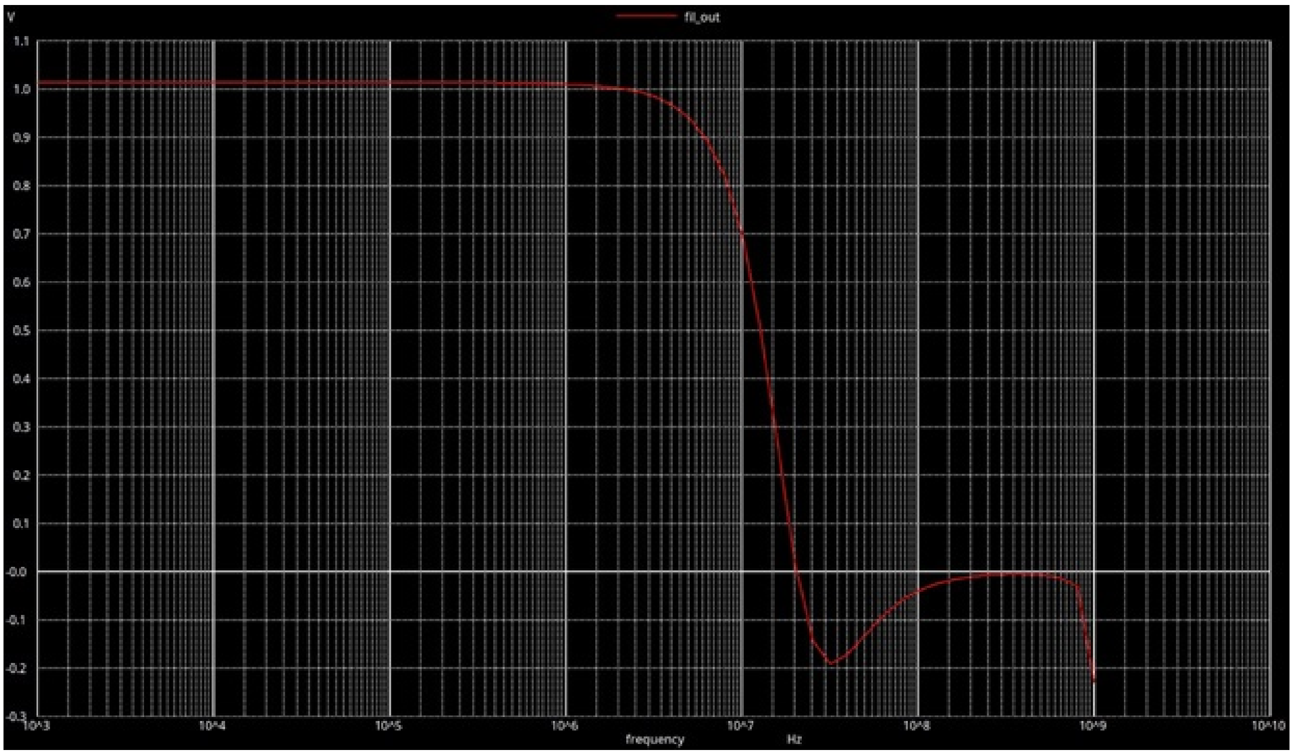


Figure 35: Filter Response

filter 22Mhz This is a second order butterworth filter intended to cut-off higher order components of the oscillators square wave output and provide a sinewave output. It keeps the DC component so as not to hit either of the chips rails.

Capacitor Array The Filter response relies on the capacitors matching relatively closely. Therefore they have been split into even sized segments and laid out according to centroid matching techniques so that on each filter they maintain relatively even responses.

How to test

Hook the outputs up to an oscilloscope or other device to measure the frequency generated.

External hardware

No external hardware is required for this module to function. The internal driving circuitry is designed around driving ~5pF, so it should be able to properly drive most high-impedance inputs to be used elsewhere in the circuit.

Pinout

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

Analog pins

ua#	analog#	Description
0	0	20Mhz Output
1	5	21Mhz Output

Analog TDC [488]

- Author: Luis Carlos Alvarez Simon
- Description: Analog part of a Time to Digital Convert
- GitHub repository
- Analog project
- Mux address: 488
- Extra docs
- Clock: 0 Hz

How it works

A Time to Digital Converter (TDC) is a circuit that measures the time interval between two events with high precision and converts this into a digital value. TDCs are fundamental components in various applications, including digital communications, radar systems, particle physics experiments, etc. The operation of a TDC can be classified into two types: direct and indirect measurement methods. Direct measurement capture time interval using a high-speed clock signal and a counter, while indirect methods employ in the first stage a Time-to-Voltage conversion followed by an analog-to-digital conversion or other similar blocks. The resolution of a TDC is determined by the frequency of the clock that drives the counter (Resolution=1/clock_frequency). Resolution in a TDC is defined as the smallest time difference that can be measured and distinguished by the converter. To achieve a balance between high resolution and wide dynamic range, it involves the use of coarse and fine resolution measurements within a single TDC architecture. Coarse resolution typically achieved using a counter that increments in every cycle of a clock. Fine resolution is employed to measure smaller segments of time within the coarse intervals, significantly improving the overall precision of the TDC often realized through techniques like time interpolation, delay line encoding, or the use of high-frequency clocks for a short duration. In a TDC using coarse and fine resolutions, the time interval between two events is first quantified in coarse units. Then, the fine resolution measures the fraction of the coarse unit that remains unaccounted for at the end of the interval. The total resolution is a combination of both resolutions, where the fine resolution refines the measurement within one coarse clock cycle. This combination offers a detailed time measurement while maintaining the capability to measure long intervals. The circuit present here comprises the analog part for the fine resolution of a Time to Digital Converter based on time interpolation. A width input pulse in the interval of 20 to 60 nanoseconds is converted to a periodic square signal with a period that is 90 times wider than the input pulse. So, we can use a counter with a low frequency clock to convert the time input pulse to digital format. For example, if we use a 25MHz clock frequency, we can measure the time of the input pulse with a resolution of 444ps.

In the Fig. 1 we can see the block diagram of the System. We introduce an input pulse on the input "Time" (after applying a reset on the capacitors) and charge the two capacitors in the first block with a constant current. They stop charging when the pulse finishes. The second block uses the voltages generated in the first block as references to charge and discharge a capacitor, generating a square signal at the output. The period of the output signal is equal to: T=90*Delta_Time.

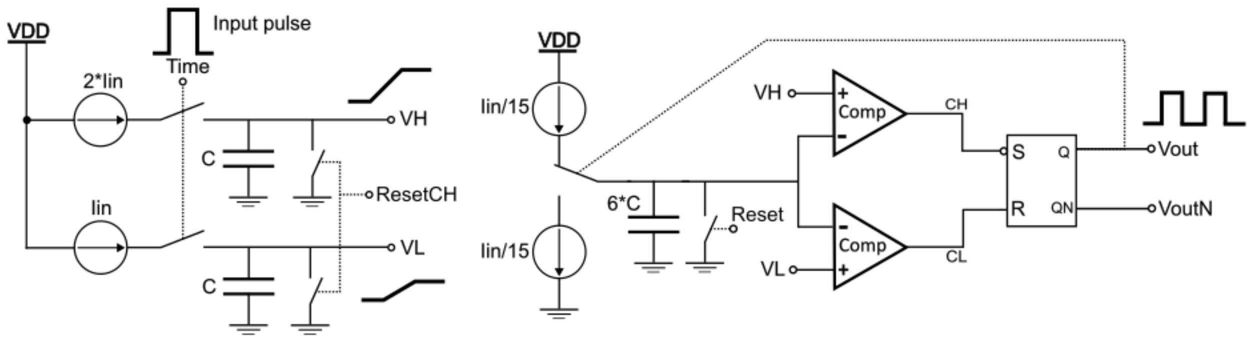


Figure 1. Block diagram of the System.

How to test

Figure 2 shows the block of the System with all the inputs and outputs that we can access in the circuit. The diagram shows the signal that we need to apply for to get the desired output signal. We can apply a width pulse between 20 to 60ns (at the pin pulse), besides the reset signals to discharge the capacitors (reset and resetCH pines), one before and one after the pulse, as shown the Figure 2. Connecting an oscilloscope to the output (Vout and VoutN) allows us to see the square periodic signal with a period proportional to the width of the input pulse, as explained in the first section. CH and CL are only used to monitor the behaviour of the comparators.

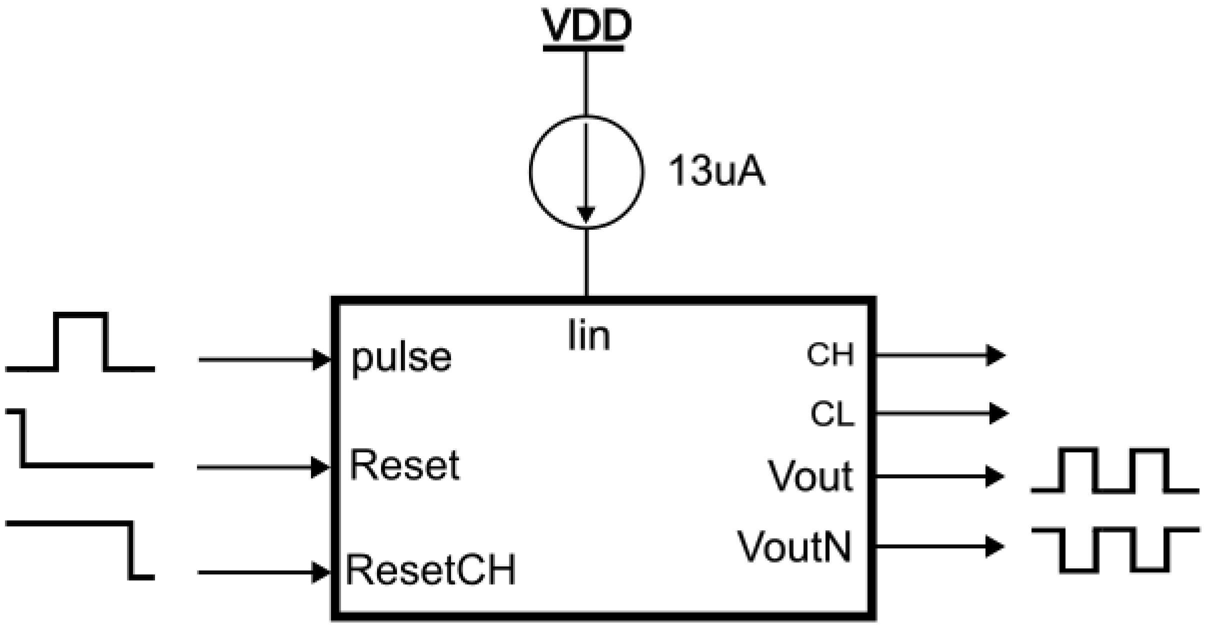


Figure 2. Inputs and Outputs of the System and test signals.

Look at Figure 3 for the circuit's connection to the chip frame, and at Figure 4 for the simulation results showing how the circuit behaves.

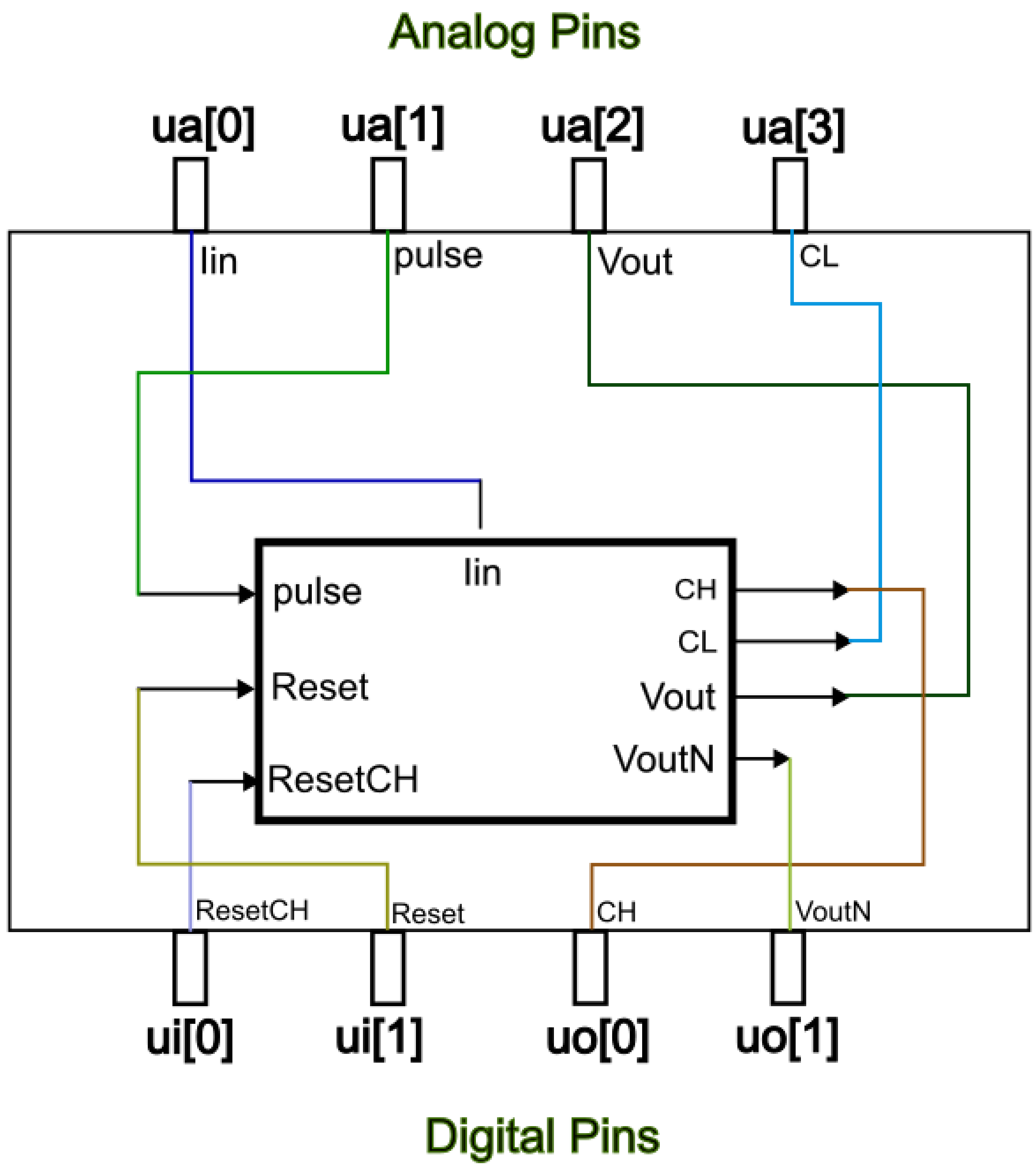


Figure 3. Pins used from the frame.

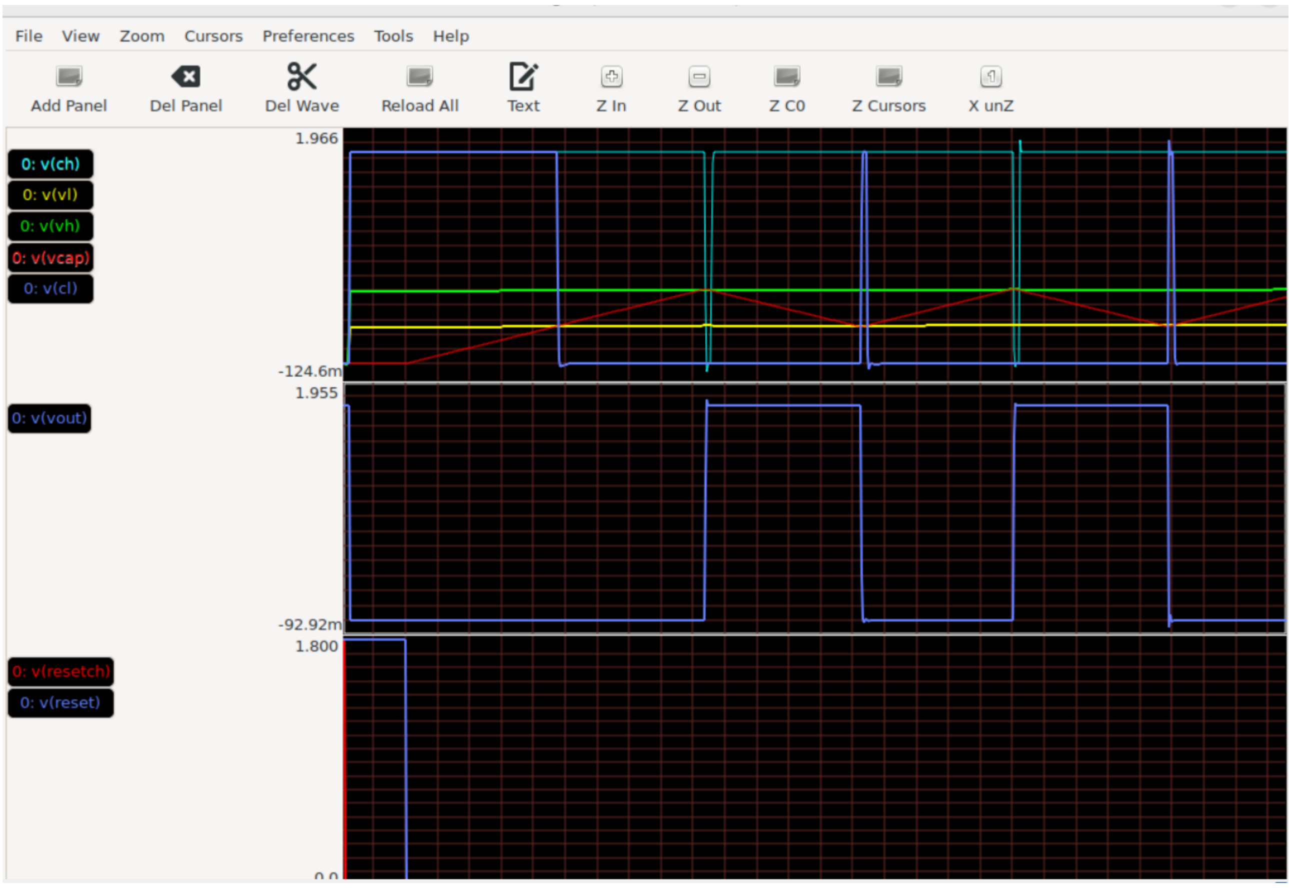


Figure 4. Simulation results.

External hardware

You need only a digital signal generator to generate the input pulse and reset signals, current source of 13uA to biasing and an oscilloscope to see the results.

Pinout

#	Input	Output	Bidirectional
0	ResetCH	CH	
1	Reset	VoutN	
2			
3			
4			
5			
6			
7			

Analog pins

ua#	analog#	Description
0	4	Iin
1	1	Pulse
2	3	Vout
3	2	CL

12 bit SAR ADC [490]

- Author: Ricardo Nunes
- Description: A 12 bit differential (11 bit single-ended) SAR ADC
- GitHub repository
- Analog project
- Mux address: 490
- Extra docs
- Clock: 0 Hz

How it works

This is a 12 bit SAR ADC. The ADC uses ~ 0.8 fF unit capacitors built with the fringe and parallel plate capacitance between the metal1 and metal2 layers.
The input signal (VIN_P - VIN_N) is sampled for 2 clock cycles if the previous conversion finished, there's a rising edge of the clock signal and the START input is high. After the sampling phase, the ADC determines the 12 bits by comparing the sampled input signal with a DAC voltage using a binary search.
The conversion result can be obtained from the DATA[5:0] outputs. The 6 most-significant bits are sampled at the rising edge of the CLK_DATA output and the 6 least-significant bits at the falling edge.
The START signal can be kept always high to convert continuously the input signal.

The comparator has a preamplifier with a gain of 20 to 30x. The preamplifier helps to relax the noise requirements of the strongarm latched comparator and makes it more predictable (not possible to run transient simulations with noise in ngspice). The comparator can calibrate its own offset. To do so, the EN_CAL_OFFSET input needs to be high. At the end of a conversion the comparator inputs are shorted, a comparison is triggered and an offset is introduced on purpose to counteract the comparator offset.

The ADC can be configured to convert a differential signal or a single-ended signal. For a single-ended configuration the ADC has a resolution of 11 bit and for a differential configuration 12 bit. For a differential configuration the inputs (VIN_P and VIN_N) should have a typical common-mode voltage of 0.6 V. For a single-ended configuration the VIN_N input should be close to VSS.

Pinout

Name	Direction	Type	Description
VDD	Input	Supply	1.8 V supply input.

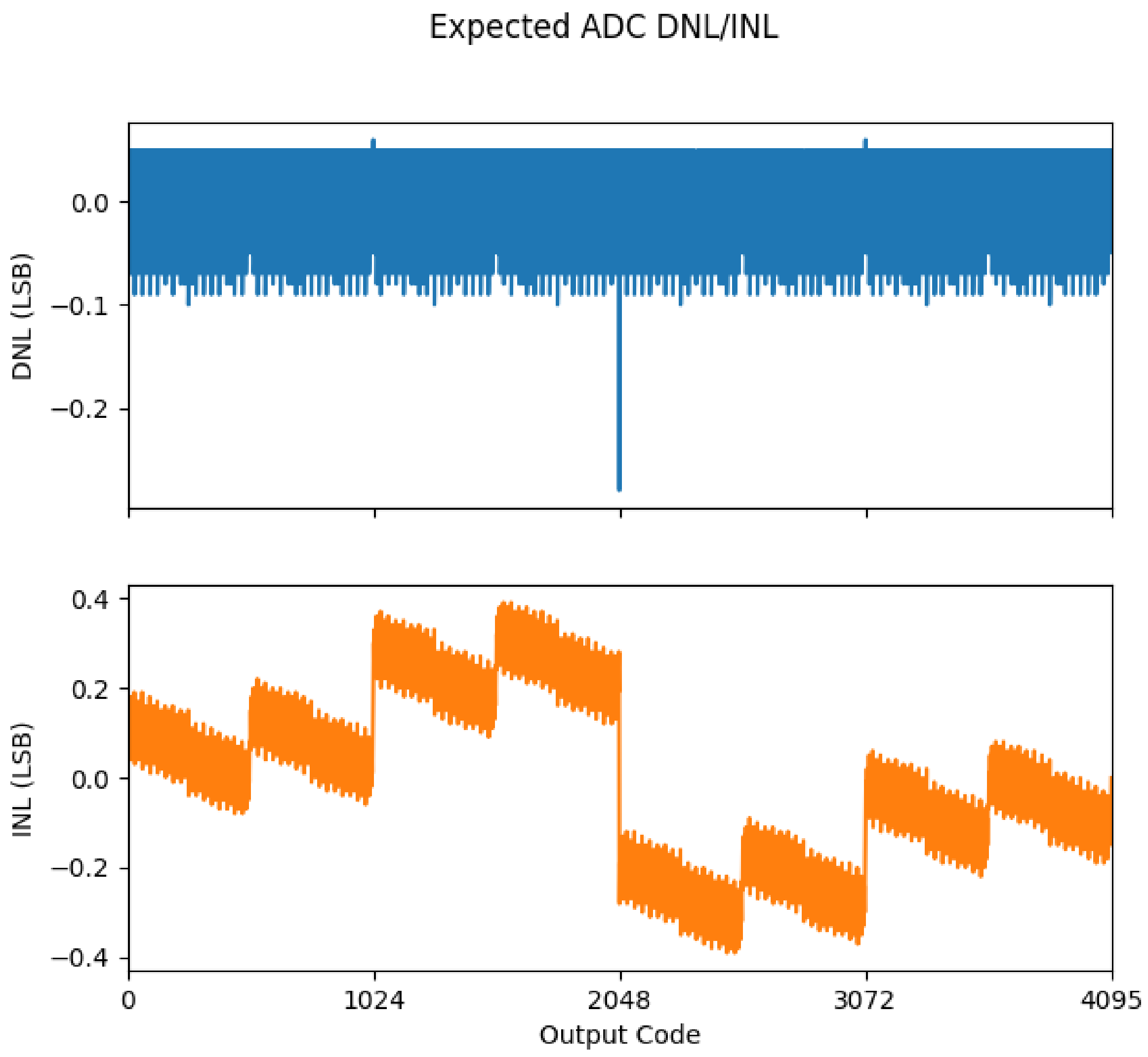
Name	Direction	Type	Description
VSS	Input	Supply	Ground.
VREF	Input	Analog	1.2 V reference voltage.
VREF_GND	Input	Analog	Reference voltage ground.
VCM	Input	Analog	0.6 V common-mode voltage.
VIN_P	Input	Analog	Positive input signal.
VIN_N	Input	Analog	Negative input signal.
CLK	Input	Digital	Clock input.
RST_Z	Input	Digital	Enable input.
START	Input	Digital	Start conversion input. Keep high to convert continuously.
EN_OFFSET_CAL	Input	Digital	Enables comparator offset self-calibration.
SINGLE_ENDED	Input	Digital	Configures ADC for single-ended input and VIN_N is used as ground reference.
CLK_DATA	Output	Digital	Rising edge used to sample the 6 MSBs and falling edge used for the 6 LSBs of ADC output.
DATA[5:0]	Output	Digital	Result of the conversion.

Specification

Parameter	Min	Typical	Max	Unit
Supply Voltage	1.7	1.8	1.9	V
Power Consumption				μA
Temperature	0	27	85	ºC
Reference Voltage	1.15	1.2	1.25	V
Input Common Mode Voltage (differential)	0.5	0.6	0.7	V
Ground Reference Voltage (single-ended)	-0.1	0	0.1	V
Output resistance for analog inputs			500	Ω
Clock Frequency[1]			20	MHz
Clock Low Pulse Width[2]	10			ns
Sampling Frequency		1/16 of clock freq.		
Sampling Capacitance		$\sim$ 1.8		pF
INL[3]				LSB
DNL[3]				LSB
ENOB without noise				bit
ENOB with noise				bit
SNDR without noise				dB
SNDR with noise				dB
SFDR				dB

[1] Input signal is sampled for 2 clock cycles. Clock/sampling frequency is limited by the 500 Ω resistance from the pads to the circuit in the TinyTapeout IC.
[2] Clock falling edge triggers the latched comparator, therefore clock low pulse width has to be larger than latched comparator propagation delay.
[3] Typical values for INL/DNL based on C extraction. Mismatch is not included.
SNDR/ENOB do not include distortion introduced by TinyTapeout analog MUX.
Noise estimated, not obtained directly from simulation.
Both 2*VIN(CM)-VCM and VCM should respect the limits to ensure comparator input common-mode voltage range is respected.

DNL/INL The DNL and INL were estimated by extracting all the DAC capacitors from the C extraction netlist. A python script reads the netlist file, extracts all the relevant capacitances and simulates the output of the ADC for a sweep of the input signal in order to calculate the DNL and INL. The results can be seen in the figure below.



How to test

Apply a differential voltage with a common-mode voltage of 0.6 V to VIN_P and VIN_N. If the ADC is running in the single-ended configuration, connect VIN_N to the ground reference of the input signal and VIN_P to the input signal.
To measure offset and noise short VIN_P and VIN_N and connect them to 0.6 V.
Apply a 20 MHz clock signal to the CLK input. The latched comparator is triggered at the falling edge of the CLK signal and the output is sampled at the rising edge. Therefore, the high time of the CLK signal should be long enough to allow the DAC to fully settle and the low time should be larger than the propagation delay of the comparator. Around 10 ns low time should be enough.
The ADC requires 16 cycles for 1 conversion. The frequency of the clock signal can be increased depending on the resistance between the analog pins and the circuit.
Set the RST_Z input high to enable the circuit and set the START input high to continuously convert.
The CLK_DATA pin should oscillate with a frequency of 1.25 MHz (20 MHz / 16).
The 6 most-significant bits of the conversion result can be sampled on the rising edge of the CLK_DATA signal and the 6 least-significant ones on the falling edge.

External hardware

Pinout

#	Input	Output	Bidirectional
0	start	data[5]	
1	en_offset_cal	data[4]	
2	single_ended	data[3]	
3		data[2]	
4		data[1]	
5		data[0]	
6		clk_data	
7			

Analog pins

`ua#`	`analog#`	Description
0	0	vcm
1	4	vref

ua#	analog#	Description
2	1	vref_gnd
3	3	vin_n
4	2	vin_p

RF_peripheral_circuits [492]

- Author: Shilpa Pavithran, Vineeta V Nair, Sruthi P, Aravind S, Vyshnav P Dinesh, Aswani A R
- Description: Peripheral circuits for RF based transmission.
- GitHub repository
- Analog project
- Mux address: 492
- Extra docs
- Clock: 0 Hz

How it works

Based upon different modes diffrent circuits are selected including RF based transmission.

How to test

Select a mode with proper input so as to select the rquired Rf transmission peripheral circuit.

External hardware

VNA, Multimeter and CRO.

Pinout

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

Analog pins

ua#	analog#	Description
0	5	vth/out
1	0	vin/vin[+]
2	4	vin[1]/vin[-]
3	1	s
4	3	ant
5	2	ind

mixed_signal_pulse_gen [494]

- Author: Aravind, Allwan
- Description: Mixed signal pulse generator
- GitHub repository
- Analog project
- Mux address: 494
- Extra docs
- Clock: 0 Hz

How it works

When input is given, pulse is generated accordingly.

How to test

give input

External hardware

NA

Pinout

#	Input	Output	Bidirectional
0	pulse_period[0]	s_out_lines[1]	pulse_count[1]
1	pulse_period[1]	s_out_lines[0]	pulse_count[2]
2	pulse_period[2]		pulse_count[3]
3	pulse_period[3]		pulse_count[4]
4	percentage[0]		pulse_count[5]
5	percentage[1]	pwm_out2	pulse_count[6]
6	start	pwm_out1	pulse_count[7]
7	pulse_count[0]	vout_1	pulse_count[8]

Analog pins

ua#	analog#	Description
0	5	s_in_lines[1]
1	0	s_in_lines[0]
2	4	vss
3	1	vout_4
4	3	vout_3
5	2	vout_2

TT07 Analog Factory Test [512]

- Author: Sylvain Munaut
- Description: Test structures for TT07 analog support
- GitHub repository
- Analog project
- Mux address: 512
- Extra docs
- Clock: 0 Hz

How it works

FIXME

How to test

FIXME

External hardware

FIXME

Pinout

#	Input	Output	Bidirectional
0	ena0_n		
1	ena1		
2			
3			
4			
5			
6			
7			

Analog pins

ua#	analog#	Description
0	11	ibias
1	6	vgnd_sense
2	10	vpwr_sense
3	7	loopback[0]
4	9	loopback[1]
5	8	loopback[2]

Tiny Eater 8 Bit [514]

- Author: Jason Kaufmann
- Description: Recreation of Ben Eater's 8 bit breadboard computer
- GitHub repository
- HDL project
- Mux address: 514
- Extra docs
- Clock: 12000000 Hz

How it works

This is Ben Eater's 8 Bit computer on an ASIC!

All credit for the design, amazing instructional videos, and diagams below goes to Ben Eater.

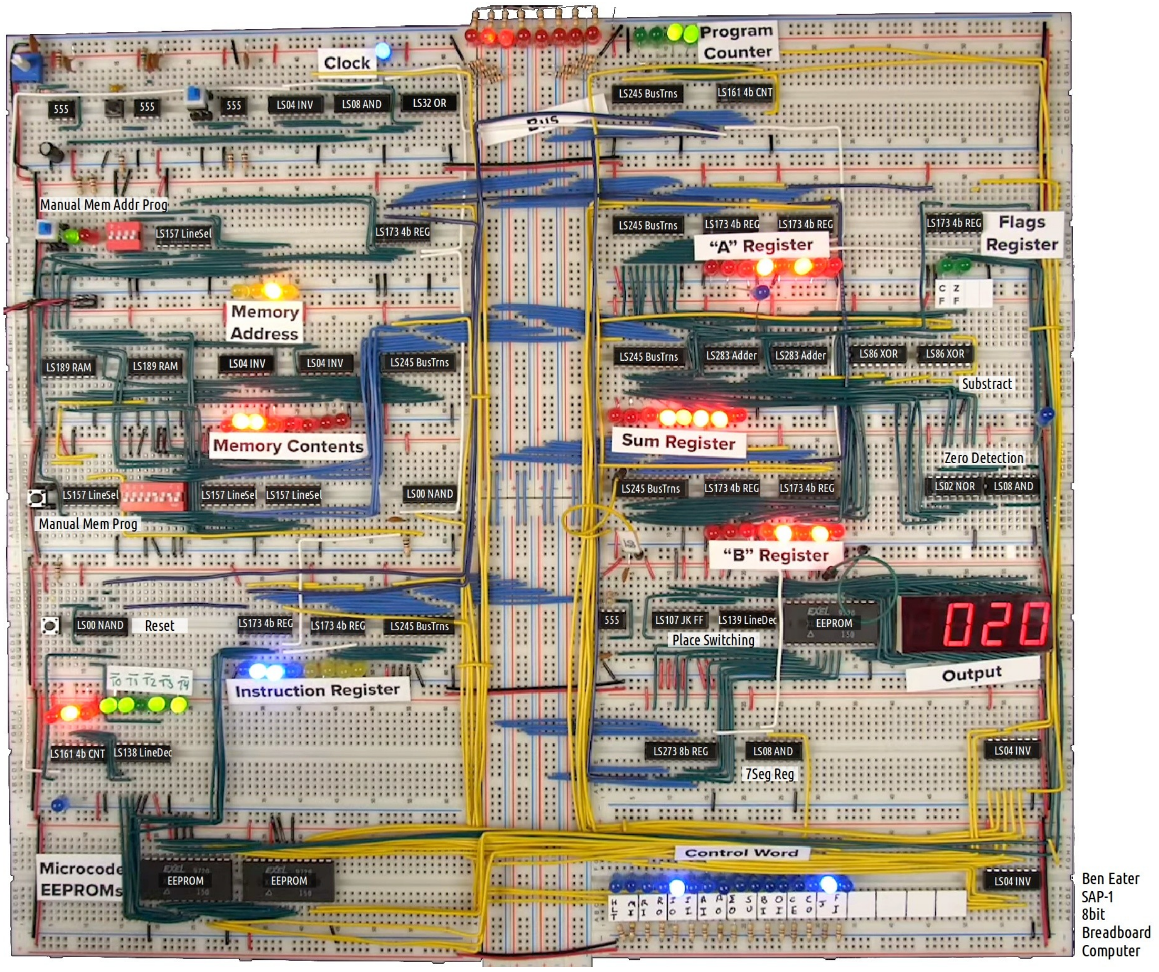
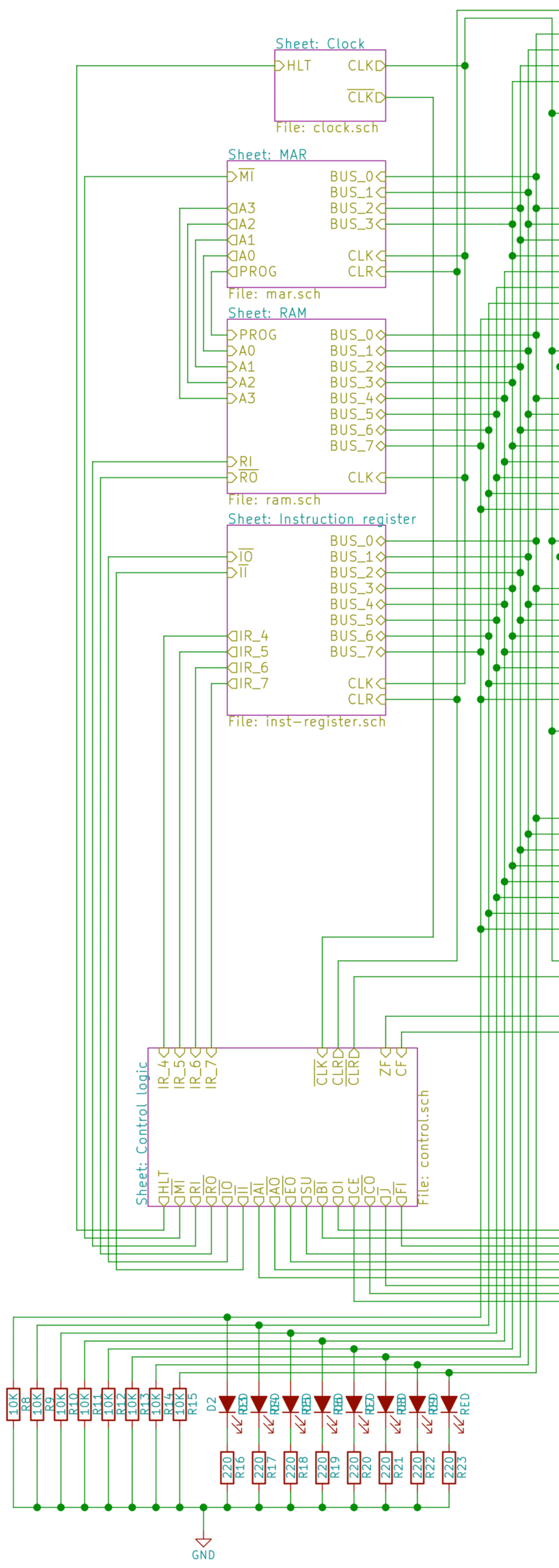
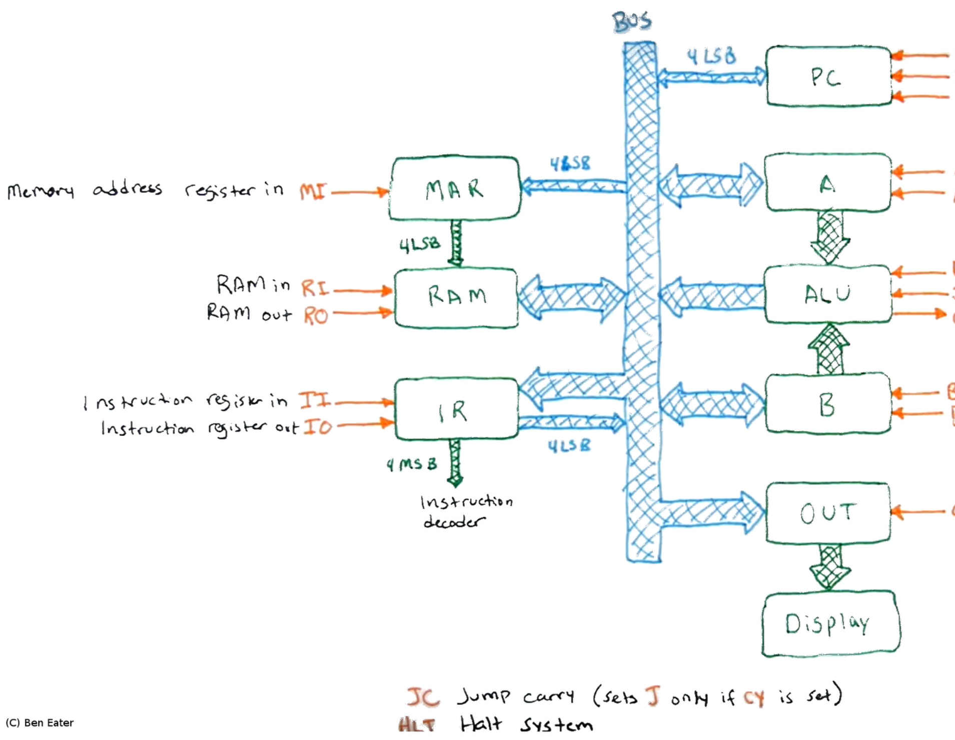


Figure 36: overview-with-chip-descriptions1



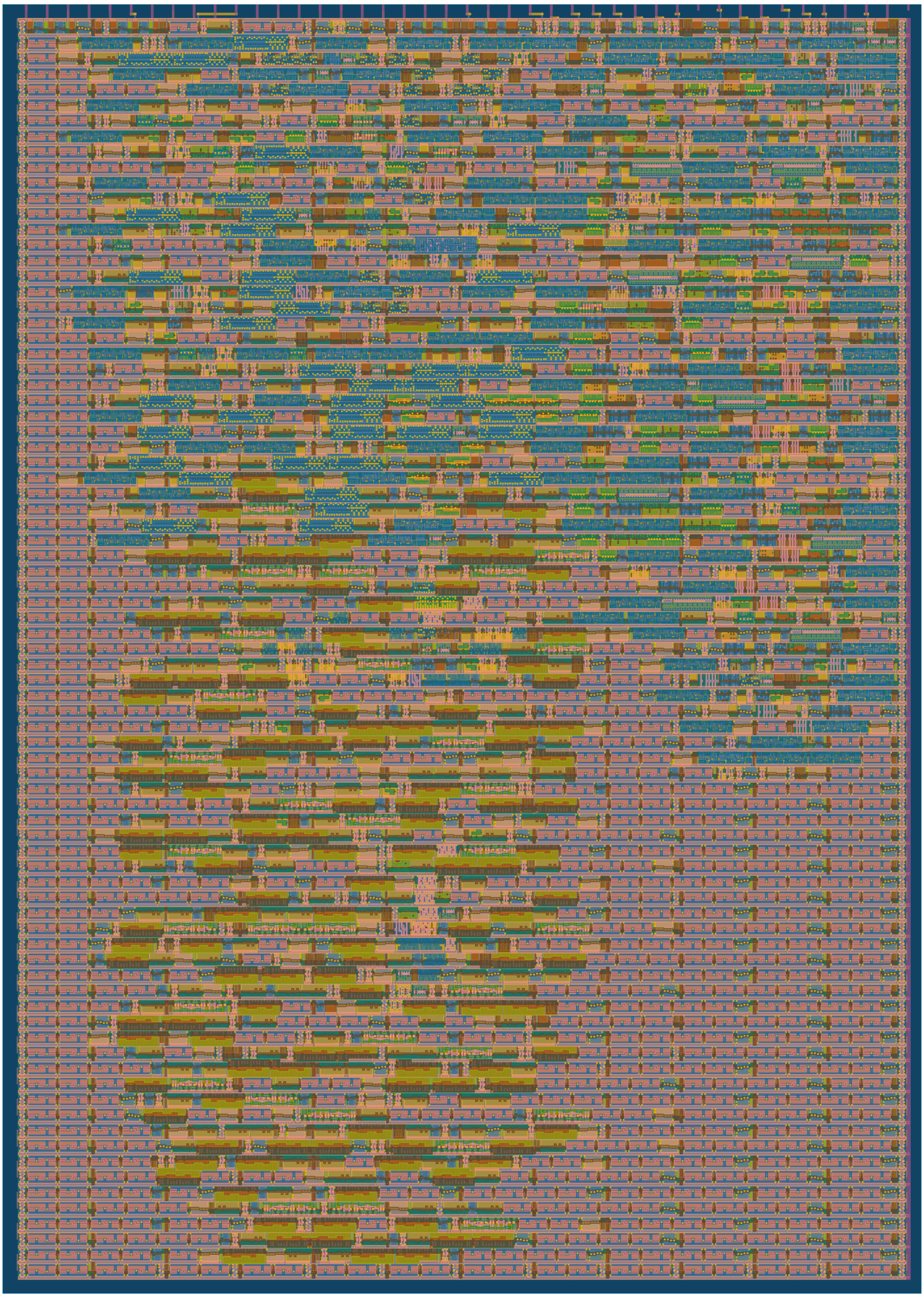
High level overview Full Computer Schematic:

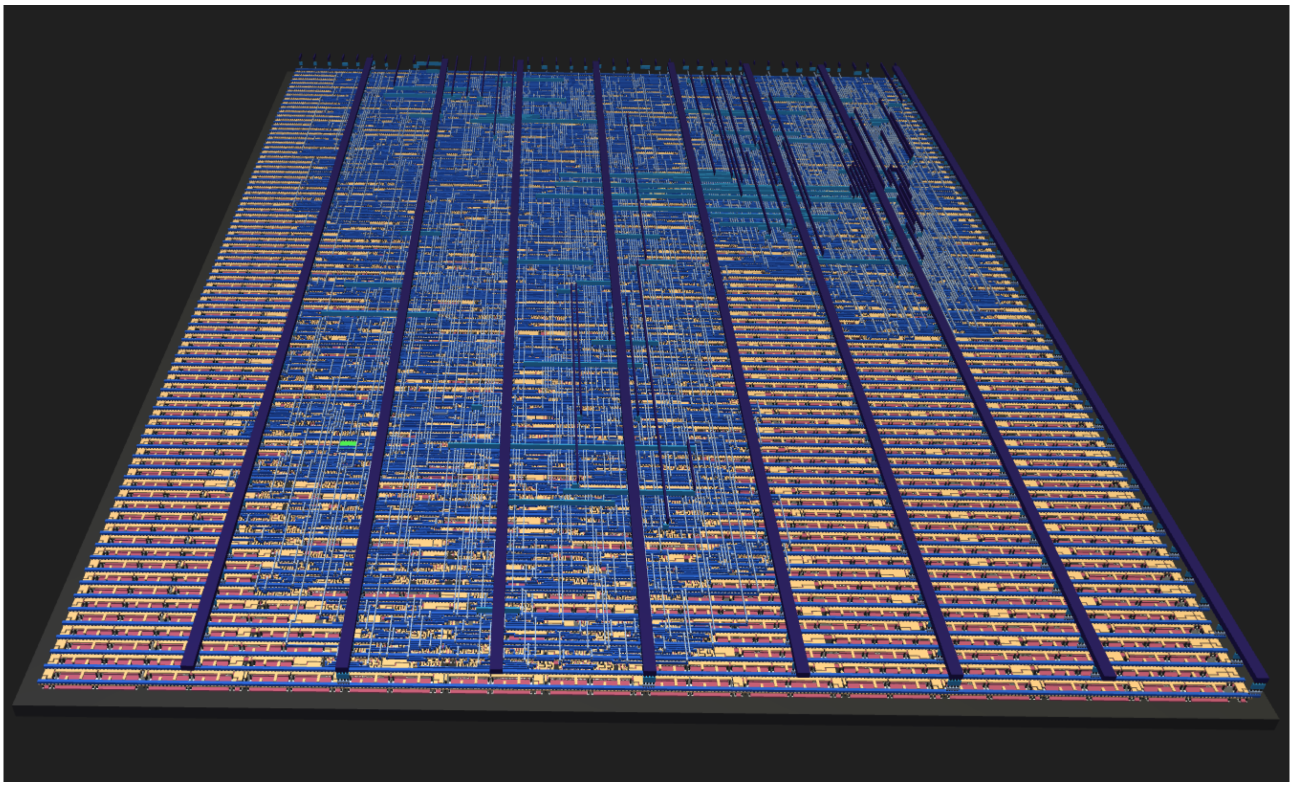

Bus
4LSB
PC
Memory address register in MI
MAR
4LSB
A
4LSB
RAM in RI
RAM out RO
RAM
ALU
Instruction register in II
Instruction register out IO
IR
4LSB
B
4MSB
Instruction decoder
OUT
Display
JC Jump carry (sets J only if CY is set)
HLT Halt system

Simple Control Signal Diagram: (C) Ben Eater

*Note: The output register and logic to display the digits is not included on the ASIC. The 8 bit output value is put on the bus and the "output register in" control signal (oi) is on an output pin. This way you can use the data bus as a general purpose interface to any display you want. (i.e. you can read in the data to the RP2040 and show it on the screen, you can build the actual output register as shown in the videos and connect it to the PMOD header, etc.)

ASIC 2D:



ASIC 3D: 

How to test

To program the computer follow these steps:

- enable my design in TT
- send prog_mode bit high
- set the four prog_address bits to the address you want to write to, put the data you want to store at that address on the I/O lines, and then pulse the clock.
- since this computer only has a 4 bit address space you can only store 16 bytes total in the internal RAM.
- see https://eater.net/8bit/ for more details.

Instructions

OPC	DEC	HEX	DESCRIPTION
NOP	00	0000	
LDA	01	0001	Load contents of memory address aaaa into register A.
ADD	02	0010	Put content of memory address aaaa into register B, add A + B, store result in A.
SUB	03	0011	Put content of memory address aaaa into register B, subtract A - B, store result in register A.
STA	04	0100	Store contents of register A at memory address aaaa.

OPC	DEC	HEX	DESCRIPTION
LDI	05	0101	Load 4 bit immediate value in register A (loads 'vvvv' in A).
JMP	06	0110	Unconditional jump. Set program counter (PC) to aaaa, resume execution from that memory address.
JC	07	0111	Jump if carry. Set PC to aaaa when carry flag is set and resume from there. When carry flag is not set, resume normally.
JZ	08	1000	Jump if zero. As above, but when zero flag is set.
	09	1001	
	10	1010	
	11	1011	
	12	1100	
	13	1101	
OUT	14	1110	Output register A to 7 segment LED display as decimal.
HLT	15	1111	Halt execution.

External hardware

You will need the RP2040 or a similar microcontroller to write the program into the internal memory. If you really wanted to, you could go old school and use DIP switches and a manual clock pulse as well.

You will want to make the output register on a breadboard to connect it to the 8 bit I/O lines from the PMOD header. See https://eater.net/8bit/output for detailed design info.

Pinout

#	Input	Output	Bidirectional
0	prog_mode	output_enable	data0
1	addr0		data1
2	addr1		data2
3	addr2		data3
4	addr3		data4
5	clock_change_mode		data5
6	clock_max_count		data6
7			data7

DJ8 8-bit CPU w/ DAC [516]

- Author: DaveX
- Description: DJ8 8-bit CPU with parallel Flash / RAM interface and 8-bit R-2R DAC
- GitHub repository
- Analog project
- Mux address: 516
- Extra docs
- Clock: 2000000 Hz

How it works

DJ8 is a 8-bit CPU featuring:

- 8 x 8-bit register file
- 3-4 cycles per instruction
- 15-bit address bus
- 8-bit data bus
- 8-bit DAC based on Tiny Tapeout Analog R2R DAC
- Built-in 256-bytes demo ROM with 2 demos

Other implementations:

- TT06 DJ8 8-bit CPU - VHDL
- TTIHP0P2 DJ8 8-bit CPU (no DAC) - Verilog

Memory Map

From	To	Description
0x0000	0x7fff	External memory
0x8000	0xffff	Internal Test ROM (256 bytes, mirrored)
0xff00	0xff00	DAC_OUT (8-bit unsigned, write-only)

External memory map if using the recommended setup (see pinout)

From	To	Description
0x2000	0x3fff	External RAM (32 bytes)
0x4000	0x5fff	External Flash ROM (16KB)

Registers There are 8 general purposes 8-bit registers (A,B,C,D,E,F,G,H), two flag registers (CF, ZF), and 16-bit PC.

For memory addressing, 16-bit combined registers EF and GH are used.

At reset time, PC is set to 0x4000. All other registers are set to 0x80.

Instruction Set For future compatibility, please set the don't care bits (?) to 0.

ALU reg, imm8: Immediate ALU operation

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	A	A	A	D	D	D	I	I	I	I	I	I	I	I

- A : ALU operation
  - 000: ADD: reg = reg + imm8
  - 001: ADC: reg = reg + imm8 + CF
  - 010: SUBC: reg = reg - (imm8 + CF)
  - 011: MOVR: reg = reg
  - 100: XOR: reg = reg ^ imm8
  - 101: OR: reg = reg | imm8
  - 110: AND: reg = reg & imm8
  - 111: MOVI: reg = imm8
- D : register
- I : imm8

ALU dest, src, A {,shift}: ALU operation with src register & register A

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	A	A	A	D	D	D	S	S	S	?	F	F	0	0

- A : ALU operation
  - 000: ADD: dest = src + A
  - 001: ADC: dest = src + A + CF
  - 010: SUBC: dest = src - (A + CF)
  - 011: MOVR: dest = src
  - 100: XOR: dest = src ^ A

  - 101: OR: dest = src | A
  - 110: AND: dest = src & A
  - 111: MOVI: dest = A

- D : dest register
- S : src register
- F : final shift operation

  - 00: No shift
  - 01: Shift right logical (shr)
  - 10: Shift right arithmetic (sar)

ALU dest, [mem], A {,shift}: ALU operation with memory & register A

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	A	A	A	D	D	D	?	?	?	M	F	F	1	0

- A : ALU operation

  - 000: ADD: dest = [mem] + A
  - 001: ADC: dest = [mem] + A + CF
  - 010: SUBC: dest = [mem] - (A + CF)
  - 011: MOVR: dest = [mem]
  - 100: XOR: dest = [mem] ^ A
  - 101: OR: dest = [mem] | A
  - 110: AND: dest = [mem] & A
  - 111: MOVI: dest = A

- D : dest register
- M: memory mode

  - 0: [GH]
  - 1: [EF]

- F : final shift operation

  - 00: No shift
  - 01: Shift right logical (shr)
  - 10: Shift right arithmetic (sar)

MOVR [mem], reg: Store content of register in memory

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	D	D	D	?	?	?	M	?	?	0	1

- D: register
- M: memory mode
  - 0: [GH]
  - 1: [EF]

Jxx imm12: Conditional or unconditional jump to absolute address

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	J	J	I	I	I	I	I	I	I	I	I	I	I	I

- J: jmpcode
  - 01: Jump if zero (JZ)
  - 10: Jump if not zero (JNZ)
  - 11: Unconditional jump (JMP)
- I: imm12
  - PC = (PC & 0xe000) | (imm12 « 1)

JMP GH: Unconditional jump to address GH

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	?	?	?	?	?	?	?	?	?	?	?	?	?	?

Pinout Due to TT07 IO constraints, pins are shared between *Address bus LSB* and *Data bus OUT*. It means that during memory write instructions, the address space is only 128 bytes.

Pins	Standard mode	During memory write execute+writeback cycles
ui[7..0]	Data bus IN	Data bus IN
uio[7..0]	Address bus LSB (7..0)	***Data bus OUT***
uo[6..0]	Address bus MSB (14..8)	Address bus MSB (14..8)
uo[7]	Write Enable	Write Enable

Pins	Standard mode	During memory write execute+writeback cycles
ua[0]	DAC output	DAC output

You can connect a 8KB parallel Flash ROM + 32b SRAM without external logic and use uo[6] for RAM OE# and uo[5] for Flash ROM OE#.

To get a bidirectional data bus (needed for SRAM), uio bus must be connected to ui bus with resistors. To be tested!

How to test

An internal test ROM with two demos is included for easy testing. Just select the corresponding DIP switches at reset time to start the demo (technically, a ***jmp GH*** instruction will be seen on the data bus thanks to the DIP switches values, with GH=0x8080 at reset).

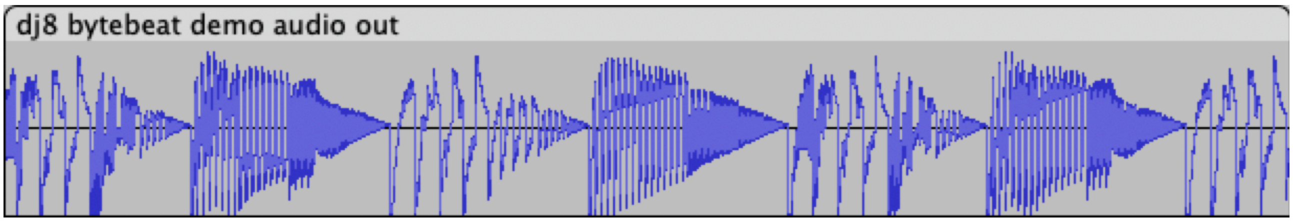
Demo 1: Rotating LED indicator

SW1	SW2	SW3	SW4	SW5	SW6	SW7	SW8
0	0	0	0	0	0	1	0

No external hardware needed. This demo shows a rotating indicator on the 7-segment display. Its speed can be changed with DIP switches, the internal delay loop is entirely deactivated when all switches are reset.

Demo 2: Bytebeat Synthetizer

SW1	SW2	SW3	SW4	SW5	SW6	SW7	SW8
0	0	0	0	0	1	1	0



Modem handshakes sound like music to your hears? It's your lucky day! Become a bit-crunching DJ thanks to 256 lo-fi glitchy settings.

Connect ua[0] -> amp(TBD?) -> speaker. Play with the DIP switches to change the loop settings. Suggested frequency/amp/passives TBD.

External hardware

- No external hardware for Demo 1
- Speaker (+ amp?) for Demo 2
- Otherwise: Parallel Flash ROM + optional SRAM

Pinout

#	Input	Output	Bidirectional
0	data in 0	address out 8	address out 0 / data out 0
1	data in 1	address out 9	address out 1 / data out 1
2	data in 2	address out 10	address out 2 / data out 2
3	data in 3	address out 11	address out 3 / data out 3
4	data in 4	address out 12	address out 4 / data out 4
5	data in 5	address out 13	address out 5 / data out 5
6	data in 6	address out 14	address out 6 / data out 6
7	data in 7	write enable	address out 7 / data out 7

Analog pins

`ua#`	`analog#`	Description
0	10	DAC output

Twin Tee Sine Wave Generator [518]

- Author: Matt Venn
- Description: Opamp plus notch filter = sine wave out
- GitHub repository
- Analog project
- Mux address: 518
- Extra docs
- Clock: 0 Hz

How it works

The circuit uses a "Twin Tee" filter along with an opamp to generate a sine wave.

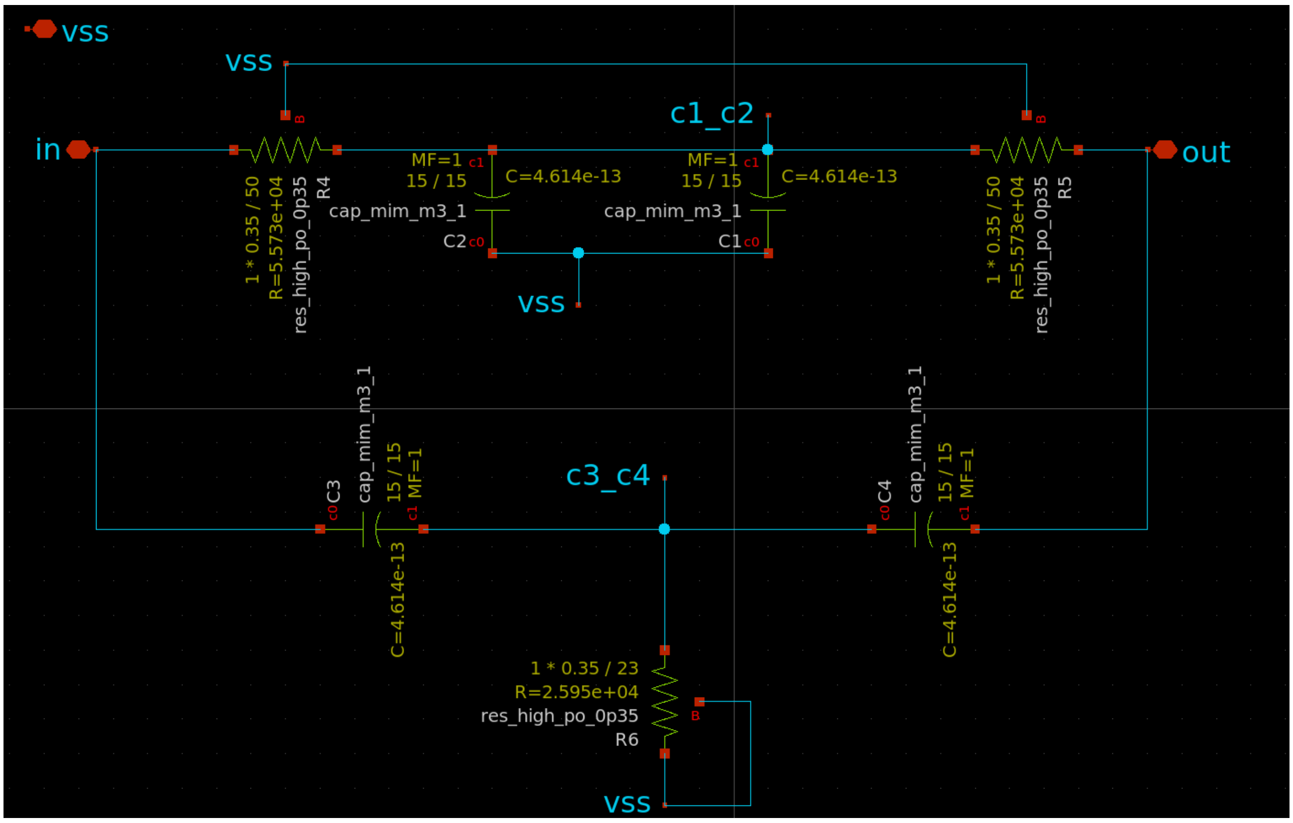


Figure 37: Twin Tee Notch Filter

https://www.electronics-tutorials.ws/oscillator/twin-t-oscillator.html

```
f = 1 / 2 * pi * RC
```

Notch filter AC analysis

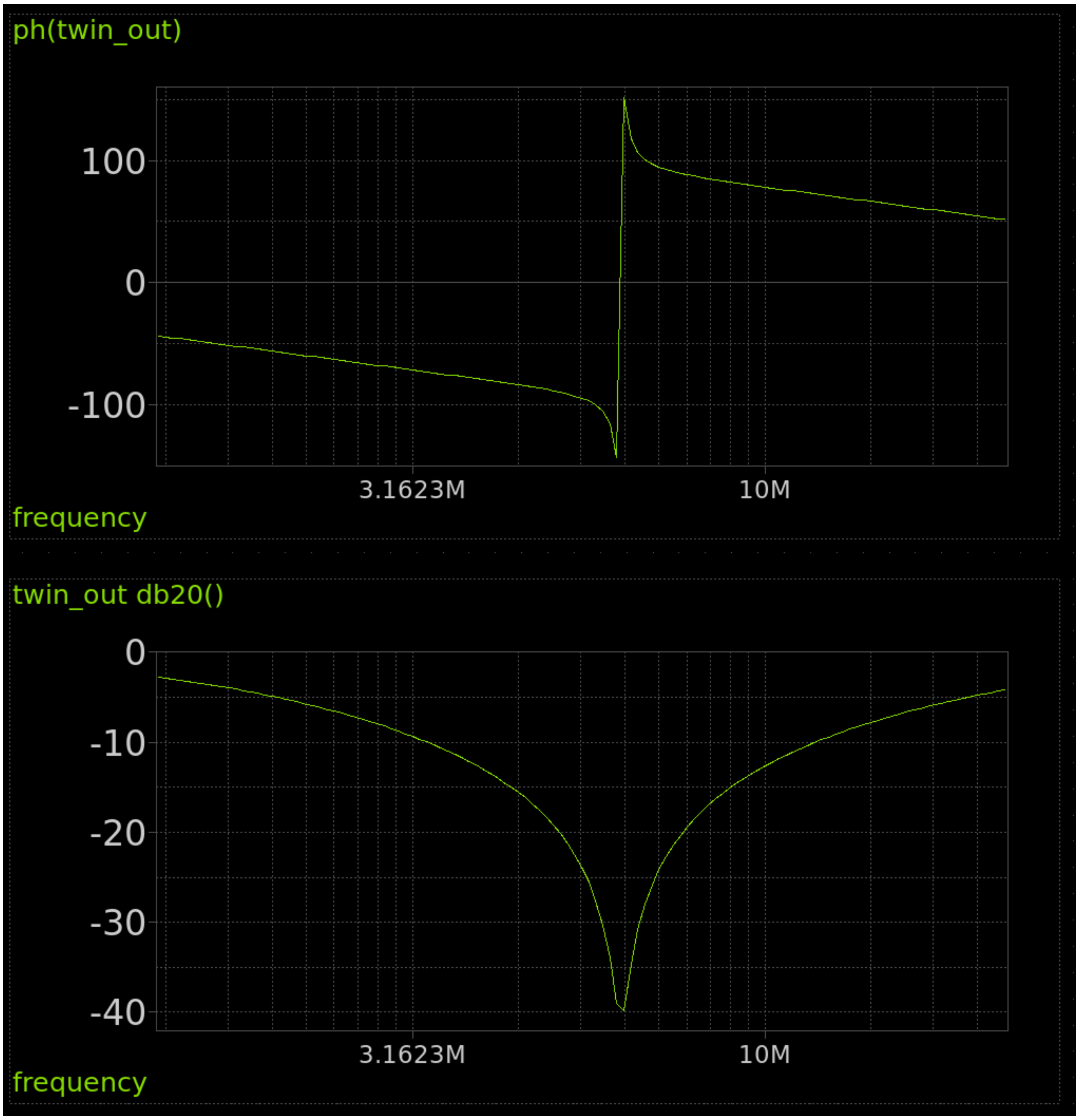


Figure 38: Notch filter AC analysis

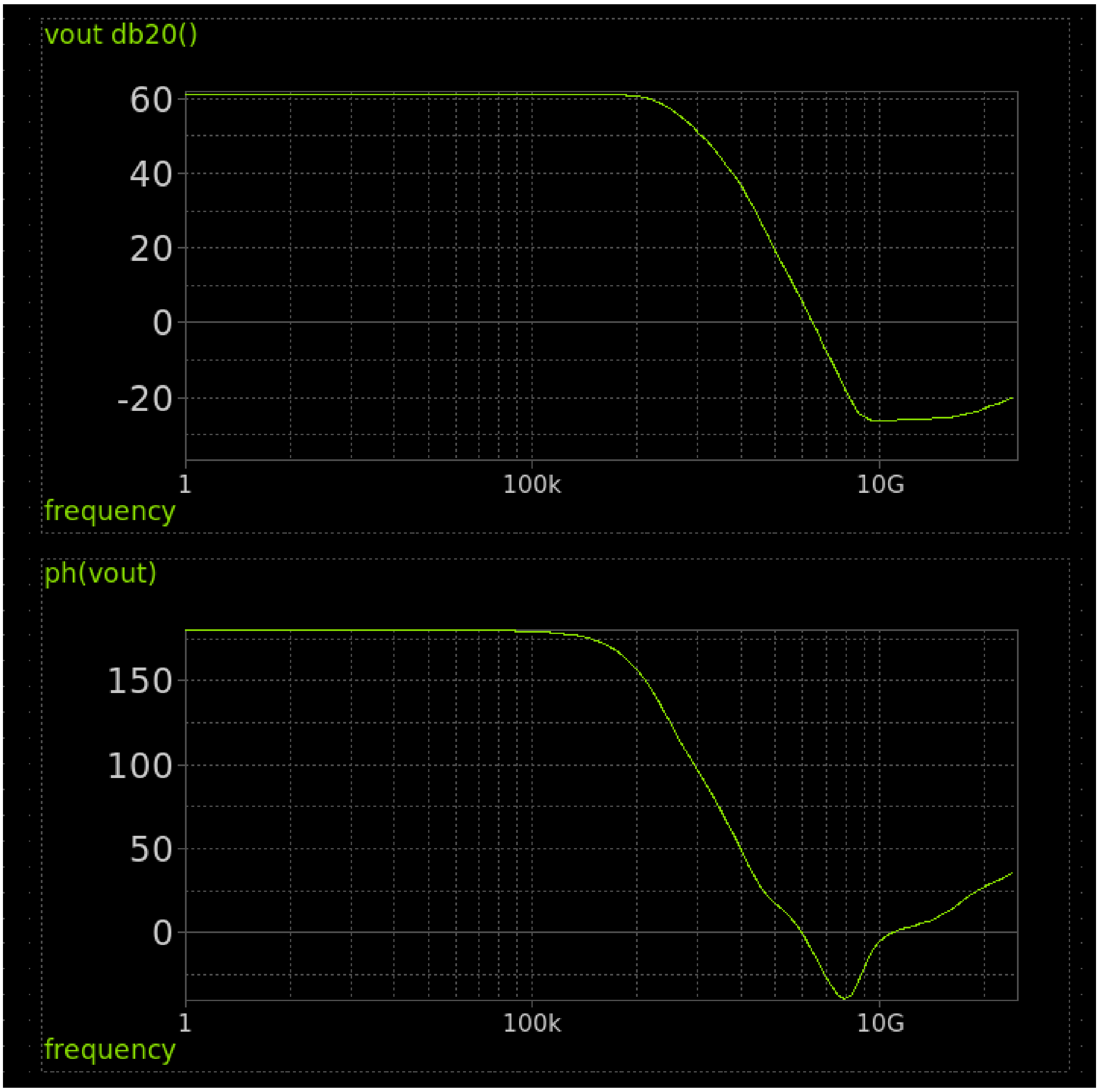


Figure 39: Opamp AC analysis

Opamp AC analysis

Transient analysis This simulation was done with a "montecarlo mismatch corner" which aims to test how well the circuit will work as the resistance and transistors change across process corners. Each run is shown in a different colour. If R6 resistor in the Twin Tee filter is too high, the oscillations quickly die out, so it's deliberately undersized at the expense of distortion in the sine wave.

This simulation includes extracted parastitics and a model of the pin.

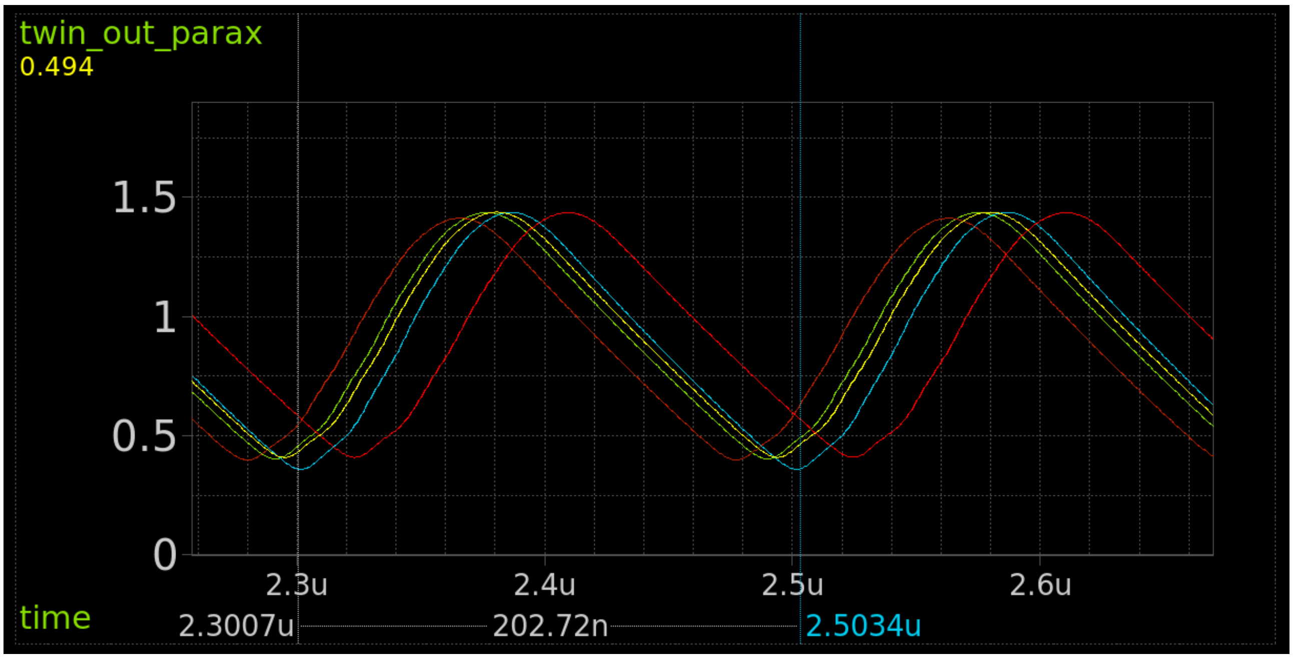


Figure 40: Twin Tee output transient analysis

Acknowledgements

- The opamp design comes from Sai
- The opamp layout comes from Pat Deegan
- Inspired by this video by Alan Wolke.

How to test

After the project is enabled, you should see a sine wave at around 4.93MHz on analog output pin 0.

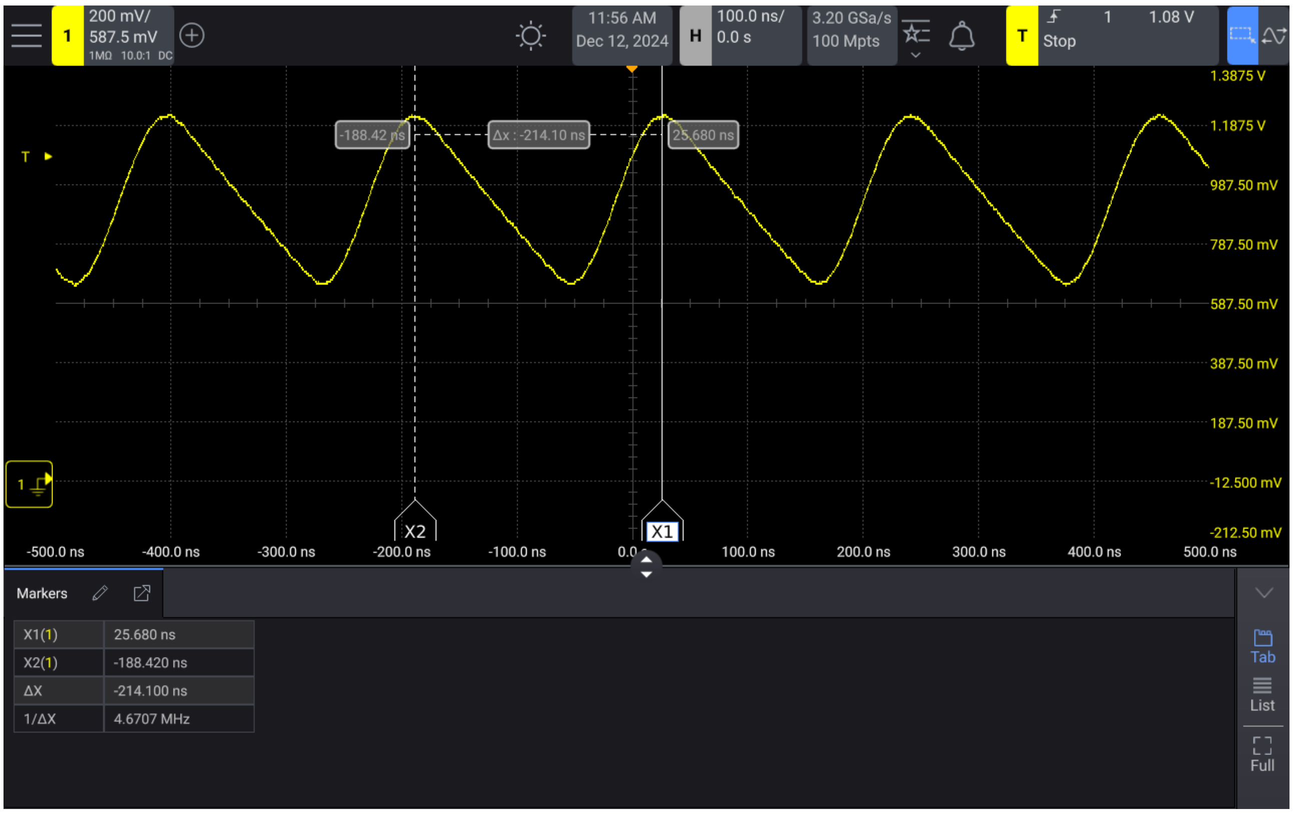
There is also a 20 bit digital counter connected to the oscillator output. The top 8 bits are connected to the LEDs. So you should also see the most significant bit (the dot) flashing about 5 times per second.

External hardware

Oscilloscope.

Silicon results

The measurement was made with a Keysight HD3 scope. The design is very sensitive to loading, so a scope probe was connected direct to the pad (after removing the pulldown).



X1(1)	25.680 ns
X2(1)	-188.420 ns
ΔX	-214.100 ns
1/ΔX	4.6707 MHz

Figure 41: measurement

The frequency was measured at 4.67MHz, which is about 5% slower than anticipated.

Pinout

#	Input	Output	Bidirectional
0		counter bit 12	
1		counter bit 13	
2		counter bit 14	
3		counter bit 15	

#	Input	Output	Bidirectional
4		counter bit 16	
5		counter bit 17	
6		counter bit 18	
7		counter bit 19	

Analog pins

`ua#`	`analog#`	Description
0	11	sine out

Classic 8-bit era Programmable Sound Generator AY-3-8913 [520]

- Author: ReJ aka Renaldas Zioma
- Description: The AY-3-8913 is a 3-voice programmable sound generator (PSG) chip from General Instruments. The AY-3-8913 is a smaller variant of AY-3-8910 or its analog YM2149.
- GitHub repository
- Analog project
- Mux address: 520
- Extra docs
- Clock: 2000000 Hz

How it works

A simple 8 bit R2R DAC. Driven externally or by an OpenLane generated sawtooth waveform generator.

How to test

Drive externally Set the `external data` input high to provide the DAC with external data. Then drive the 8 inputs and observe the analog output.

Drive with internal sawtooth wave generator Set the `external data` input low to enable the sawtooth generator. A sawtooth wave should be seen on the analog output.

To change the frequency, set the inputs and then raise the 'load divider' input.

External hardware

A multimeter to measure the output voltage on analog pin 0.

#	Input	Output	Bidirectional

Pinout

#	Input	Output	Bidirectional
0	multiplexed data/address bus LSB	(pwm) channel C	(in) **BC1** bus control
1	multiplexed data/address bus	(pwm) channel B	(in) **BDIR** bus direction
2	multiplexed data/address bus	(pwm) channel A	(in) **SEL0** clock divider
3	multiplexed data/address bus		(in) **SEL1** clock divider
4	multiplexed data/address bus		
5	multiplexed data/address bus		
6	multiplexed data/address bus		
7	multiplexed data/address bus MSB		

Analog pins

ua#	analog#	Description
0	11	(out) channel C
1	6	(out) channel B
2	10	(out) channel A

Double Balanced Mixer and Quadrature Divider [522]

- Author: Bruce MacKinnon
- Description: Takes an RF input (with DC bias) and a digital LO signal and mixes to provide a differential IF output. Simulated at 7 MHz.
- GitHub repository
- Analog project
- Mux address: 522
- Extra docs
- Clock: 0 Hz

Overview

The project name notwithstanding, this is a simple double-balanced RF mixer for the HF frequency range (around 7 MHz). The design is a switching mixer, so the local oscillator (LO) clock is a digital signal.

The RF input should be DC biased. Expected bias is around 0.6 volts.

A differential IF output is provided. The IF output should be the difference of IFOUT_P and IFOUT_N.

The detailed document here gets into the mathematical basis for the circuit design used.

The tile also has a simple digital component called a quadrature divider. This takes a digital clock and creates two output clocks with quadrature relationship (90 degrees of phase difference). This would be useful for certain types of RF modulators and demodulators.

The mixer and the quadrature divider are entirely independent circuits at this time.

Pinouts

- ua[0] - Analog RF input with DC bias.
- ua[1] - Analog IF output (positive phase)
- ua[2] - Analog IF output (negative phase)
- uio_in[0] - Digital local oscillator (LO) input.
- uio_in[1] - Digital clock input for quadrature divider.
- uo_out[0] - Digital quadrature divider output, I phase.
- uo_out[1] - Digital quadrature divider output, Q phase.

Pinout

#	Input	Output	Bidirectional
0	Local Oscillator input for mixer	I phase output from quadrature divider	
1	Clock input for quadrature divider	Q phase output from quadrature divider	
2			
3			
4			
5			
6			
7			

Analog pins

ua#	analog#	Description
0	11	RF input (with DC bias)
1	6	IF output (Positive phase)
2	10	IF output (Negative phase)

TT07 Differential Receiver test [524]

- Author: Sylvain Munaut
- Description: Small test module to test functionality of a differential input receiver
- GitHub repository
- Analog project
- Mux address: 524
- Extra docs
- Clock: 0 Hz

How it works

FIXME

How to test

FIXME

External hardware

FIXME

Pinout

#	Input	Output	Bidirectional
0	debug	q[0]	q[8]
1	bias_sel	q[1]	q[9]
2		q[2]	q[10]
3		q[3]	q[11]
4		q[4]	q[12]
5		q[5]	q[13]
6		q[6]	q[14]
7		q[7]	q[15]

Analog pins

ua#	analog#	Description
0	11	clk_n
1	10	clk_p
2	7	data_n
3	9	data_p
4	8	ibias

QIF Neuron [526]

- Author: Katie Burrows and David Parent
- Description: Models a QIF spiking neuron
- GitHub repository
- Analog project
- Mux address: 526
- Extra docs
- Clock: 0 Hz

How it works

When the input hits a certain voltage, the output will spike and reset to a known value E. J. Basham and D. W. Parent, "Compact digital implementation of a quadratic integrate-and-fire neuron," 2012 Annual International Conference of the IEEE Engineering in Medicine and Biology Society, San Diego, CA, USA, 2012, pp. 3543-3548, doi: 10.1109/EMBC.2012.6346731. keywords: {Mathematical model;Clocks;Equations;Vectors;Computational modeling;Field programmable gate arrays;Neurons},

How to test

Input a content 8-bit digital singal when the voltage crosses the threshold, and see if it spikes. See: https://www.dropbox.com/s/6pjsgwxqhryaggs/ee122_lab_manual.pdf?dl=0

External hardware

ADLAM2000

Pinout

#	Input	Output	Bidirectional
0			
1			
2			
3			
4			
5			

#	Input	Output	Bidirectional
6			
7			

Analog pins

`ua#`	`analog#`	Description
0	6	Vout
1	10	Vinp
2	7	Vinn
3	9	VoutM
4	8	VinM

badGPU [582]

- Author: Emery Nagy
- Description: Basic GPU for rasterizing polygons
- GitHub repository
- HDL project
- Mux address: 582
- Extra docs
- Clock: 25000000 Hz

How it works

This project implements a GPU capable of rasterizing 4 triangles. It takes commands from any microcontroller via the SPI bus and draws them on screen at 640x480 @60Hz. All colors are 6 bits in depth (i.e rrggbb), giving up to 64 unique colors. It can also set a unique background color of your choosing.

Design expects a 25Mhz master clock frequency.

Triangles are supplied by their vertices. Note that not all vertex values are supported, only 80 X vertex positions and 60 Y vertex positions are allowed. This is acheived by taking the desired vertex position in the 640x480 native display resolution and dividing it by 8 on the host microcontroller side.

How to test

The device can draw up to 4 polygons, A, B, C, D at a time. Since there is no Z-buffer, polygon A can be thought of as "closest" to the viewer, B "second closest", etc. If there is a region where your set polygons overlap, A will be rasterized over B and so on.

The "GPU" should be connected to the host microcontroller via SPI (tested with up to 4Mhz). Note that SPI communication here needs to be with LSB first formatting. During each frame there is a certain amount of time which is not used to display any image. Here, the GPU will assert the INT pin, telling the host microcontroller that it is able to send new commands via SPI.

Example SPI transfers initiated after INT pin asserted

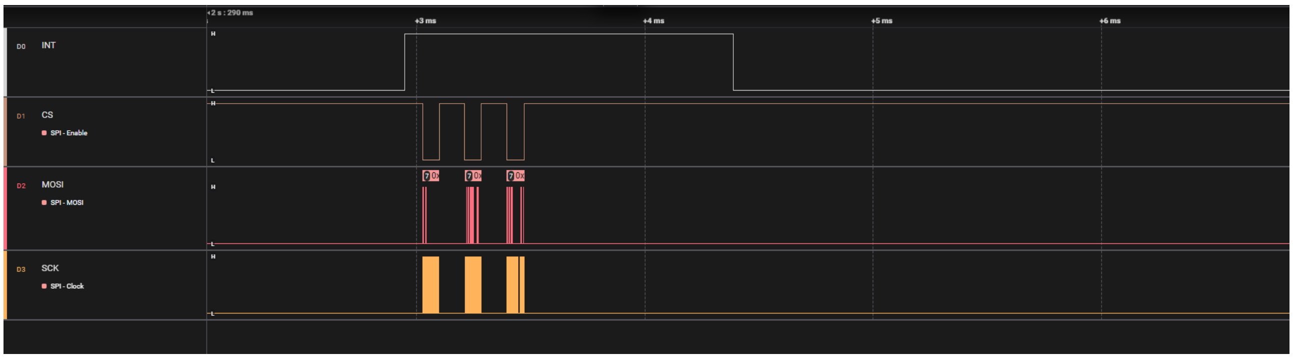


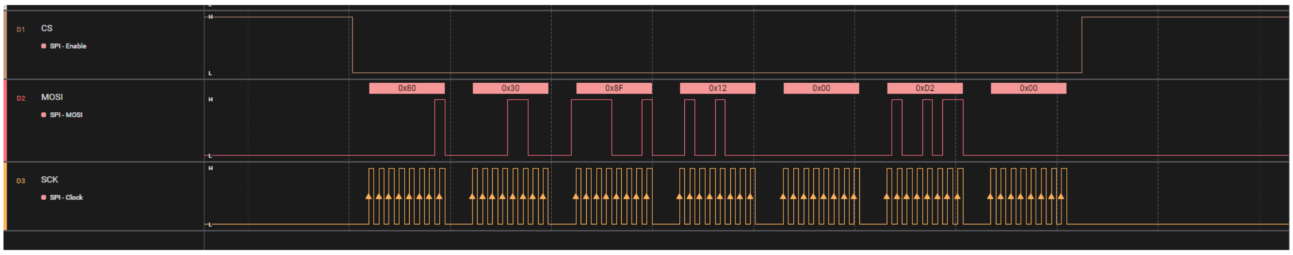
Figure 42: image

Command set Each SPI command contains a CMD byte along with 6 data bytes. The 6 data bytes are a packed bitfield with the following formatting:

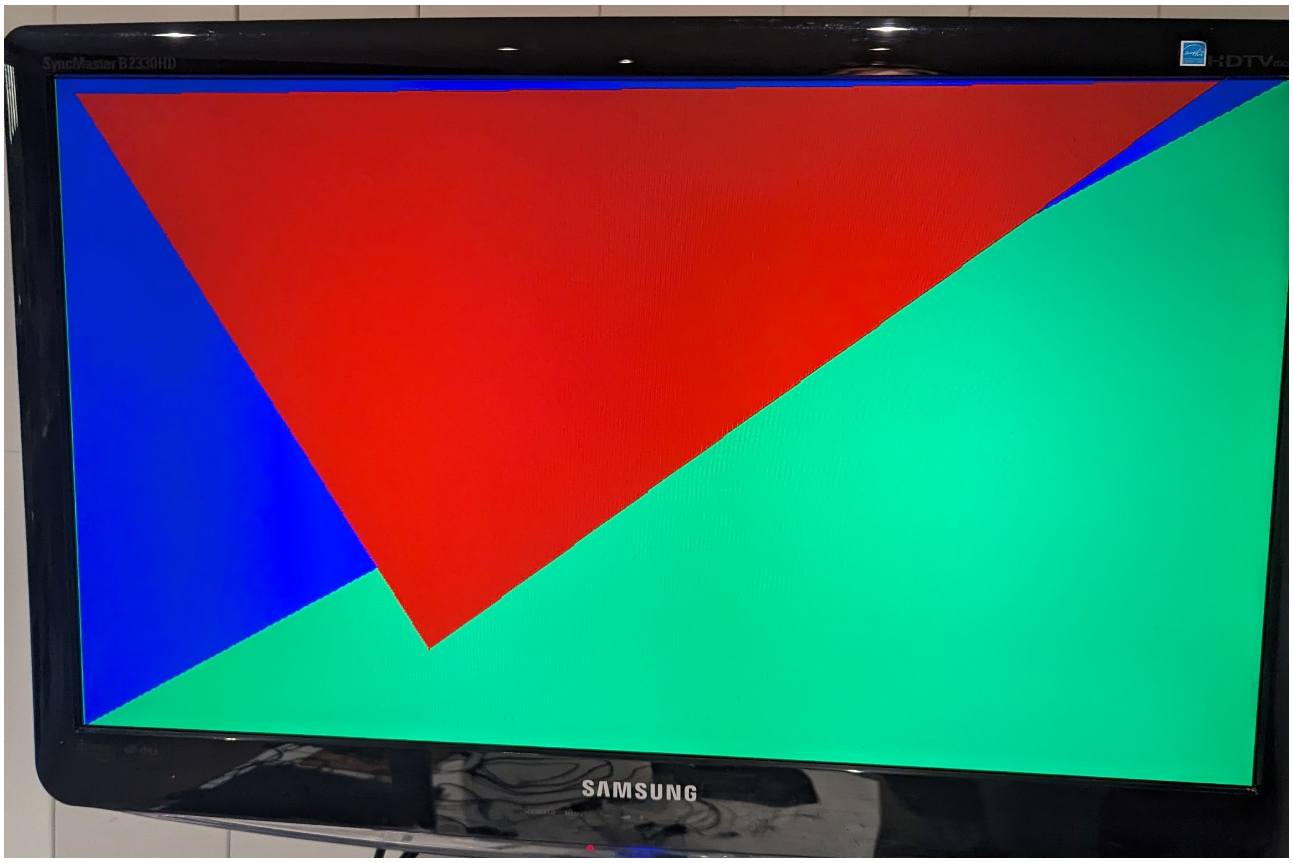
[CMD - 8 bit] + [Color(r0r1g0g1b0b1) - 6 bit][Vertex 0 X - 7 bit][Vertex 1 X - 7 bit][Vertex 2 X - 7 bit][Vertex 0 Y - 6 bit][Vertex 1 Y - 6 bit][Vertex 3 Y - 6 bit][Unused - 3 bits]

Available Commands:
SPI_CMD_WRITE_POLY_A = 0x80
SPI_CMD_CLEAR_POLY_A = 0x40
SPI_CMD_WRITE_POLY_B = 0x81
SPI_CMD_CLEAR_POLY_B = 0x41
SPI_CMD_WRITE_POLY_C = 0x82
SPI_CMD_CLEAR_POLY_C = 0x42
SPI_CMD_WRITE_POLY_D = 0x83
SPI_CMD_CLEAR_POLY_D = 0x43
SPI_CMD_SET_BG_COLOR = 0x01

Note the SPI_CMD_SET_BG_COLOR command only utilizes the 6-bit ‘Color’ field, all other fields are ignored.



Example command setting a blue triangle in the top left corner.



Example of rendering 2 triangles + background color.

External hardware

In order to use the project you will need the following:

- TinyVGA PMOD https://github.com/mole99/tiny-vga
- VGA screen
- Host microcontroller with SPI enabled

Pinout

#	Input	Output	Bidirectional
0		R1	CS
1		G1	MOSI
2		B1	MISO
3		vsync	SCK
4		R0	INT
5		G0	
6		B0	

#	Input	Output	Bidirectional
7		hsync	

KianV RISC-V RV32E Baremetal SoC [588]

- Author: Dipl.-Ing. Hirosh Dabui
- Description: A baremetal RISC-V RV32E ASIC with audio, spi, uart
- GitHub repository
- HDL project
- Mux address: 588
- Extra docs
- Clock: 50000000 Hz

How it works

After implementing a KianV uLinux TT06, I felt like implementing a KianV bare metal edition, which is an RV32E RISC-V32 SoC. This SoC is equipped with a UART, qspi memory controller (psram/flash), a generic SPI interface, and a sigma-delta emulator for playing audio files. In the firmware folder, the kernelboot.c and crt0.S files display all hardware registers and their initialization in the code.

How to test

First, one must build the toolchain for an RV32E, as you can see here:

```
sudo apt-get update
sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-
git clone --recursive https://github.com/riscv/riscv-gnu-toolchain.git
cd riscv-gnu-toolchain
./configure --prefix=/opt/riscv32e --with-arch=rv32e --with-abi=ilp32e
make
export PATH=/opt/riscv32e/bin:$PATH
```

The following hardware addresses are given:

```
#define LSR_DR 0x01
#define LSR_TEMT 0x40
#define LSR_THRE 0x20
#define PWM_ADDR (IO_BASE + 0x14)
#define REG_DIV (IO_BASE + 0x10)
#define SPI_DIV (IO_BASE + 0x500010)
#define UART_LSR (IO_BASE + 0x5)
```

```
#define UART_RX (IO_BASE)
#define UART_TX (IO_BASE)
```

The use of the registers can be determined from the C, linker script and assembly program. The CPU starts to execute the instruction stored in the NOR Flash at an offset of 1MiB. When the chip comes into my hands, I will provide demos that I test on the chip, including audio playback with appropriate documentation.

External hardware

It's very important to use the PMOD Flash + PSRAM. We only use 8MB of PSRAM address space.

Pinout

#	Input	Output	Bidirectional
0	uart_rx	spi_cen0	ce0 flash
1	spi_sio1_so_miso0	spi_sclk0	sio0
2		spi_sio0_si_mosi0	sio1
3		pwm_o	sck
4		uart_tx	sd2
5		led[0]	sd3
6		led[1]	cs1 psram
7		led[2]	always high

TinyTPU [590]

- Author: Refik
- Description: TPU Unit with 2x2 matrix multiplication support
- GitHub repository
- HDL project
- Mux address: 590
- Extra docs
- Clock: 50000 Hz

How it works

The tiny TPU is a accelerator board for multiplying matricies. This version only supports 2x2 matrix multiplication. However, I will be updating the project to muhc more complex computations.

How to test

You can run the test file and see how random two 2x2 matricies are multiplied and passed out.

External hardware

There arent any hardware required for this project. However, you do need a driver to convert matricies and push as an input, then read as a output. I will be providing a simple driver once im done with the RTL design

Pinout

#	Input	Output	Bidirectional
0	ui_in[0]	uo_out[0]	
1	ui_in[1]	uo_out[1]	
2	ui_in[2]		
3	ui_in[3]		
4			
5			
6			
7			

8-bit Vector Compute-in-SRAM [642]

- Author: Ramyad Hadidi
- Description: 8-bit Vectorized Compute-in-SRAM
- GitHub repository
- HDL project
- Mux address: 642
- Extra docs
- Clock: 0 Hz

How it Works

This design is a vector multiplier with stationary weights implemented on 4 tiles in tiny tapeout. The entire design, tests, and documentation took around 10 hours. It contains 8 multiply-and-add (MAC) units, each equipped with two registers, and an adder tree that sums the multiplication results of an 8-element vector without the loss of precision. The design allows for weights and activations to be programmed separately into each MAC unit using specific operation codes (OPs). The weights remain stationary, meaning they are programmed once and reused for multiple activation inputs.

Components and Operation

1. **Multiply-and-Add (MAC) Units:**

   - Each MAC unit consists of two registers: one for weights (W) and one for activations (A). The MAC unit performs the multiplication of these two values and stores the result.
   - The weights and activations are loaded into the MAC units using distinct OP codes (`LOAD_W` and `LOAD_A`). The `LOAD_W` OP code (`0b00`) loads the weight into the MAC unit, while the `LOAD_A` OP code (`0b01`) loads the activation value.
   - After the weights and activations are loaded, the MAC unit multiplies these values and stores the result for further processing by the adder tree.

2. **Adder Tree:**

   - The adder tree is responsible for summing the outputs of the 8 MAC units. It ensures that the multiplication results are accumulated accurately without precision loss.
   - The adder tree is structured hierarchically in three levels to sum the results efficiently.
     - **Level 1:** Adds pairs of MAC outputs.

    - **Level 2:** Adds the results of Level 1.
    - **Level 3:** Adds the results of Level 2 to produce the final sum.
  - This hierarchical structure ensures that the final sum of all MAC outputs is computed correctly and efficiently.

3. **Readout Mechanism:**

   - Due to the limited width of the output interface (8 bits), the readout mechanism reads the final result of the adder tree (`s_adder_tree`) over multiple clock cycles.
   - The `READ_S` OP code (`0b10`) is used to initiate the readout process. The final sum is split into three 8-bit chunks, which are read sequentially.
   - The readout process ensures that the most significant bits (MSBs) are read first, followed by the least significant bits (LSBs), reconstructing the complete 19-bit result correctly.

4. **Programming and Operation:**

   - **Loading Weights and Activations:** The design supports separate loading of weights and activations. Each MAC unit is addressed individually, and the corresponding values are loaded using the appropriate OP codes.
   - **Vector Multiplication:** Once the weights and activations are loaded, the MAC units perform the multiplication, and the results are summed by the adder tree.
   - **Result Readout:** The final sum is read out using the `READ_S` OP code, ensuring the complete result is available for further processing or verification.

This design efficiently handles the multiplication and accumulation of vectors, ensuring high precision and accuracy despite the limitations in the size (even with 4 tiles). It provides a robust mechanism for programming and reading the results, making it suitable for various applications requiring vector multiplications.

How to Test

There are several tests under `test/test.py` that would help anyone to understand how the design works. Due to the limitation of access to the external signals, most of the tests are commented out. Each test also contains its own commented Verilog code since some tests were initially developed to test and verify individual units such as MACs and adders. There are four categories of tests:

- **Single MAC operation test**: This test verifies the basic functionality of a single MAC unit, ensuring that it correctly performs multiplication and stores the result.
- **Multiple MAC units loading weights and activations**: This test checks that multiple MAC units can load weights and activations correctly and simultaneously. It ensures that each MAC unit receives and processes its assigned data independently of the others.
- **Adder tree tests verifying that all levels of adder tree are summing to correct numbers**: These tests validate the correctness of the adder tree at all levels. They ensure that the outputs from the MAC units are correctly summed through the hierarchical adder tree structure, producing accurate intermediate and final sums.
- **Read result tests that would test read out circuit**: These tests focus on the readout circuit, verifying that the `s_adder_tree` result can be correctly read out in multiple 8-bit chunks. This ensures that the readout mechanism accurately reconstructs the full result over several clock cycles.
-
  - **Read only with external signals**: These tests focus on the entire design only using the external signals. This is the only test not commented in the final version.

The last three tests work on several test vectors to ensure correct operation with various numbers. These test vectors include a wide range of values and scenarios to thoroughly exercise the design and confirm its correctness under different conditions.

OP 00: `LOAD_W` The `LOAD_W` function is an integral part of the testbench, designed to load weight values into the MAC units. This function is invoked by setting the `LOAD_W` opcode, which corresponds to the value `0b00 &amp;lt;&amp;lt; 6`. The opcode is combined with the MAC address to target a specific MAC unit for the weight load operation. The MAC address is specified in the lower bits of the `ui_in` signal, allowing precise selection of the MAC unit.

The process begins by setting the `ui_in` input to the `LOAD_W` opcode combined with the target MAC address. This action signals the DUT to prepare for loading the weight value into the specified MAC unit. Simultaneously, the weight value is provided via the `uio_in` input. To ensure that the command and data are properly registered and processed, the function waits for one clock cycle.

By following these steps, the `LOAD_W` function effectively communicates with the DUT to load weight values into the desired MAC units. This operation is crucial for initializing the MAC units with the appropriate weights for subsequent computations.

OP 00: `LOAD_W` Code Snippet for Reference:

```
async def write_weight(mac_address, weight):
    # Set the op code to 00 (write weight) and address
    dut.ui_in.value = (0b00 << 6) | mac_address
    # Set the weight data
    dut.uio_in.value = weight
    # Wait for a clock cycle to simulate the write
    await ClockCycles(dut.clk, 1)
```

OP 01: `LOAD_A` The `LOAD_A` function is another essential component of the testbench, designed to load activation values into the MAC units. This function is activated by setting the `LOAD_A` opcode, which corresponds to the value `0b01 &amp;lt;&amp;lt; 6`. Similar to `LOAD_W`, the opcode is combined with the MAC address to target a specific MAC unit for the activation load operation.

The process starts by setting the `ui_in` input to the `LOAD_A` opcode combined with the target MAC address. This action instructs the DUT to prepare for loading the activation value into the specified MAC unit. Concurrently, the activation value is provided via the `uio_in` input. To ensure the command and data are correctly registered and processed, the function waits for one clock cycle.

By adhering to these steps, the `LOAD_A` function successfully communicates with the DUT to load activation values into the designated MAC units. This operation is vital for initializing the MAC units with the appropriate activation values, enabling accurate computation during the subsequent processing stages.

OP 01: `LOAD_A` Code Snippet for Reference:

```
async def write_act(mac_address, a_value):
    # Set the op code to 01 (write a value) and address
    dut.ui_in.value = (0b01 << 6) | mac_address
    # Set the a value data
    dut.uio_in.value = a_value
    # Wait for a clock cycle to simulate the write
    await ClockCycles(dut.clk, 1)
```

OP 10: `read_s` The `read_s` function is a critical part of the testbench designed to read the final result from the adder tree, known as `s_adder_tree`. Since the output interface of the system can only handle 8 bits at a time, the function retrieves the complete result over multiple clock cycles. The process begins by initializing the `READ_S` command. This is achieved by setting the `ui_in` input to the value corresponding to the `READ_S` opcode (`0b10 &amp;lt;&amp;lt; 6`). This command instructs the system to prepare the `s_adder_tree` result for reading. To ensure the command is properly registered and processed by the Device Under Test (DUT), the function waits for one clock cycle.

Following the initialization of the `READ_S` command, the function sets the `ui_in` input to a non-operational value (`0b11 &amp;lt;&amp;lt; 6`). This step ensures that the command remains stable and does not interfere with the readout process. Another clock cycle wait is introduced to guarantee that the data is ready to be read.

The core of the `read_s` function involves reading the `s_adder_tree` result in 8-bit chunks. The function initializes a variable, `result`, to store the combined output. It then enters a loop that iterates three times, corresponding to the three 8-bit chunks required to construct the 24-bit result. During each iteration, the function waits for the rising edge of the clock to synchronize with the DUT's data output. This synchronization is crucial for accurate data retrieval. The function reads the current 8-bit chunk from the `uo_out` output, shifts the previously read data left by 8 bits, and combines it with the new chunk using a bitwise OR operation. This method ensures that the first chunk read corresponds to the most significant bits (MSBs) and the last chunk read corresponds to the least significant bits (LSBs).

After all three chunks have been read and combined, the function returns the complete `result`, which represents the full 19-bit `s_adder_tree` value. This process highlights the importance of synchronization with the clock signal and handling data in multiple cycles due to the 8-bit limitation of the output interface. By following these steps, the `read_s` function effectively reads and reconstructs the adder tree result, ensuring accurate and reliable verification of the system's computation.

OP 10: `read_s` Code Snippet for Reference:

```
async def read_s():
    # Set the op code to 10 (read s_adder_tree)
    dut.ui_in.value = 0b10 << 6
    await ClockCycles(dut.clk, 1)

    # Set to non-operational value to avoid interference
    dut.ui_in.value = 0b11 << 6
    await ClockCycles(dut.clk, 1)

    result = 0
    for i in range(3):
        await RisingEdge(dut.clk)
        result = (result << 8) | int(dut.uo_out.value)
    return result
```

External hardware

Currently, the compute in SRAM does not interface with any external hardware components, and in reality in should not! It operates entirely within it own resources with its own defined set of control commands. Some external signals might be needed to orchestrate operation of multiple units with a control processor.

Pinout

#	Input	Output	Bidirectional
0	Address bit 0	Data out bit 0	Data in bit 0
1	Address bit 1	Data out bit 1	Data in bit 1
2	Address bit 2	Data out bit 2	Data in bit 2
3	Address bit 3	Data out bit 3	Data in bit 3
4	Address bit 4	Data out bit 4	Data in bit 4
5	Address bit 5	Data out bit 5	Data in bit 5
6	Op Code bit 0	Data out bit 6	Data in bit 6
7	Op Code bit 1	Data out bit 7	Data in bit 7

Mandelbrot Set Accelerator (32-bit IEEE 754) [654]

- Author: Uri Shaked
- Description: Calculates `z = z^2 + c` on every clock cycle using 32-bit IEEE 754 floating point arithmetic.
- GitHub repository
- HDL project
- Mux address: 654
- Extra docs
- Clock: 20000000 Hz

How it works

A mandelbrot set is a set of complex numbers that satisfy a certain mathematical property. The set is defined by iterating a function on a complex number, and checking if the result of the iteration is bounded. If the result is bounded, the complex number is part of the mandelbrot set. If the result is unbounded, the complex number is not part of the mandelbrot set.

This project calculates the function `z = z^2 + c` iteratively, where `z` and `c` are complex numbers. The function is iterated on every clock cycle, and the result is checked to see if it is bounded (`|z| &amp;lt;= 2`). When the result is unbounded, the `unbounded` signal is set high, and the `iter` signal is set to the number of iterations it took for the result to become unbounded.

All the calculations are done in 32-bit IEEE 754 floating point format. The floating point addition code is taken from Caravel FPU, and the floating point multiplication code was generated by GPT-4o.

How to test

Load the value of the complex number `c` that you want to test into the `Cr` and `Ci` registers. Each register holds a 32-bit IEEE 754 floating point number. The value of `c` is `Cr + Ci * i`, where `i` is the imaginary unit.

The registers are shifted in LSB first, 8 bits at a time. When shifting the last byte, the corresponding `load` signal should be set high to load the value into the register.

For example, to load the real part of `c` into `Cr`, you would need four clock cycles:

1. Set `data_in` to the least significant byte (`Cr[7:0]`)
2. Set `data_in` to the second byte (`Cr[15:8]`)
3. Set `data_in` to the third byte (`Cr[23:16]`)

4. Set `data_in` to the most significant byte (`Cr[31:24]`), and set `load_Cr` high to load the value into the register.

Do the same for the imaginary part of `c` and `Ci`. In case you want to load the same value into both `Cr` and `Ci`, you can set `load_Cr` and `load_Ci` high at the same time.

Strobe the `start` signal to begin the calculation. The design will iterate the function `z = z^2 + c`, one iteration per clock cycle. When the result is unbounded, the `unbounded` signal will be set high. For numbers that are part of the mandelbrot set, the `unbounded` signal will remain low as the design iterates the function.

The values of `Cr` and `Ci` are buffered, so you can load new values into the registers while the design is calculating the mandelbrot set for the previous values. When you strobe the `start` signal, the design will begin calculating the mandelbrot set for the new values of `c`.

The following example illustrates how to load the value `c = 1.2 + 1.4i` into the registers and start the calculation:

Clock	`data_in`	`load_Cr`	`load_Ci`	`start`	`unbounded`
1	0x9A	0	0	0	0
2	0x99	0	0	0	0
3	0x99	0	0	0	0
4	0x3F	1	0	0	0
5	0x33	0	0	0	0
6	0x33	0	0	0	0
7	0xB3	0	0	0	0
8	0x3F	0	1	1	0
9	0x00	0	0	0	0
10	0x00	0	0	0	1

Where:

- 0x3F99999A is the IEEE 754 floating point representation of the real part of `c` (1.2)
- 0x3FB33333 is the IEEE 754 floating point representation of the imaginary part of `c` (1.4)
- Unbounded goes high two clock cycles after the start signal is strobed, indicating that the result is unbounded.

When the calculation concludes, the `iter` signal will hold the number of iterations it took for the result to become unbounded. The `iter` signal is valid when the `unbounded` signal is set high.

External hardware

None

Pinout

#	Input	Output	Bidirectional
0	start	unbounded	data_in[0]
1	load_Cr	iter[0]	data_in[1]
2	load_Ci	iter[1]	data_in[2]
3		iter[2]	data_in[3]
4		iter[3]	data_in[4]
5		iter[4]	data_in[5]
6		iter[5]	data_in[6]
7		iter[6]	data_in[7]

raybox-zero TT07 edition [714]

- Author: algofoogle (Anton Maurovic)
- Description: TT07 improved resub of ‘simple VGA ray caster game demo’ from TT04
- GitHub repository
- HDL project
- Mux address: 714
- Extra docs
- Clock: 25000000 Hz

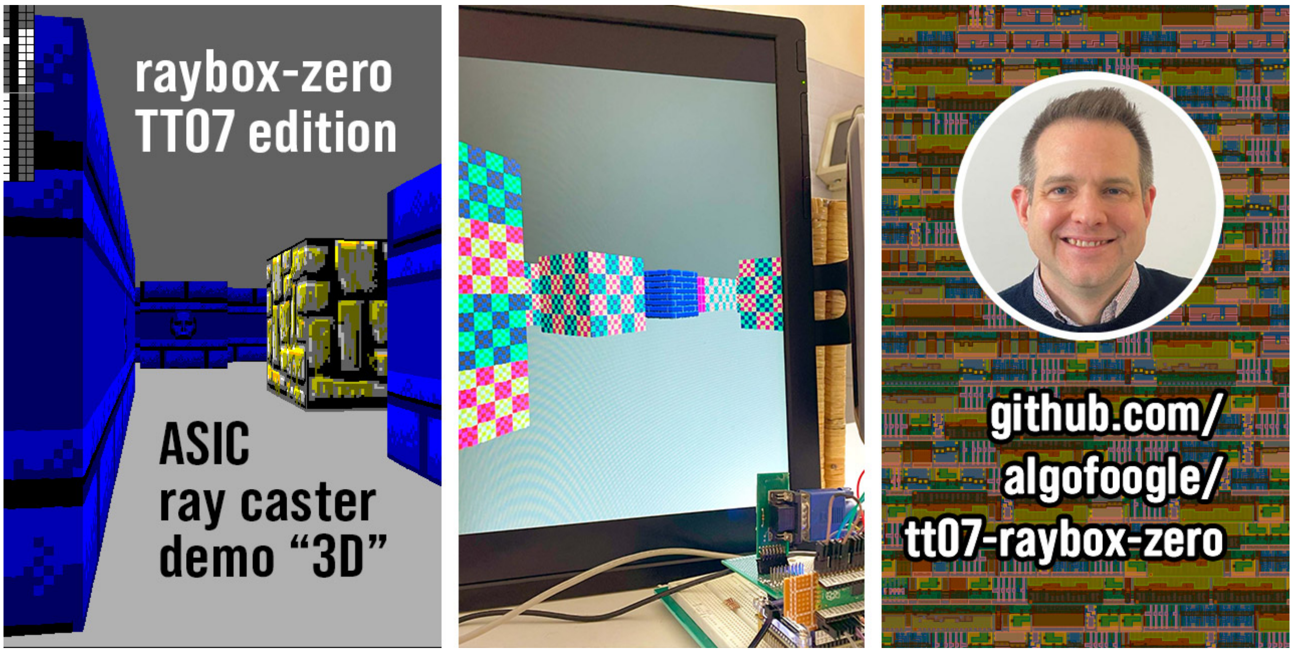


Figure 43: TT07 raybox-zero showing 3D views in simulation and on an FPGA

How it works

This is a framebuffer-less VGA display generator (i.e. it is ‘racing the beam’) that produces a simple implementation of a “3D”-like ray casting game engine… just the graphics part of it. It is inspired by Wolfenstein 3D, using a map that is a grid of wall blocks, with basic texture mapping.

There is nothing yet but textured walls, and flat-coloured floor and ceiling. No doors or sprites, sorry. Maybe that will come in a future version (stay tuned for TT08 or some later release, maybe?)

The ‘player’ POV (“point of view”) is controlled by SPI, which can be used to write the player position, facing X/Y vector, and viewplane X/Y vector in one go.

NOTE: To optimise the design and make it work without a framebuffer, this renders what is effectively a portrait view, rotated. A portrait monitor (i.e. one rotated 90 degrees anti-clockwise) will display this like the conventional first-person shooter view, but it could still be used in a conventional landscape orientation if you imagine it is for a game where you have a first-person perspective of a flat 2D platformer, endless runner, "Descent-style" game, whatever.

NOTE: This is a resubmission of an updated version of what originally went to TT04. While the TT04 version used 4x2 tiles, this TT07 version adds some new features (namely the option for an external SPI texture ROM), and so uses 6x2 tiles (which is about 0.25mm2). The opportunity to do this was largely supported by Uri Shaked and Matt Venn of Tiny Tapeout, who graciously offered a resubmission after it was found that the TT04 version had suffered from a synthesis bug in an older version of OpenLane, which led to severe glitches in the rendering (due to an unintended alteration of the logic).

TBC. Please contact me if you want to know something in particular and I'll put it into the documentation!

How to test

TBC. Please contact me if you want to know something in particular and I'll put it into the documentation!

Supply a clock in the range of 21-31.5MHz; 25.175MHz is ideal because this is meant to be "standard" VGA 640x480@59.94Hz.

Start with `gen_tex` set high, to use internally-generated textures. You can optionally attach an external QSPI memory (`tex_...`) for texture data instead, and then set `gen_tex` low to use it.

`debug` can be asserted to show current state of POV (point-of-view) registers, which might come in handy when trying to debug SPI writes.

If `reg` input is high, VGA outputs are registered. Otherwise, they are just as they come out of internal combo logic. I've done it this way so I can test the difference (if any).

`inc_px` and `inc_py` can be set high to continuously increment their respective player X/Y position register. Normally the registers should be updated via SPI, but this allows someone to at least see a demo in action without having to implement the SPI host controller. NOTE: Using either of these will suspend POV updates via SPI.

The “SPI2” ports (`reg_sclk`, etc.) are for access to all other registers that we can play with. I decided to keep these separate because I implemented them very late, and didn’t want to break the existing SPI interface for POV register access.

External hardware

Tiny VGA PMOD on dedicated outputs (`uo`).

Optional SPI controllers to drive `ui_in[2:0]` (point-of-view aka vectors) and `uio_in[4:2]` (other control/display registers).

Optional external SPI ROM for textures.

TBC. Please contact me if you want to know something in particular and I’ll put it into the documentation!

Pinout

#	Input	Output	Bidirectional
0	SPI in: pov_sclk	red[1]	Out: tex_csb
1	SPI in: pov_mosi	green[1]	Out: tex_sclk
2	SPI in: pov_ss_n	blue[1]	In: “SPI2” reg_sclk
3	debug	vsync_n	In: “SPI2” reg_mosi
4	inc_px	red[0]	In: “SPI2” reg_ss_n
5	inc_py	green[0]	I/O: tex_io0
6	reg	blue[0]	In: tex_io1
7	gen_tex	hsync_n	In: tex_io2

tiny sha256 [718]

- Author: xenia dragon
- Description: a minimal single-cycle-round sha256 core intended to fit in one tile
- GitHub repository
- HDL project
- Mux address: 718
- Extra docs
- Clock: 50000000 Hz

How it works

This is a minimal SHA-256 hash core implemented in a single-cycle-round architecture. TODO: expand on this

How to test

TODO: write instructions

External hardware

No external hardware is needed besides some method of interacting with the bus to transfer commands and data.

Pinout

#	Input	Output	Bidirectional
0	addressed register bit 0	ready	data bit 0
1	addressed register bit 1	IO read/write selected	data bit 1
2	addressed register bit 2	todo0	data bit 2
3	addressed register bit 3	todo1	data bit 3
4	addressed register bit 4	todo2	data bit 4
5	addressed register bit 5	todo3	data bit 5
6	IO read/write select	todo4	data bit 6
7	IO clock	todo5	data bit 7

SPELL [782]

- Author: Uri Shaked
- Description: SPELL is a minimal, cryptic, stack-based programming language crafted for The Skull CTF
- GitHub repository
- HDL project
- Mux address: 782
- Extra docs
- Clock: 10000000 Hz

How it works

SPELL is a minimal, stack-based programming language created for The Skull CTF.

The language is defined by the following cryptic piece of Arduino code:

```
void spell() {

                  uint8_t*a,pc=16,sp=0,
               s[32]={0},op;while(!0){op=
            EEPROM.read(pc);switch(+op){case
          ',':delay(s[sp-1]);sp--;break;case'>':
         s[sp-1]>>=1|1;break;case'<':s[sp-1]<<=1;
        break;case'=':pc=s[sp-1]-1;sp--;break;case
       '@':if(s[sp-2]){s[sp-2]--;pc=s[sp-1]-1;sp+=
      1;}sp-=2;break;case'&':s[sp-2]&=s[sp-1];sp-=1;
      break;case'|':s[sp-2]|=s[sp-1];sp-=1;break;case
    '^':s[sp-2]^=s[sp-1];sp--;break;case'+':s[sp-2]+=
   s[sp-1];sp=sp-1;break;case'-':s[sp-2]-=s[sp-1];sp--;
   break;case'2':s[sp]=s[sp-1];sp=sp+1;break;case'?':s[
 sp-1]=EEPROM.         read(s[sp-1]|0        );break;case
  "!!!"[0]:             666,EEPROM              .write(s
   [sp-1]                ,s[sp-2]                );sp=+
    sp-02;               ;break;                 1;case
    "Arr"[               1]:  s[+               sp-1]=
    *(char*)            (s[+   sp-1           ]);break
      ;case'w':*   (char*)(     s[+sp-1])  =s[sp-+2];
        sp-=2;break;case+       'x':s[sp] =s[sp-1
           ];s[sp-1]=s[sp   +    -2];s[sp-2]=s[
            0|sp];break;    ;;    case"zzz"[0
             ]:sleep();"   Arrr  ";break;case
             255   :return;;  default:s  [sp]
              =+    op;sp+=    1,1    ;}pc=
               +     pc   +     1;       %>

}
```

This design is an hardware implementation of SPELL with the following features:

- 256 bytes of program memory (volatile, simulates EEPROM)
- 32 bytes of stack memory
- 32 bytes of data memory
- 8 bidirectional pins and up to 8 output-only pins

Initially, all the program memory is filled with `0xFF`, and the stack and data memory are filled with `0x00`. The program counter is set to `0x00`, and the stack pointer is set to `0x00`.

To load a program or inspect the internal state, the design provides access to the following registers via a simple serial interface:

Address	Register name	Description
0x00	PC	Program counter
0x01	SP	Stack pointer
0x02	EXEC	Execute-in-place (write-only)
0x03	STACK	Stack access (read the top value, or push a value)

The serial interface is implemented using a shift register, which is controlled by the following signals:

Pin	Type	Description
`reg_sel`	input	Select the register to read/write
`load`	input	Load the selected register with the value from the shift register
`dump`	input	Dump the selected register value to the shift register
`shift_in`	input	Serial data input
`shift_out`	output	Serial data output (when `porta[3]` is disabled)

When `load` is high, the value from the shift register is loaded into the selected register. When `dump` is high, the value of the selected register is dumped into the shift register, and can be read after two clock cycles by reading `shift_out` (MSB first).

For example, if you want to read the value of the program counter (PC), you would:

1. Set `reg_sel` to 0x00 and set `dump` to 1
2. Wait for two clock cycles for the first bit (MSB) to appear on `shift_out`.
3. Read the remaining bits from `shift_out` on each clock cycle.

To write a value to the program counter, you would:

1. Write the value to the shift register, one bit at a time, starting with the **MSB**.
2. Set `reg_sel` to 0x00 and set `load` to 1.
3. Wait for a single clock cycle for the value to be loaded.

Writing an opcode to the `EXEC` register will execute the opcode in place, without modifying the program counter (unless the opcode is a jump instruction).

The `STACK` register is used to push a value onto the stack or read the top value from the stack (for debugging purposes).

Data memory and registers The data memory space is divided into two regions:

Address range	Description
0x00 - 0x1F	General-purpose data storage (data memory)
0x20 - 0x5F	I/O and control registers

Other addresses are reserved for future use, and should not be accessed.

The following registers are available in the data memory space:

Address	Name	Description
0x36	PINB	Read the value of the `portb` pins, or toggle the output when written to
0x37	DDRB	Set the direction of the `portb` pins (0 = input, 1 = output)
0x38	PORTB	Write to the `portb` pins
0x39	PINA	Toggle the output on `porta` pins (write only; read returns 0x00)
0x3A	DDRA	Enables of the `porta` pins (0 = disabled, 1 = output)
0x3B	PORTA	Write to the `porta` (output only) pins

For example, to toggle the value of the `portb[2]` (`uio[2]`) pin, you would write `0x04` to the `PINB` register.

The `porta[3:0]` pins are also used for debug output, and their function is determined by the `DDRA` register:

Output pin	DDRA[n] == 0	DDRA[n] == 1
0	`sleep`	`porta[0]`
1	`stop`	`porta[1]`
2	`wait_delay`	`porta[2]`
3	`shift_out`	`porta[3]`
4	0	`porta[4]`
5	0	`porta[5]`
6	0	`porta[6]`
7	0	`porta[7]`

How to test

To test SPELL, you need to load a program into the program memory and execute it. You can load the program by repeatedly executing the following steps for each byte of the program:

1. Write the byte to the top of the stack (using the `STACK` register)
2. Write the address of the byte in the program memory to top of the stack
3. Write the opcode `!` to the `EXEC` register

After loading the program, you can execute it by writing the address of the first byte in the program memory to the `PC` register, and then pulsing the `run` signal.

Test programs The following program will spell "SPELL" on the Tiny Tapeout demo board's 7-segment display: (see what we did there?)

```
[127, 58, 119, 0, 129, 57, 57, 244, 62, 116, 109, 50, 0, 38, 94, 59, 119,
```

The program bytes should be loaded into the program memory starting at address 0.

And of course, the obligatory blink, rapidly blinking an LED connected to the `uio[0]` pin:

```
[1, 55, 119, 1, 54, 119, 250, 44, 3, 61]
```

External hardware

None

Errata

When reseting the chip, bytes 0-3 and 128-131 of the program memory are not reset to `0xFF`, and retain their values from the last program loaded (or a random value on power-up). All other program memory bytes are reset to `0xFF`, as expected. This happens due to a timing issue with the DFFRAM write operation that affects the first word of each program memory bank.

Pinout

#	Input	Output	Bidirectional
0	run	sleep/porta[0]	portb[0]
1	step	stop/porta[1]	portb[1]
2	load	wait_delay/porta[2]	portb[2]
3	dump	shift_out/porta[3]	portb[3]
4	shift_in	porta[4]	portb[4]
5	reg_sel[0]	porta[5]	portb[5]
6	reg_sel[1]	porta[6]	portb[6]
7		porta[7]	portb[7]

LISA Microcontroller with TTLC [846]

- Author: Ken Pettit
- Description: 8-Bit Microcontroller SOC with Tiny Tapeout Logic Controller
- GitHub repository
- HDL project
- Mux address: 846
- Extra docs
- Clock: 50000000 Hz

What is LISA?

LISA is a Microcontroller built around a custom 8-Bit Little ISA (LISA) microprocessor core. It includes several standard peripherals that would be found on commercial microcontrollers including timers, GPIO, UARTs and I2C. The following is a block diagram of the LISA Microcontroller:

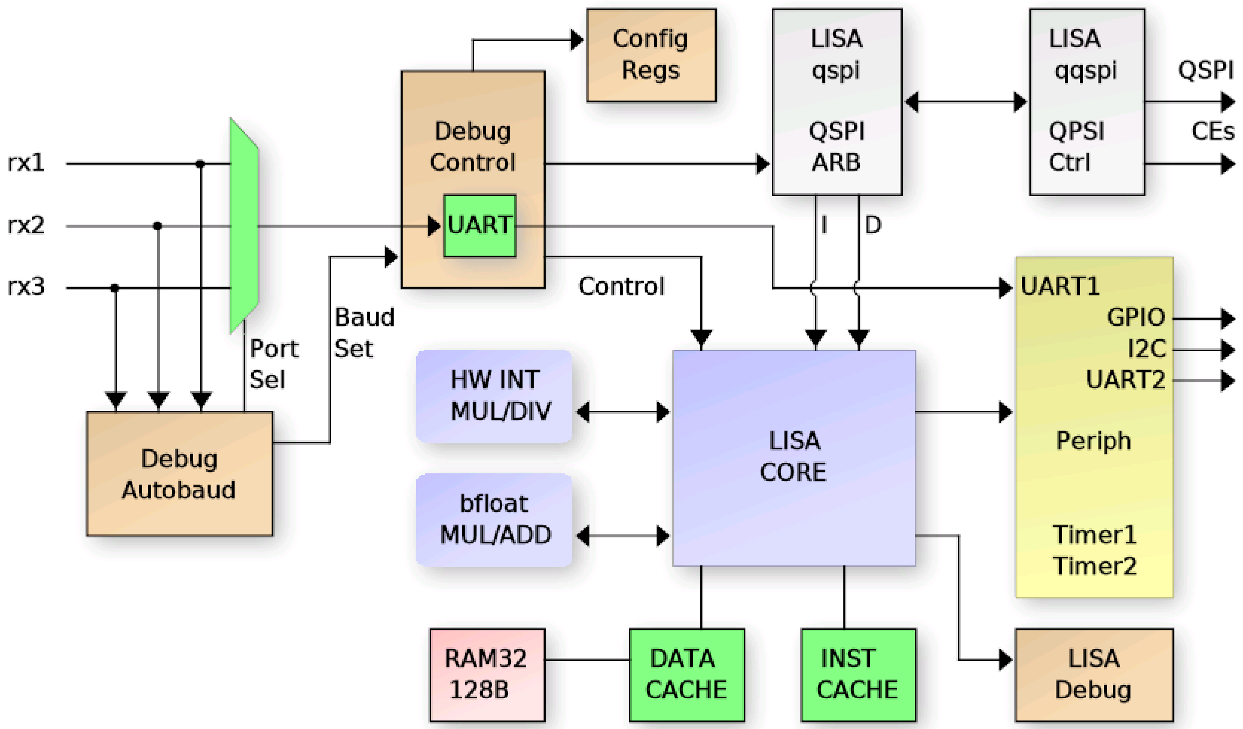


Figure 974.1: LISA Microcontroller Block Diagram

- The LISA Core has a minimal set of register that allow it to run C programs:
  - Program Counter + Return Address Resister
  - Stack Pointer and Index Register (Indexed DATA RAM access)
  - 8-bit Accumulator + 16-bit BF16 Accumulator and 4 BF16 registers

Deailed list of the features

- Harvard architecture LISA Core (16-bit instruction, 15-bit address space)
- Debug interface
    - UART controlled
    - Auto detects port from one of 3 interfaces
    - Auto detects the baud rate
    - Interfaces with SPI / QSPI SRAM or FLASH
    - Can erase / program the (Q)SPI FLASH
    - Read/write LISA core registers and peripherals
    - Set LISA breakpoints, halt, resume, single step, etc.
    - SPI/QSPI programmability (single/quad, port location, CE selects)
- (Q)SPI Arbiter with 3 access channels
    - Debug interface for direct memory access
    - Instruction fetch
    - Data fetch
    - Quad or Single SPI. Hereafter called QSPI, but supports either.
- Onboard 128 Byte RAM for DATA / DATA CACHE
- Data bus CACHE controller with 8 16-byte CACHE lines
- Instruction CACHE with a single 4-instruction CACHE line
- Two 16-bit programmable timers (with pre-divide)
- Debug UART available to LISA core also
- Dedicated UART2 that is not shared with the debug interface
- 8-bit Input port (PORTA)
- 8-bit Output port (PORTB)
- 4-bit BIDIR port (PORTC)
- I2C Master controller
- Hardware 8x8 integer multiplier
- Hardware 16/8 or 16/16 integer divider
- Hardware Brain Float 16 (BF16) Multiply/Add/Negate/Int16-to-BF16
- Programmable I/O mux for maximum flexibility of I/O usage.

It uses a 32x32 1RW DFFRAM macro to implement a 128 bytes (1 kilobit) RAM module. The 128 Byte ram can be used either as a DATA cache for the processor data bus, giving a 32K Byte address range, or the CACHE controller can be disabled, connecting the Lisa processor core to the RAM directly, limiting the data space to 128 bytes. Inclusion of the DFFRAM is thanks to Uri Shaked (Discord urish) and his DFFRAM example.

Reseting the project **does not** reset the RAM contents.

Connectivity

All communication with the microcontroller is done through a UART connected to the Debug Controller. The UART I/O pins are auto-detected by the debug_autobaud module from the following choices (RX/TX):

```
ui_in[3]  / ui_out[4]     RP2040 UART interface
uio_in[4] / uio_out[5]    LISA PMOD board (I am developing)
uio_in[6] / uio_out[5]    Standard UART PMOD
```

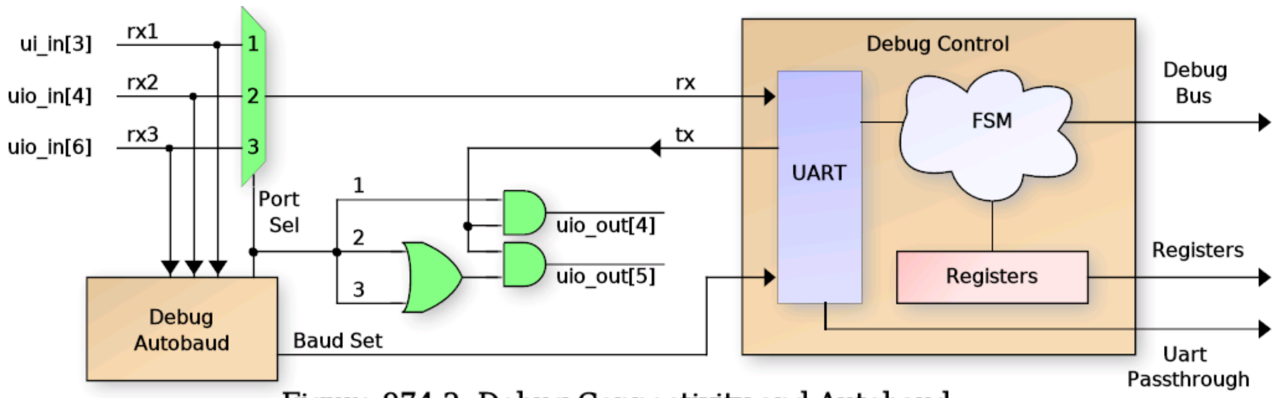


Figure 974.2: Debug Connectivity and Autobaud

The RX/TX pair port is auto-detected after reset by the autobaud circuit, and the UART baud rate can either be configured manually or auto detected by the autobaud module. After reset, the ui_in[7] pin is sampled to determine the baud rate selection mode. If this input pin is HIGH, then autobaud is disabled and ui_in[6:0] is sampled as the UART baud divider and written to the Baud Rate Generator (BRG). The value of this divider should be: clk_freq / baud_rate / 8 - 1. Due to last minute additions of complex floating point operations, and only 2 hours left on the count-down clock, the timing was relaxed to 20MHz input clock max. So for a 20MHz clock and 115200 baud, the b_div[6:0] value would be 42 (for instance).

If the ui_in[7] pin is sampled LOW, then the autobaud module will monitor all three potential RX input pins for LINEFEED (ASCII 0x0A) code to detect baud rate and set the b_div value automatially. It monitors bit transistions and searches for three successive bits with the same bit period. Since ASCII code 0x0A contains a "0 1 0 1 0" bit sequence, the baud rate can be detected easily.

Regardless if the baud rate is set manually or using autobaud, the input port selection will be detect automatically by the autobaud. In the case of manual buad rate selection, it simply looks for the first transition on any of the three RX pins. For autobaud, it selects the RX line with three successive equivalent bit periods.

Debug Interface Details The Debug interface uses a fixed, Verilog coded Finite State Machine (FSM) that supports a set of commands over the UART to interface with the microcontroller. These commands are simple ASCII format such that low-level testing can be performed using any standard terminal software (such as minicom, tio. Putty, etc.). The 'r' and 'w' commands must be terminated using a NEWLINE (0x0A) with an optional CR (0x0D). Responses from the debug interface are always terminated with a LINFEED plus CR sequence (0x0A, 0x0D). The commands are as follows (responsce LF/CR ommited):

Command	Description
v	Report Debugger version. Should return: lisav1.2
wAAVVVV	Write 16-bit HEX value 'VVVV' to register at 8-bit HEX address 'AA'.
rAA	Read 16-bit register value from 8-bit HEX address 'AA'.
t	Reset the LISA core.
l	Grant LISA the UART. Further data will be ignored by the debugger.
+++	Revoke LISA UART. NOTE: a 0.5s guard time before/after is required.

NOTE: All HEX values must be a-f and not A-F. Uppercase is not supported.

Debug Configuration and Control Registers The following table describes the configuration and LISA debug register addresses available via the debug 'r' and 'w' commands. The individual register details will be described in the sections to follow.

ADDR	Description	ADDR	Description
0x00	LISA Core Run Control	0x12	LISA1 QSPI base address
0x01	LISA Accumulator / FLAGS	0x13	LISA2 QSPI base address
0x02	LISA Program Counter (PC)	0x14	LISA1 QSPI CE select
0x03	LISA Stack Pointer (SP)	0x15	LISA2 QSPI CE select
0x04	LISA Return Address (RA)	0x16	Debug QSPI CE select
0x05	LISA Index Register (IX)	0x17	QSPI Mode (QUAD, flash, 16b)
0x06	LISA Data bus	0x18	QSPI Dummy read cycles
0x07	LISA Data bus address	0x19	QSPI Write CMD value
0x08	LISA Breakpoint 1	0x1a	The '+++' guard time count
0x09	LISA Breakpoint 2	0x1b	Mux bits for uo_out
0x0a	LISA Breakpoint 3	0x1c	Mux bits for uio
0x0b	LISA Breakpoint 4	0x1d	CACHE control
0x0c	LISA Breakpoint 5	0x1e	QSPI edge / SCLK speed
0x0d	LISA Breakpoint 6	0x20	Debug QSPI Read / Write
0x0f	LISA Current Opcode Value	0x21	Debug QSPI custom command
0x10	Debug QSPI Address (LSB16)	0x22	Debug read SPI status reg

ADDR	Description	ADDR	Description
0x11	Debug QSPI Address (MSB8)		

LISA Processor Interface Details The LISA Core requires external memory for all Instructions and Data (well, sort of for data, the data CACHE can be disabled then it just uses internal DFFRAM). To accomodate external memory, the design uses a QSPI controller that is configurable as either single SPI or QUAD SPI, Flash or SRAM access, 16-Bit or 24-Bit addressing, and selectable Chip Enable for each type of access. To achieve this, a QSPI arbiter is used to allow multiple accessors as shown in the following diagram:

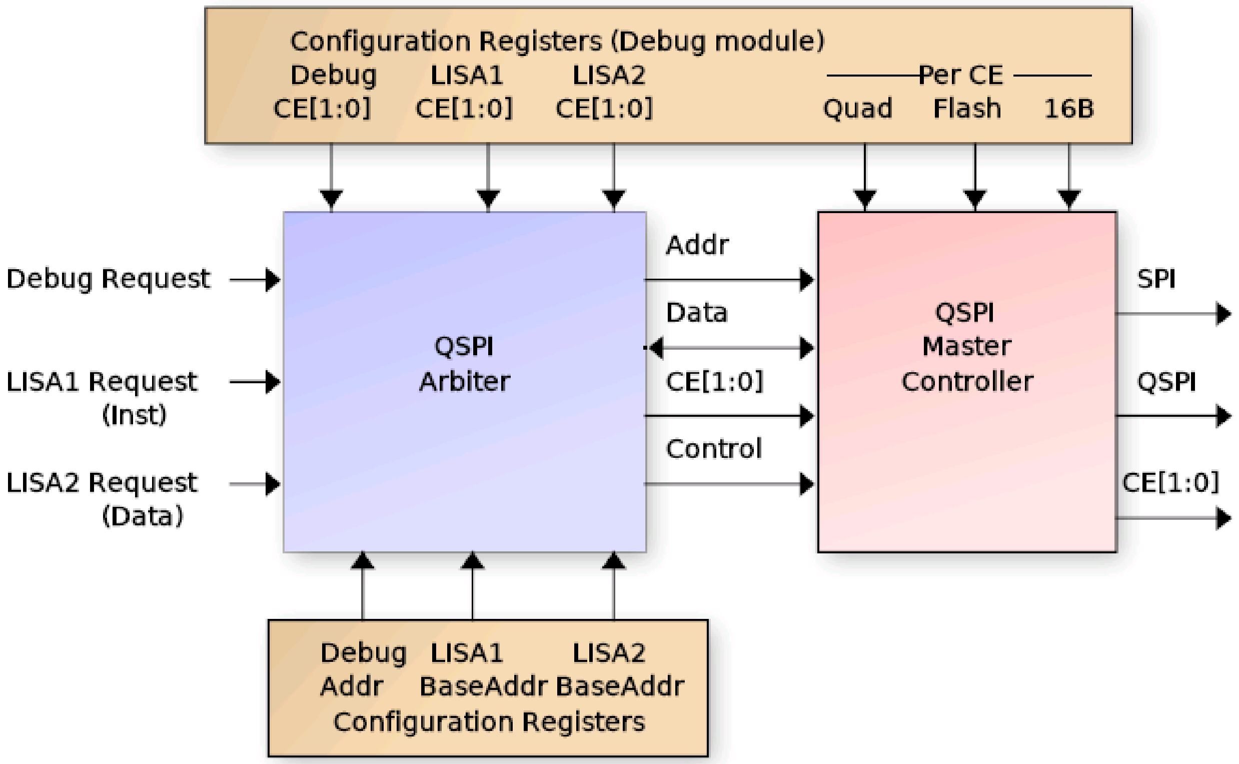


Figure 974.3: (Q)SPI Controller Interface Diagram

The arbiter is controlled via configuration registers (accessible by the Debug controller) that specify the operating mode per CE, and CE selection bits for each of the three interfaces:

- Debug Interface
- LISA1 (Instruction fetch)
- LISA2 (Data read/write)

The arbiter gives priority to the Debug accesses and processes LISA1 and LISA2 requests using a round-robbin approach. Each requestor provides a 24-bit address along with 16-bit data read/write. For the Debug interface, the address comes from the configuration registers directly. For LISA1, the address is the Program Counter (PC) + LISA1 Base and for LISA2, it is the Data Bus address + LISA2 Base. The LISA1 and LISA2 base addresses are programmed by the Debug controller and set the upper 16-bits in the 24-bit address range. The PC and Data address provide the lower 16 bis (8-bits overlapped that are 'OR'ed together). The BASE addresses allow use of a single external QSPI SRAM for both instruction and data without needing to worry about data collisions.

When the arbiter chooses a requestor, it passes its programmed CE selection to the QSPI controller. The QSPI controller then uses the programmed QUAD, MODE, FLASH and 16B settings for the chosen CE to process the request. This allows LISA1 (Instruction) to either execute from the same SRAM as LISA2 (Data) or to execute from a separate CE (such as FLASH with permanent data storage).

Additionally the Debug interface has special access registers in the 0x20 - 0x22 range that allow special QSPI accesses such as FLASH erase and program, SRAM programming, FLASH status read, etc. In fact the Debug controller can send any arbitrary command to a target device, using access that either provide an associated address (such as erase sector) or no address. The proceedure for this is:

1. Program Debug register 0x19 with the special 8-bit command to be sent
2. Set the 9-th bit (reg19[8]) to 1 if a 16/24 bit address needs to be sent)
3. Perform a read / write operation to debug address 0x21 to perform the action.

Simple QSPI data reads/write are accomplished via the Debug interface by setting the desired address in Debug config register 0x10 and 0x11, then performing read or write to address 0x20 to perform the request. Reading from Debug config register 0x22 will perform a special mode read of QSPI register 0x05 (the FLASH status register).

Data access to the QSPI arbiter come from the Data CACHE interface (described later), enabling a 32K address space for data. However the design has a CACHE disable mode that directs all Data accesses directly to the internal 128 Byte RAM, thus eliminating the need for external SRAM (and limiting the data bus to 128 bytes).

Programming the QSPI Controller Before the LISA microcontroller can be used in any meaningful manner, a SPI / QSPI SRAM (and optionally a NOR FLASH) must be connected to the Tiny Tapeout PCB. Alternately, the RP2040 controller on the board can be configured to emulate a single SPI (the details for configuring this are outside the scope of this documentation … search the Tiny Tapeout website for details.). For the CE signals, there are two operating modes, fixed CE output and Mux Mode 3 “latched” CE mode. Both will be described here. The other standard SPI signals are routed to dedicated pins as follows:

Pin	SPI	QSPI	Notes
uio[0]	CE0	CE0	
uio[1]	MOSI	DQ0	Also MOSI prior to QUAD mode DQ0
uio[2]	MISO	DQ1	Also MISO prior to QUAD mode DQ1
uio[3]	SCLK	SCLK	
uio[4]	CE1	CE1	Must be enabled via uio MUX bits
uio[6]	-	DQ2	Must be enabled via uio MUX bits
uio[7]	-	DQ3	Must be enabled via uio MUX bits

For Special Mux Mode 3 (Debug register 0x1C uio_mux[7:6] = 2'h3), the pinout is mostly the same except the CE signals are not constant. Instead they are "latched" into an external 7475 type latch. This mode is to support a PMOD board connected to the uio PMOD which supports a QSPI Flash chip, a QSPI SRAM chip, and either Debug UART or I2C. For all of that functionality, nine pins would be required for continuous CE0/CE1, however only eight are available. So the external PMOD uses uio[0] as a CE "latch" signal and the CE0/CE1 signals are provided on uio[1]/uio[2] during the latch event. This requires a series resistor as indicated to allow CE updates if the FLASH/SRAM is driving DQ0/DQ1. The pinout then becomes:

Pin	SPI/QSPI	Notes
uio[0]	ce_latch	ce_latch HIGH at beginning of cycle
uio[1]	ce0_latch/MOSI/DQ0	Connection to FLASH/SRAM via series resistor
uio[2]	ce1_latch/MISO/DQ1	Connection to FLASH/SRAM via series resistor
uio[3]	SCLK	
uio[6]	-/DQ2	Must be enabled via uio MUX bits
uio[7]	-/DQ3	Must be enabled via uio MUX bits

This leaves uio[4]/uio[5] available for use as either UART or I2C.

Once the SPI/QSPI SRAM and optional FLASH have been chosen and connected, the Debug configuration registers must be programmed to indicate the nature of the external device(s). This is accompilished using Debug registers 0x12 - 0x19 and 0x1C. To programming the proper mode, follow these steps:

1. Program the LISA1, LISA2 and Debug CE Select registers (0x14, 0x15, 0x16) indicating which CE to use.
   - 0x14, 0x15, 0x16: {6'h0, ce1_en, ce0_en} Active HIGH
2. Program the LISA1 and LISA2 base addresses if they use the same SRAM:

- 0x12: {LISA1_BASE, 8'h0} | {8'h0, PC}
- 0x13: {LISA2_BASE, 8'h0} | {8'h0, DATA_ADDR}

3. Program the mode for each Chip Enable (bits active HIGH)
   - 0x17: {10'h0, is_16b[1:0], is_flash[1:0], is_quad[1:0]}
4. For Quad SPI, Special Mux Mode 3, or CE1, program the uio_mux mode:
   - 0x1C:
     - [7:6] = 2'h2: Normal QSPI DQ2 select
     - [7:6] = 2'h3: Special Mux Mode 3 (Latched CE)
     - [5:4] = 2'h2: Normal QSPI DQ3 select
     - [5:4] = 2'h3: Special Mux Mode 3
     - [1:0] = 2'h2: CE1 select on uio[4]
5. For RP2040, you might need to slow down the SPI clock / delay between successive CE activations:
   - 0x1E:
     - [3:0] spi_clk_div: Number of clocks SCLK HIGH and LOW
     - [10:4] ce_delay: Number clocks between CE activations
     - [12:11] spi_mode: Per-CE FALLING SCLK edge data update
6. Set the number of DUMMY ready required for each CE:
   - 0x18: {8'h0, dummy1[3:0], dummy0[3:0]
7. For QSPI FLASH, set the QSPI Write opcode (it is different for various Flashes):
   - 0x19: {8'h0, quad_write_cmd}

NOTE: For register 0x1E (SPI Clock Div and CE Delay), there is only a single register, meaning this register value applies to both CE outputs. Delaying the clock of one CE will delay both, and adding delay between CE activations does not keep track of which CE was activated. So if two CE outputs are used and a CE delay is programmed, it will enforce that delay even if a different CE is used. This setting is really in place for use when the RP2040 emulation is being used in a single CE SRAM mode only (i.e. you have no external PMOD with a real SRAM / FLASH chip. In the case of real chips on a PMOD, SCLK and CE delays (most likely) are not needed. The Tech Page on the Tiny Tapeout regarding RP2040 SPI SRAM emulation indicates a delay between CE activations is likely needed, so this setting is provided in case it is needed.

Architecture Details

Below is a simplified block diagram of the LISA processor core. It uses an 8-bit accumulator for most of its operations with the 2nd argument predominately coming from either immediate data in the instruction word or from a memory location addressed by either the Stack Pointer (SP) or Index Register (IX).

There are also instructions that work on the 15-bit registers PC, SP, IX and RA (Return Address). As well as floating point operations. These will be covered in the sections to follow.

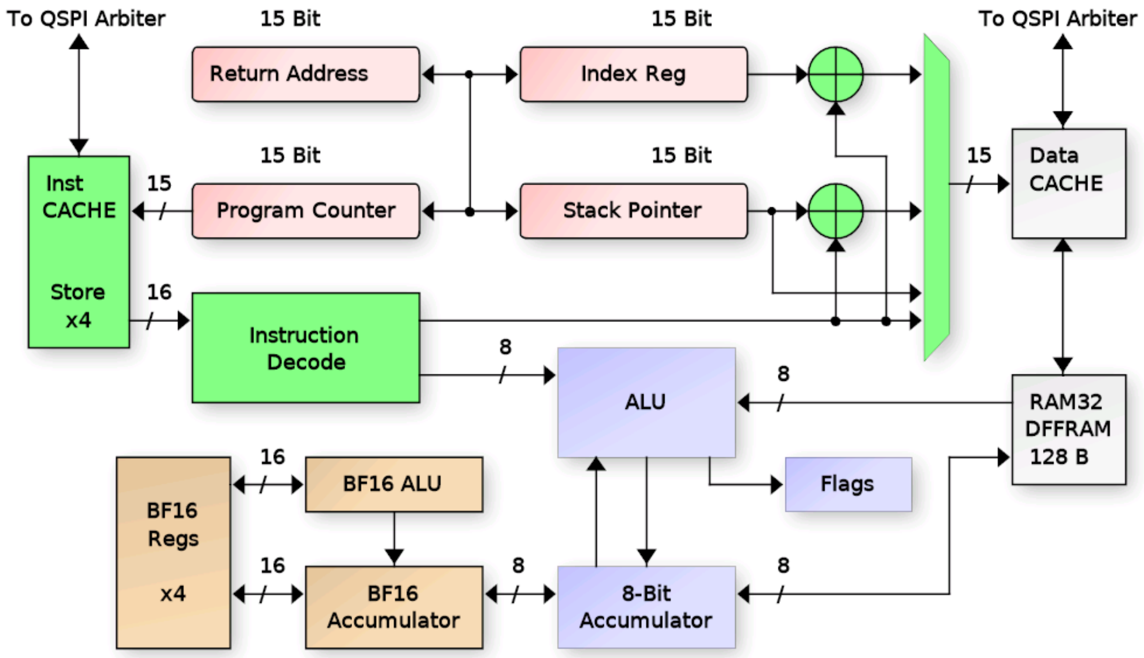


Figure 974.4: Simplified LISA Processor Block Diagram

Addressing Modes Like most processors, LISA has a few different addressing modes to get data in and out of the core. These include the following:

Mode	Data	Description
Register	Rx[n -: 8]	Transfers between registers (ix, ra, facc, etc.).
Direct	inst[n:0]	N-bit data stored in the instruction word.
NextOp	(inst+1)[14:0]	Data stored in the NEXT instruction word.
Indirect	mem[inst[n:0]]	Address of the data is in the instruction word.
Periph	periph[inst[n:0]]	Accesses to the peripheral bus.
Indexed	mem[sp/ix+inst[n:0]]	The SP or IX register is added to a fixed offset.
Stack	mem[sp]	Stack pointer points to the data (push/pop).

The Control Registers To run meaninful programs, the Program Counter (PC) and Stack Pointer (SP) must be set to useful values for accessing program instructions and data. The PC is automatically reset to zero by rst_n, so that one is pretty much automatic. All programs start at address zero (plus any base address programmed by the Debug Controller). But as far as the LISA core is concerned, it knows nothing of base addresses and believes it is starting at address zero.

Next is to program the SP to point to a useful location in memory. The Stack is a place where C programs store their local variable values and also where we store the Return Address (RA) if we need to call nested routines, etc. The stack grows down, meaning it starts at a high RAM address and decrements as things are added to the stack. Therefore the SP should be programmed with an address in upper RAM. LISA supports different Data bus modes through it's CACHE controller, including CACHE disable where it can only access 128 bytes. But for this example, let's assume we have a full range of 32K SRAM available. The LISA ISA doesn't have an opcode for loading the SP directly. Instead it can load the IX register directly with a 15-bit value using NextOp addressing, and it supports "xchg" opcodes to exchange the IX register with either the SP or RA. So to load the SP, we would write:

```
Example:
  ldx     0x7FFF      // Load IX with value in next opcode
  xchg_sp             // Exchange IX with SP
```

The IX register can be programmed as needed to access other data within the Data Bus address range. This register is useful especially for accessing structures via a C pointer. The IX then becomes the value of the pointer to RAM, and Indexed addressing mode allows fixed offsets from that pointer (i.e. structure elements) to be accessed for read/write.

Loading the PC indirectly can be done using the "jmp ix" opcode which does the operation pc <= ix. Loading ix from the pc directly is not supported, though this can be accomplished using a function call and opcodes to save RA (sra) and pop ix:

```
Example:
  get_pc:
    sra         // Push RA to the stack (Save RA)
    pop_ix      // Pop IX from the stack
    ret         // Return. Upon return, IX is the same as PC
```

Conditional Flow Processing Program flow is controlled using flags (zero, carry, sign), arithemetic mode (amode) and condition flags (cond) to determine when program branches should occur. Specific opcode update the flags and condition registers based on results of the operation (AND, OR, IF, etc.). Then conditional branches are made using bz, bnz and if (and variants ifte "if-then-else" and iftt "if-then-then"). Also available are rc "Return if Carry" and rz "Return if Zero", though these are less useful in C programs as typically a routine uses local variables and the stack must be restored prior to return, mandating a branch to the function epilog to restore the stack and often the return address. Below is a list of the opcodes used for conditional program processing:

Legend for operations below:

- acc_val = inst[7:0]
- pc_jmp = inst[14:0]
- pc_rel = pc + sign_extend(inst[10:0])

Opcode	Operation	Encoding	Description
jal	pc <= pc_jmp ra <= pc	0aaa_aaaa_aaaa_aaaa	Jump And Link (call).
ret	pc <= ra	1000_1010_0xxx_xxxx	Return
reti	pc <= ra acc <= acc_val	1000_11xx_iiii_iiii	Return Immediate.
br	pc <= pc_rel	1011_0rrr_rrrr_rrrr	Branch Always
bz	pc <= pc_rel if zero=1	1011_1rrr_rrrr_rrrr	Branch if Zero.
bnz	pc <= pc_rel if zero=0	1010_1rrr_rrrr_rrrr	Branch if Not Zero.
rc	pc <= ra if carry=1	1000_1011_0xxx_xxxx	Return if Carry
rz	pc <= ra if zero=1	1000_1011_1xxx_xxxx	Return if Zero
call_ix	pc <= ix ra <= pc	1000_1010_100x_xxxx	Call indirect via IX
jump_ix	pc <= ix	1000_1010_101x_xxxx	Jump indirect via IX
if	cond <= ??	1010_0010_0000_0ccc	If. See below.
iftt	cond <= ??	1010_0010_0000_1ccc	If then-then. See below.
ifte	cond <= ??	1010_0010_0001_0ccc	If then-else. See below.

The IF Opcode The "if" opcode and it's variants "if-then-then" and "if-then-else" control program flow in a slightly different manner than the others. Instead of affecting the value of the PC directly, they set the two condition bits “cond[1:0]” to indicate which (if any) of the two following opcodes should be executed. the cond[0] bit represents the next instruction and cond[1] represents the instruction following that. All three “if” forms take an argument that checks the current value of the FLAGS to set the condition bits. The argument is encoded as the lower three bits of the instruction word ard operate as shown in the following table:

Condition	Test	Encoding	Description
EQ	zflag=1	3’h0	Execute if Equal
NE	zflag=0	3’h1	Execute if Not Equal
NC	cflag=0	3’h2	Execute if Not Carry
C	cflag=1	3’h3	Execute if Carry
GT	~cSigned & ~zflag	3’h4	Execute if Greater Than
LT	cSigned & ~zflag	3’h5	Execute if Less Than
GTE	~cSigned	zflag	3’h6
LTE	cSigned	zflag	3’h7

The “if” opcode will set cond[0] based on the condition above and the cond[1] bit to HIGH. It only affects the single instruction following the “if” opcode. The “iftt” opcode will set both cond[0] and cond[1] to the same value based on the condition above. It means “if true, execute the next two opcodes”. And the “ifte” opcode will set cond[0] based on the condition above and cond[1] to the OPPOSITE value, meaning it will execute either the following instruction OR the one after that (then-else).

```
Example:
  ldi      0x41        // Load A immediate with ASCII 'A'
  cpi      0x42        // Compare A immediate with ASCII 'B'
  ifte     eq          // Test if the compare was "Equal"
  jal      L_equal     // Jump if equal
  jal      L_different // Jump if different
```

The above code will load the “jal L_equal” opcode but will not execute it since the compare was Not Equal. Then it will execute the “jal L_different” opcode. Note that if the compare were “ifte ne”, it would call the L_equal function and then upon return would not execute the “L_different” opcode. This is because the cond[1] code is saved with the Return Address (RA) during the call and restored upon return. This means the FALSE cond[1] code would prevent the 2nd opcode from executing. As an opcode gets executed, the cond[1] value is shifted into the cond[0] location, and the cond[1] is loaded with 1’b1.

Direct Operations To do any useful work, the LISA core must be able to load and operate on data. This is done through the accumulator using the various addressing modes. The diagram below details the Direct addressing mode where data is stored directly in the opcode / instruction word:

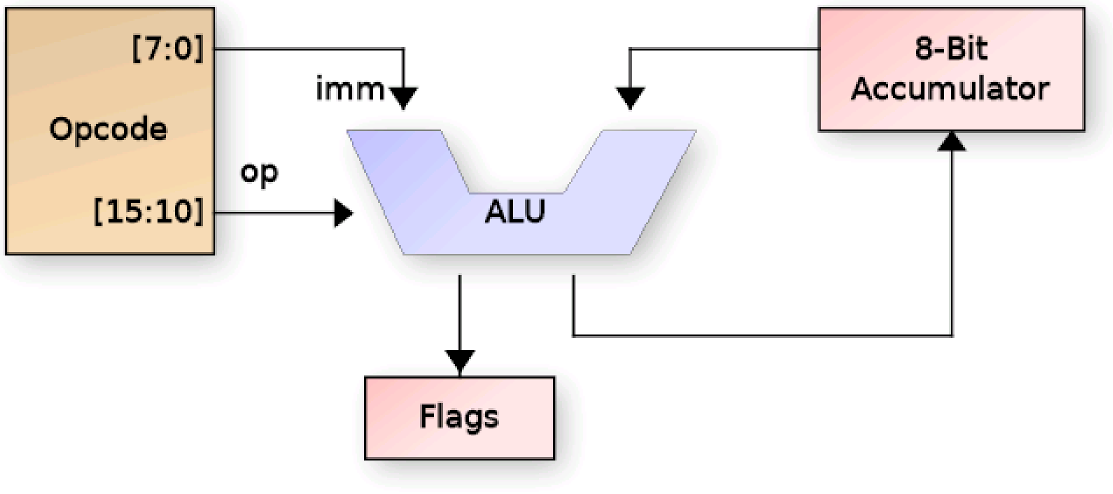


Figure 974.5: Accumulator Direct Operations Diagram

The instructions that use direct addressing are:

Opcode	Operation	Encoding	Description
adc	A <= A + imm + C	1001_00xx_iiii_iiii	ADD immediate with Carry
ads	SP <= SP + imm	1001_01ii_iiii_iiii	ADD SP + signed immediate
adx	IX <= IX + imm	1001_10ii_iiii_iiii	ADD IX + signed immediate
andi	A <= A & imm	1000_01xx_iiii_iiii	AND immediate with A
cpi	Z,C <= A >= imm	1010_01xx_iiii_iiii	Compare A >= immediate
cpi	Z,C <= A >= imm	1010_01xx_iiii_iiii	Compare A >= immediate

Accumulator Indirect Operations The Accumulator Indirect operations use immediate data in the instruction word to index indirectly into Data memory. That memory address is then used to load, store or both load and store (swap) data with the accumulator.

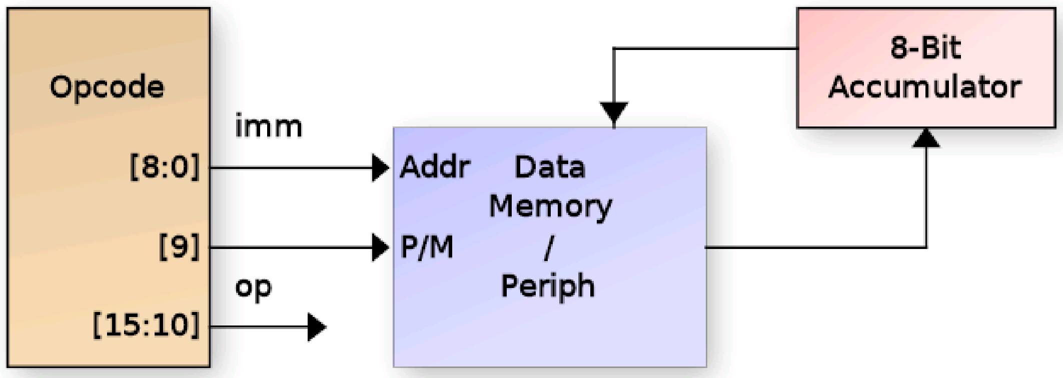


Figure 974.6: Accumulator Indirect Operations Diagram

Opcode	Operation	Encoding	Description
lda	A <= M[imm]	1111_01pi_iiii_iiii	Load A from Memory/Peripheral
sta	M[imm] <= A	1111_11pi_iiii_iiii	Store A to Memory/Peripheral
swapi	A <= M[imm] M[imm] <= A	1101_11pi_iiii_iiii	Swap Memory/Peripheral with A

- p = Select Peripheral (1'b1) or RAM (1'b0)
- iiii = Immediate data

Indexed Operations Indexed operations use either the IX or SP register plus a fixed offset from the immediate field of the opcode. The selection to use IX vs SP is also from the opcode[9] bit. The immediate field is not sign extended, so only positive direction indexing is supported. This was selected because this mode is typically used to access either local variables (when using SP) or C struct members (when using IX), and in both cases, negative index offsets aren't very useful. The following is a diagram of indexed addressing:

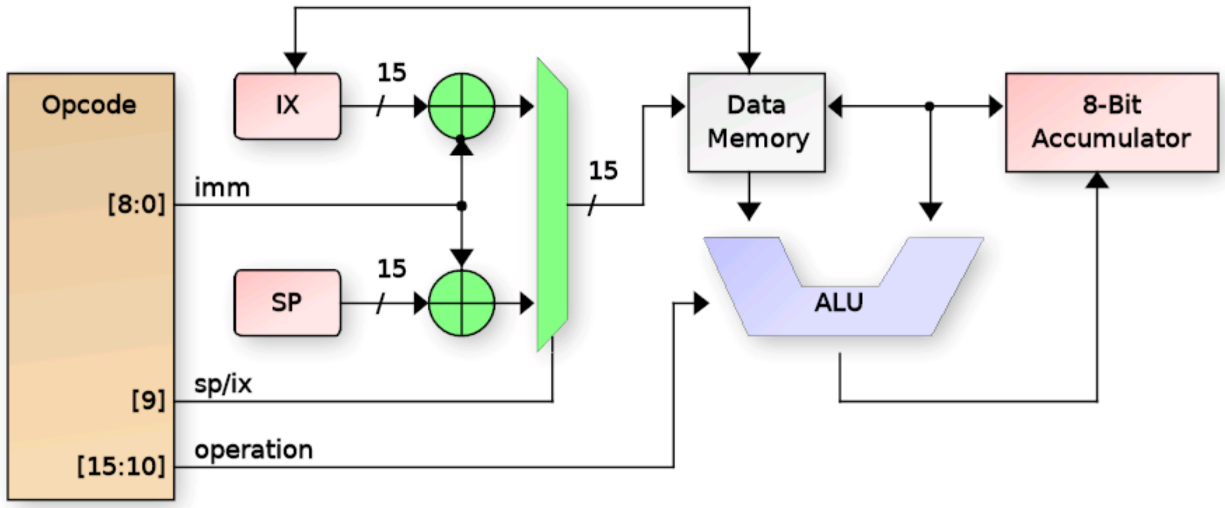


Figure 974.7: Indexed Addressing Diagram

Opcode	Operation	Encoding	Description
add	A <= A+ M[ind]	1100_00si_iiii_iiii	ADD index memory to A
and	A <= A & M[ind]	1101_00si_iiii_iiii	AND A with index memory
cmp	A >= M[ind]?	1110_10si_iiii_iiii	Compare A with index memory
dcx	M[ind] -= 1	1001_11si_iiii_iiii	Decrement the value at index memory
inx	M[ind] += 1	1110_01si_iiii_iiii	Increment the value at index memory
ldax	A <= M[ind]	1111_00si_iiii_iiii	Load A from index memory
ldxx	IX <= M[SP+imm]	1100_110i_iiii_iiii	Load IX from memory at SP+imm
mul	A <= A*M[ind]L	1100_10si_iiii_iiii	Multiply index memory * A, keep LSB
mulu	A <= A*M[ind]H	1000_01si_iiii_iiii	Multiply index memory * A, keep MSB
or	A <= A	M[ind]	1101_10si_iiii_iiii
stax	M[ind] <= A	1111_10si_iiii_iiii	Store A to index memory
stxx	M[SP+imm] <= IX	1100_111i_iiii_iiii	Save IX to memory at SP+imm
sub	A <= A-M[ind]	1100_10si_iiii_iiii	SUBtract index memory from A
swap	A <= M[ind]	1110_11si_iiii_iiii	Swap A with index memory
	M[ind] <= A		
xor	A <= A ^ M[ind]	1110_00si_iiii_iiii	XOR A with index memory

Legend for table above:

- ind = IX or SP + immediate
- s = Select IX (zero) or SP (one)
- iiii = Immediate data

The Zero and Carry flags are updated for most of the above operations. The Carry flag is only updated for math operations where a Carry / Borrow could occur.

Carry	Zero
adc	add and
add	or xor
sub	cmp sub
cmp	dcx inx
dcx	swap ldax
inx	mul mulu

Stack Operations Stack operations use the current value of the SP register to PUSH and POP items to the stack in opcode. As items are PUSHed to the stack, the SP is decremented after each byte, and as they are POPed, the SP is incremented prior to reading from RAM.

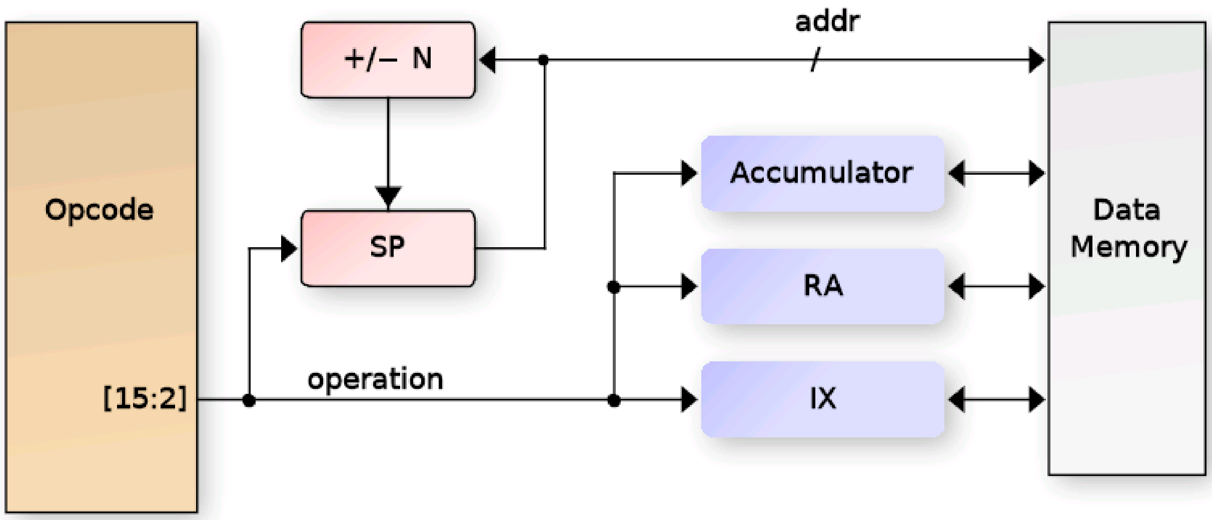


Figure 974.8: Stack Addressing Diagram

Opcode	Operation	Encoding	Description
lra	RA <= M[SP+1] SP += 2	1010_0001_0110_01xx	Load {cond,RA} from stack

Opcode	Operation	Encoding	Description
sra	M[SP] <= RA SP -= 2	1010_0001_0110_00xx	Save {cond,RA} to stack
push_ix	M[SP] <= IX SP -= 2	1010_0001_0110_10xx	Save IX to stack
pop_ix	IX <= M[SP+1] SP += 2	1010_0001_0110_11xx	Load IX from stack
push_a	M[SP] <= A SP -= 1	1010_0000_100x_xxxx	Save A to stack
pop_a	A <= M[SP+1] SP += 1	1010_0000_110x_xxxx	Load A from stack

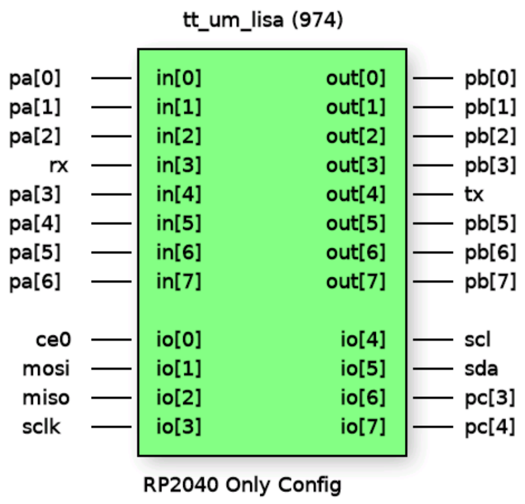
How to test

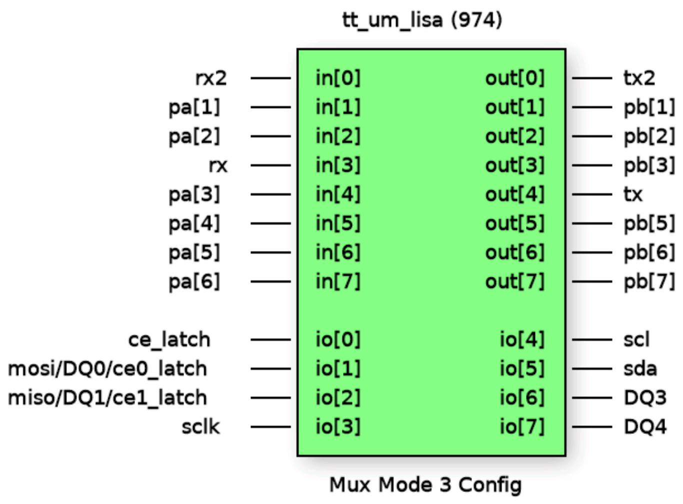
You will need to download and compile the C-based assembler, linker and C compiler I wrote (will make available) Also need to download the Python based debugger.

- Assembler is fully functional
  - Includes limited libraries for crt0, signed int compare, math, etc.
  - Libraries are still a work in progress
- Linker is fully functional
- C compiler is somewhat functional (no float support at the moment) but has *many* bugs in the generated code and is still a work in progress.
- Python debugger can erase/program the FLASH, program SPI SRAM, start/stop the LISA core, read SRAM and registers.

Legend for Pinout

- pa: LISA GPIO PortA Input
- pb: LISA GPIO PortB Output
- b_div: Debug UART baud divisor sampled at reset
- b_set: Debug UART baud divisor enable (HIGH) sampled at reset
- baud_clk: 16x Baud Rate clock used for Debug UART baud rate generator
- ce_latch: Latch enable for Special Mux Mode 3 as describe above
- ce0_latch: CE0 output during Special Mux Mode 3
- ce1_latch: CE1 output during Special Mux Mode 3
- DQ1/2/3/4: QUAD SPI bidirection data I/O
- pc_io: LISA GPIO Port C I/O (direction controllable by LISA)





Pinout

#	Input	Output	Bidirectional
0	pa[0]/b_div[0]/rx2	pb[0]/tx2	ce0/ce_latch
1	pa[1]/b_div[1]/rx2	pb[1]/tx2	mosi/dq1/ce0_latch
2	pa[2]/b_div[2]/rx2	pb[2]/tx2	miso/dq2/ce1_latch
3	pa[3]/b_div[3]/rx	pb[3]	sclk
4	pa[4]/b_div[4]	pb[4]/tx	rx /pc_io[0]/scl/ce1
5	pa[5]/b_div[5]	pb[5]	tx /pc_io[1]/sda
6	pa[6]/b_div[6]	pb[6]	scl /pc_io[2]/dq2/rx
7	pa[7]/b_set(autobaud_disable)	pb[7]/baud_clk	sda/pc_io[3]/dq3

Simon Says memory game [897]

- Author: Uri Shaked
- Description: Repeat the sequence of colors and sounds to win the game
- GitHub repository
- HDL project
- Mux address: 897
- Extra docs
- Clock: 50000 Hz

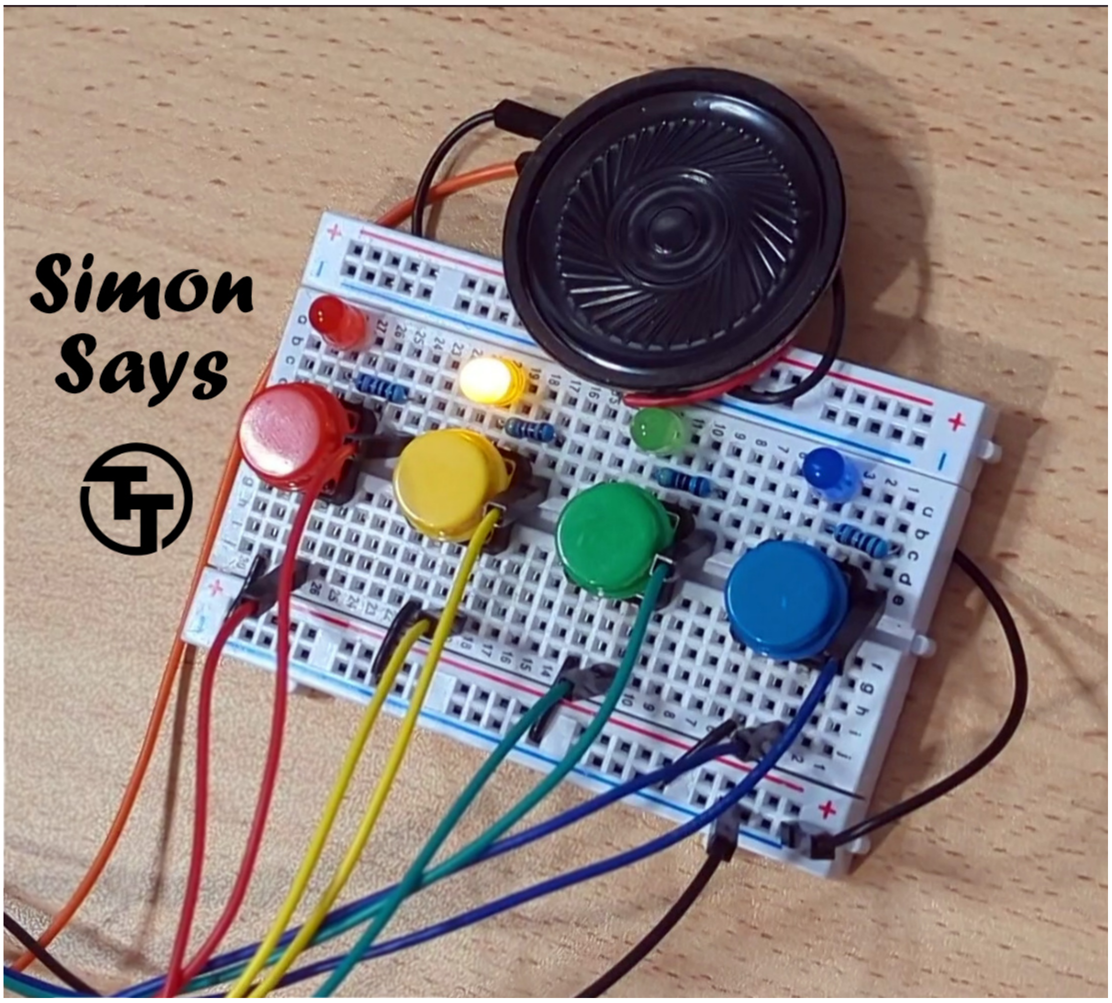


Figure 44: Simon Says Game

How it works

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

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

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

Check out the online simulation at https://wokwi.com/projects/397436605640509441 (including wiring diagram).

How to test

You need four buttons, four LEDs, resistors, and optionally a speaker/buzzer and a two digit 7-segment display for the score.

Ideally, you want to use 4 different colors for the buttons/LEDs (red, green, blue, yellow).

1. Connect the buttons to pins `btn1`, `btn2`, `btn3`, and `btn4`, and also connect each button to a pull down resistor.
2. Connect the LEDs to pins `led1`, `led2`, `led3`, and `led4`, matching the colors of the buttons (so `led1` and `btn1` have the same color, etc.)
3. Connect the speaker to the `speaker` pin.
4. Connect the seven segment display as follows: `seg_a` through `sev_g` to individual segments, `dig1` to the common pin of the tens digit, `dig2` to the common pin of the ones digit. Set `seginv` according to the type of 7 segment display you have: high for common anode, low for common cathode.
5. Reset the game, and then press any button to start it. Enjoy!

Note: the game requires 50KHz clock input.

External Hardware

Four push buttons (with pull-down resistors), four LEDs, and optionally a speaker/buzzer and two digit 7-segment display

#	Input	Output	Bidirectional

Pinout

#	Input	Output	Bidirectional
0	btn1	led1	seg_a
1	btn2	led2	seg_b
2	btn3	led3	seg_c
3	btn4	led4	seg_d
4	seginv	speaker	seg_e
5		dig1	seg_f
6		dig2	seg_g
7			

Reversible logic based Ring-Oscillator Physically Unclonable Function (RO-PUF) [899]

- Author: Syed Farah Naz, Shivam Bhardwaj, Ambika Prasad Shah
- Description: Reversible logic based Ring-Oscillator Physically Unclonable Function (RO-PUF)
- GitHub repository
- HDL project
- Mux address: 899
- Extra docs
- Clock: 0 Hz

How it works

We have introduced a fault-tolerant system featuring a ring-oscillator (RO) based Physcially Unclonable Function (PUF), utilizing a reversible logic (RL) design. The proposed system comprises of Fault-Tolerant RL-based inverter design and the Reversible RO-PUF module. This RO-PUF design consists of several cascaded chains of DFGs (functioning as inverter/buffer as and when required). We have used 64 ROs, divided into groups of 32 ROs in our proposed RO-PUF architecture. Each of these 32 ROs is connected to two 32×1 MUXs (Mux-1 and Mux-2), which select an RO according to the bit combination given in the select lines. The output of the selected RO is fed to the input of the counter, which counts the number of pulses generated by the RO until a certain condition is reached, as specified in the comparator design. The comparator takes input from the output of the two counters (Counter-1 and Counter-2), which are continuously changing. When any of the two counters reaches its specified maximum value of 32 bits, the current 32-bit count value of the slower counter is latched onto the output of the PUF. The 5-bit challenges are randomly generated with the help of a linear feedback shift register (LFSR). The select lines of both the 32×1 MUXs are connected with challenge bits so that unique ROs are selected for comparison. Since the frequency of each RO is different due to process variations, when one of the two chosen ROs attains its specified maximum count value, the frequency count of the other RO will be different, and we will get a unique response for each challenge.

How to test

We have tested the design on Vivado and OpenRoad Flow Script.

External hardware

Default

Pinout

#	Input	Output	Bidirectional
0	rst_n	Respose_1	
1	ena	Respose_2	
2	Challenge_1	Respose_3	
3	Challenge_2	Respose_4	
4	Challenge_3	Respose_5	
5	Challenge_4	Respose_6	
6		Respose_7	
7		Respose_8	

CRC-8 CCITT [901]

- Author: Aiden Fox Ivey
- Description: A simple parallel implementation of CRC-8 following CCITT specs. This implies 0x00 is the start value.
- GitHub repository
- HDL project
- Mux address: 901
- Extra docs
- Clock: 0 Hz

How it works

`ui` should have the two bytes you want added to the CRC8. If you want to restart the internal CRC value, then pull `rst_n` low. That will set it back to the default 0x00. `enable` should be high unless you want to ignore the new calculated value from the specific clock cycle. You can add any number of two byte combinations to it and it will calculate the CRC8 CCITT value for the given combination.

https://crccalc.com can help you calculate the CRC8 if you want.

The specific polynomial in this case is 1+x^{1+x}2+xˆ8.

How to test

Run `make` in the `/test` directory.

External hardware

None required! The design is combinational, requiring only a small buffer to store the current CRC value. As a result, it's quite simple.

Pinout

#	Input	Output	Bidirectional
0	CRC input pin 0.	CRC output pin 0.	Represents whether or not to ingest the values on ui to the CRC.

#	Input	Output	Bidirectional
1	CRC input pin 1.	CRC output pin 1.	
2	CRC input pin 2.	CRC output pin 2.	
3	CRC input pin 3.	CRC output pin 3.	
4	CRC input pin 4.	CRC output pin 4.	
5	CRC input pin 5.	CRC output pin 5.	
6	CRC input pin 6.	CRC output pin 6.	
7	CRC input pin 7.	CRC output pin 7.	

VGA clock [903]

- Author: Matt Venn
- Description: Shows the time on a VGA screen
- GitHub repository
- HDL project
- Mux address: 903
- Extra docs
- Clock: 31500000 Hz

How it works

Races the beam! Font is pre generated and loaded into registers. 6 bit colour keeps register count low.

Every minute the colours cycle.

How to test

Hook up a VGA monitor to the outputs and provide a clock at 31.5 MHz.

Adjust time with the inputs[2:0], and choose the type of VGA PMOD with the input[3].

External hardware

VGA PMOD - you can use one of these VGA PMODs:

- https://github.com/mole99/tiny-vga
- https://github.com/TinyTapeout/tt-vga-clock-pmod

Set input[3] low to use tiny-vga and high to use vga-clock

#	Input	Output	Bidirectional

Pinout

#	Input	Output	Bidirectional
0	adjust hours	hsync / R1	
1	adjust minutes	vsync / G1	
2	adjust seconds	B0 / B1	
3	PMOD type select	B1 / VS	
4		G0 / R0	
5		G1 / G0	
6		R0 / B0	
7		R1 / HS	

Padlock [905]

- Author: J. Rosenthal & htfab
- Description: Set a code for your precious safe
- GitHub repository
- Wokwi project
- Mux address: 905
- Extra docs
- Clock: 0 Hz

How it works

Set a code for your precious safe! **Controls**

- Pin 1 is used to reset the safe.
- Pin 7 is used to set your code (ON = set, OFF = locked)
- Pins 2 to 4 are used to set the code.
- The clock button is used to enter your code.

How to test

Set the clock to manual mode. Set your desired code using pins 2 to 4. Once you've done so, toggle pin 7 to ON, press the clock button then toggle pin 7 back OFF–the safe is now set! Turn ON pin 1, and press the clock button. The seven segment display should show "L" (for locked). Next turn OFF pin 1 to begin entering codes.

External hardware

None

Pinout

#	Input	Output	Bidirectional
0		segment a	
1	reset	segment b	
2	code bit 0	segment c	
3	code bit 1	segment d	
4	code bit 2	segment e	

#	Input	Output	Bidirectional
5		segment f	
6		segment g	
7	set code		

7-Seg 'Tiny Tapeout' Display [907]

- Author: J. Rosenthal & htfab
- Description: This circuit will output a string of characters ('tiny tapeout') to the 7-segment display.
- GitHub repository
- Wokwi project
- Mux address: 907
- Extra docs
- Clock: 0 Hz

How it works

The logic to light the characters appears in the bottom half of the simulation window. The top half of the simulation window implements a modulo-12 counter. In other words, the counter increments up to 11 then resets. This counter is used to determine which character we should output to the 7-segment display. The truth table for the design can be found in the Design Spreadsheet.

How to test

Turn all pins OFF, keep the clock running and watch the rolling text on the 7-segment display. Or turn pin 3 ON and manually select a letter using pins 4 to 7.

External hardware

None

Pinout

#	Input	Output	Bidirectional
0		segment a	
1	reset (sync)	segment b	
2		segment c	
3	clock override	segment d	
4	clock bit 3	segment e	
5	clock bit 2	segment f	
6	clock bit 1	segment g	

#	Input	Output	Bidirectional
7	clock bit 0		

clock [909]

- Author: ender
- Description: a clock usb digital tube
- GitHub repository
- HDL project
- Mux address: 909
- Extra docs
- Clock: 32768 Hz

this is a clock

How it works

pcb will be public at :https://github.com/ender110/tiny-tape-out-clock-pcb 3d shell will be public at:https://github.com/ender110/tiny-tape-out-clock-shell

How to test

follow How it works

External hardware

pcb shell Digital tube

Pinout

#	Input	Output	Bidirectional
0	addr[0]	data[0]	
1	addr[1]	data[1]	
2	addr[2]	data[2]	
3	addr[3]	data[3]	
4	addr[4]	data[4]	
5	addr[5]	data[5]	
6	addr[6]	data[6]	
7	addr[7]	data[7]	

Tiniest GPU [910]

- Author: Matt Pongsagon
- Description: A GPU that can render only a triangle
- GitHub repository
- HDL project
- Mux address: 910
- Extra docs
- Clock: 50000000 Hz

What is it

- This is a tiniest ASIC GPU. It can render a quad using two triangles with texture mapped.
- The chip comes with two texture ROM images. (My schools' logo)
- The transformation, lighting and rasterization are done in the GPU.
- It support solid shading with one directional light source and affine texture mapping.
- All 3D data (coordinates, transformation, render mode) are sent from the PC each frame via a COM port.
- The output is sent to the VGA monitor using TinyVGA. The output resolution is 640x480 pixels, 6-bit RGB.
- The clock fequency is 50 Mhz.

Folders

https://github.com/pongsagon/tt07-tiniest-gpu

1. src: ASIC Verilog version
2. Basys3: Verilog version targeted Basys3 FPGA board
3. Verilator_sim: Verilog simulation version using Verilator and SDL
4. test_software: PC app used to sending data to the GPU

How to use

Plugin a TinyVGA PMOD, connect at the port uio.
Send UART command to control the GPU via serial console, ui_in[3] - RX

Please go to https://github.com/pongsagon/tt07-tiniest-gpu/tree/main/test_software to get the testing app.

ASIC GPU testing app written in C run on Windows.
To be able to run on different OS, you may need to use different UART library.
The app will send these data, 60 bytes, each frame @11520 baud rate to the GPU

- 4 vertices world coordinate that form a quad
- 1 normalize normal
- 1 normalize light direction
- 3x4 ModelViewProjection matrix. (third row is not used)
- 1 byte render mode
    - solid shading, texture, alpha masked

The data are in the format of fixed point Q8.8 except for the 1-byte render mode.
The code has been successfully tested with the Basys3 board, sending data at 60fps.

Run/Edit the code

1. The code rely on SFML library for the input and windows. https://www.sfml-dev.org/tutorials/2.6/start-vc.php.
   Please install SFML first.
2. Change the COM port number in C code to match with the ASIC/FPGA port (main_serial.cpp, line number 330)
3. short cut keys
    - arrow key: yaw pitch
    - as: zoom
    - df: change model size
    - er: X translation
    - 012: render mode
    - 34: change texture
    - 6789: change triangle 1 color
    - uiop: change triangle 2 color
4. You can also changes the vertices coordinate, quad normal and light direction using code. I have not write short cut keys for setting them.

How it work

- Fixed point
    - All of the calculation are done in fixed point.

- The format of the fixed point is depend on the type of variables to save register space as much as possible. Thus, they are a lot of bit operation in the code to transiton between fixed point formats.

- Modules
  - ia.v (input assembly)
    - manage reading data (60 bytes each frame) from UART and save to the registers
  - vs.v (vertex stage)
    - transform vertices from world space to screen space
    - compute lighting intensity color for each triangle
    - compute triangles' edge parameters and barycentric coordinates
  - raster.v
    - rasterization two triangles, interpolate color, texture mapping
- No framebuffer or linebuffer
  - Each pixel color has to be computed in 2 clock cycles.
  - the rasterization is running in parallel with the vertex stage.
  - Using incremental edge function to do pixel-triangle inside test.
- Computation steps
  1. read data from the PC via UART (in project.v, ia.v)
  2. for each frame (in vs.v)
     - transform vertices to screen space and compute lighting
     - (done during VBlank) compute triangles' edge parameters and barycentric coordinates
     - all of these calculation are done in 82 states and use around 2,000 clock cycles.
  3. for each scanline (in vs.v, raster.v)
     - done in 1 clock cycle
     - increment edge functions parameters of each scanline
     - increment barycentric parameters of each scanline
  4. for each pixel (in raster.v)
     - 1st clock cycle:
       - check pixel inside/outside of which triangles
       - compute interpolated (u,v) of this pixel, get texel color from this (u,v)
     - 2nd clock cycle:
       - color the pixel using texel color or light intensity color
       - actually the texture ROM is monochorme, the color is hardcoded using (u,v) coordinate.

Pinout

#	Input	Output	Bidirectional
0			R1
1			G1
2			B1
3	RX		vsync
4			R0
5			G0
6			B0
7			hsync

SUBNEG CPU [911]

- Author: Pawel Bialic
- Description: SUBNEG CPU requiring external parallel SRAM
- GitHub repository
- HDL project
- Mux address: 911
- Extra docs
- Clock: 1000 Hz

How it works

Implementation of a simple 8-bit SUBNEG CPU. The CPU interfaces to external SRAM memory through address output latch. CPU output can be implemented using a second output latch. The program to be executed has to be written to the SRAM by external means (e.g. a microcontroller) prior to setting CPU enable pin high. 3 inputs are provided for this purpose (CPU enable, External SRAM address latch CLK, External SRAM WEn).

External hardware

3.3V SRAM memory (e.g. AS6C6264). Memory address latch (e.g. 74HC574). CPU output latch (e.g. 74HC574). Device capable of displaying 8-bit output value.

Pinout

#	Input	Output	Bidirectional
0	CPU enable	SRAM address latch CLK	CPU bi-directional bus
1	External SRAM address latch CLK	SRAM OEn	CPU bi-directional bus
2	External SRAM WEn	SRAM WEn	CPU bi-directional bus
3		CPU output latch CLK	CPU bi-directional bus
4		Internal CPU state bit 0	CPU bi-directional bus
5		Internal CPU state bit 1	CPU bi-directional bus
6		Internal CPU state bit 2	CPU bi-directional bus
7		Internal CPU state bit 3	CPU bi-directional bus

Stopwatch Project [961]

- Author: A.J. Stein
- Description: A TinyTapeout project to display a stopwatch counter one digit at a time
- GitHub repository
- HDL project
- Mux address: 961
- Extra docs
- Clock: 20000000 Hz

How it works

This is a Tiny Tapeout project (designed to start for Tiny Tapeout 7 in May 2024) to use the 20 MHz of the ASIC chip to make a stopwatch that can count seconds in decimal format for ten second increments (from 0 to 9 and loops back to 0).

How to test

This is a simple project and has limited testing infrastructure. To test in simulation and analyze changes to the logic with a waveform analyzer, you can use the published MakerChip project associated with this GitHub repo and check the `clock_speed` to have smaller clock speeds and increment the decimal stopwatch much quicker than the necessary hardware clock speed.

To test this with a Tiny Tapeout 3 Demo Board (v 2.2.5) and the ASIC Simulator (v1.2) using this test harness repo and use `dfu-util` to flash it on this device.

External hardware

This project is a simple first experiment that does not require additional external inputs or outputs.

Pinout

#	Input	Output	Bidirectional
0	N/A	N/A	N/A
1	N/A	N/A	N/A
2	N/A	N/A	N/A

#	Input	Output	Bidirectional
3	N/A	N/A	N/A
4	N/A	N/A	N/A
5	N/A	N/A	N/A
6	N/A	N/A	N/A
7	N/A	N/A	N/A

calculator [963]

- Author: ZHU QUANHAO
- Description: input two number and do all kinds of calculation base on it
- GitHub repository
- HDL project
- Mux address: 963
- Extra docs
- Clock: 100 Hz

How it works

Adding, and, or, xor numbers

How to test

By viewing the input and predict the output to see if it match with the display

External hardware

4Nixie tube

Pinout

#	Input	Output	Bidirectional
0	bit1_XOR	display	display
1	bit2_OR	display	display
2	bit3_AND	display	display
3	bit4_ADD	display	display
4	numbersel_NOT	display	display
5	positionsel_dispB	display	display
6	signsel_dispA	display	power13
7	modesel	display	power24

Mini Light Up Game [965]

- Author: Dyrick Williams
- Description: This is a small game where the objective is to light up the segments to form a '0' by toggling a switch at the correct moment.
- GitHub repository
- Wokwi project
- Mux address: 965
- Extra docs
- Clock: 10 Hz

How it works

The underlying selector is controlled by a circular buffer composed of D Flip-Flops which acts as a circular bit shifter. The clock signal performs the shift every clock rising edge. The selection is done by toggling the state of the input switch and rising edges and falling edges are turned into pulses. The memory component of what the player selects is also implemented with D Flip-Flops. The rest of the output logic for the segment display is combinational logic. The reset signal sets only one of the bits in the circular buffer and clears the memory component that is controlled by the player.

How to test

First perform a reset. As this is designed within Wokwi, the testing can be done by trying to light up the segments and form the '0' at a low clock frequency. Once the '0' is formed, the next clock cycle should then display only the dot segment.

Clock configuration

The generated clock frequency from the RP2040 may be lowered to a reasonable, visually observable frequency (3-20Hz).

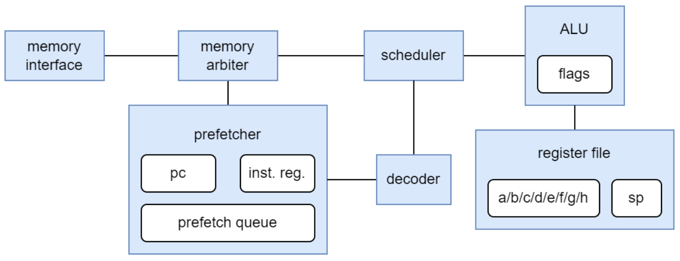
Pinout

#	Input	Output	Bidirectional
0	select	S1	
1		S2	
2		S3	

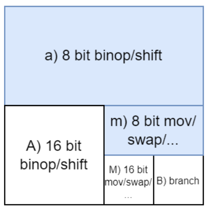
#	Input	Output	Bidirectional
3		S4	
4		S5	
5		S6	
6			
7		SDOT	

Basilisc-2816 v0.1a CPU [967]

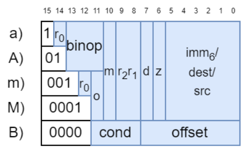
- Author: Toivo Henningsson
- Description: Small 2-bit serial 8/16 bit CPU
- GitHub repository
- HDL project
- Mux address: 967
- Extra docs
- Clock: 50000000 Hz



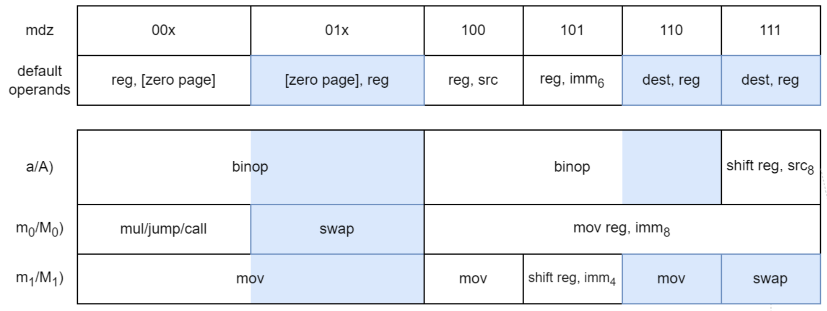
Major parts and interconnections of the processor.



Major division of the instruction encoding space based on the top 4 bits. 8 bit instructions use 3 bits to choose between the 8 registers:a/b/c/d/e/f/g/h, 16 bit instructions use 2 bits to choose between the 4 registers ba/dc/fe/hg.

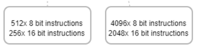


Bit fields of the major instruction forms. The o bit selects between m_0/m_1 or M_0/M_1.

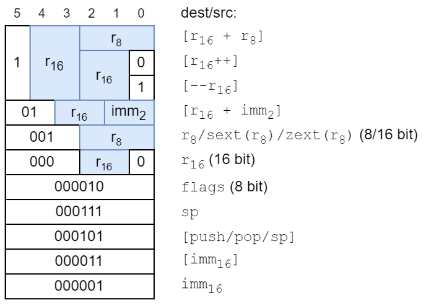


mdz	00x	01x	100	101	110	111
default operands	reg, [zero page]	[zero page], reg	reg, src	reg, imm_6	dest, reg	dest, reg
a/A)	binop		binop			shift reg, src_8
m_0/M_0)	mul/jump/call	swap	mov reg, imm_8			
m_1/M_1)	mov		mov	shift reg, imm_4	mov	swap

Further division of the a/A/m/M) encoding spaces into instruction forms. Shaded instructions write to memory.



512x 8 bit instructions 256x 16 bit instructions
4096x 8 bit instructions 2048x 16 bit instructions



5 4 3 2 1 0	dest/src:
1, r_{16}, r_8	$[r_{16} + r_8]$
1, r_{16}, r_{16}, 0	$[r_{16}{+}{+}]$
1, r_{16}, r_{16}, 1	$[{-}{-}r_{16}]$
01, r_{16}, imm_2	$[r_{16} + imm_2]$
001, r_8	r_8/sext(r_8)/zext(r_8) (8/16 bit)
000, r_{16}, 0	r_{16} (16 bit)
000010	flags (8 bit)
000111	sp
000101	[push/pop/sp]
000011	$[imm_{16}]$
000001	imm_{16}

Encoding of the dest/src field.

Overview

Basilisc-2816 v0.1 is a small 2-bit serial 2/8/16 bit processor that fits into one Tiny Tapeout tile. It has been designed around the constraints of

- small area,
- 4 pin serial memory interface to a RAM emulator implemented in an RP2040 microcontroller (which can be supported by the RP2040 microcontroller on the Tiny Tapeout 7 Demo Board),
- to be suitable to be included in in the next version of the AnemoneGrafx-8 retro console https://github.com/toivoh/tt06-retro-console, which motivates the other constraints.

Features:

- 2-bit serial execution:

  - ALU results etc are calculated at 2 bits/cycle
  - 2-bit-serial register file with two read/write ports
  - Addresses and data are sent to/from memory at 2 bits/cycle
    - The processor starts to operate on each bit of incoming read data as it arrives
  - Saves area compared to processing 8/16 bits per cycle / using a parallel access register file
  - No point in calculating faster than the memory interface allows

- 8x 8-bit general purpose registers that can be paired into 4x 16-bit general purpose registers, plus an 8 bit stack register
- 8 bit and 16 bit versions of almost all instructions
- 64 kB address space
- 16 bits/instruction
- Quite regular and orthogonal instruction encoding, most instructions can use most addressing modes
  - `op reg, src` and `op src, reg` instruction forms
- Instructions:
  - `mov`, `swap`
  - `binop`: `add/adc/sub/sbc/and/or/xor/cmp/test`
    - for register-to-register also: `neg/negc/revsub/revsbc/and_not/or_not/xor_not/not`,
  - `shl/shr/sar/rol/ror` with variable or immediate shift count,
  - `mul`: 8x8 and 8x16 bit multiply instructions, producing 2 result bits per cycle like everything else,
  - `branch cc, offset`: relative branch
    - unconditional/call/12 conditions including signed/unsigned comparisons,
  - `jump/call`: absolut direct/indirect jump/call,
  - additional functionality through combination with addressing modes, e g, `ret = jump [pop]`
- Addressing modes:
  - `[imm7]` / `[imm7*2]`: zero page
  - `[r16 + imm2]`
  - `[r16 + r8]`
  - `[r16]` with postincrement/predecrement
  - `[push]` / `[pop]` / `[top-of-stack]` depending on whether the operand is written/read/modified

    - `[imm16]`

- Sign/zero extension of any 8 bit register as source operand to 16 bit instructions
- `imm16` / `[imm16]` operands supported using extra instruction word
- 2-4 word instruction prefetch queue

Contents:

- Basilisc-2816 v0.1 variants in Tiny Tapeout 7
- Interface / pins
- Programmer's view
- Execution timing
- How it works

Basilisc-2816 v0.1 variants

Basilisc-2816 v0.1 has been taped out in three variants for Tiny Tapeout 7:

	mul instruction	Prefetch queue size	Hardened with	Uses latches	Mux address
v0.1a	yes	2	OpenLane 1	no	967
v0.1b	no	3	OpenLane 2	no	202
v0.1c	yes	4	OpenLane 2	yes	72

successively more experimental. Longer prefetch queue should help contribute to better performance, especially with long memory access latencies.

The target clock frequency is 40 MHz for v0.1a, and 50 MHz for the others, but the max frequency may be higher or lower in practice. It is unknown how the use of latches will affect max clock frequency (and correctness). Reducing the memory access latency may be more important to performance than using a high clock frequency.

The three variants are availble at

- v0.1a: https://github.com/toivoh/tt07-basilisc-2816-cpu
- v0.1b: https://github.com/toivoh/tt07-basilisc-2816-cpu-OL2
- v0.1c: https://github.com/toivoh/tt07-basilisc-2816-cpu-experimental

This is the v0.1a version.

Interface / pins

The TX/RX interface is used to send commands to the RAM/RAM emulator, and to receive read data. Start bits are used to let each side initiate a message when appropriate, with subsequent bits sent on subsequent clock cycles. The `tx_out` and `rx_in` pins must remain low when no messages are sent, to avoid being interpreted as start bits.

The TX channel / `tx_out[1:0]` pins are used for messages from the CPU:

- a message is initiated with one cycle of `tx_out[1:0] = 1` (low bit set, high bit clear/don't care),
- during the next cycle, `tx_out[1:0]` contains the 2 bit *TX header*, specifying the message type,
- during the following 8 cycles, a 16 bit payload is sent through `tx_out[1:0]`, from lowest the bits to the highest.

The RX channel / `rx_in[1:0]` pins are used for messages to the console:

- a message is initiated with one cycle when `rx_in[1:0] != 0`, specifying the *RX header*, i e, the message type,
- the value of `rx_in` during the next cycle is ignored,
- during the following 8 cycles, a 16 bit payload is sent through `rx_in[1:0]`, from lowest bits to highest.

All TX messages use the same start bit and the same length, to make them easier to receive for the RAM emulator. The RX message has been prolonged to the same length as the TX messages so that the CPU can respond to an incoming RX message with an outgoing TX message without any delay.

TX message types:

- 0: 16 bit read request: Read 16 bit data. Payload is the byte address (can be uneven).
- 2: 8 bit write request: Write bottom 8 bits of payload to last read address.
- 3: 16 bit write request: Write payload to last read address.

There is only one RX message type: 1: 16 bit read response. Payload is 16 bit data. Each read request must get exactly one read response, in the same order as the requests. Write requests must not get any response.

The two remaining output pins give additional information about the current TX message:

- `tx_fetch` is high when the current TX message is a read request for an instruction word,

- `tx_jump` is high when `tx_fetch` is high and the current fetch is for a jump destination.

The RAM emulator does not need to use these pin values to operate correctly, but they give additional information about what the CPU is doing. They are used by the gate level test.

The 2-bit serial RAM interface is motivated by the AnemoneSynth synth in https://github.com/toivoh/tt06-retro-console, which needs to be able to read and write 2 bits/cycle to and from memory for fast enough context switching between voices. The 2-bit serial interface in turn shapes the design of the rest of the processor.

Programmer's view

Registers The CPU has the following registers:

- 8 general purpose 8 bit registers `a - h`,
  - also available as four general purpose 16 bit register pairs `ba / dc / fe / hg`.
- 16 bit program counter `pc`, keeping the adress of the current instruction.
- Stack pointer register pair `sp`:
  - bottom half `p` is an 8 bit register,
  - top half `s` always reads as 1, making all stack operations work on the address range `0x100 - 0x1ff` (actually, `0xfe - 0x1ff`, see below).
- Flags register `flags`:
  - zero flag `z`, (bit 0)
  - sign flag `s`, (bit 1)
  - signed carry flag `v`, (bit 2)
  - carry flag `c` (bit 3).

At reset, the `pc` register starts at `0xfffc`, which is just enough to fit a `jump imm16` instruction before the `pc` wraps around. All other registers are uninitialized at reset.

Instructions Most instructions operate on one general purpose register and one additional operand, which can be, e g, a register, memory, or an immediate value. The `branch / jump / call / ret` instructions are always 16 bit; all others have corresponding 8/16 bit forms. Generally, 8 bit instructions operate on 8 bit registers while 16 bit instructions operate on register pairs.

In the instruction descriptions below,

- `reg` is a general purpose register (pair) operand to an 8 (16) bit instruction.
- `dest/src` can be a general purpose register (/pair), a memory location, an immediate value (for `src`), among other things (see addressing modes below).
- `imm4/imm6/imm8` is a 4/6/8 bit immediate value, usually sign extended.
- `[zp]` is a *zero page memory location* (see addressing modes below).

The following types of instructions are supported:

`mov dest, src` Copy value from `src` to `dest`. Supported forms:

```
mov reg, src
mov dest, reg
mov reg, imm8  // imm8 is sign extend if reg is 16 bit
mov reg, [zp]
mov [zp], reg
```

`swap dest, reg` Swap the value of `dest` and `reg`. Supported forms:

```
swap dest, reg
swap [zp], reg
```

`binop dest, src` Perform a binary operation on `dest` and `src`, write the result to `dest`, and update `flags` to reflect the result.

For the binary operations

```
add dest, src  // dest = dest + src
sub dest, src  // dest = dest - src
adc dest, src  // dest = dest + src + c
sbc dest, src  // dest = dest - src + c - 1
and dest, src  // dest = dest & src
or  dest, src  // dest = dest | src
xor dest, src  // dest = dest ^ src
```

the following forms are supported:

```
binop reg, src
binop dest, reg
binop reg, imm6  // imm6 is sign extend
binop reg, [zp]
binop [zp], reg
```

There are additional binary operations

```
revsub dest, src  // dest = src - dest
revsbc dest, src  // dest = src - dest + c - 1
cmp    dest, src  // Update flags according to dest - src,
                     don't update dest
test   dest, src  // Update flags according to dest & src
                     don't update dest
```

which support fewer forms:

```
revsub reg, imm6  // replaces sub dest, imm6
revsub reg1, reg2

revsbc reg, imm6  // replaces sbc dest, imm6
revsbc reg1, reg2

cmp reg, src
cmp reg, imm6
cmp reg, [zp]

test reg, src
test reg, [zp]
```

To get a non-reverse `sub reg, imm6`, use `add reg, -imm6` instead.

The `binop reg1, reg2` form is also supported for the additional operations

```
neg     dest, src  // dest = -src
negc    dest, src  // dest = -src + c - 1
and_not dest, src  // dest = dest & ~src
or_not  dest, src  // dest = dest | ~src
xor_not dest, src  // dest = dest ^ ~src
not     dest, src, // dest = ~src
```

The binary operations `add/adc/sub/sbc/revsub/revsbc/cmp/neg/negc` update the `c, v` flags. The `v` flag is set calculated so that signed and unsigned comparisons using `cmp` work the same, except that signed comparisons rely on the `v` flag instead of the `c` flag. All binary operations update the `s, z` flags.

`shift reg, src8` Perform a shift operation on `reg` using shift count from `src`. The shift operation can be

```
shl reg, src8  // reg = reg << src
shr reg, src8  // reg = reg >> src, unsigned shift
sar reg, src8  // reg = reg >> src, signed shift
rol reg, src8  // rotate reg left
ror reg, src8  // rotate reg right
```

Supported forms:

```
shift reg, src8
shift reg, imm4
```

The `src8` argument is always taken to be 8 bit even for 16 bit shifts. 16 bit shifts always use the bottom 4 bits of the shift count. `shr` and `sar` use the bottom 4 bits also for 8 bit shifts, while the others use the bottom 3 bits. (Timing wise, right shifts use the bottom 4 bits and left shifts use the bottom 3 bits for 8 bit shifts).

`mul reg, src` Unsigned multiply of 8/16 bit `reg` and 8 bit `src`, producing a 16/24 bit result. Store the bottom part of the result in `reg`, and store the top 8 bits in `h`, unless `reg` is `h` or `hg`. (Instruction takes 4 cycles less to execute if top 8 bits are not stored). Supported forms:

```
mul reg, src8
mul reg, imm6  // imm6 is unsigned
```

`reg` can not be `a`, `b`, or `ba`.

branch cc, imm8 Relative conditional branch: if the specified condition is true, jump imm8 instruction words ahead of the current instruction. imm8 is signed. `branch cc, 0` jumps to itself. The encoding for `branch always, 0` is 0, so if the processor encounters a zero instruction word, it enters an infinite loop. (This might be explicitly designated as an illegal instruction in future versions.)

Supported conditions:

```
always
call          // like always, but push address
                 of next instruction before branching

z / e         // zero / equal
nz / ne       // not zero / not equal
s             // signed
ns            // not signed

// unsigned comparisons:
c / ae / nb   // carry / above equal / not below
nc / nae / b  // not carry / not above equal / below
a / nbe       // above / not below
na / be       // not above / below

// signed comparisons:
v / ge / nl   // signed carry / greater equal / not less
nv / nge / l  // not signed carry / not greater equal / less
g / nle       // greater / not less
ng / le       // not greater / less
```

jump src16, call src16, ret `jump src16` performs an absolute unconditional jump to `pc = src16`.

Supported forms:

```
jump src16
jump [zp]

call src16
```

`call` is like `jump`, but pushes address of the next instruction before jumping. It can be used for calling a subroutine.

ret is a pseudoinstruction for jump [pop], which pops a pc value from the stack and jumps to it. It can be used for returning from a subroutine.

Additional pseudoinstructions Some more useful pseudoinstructions that can be realized using the existing instructions:

```
push reg  // mov [push], reg
pop reg   // mov reg, [pop]

rlc1 reg  // adc reg, reg     // rotate left one step through carry flag
```

There are probably more.

Addressing modes For instructions that use a dest operand, dest can be one of

```
reg                 // 8/16 bit general purpose register,
                       depending on size of the operand.
sp/p                // Stack pointer.
                       Read LSB p for 8 bit operands.
                       Write only LSB p for any operand.
flags               // Only for 8 bit operands. Not for shift/mul.
[r16 + zext(r8)]    // r16 and r8 can not overlap.
[r16++]             // Postincrement: increase r16 after calculating
                       the address. Increase by 2 for 16 bit operands.
[--r16]             // Predecrement: decrease r16 before calculating
                       the address. Decrease by 2 for 16 bit operands.
[r16 + zext(imm2)]// imm2 is multiplied by 2 for 16 bit operands.
[imm16]             // imm16 value follows in next instruction word.
[push] / [--sp]     // Push the result onto the stack.
                       Only for operations that don't depend on
                       the value of dest.
                       Decreases sp by 2 for 16 bit operands.
[sp]                // Top of stack. Only for operations that depend
                       on the value of dest.
```

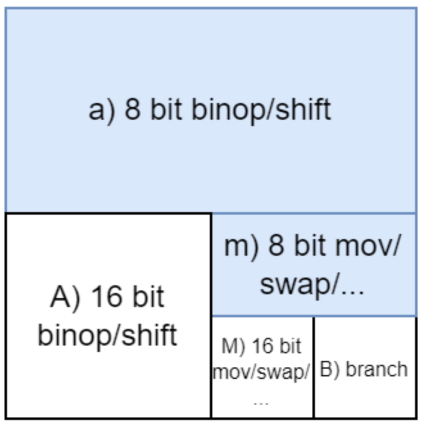
For instructions that use a src operand, src can be anything that dest can be except [pop], [sp], and can also be

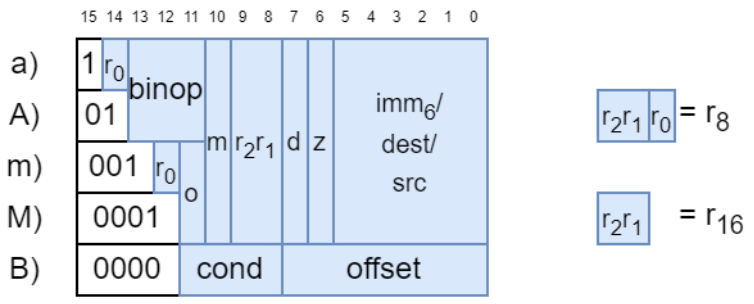
```
sext(r8)         // Sign extend r8, only for 16 bit operations.
zext(r8)         // Zero extend r8, only for 16 bit operations,
                    not for cmp or test.
imm16            // imm16 value follows in next instruction word.
[pop] / [sp++]   // Pop the source from the stack.
                    Increases sp by 2 for 16 bit operands.
```

The [push] and [pop] operands do not play well with each other when sp wraps around. It is recommended to initialize sp to 0x1ff or 0x1fe when the stack is empty (by writing 0xff or 0xfe to the p register). The [push] operand will write a byte to [sp - 1] or two bytes to [sp - 2], [sp - 1], which can reach as low as 0xfe. 8-bit wraparound will only affect the updated value of sp after the [push] operation.

Some instructions can use a [zp] operand, indicating a 7 bit memory adress [imm7], which allows reaching the first 128 bytes in the address space. For 16 bit operands, [2*imm7] is used instead, which can reach the first 256 bytes in the address space, as aligned 16 bit words.

Instruction encoding Each instruction is encoded into one of five major forms a / A / b / B / M):





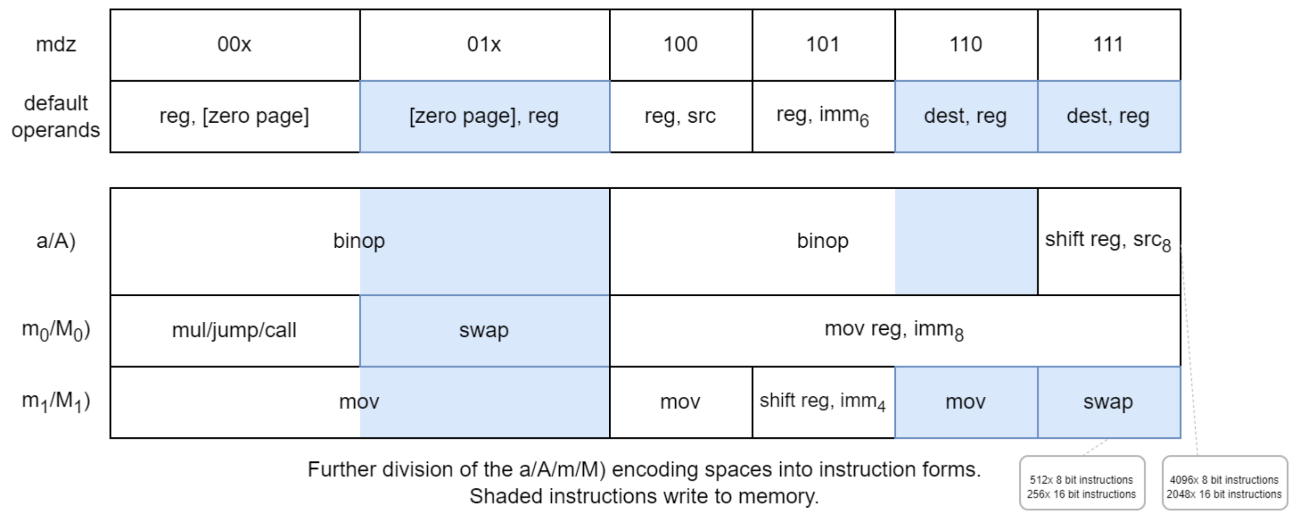
Bit fields of the major instruction forms.
The o bit selects between m_0/m_1 or M_0/M_1.

Major division of the instruction encoding space based on the top 4 bits.
8 bit instructions use 3 bits to choose between the 8 registers a/b/c/d/e/f/g/h,
16 bit instructions use 2 bits to choose between the 4 registers ba/dc/fe/hg.

where

- a/A) forms are mostly used for 8/16 bit binops,
- m/M) forms are used for 8/16 bit moves and other things, and
- B) form is used for branches.

The `mdz` bits are used togther with the major form to choose instruction form:



Further division of the a/A/m/M) encoding spaces into instruction forms.
Shaded instructions write to memory.

where

- The form of the instruction's operands is decided by the `mdz` value, except for the cases `mov reg, imm8`, `shift reg, src8`, and `mul/jump/call`.
- The `o` bit chooses between the `m0/M0` (`o = 0`) and `m1/M1` (`o = 1`) columns.
- `A)` and `M)` form instructions are 16 bit while `a)` and `m)` form instructions are 8 bit, except for `jump/call` which are all 16 bit.

Most instructions encode one general purpose register in `r2r1`/`r2r1r0`, and a source/destination in `imm6`. The interpretation of the `imm6`/`dest`/`src` bits depend on the form of the instruction, including whether it is a source or a destination. Zero page instructions use the `z` bit to extend the `imm6` field to a 7 bit immediate address, and `mov reg. imm8` instructions use the `dz` bits to extend it to an 8 bit immediate.

The registers `a - h` are represented with the numbers 0 - 7; `ba - hg` with the numbers 0 - 3 (stored in a 2 bit field such as `r2r1`). For `A/M)` form instructions, `reg` encoded in `r2r1`. For `a/m)` form instructions, `reg` encoded in `r2r1r0`, but the `r0` bit is stored in different places in `a)` and `m)` form instructions.

The `binop` field in the `a/A)` forms selects which binary operation to use:

binop	operation	alternate
0	add	neg
1	sub	revsub
2	adc	negc
3	sbc	revsbc
4	and	and_not
5	or	or_not
6	xor	xor_not

7	`cmp/test`

The alternate form is used in for `binop` instructions with `mdz = 110` and two register operands (use `mdz = 100` for the regular form). In the same way, `not` is the alternate form of `mov`, and is used under the same circumstances. `sub reg, imm6` is replaced by `revsub reg, imm6` (use `add reg, -imm6` for `sub reg, imm6`). `cmp` is replaced by `test` when `d=1`, but the form is kept as `cmp reg, src`.

The `binop` field is also used to specify the shift operation in `shift reg, src8`:

binop	shift operation
0	`ror`
2	`sar`
4	`shr`
6	`shl`
7	`rol`

For `shift reg, imm4`, the top two bits of `binop` are stored in the top two bits of `imm6` (the bottom bit is implicitly zero), while the bottom 4 bits store the shift count. `rol reg, imm4` should be encoded as

```
ror reg, 16 - imm4       // for 16 bit reg
ror reg, (8 - imm4) & 7  // for  8 bit reg
```

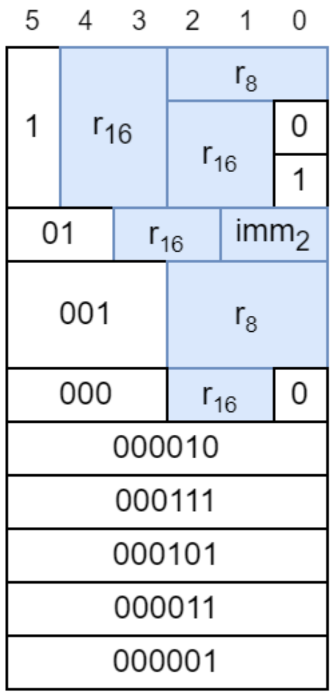
Multiplication instructions are encoded in the `mul/jump/call` case with `r2r1 != 0` (which excludes the use of `a/b/ba` for the `reg` operand). When `r2r1 = 0`, absolute jumps and calls are encoded instead:

```
111111
5432109876543210
0010000000iiiiii    jump src16
0010000001iiiiii    call src16
000100000ziiiiii    jump [zp]
```

For branches, `offset` is the `imm8` offset, which is sign extended and multiplied by 2 and added to the address of the branch instruction to get the target `pc` value. The `cond` field selects the branch condition according to

```
cond   branch condition

  0    always
  1    call          // like always, but push address
                        of next instruction before branching

  4    z / e         // zero / equal
  5    nz / ne       // not zero / not equal
  6    s             // signed
  7    ns            // not signed

  8    c / ae / nb   // carry / above equal / not below
  9    nc / nae / b  // not carry / not above equal / below
 10    a / nbe       // above / not below
 11    na / be       // not above / below

 12    v / ge / nl   // signed carry / greater equal / not less
 13    nv / nge / l  // not signed carry / not greater equal / less
 14    g / nle       // greater / not less
 15    ng / le       // not greater / less
```

When `dest/src` is used in an instruction, it is encoded into the `imm6/dest/src` field according to



dest/src:	
`[`r_{16}` + zext(`r_8`)]`	For non-overlapping r_{16} and r_8
`[`r_{16}`++]`	Both r_{16} fields have the same value
`[--`r_{16}`]`	
`[`r_{16}` + zext(`imm_2`)]`	
`sext(`r_8`)/zext(`r_8`)`	For 16 bit operands, src only
r_8	For 8 bit operands
r_{16}	For 16 bit operands
`flags`	For 8 bit operands
`sp`	Low 8 bits `s` for 8 bit operands
`[push/pop/sp]`	Variant depends on instruction
`[`imm_{16}`]`	
imm_{16}	

Encoding of the dest/src field.

The `d` bit chooses between `sext(r8)` (`d=0`) and `zext(r8)` (`d=1`), except that the `test` instruction uses `sext(r8)` even though it is encoded with `d=1`. For the `imm16` and `[imm16]` cases, the `imm16` value follows the instruction word. For 8 bit instructions, `imm16` still takes 16 bits, but only the lower 8 bits are used by the instruction.

Execution timing

The that it takes to execute differemt instructions is influenced by 4 main factors:

- the number of execution cycles needed for a given instruction,
- cycles needed to wait for access to the TX channel, for instructions that send read/write requests to memory,
- cycles needed to wait for read response data, for instructions that read memory, and
- availability of new instructions to execute from the prefetch queue (which is flushed by jumps).

See the Inferface / Pins section for details of the memory interface. In the processor, the memory interface is shared between

- the *prefetcher*, which tries to fetch new instruction words a few steps ahead of the current program position, and
- the *scheduler*, which executes instructions, including making any memory accesses that they need to perform.

If the TX channel is idle and both try to start a new message TX message at the same time, the scheduler has priority.

Instruction stages The scheduler divides the execution of a given instruction into a subset of the following stages:

```
                               ror1 stage -> rotate stage
address stage -> data stage ->
                               mul1 stage -> mul2 stage
```

executed in the above order (the top and bottom tracks are mutually exclusive), where

- the *address stage* is used by instructions that need to send a memory address to RAM,

- the *data stage* is the main stage, and is used by most instructions to calculate and store their result,
- the *ror1* and *rotate stages* are used by `shift` instructions, and
- the *mul1* and *mul2* stages are used by `mul` instructions.

Stages that are not needed for the current instruction are skipped, and when the last stage that is needed is done, the instruction finishes. No extra cycles are needed to skip the address stage or to finish an instruction, but otherwise it takes one cycle to skip an unused stage.

The address stage always sends a read request on the TX channel, and takes 10 cycles starting from when the scheduler gains access to the channel.

The data stage takes a different number of cycles depending on the instruction:

- 4/8 cycles are needed to calculate and/or store an 8/16 bit result.
- If the instruction reads memory, the data stage needs to wait for the read response data to start arriving.
  - It does not need to wait for the read message to finish before finishing (when reading 16 bits but only needing 8).
- If the instruction writes memory, the data stage needs 6/10 cycles to send a write message on the TX channel (this includes two cycles to send a message header and 4/8 cycles to calculate and send the payload data, for 8/16 bit data).
- For read-modify-write instructions, the write request message will be initiated in the same cycle as the start bit for the read response is received for the read sent during the address stage.
- If the data stage reads and/or writes the `pc` register, (calls read from `pc`, all kinds of jumps write `pc`) it needs to wait for the prefetcher to be idle first.
  - The prefetcher is blocked while reading from the `pc`.
  - Writing to the `pc` is combined with sending the first read request to start prefetching from the jump destination, and is also combined with reading from `pc` for branches.
- Branches use only the data stage, and untaken branches finish after one cycle in the data stage.

Read-modify-write instructions send a read command on the TX channel, and reply with a write command when the read response is received, updating the data at the address just read. Since they need to be able to respond with a TX message as soon as an RX message arrives, and because the destination address for the write is given by the last read address, read-modify-write instructions block the prefetcher from sending read requests while waiting for their read response.

`mov` and `swap` are not considered to be read-modify-write instructions. They send a read request to specify the address, immediately followed by a write request to write the data. `swap` instructions need to wait for the read response before they can finish, but the prefetcher is allowed read access during the wait.

`shift` and `mul` instructions with immediate second operand skip the address and data stages completely. Other `shift` and `mul` instructions use the data stage (and address stage if needed) to load the shift count / multiplication factor from the second operand, which is always treated as 8 bit, also timing wise.

The ror1 stage is used by most shifts. It rotates the destination register right by a single bit if needed, clears the needed bits for `shl`, and finds the value of the sign bit for `sar`. It takes 4-5/8-9 cycles for 8/16 bit shifts (the extra cycle is a rotate right one step is needed). The ror1 stage is skipped if either

- the shift count is zero (the shift instruction is finished immediately from the ror1 stage), or
- the shift count is even, except for `sar` and `shl` instructions.

The rotate stage is used by all shifts except when the shift count is zero. It rotates the destination register right by 2 bits/cycle, taking as many cycles as needed to produce the desired rotation. It also feeds in zero/sign bits for `shr` and `sar` instructions.

The mul1 stage is used by all `mul` instructions. It calculates the bottom part of the product (the part that is stored into the first operand), and takes 4/8 cycles for 8/16 bit operands. The mul2 stage is used to write the top 8 bits of the product to the `h` register. It takes 4 cycles, but the instruction is finished after the mul1 stage if the `h` register overlaps with the destination.

The prefetcher The objective of the prefetcher is to try to keep the decoder and scheduler fed with instruction words, even though it might take some time between the point when a read request is sent and the read response data is received. The prefetcher has a prefetch queue of 2 - 4 instruction words (see Basilisc-2816 v0.1 variants above).

When the current instruction is finished (or there is no currently executing instruction), the decoder tries to load the next instruction from the head of the prefetch queue to start executing it. Instructions that use an `imm16` operand (including `[imm16]`) wait for and load an additional instruction word from the prefetch queue before they can start. This takes at least one cycle.

An instruction word is considered to be in the prefetch queue from the point when it the read request is sent to when it is loaded into the instruction and/or immediate register, or is flushed due to a jump. As long as the prefetch queue is not full, the prefetcher tries to send new read requests to fill it.

When a read response for a prefetch arrives, the instruction word is stored in the 2 - 4 prefetch queue buffers (same number as the size of the prefetch queue) waiting to be consumed by the decoder and/or scheduler. The whole instruction word must arrive into the first buffer before it is considered valid. Even if the other prefetch queue buffers are empty, it takes on cycle for an instruction word to traverse each one.

Jumps Whenever a jump is taken, the current contents of the prefetch queue are flushed, meaning that they will not be used (including responses to read requests that have already been sent, but where the response has not yet arrived).

Jumps update the `pc` register during the data stage. At the same time, they send the first read request to prefetch from the jump destination.

Calls (whether relative or absolute) are special in that they are scheduled like two instructions in sequence: `push pc+2 / push pc+4` followed by the corresponding `jump` instruction.

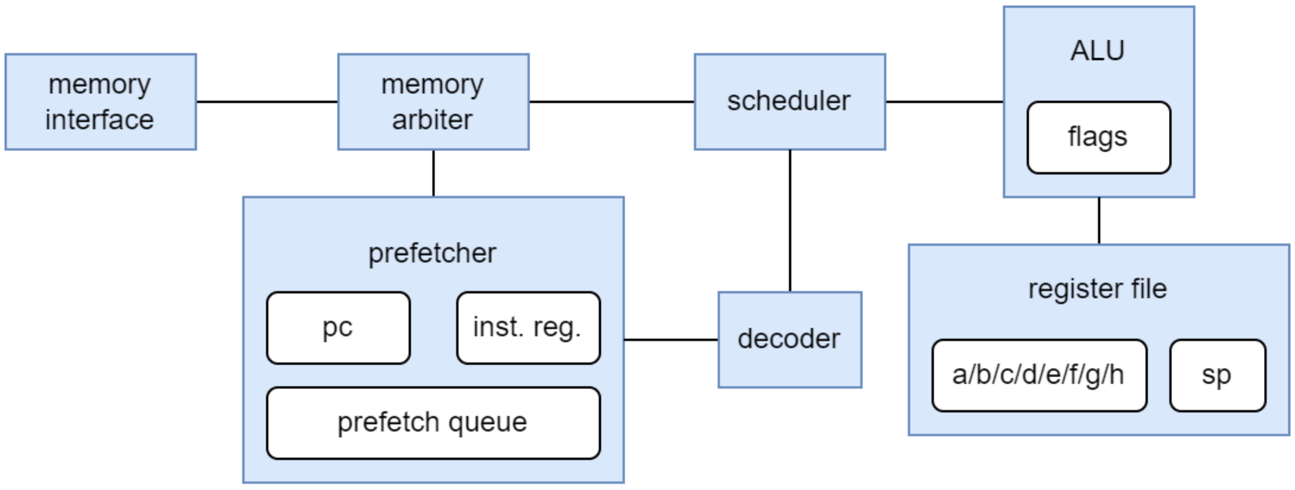
Performance considerations Performance characteristics of different types of instructions:

- Instructions that operate on only registers and immediates are generally fast, don't need to wait for memory access, and allow the prefetcher to work in parallel.
  - 8/16 bit `mov/swap/binop` with only register (including `sp/p/flags/ sext(r8)/zext(r8)`) and immediate operands generally take 4/8 cycles.
  - `shift` and `mul` instructions usually take more cycles (see above), but still allow the prefetcher to work in parallel, except for in the address stage (if any).
    * `shift` and `mul` instructions with immediate second operand are faster since they don't need to spend time on loading the shift count/factor.
    * For `shift` instructions, the ror1 stage can be skipped under certain circumstances, and the length of the rotate stage depends on the shift count: small right shifts / big left shifts are faster (see above).
- `mov/binop reg, mem` instructions need to send a read request and wait for the response, but allow the prefetcher to work while waiting for the response. Likely, the prefetcher has a chance to get ahead while waiting.
- `mov mem, reg` instructions need to send a read request and a write request, but don't need to wait for a response.

- `swap mem, reg` instructions need to send a read request and a write request, and wait for the read response, but allow the prefetcher to run while waiting.
- Read-modify-write instructions `binop mem, reg` need to send a read request and a write request, and block the prefetcher while waiting for the read response.
- (Taken) jumps flush the prefetch queue, causing execution to stall until new instructions can arrive at the decoder.
- `imm16` / `[imm16]` operands require an extra instruction word to be prefetched (and require one additional cycle to load). They still allow the data to be prefetched, unlike with reads initiated by the instruction.

The things to be most careful about performance wise are probably read-modify-write instructions and jumps. On the other hand, read-modify-write instructions can help relieve register pressure, and jumps are often necessary and sometimes save more time than they cost.

How it works



Major parts and interconnections of the processor.

The CPU consists of a number of parts:

- The *memory interface* communicates with the external memory (RAM emulator) to send and receive serial messages.
- The *memory arbiter*
  - arbitrates memory command transmission between the prefetcher and scheduler, and
  - keeps track of outstanding responses, directing incoming responses to the intended recepient.
- The *prefetcher*

  - keeps track of and updates the *program counter* (PC) register,
  - tries to fetch instruction words a few steps ahead of the currently executing instruction,
  - buffers prefetched instruction words in the prefetch queue buffer, and
  - keeps track of the currently executing instruction in the *instruction register* and the *immediate register*.

- The *decoder*
  - recieves the current instruction from the prefetcher/instruction register, and
  - decodes it into control signals for the scheduler.
- The *scheduler*
  - evaluates branch conditions,
  - divides instructions into stages,
  - runs the stages needed for each instruction in order, and
  - sets control and data signals to the ALU and other parts of the CPU as needed to execute each stage.
- The *ALU* (arithmetic logic unit)
  - calculates ALU operations (binary operations `add/sub/adc/sbc/and/or/xor` etc), shifts, and multiplies,
  - breaks down operations on 16 bit register pairs into back-to-back operations on 8 bit registers,
  - breaks down and schedules calculations into 2-bit serial steps,
  - updates the flags based on the results of a computation,
  - reads external inputs and produces an external result when needed, and
  - reads and updates the register file.
- The *register file*
  - holds most of the CPUs registers: general purpose registers `a-h` and `sp`, and
  - mediates access to the flags as the `flags` register.

2-Bit serial operation The CPU uses 2-bit-serial operation wherever it can. This means that instead of operating on all bits of a value in parallel, it operates on 2 bits per cycle. Since the memory interface can only send/reveive 2 bits/cycle, there is little point in making most other things operate faster. Serial operation saves area compared to parallel operation for both computions and registers.

Consider adding two values that are stored in separate registers. Addition starts from the least significant 2 bits (lsbs), and proceeds upwards, 2 bits/cycle. To facilitate this, the registers are organized as shift registers, which can be right-shifted two bits per cycle, feeding out the 2 bottom bits. At the same time, two new bits are fed into the top.

The two lsbs from each register respresent a value between 0 and 3, and the sum has a value between 0 and 6. With a carry bit coming in, the sum can be between 0 and 7. The bottom 2 bits of the sum are used as the result at the current 2-bit position, while the top bit is used as the carry into the next 2-bit position. A flip-flop is needed to store the carry from one cycle to the next. At each step (cycle), the registers are shifted right by two bits, so that the 2 bits to be operated on are always at the bottom of each register.

The addition can be performed in place, replacing the value in one of the registers with the result: The result bits are just shifted into the top of the register. If a register should keep its value after the operation, the lsb bits shifted out are shifted back into the top at the same time. When the register contents have been shifted around to the starting position, the addition is completed.

Memory arbiter The memory arbiter keeps track of which block is currently using the TX channel, if any. If both the prefetcher and scheduler try to start a transaction at the same time, the scheduler gets priority. The scheduler has been designed under the assumption that it will get priority whenever it raises the `reserve_tx` signal, which also blocks the prefetcher from sending any new TX messages. This is needed for read-modify-write instructions, which send a read request followed by a write request, where the read address is reused as write address for the write request.

The memory arbiter has a FIFO that keeps track of outstading requests (reads), which have been started but where the response has not yet finished. The FIFO keeps track of whether a given read response should be handled by the prefetcher or the scheduler, or if it should be ignored completely. The ignore feature is used for write instructions (`mov mem, reg`, `push pc+2/4` during `calls`), where the read message is just used to set the write address. This allows the scheduler to move on without waiting for a read response. The scheduler is designed to have only one outstanding (non-ignored) read, so that when it receives a read response, there is no doubt about what the data is for.

Prefetcher The prefetcher holds the 16 bit program counter register `pc`, and updates it using 2-bit serial operations, which can add a delta to the `pc` or set a new value. The least significant bit of `pc` is always expected to be zero, but the storage space is still needed since 2-bit serial addition causes the `pc` value to rotate around through the `pc` bits.

Except when jumping, the pc register is incremented by 2 for each prefetch. When the prefetcher sends a read message on the TX channel, it scans through the pc register and increments it at the same time, sending the incremented pc value as the read address. The bits of the incremented pc value are sent as soon as they are calculated, starting from the lsbs.

The prefetcher also keeps track of the number of prefetched and flushed instruction words. The pc register does not store the address of the currently executing instruction, instead it stores the address of the last instruction that was prefetched, `curr_inst_addr + 2*num_prefetched`.

An instruction word is considered prefetched between when a read request is first sent for it and when it leaves the prefetch queue to enter the instruction and/or immediate register, or is considered flushed by updating the `num_flushed` register. The prefetcher has space to store 2-4 prefetched instruction words that have been read but not yet executed. It takes a cautious approach, and will not send prefetch read transactions unless there is currently space to store the read response data in the prefetch queue buffers when it arrives. The first buffer stage is a serial shift register to receive incoming serial data. Each remaining buffer stage is loaded in parallel from the previous stage. This occurs when one stage has valid data and the next one does not, making the instruction words fall through the prefetch queue buffers until there is no free space to fall into.

The prefetcher also holds the instruction and immediate registers. When loading a new value into the instruction register, the same value is loaded into the immediate register as well. For instructions that use an `imm16` or `[imm16]` argument, the immediate register is loaded with the next value from the prefetch queue.

The instruction register is used as the input to the decoder. The immediate register is used to feed the ALU with an immediate value, typically shifting out 2 bits per cycle. This works since the immediate field is always placed in the least significant bits of an instruction. The bottom half of the immediate register is also used to store the shift count of `shift` instructions and 2nd operand of `mul` instructions. `shift/mul` instructions that don't have these stored as immediates shift the needed data into the immediate register during the data stage. The top half of the immediate register is also used as scratch space by the `mul` instruction.

When reading out the pc (for branches and calls), the adder that is normally used to increment it is used to subtract `2*num_prefetched` to compute the address of the current instruction. The prefetcher must be blocked from sending new read requests to allow reading out the pc register. In the case of branches, the pc is read out, adjusted, updated, and used to send a prefetch read request for the jump destination, all at the same time: The adjusted pc value is sent to the ALU, which adds the offset from the branch, and sends the branch target pc value to be sent as prefetch address and stored in the `pc` register. Jumps are similar to branches, but don't need to read the current value of the `pc`, just to set it to the value output from the ALU.

The reason that the prefetcher outputs the incremented `pc` value as read address is so that the same target address can be stored into the `pc` register and output in a prefetch read request when jumping, saving an adder.

When jumping, the prefetcher also sets `num_prefetched = 1`, to reflect that the jump is started with a single prefetch. Before that, `num_flushed` is set to `num_prefetched`, to flush all instruction words prefetched after the the jump instruction that was executed. As long as `num_flushed` is nonzero, instruction words that arrive at the output of the prefetch queue are discarded immediately, instead of feeding the instruction or immediate registers, or waiting to be consumed. The `num_flushed` register is decreased by one for each time an instruction word is discarded in this way.

Prefetch queue using latches instead of flip flops The experimental v0.1c version of the CPU uses latches instead of flip flops in the prefetch queue buffers and instruction register. This saves space, but is a bit more tricky to work with. Hopefully, it works as intended.

To try to make the latches behave as desired, the design

- avoids glitches on the gate input of each latch by feeding it directly from a flip flop, and
- makes sure that the input data to each latch stays stable one cycle after the gate was closed.

Each buffer stage includes 16 latches, a valid flag register, and a write enable register. The cycle-by-cycle transfer of data from one stage to the next proceeds as follows:

1. When one buffer stage has valid data and the next stage does not, the write enable signal of the second stage is raised.
2. The second stage's write enable register goes high, opening the latch gates in the second stage to read the data from the first stage.
    - At the same time, the second stage's valid register goes high, indicating that the output data is valid, since the latch is transparent. This causes the write enable signal to go low.
3. The valid flag of the first stage goes low, due to the write enable that was high during the previous cycle.
4. The first stage may change its data during this cycle, since the valid flag was low during the previous cycle.

Compared to using flip flops for the prefetch queue buffers, the concern is mostly about keeping the input data to each latch stage stable for one cycle after the gate goes low. Instruction words can still propagate through the queue at one stage per cycle, if there is space.

The instructon register works much like the final stage of the prefetch queue buffer, and can be treated in the same way. The last stage in the prefetch queue buffer can also feed the immediate register or be flushed, but this mostly adds some logic for when to clear its valid flag.

Decoder The decoder decodes instruction words according to instruction type, instruction form, and addressing mode, and instructs the scheduler which operations need to be performed: which operation, sources, and destination to use for the data stage, which operation and sources to use for the address stage (if any source or destination in the data stage needs one), whether to update flags, if special features like swapping, condition codes, shifting, and multiplication should be activated, etc.

The decoder also schedules a `push pc+2` or `push pc+4` as the first stage of executing a call (including `branch call`) instruction, which is otherwise executed just like a `jump/branch`, calculating the size of the instruction already at the push stage. Philosophically, this falls into the domain of the scheduler, but practically, it was simpler to let the decoder take care of this scheduling task to avoid complicating the stage structure of the scheduler.

Register file The register file is indexed using 4 bit register indices. The first 8 indices are connected to the general register file, while additional indices are used for the `s, p`, and `flags` registers. The general register file is implemented using a bank of addressable shift registers; the additional registers are implemented with some extra glue logic and storage.

The general register file has two read/write ports, with separate 3 bit register indices, enable signals, inputs and outputs. It is implemented as 8 addressable 8 bit shift registers. Registers that are addressed through a read/write port that is enabled are shifted, shifting in the input value for the appropriate port. The input from port 1 has priority.

If a value should be read out without being modified, the user of the register file (the ALU) needs to recirculate the bits that are read out from the port back to its input. Each port is generally expected to be enabled 4 times in sequence with the same register index, so that values can circulate back to their original bit positions. The exception is shift operations, which vary the number of enable cycles to achieve different bit shifts.

Addressing modes When sending a read/write address using a read request on the TX channel, the address bits need to be retrieved/computed 2 bits at a time. The ALU and register file have nothing else to do at this time, and are used to calculate the address. Either the ALU's second argument is used as the result (just like in a `mov` instruction, which ignores the destination's value), or an addition is used, sometimes exploiting the ALU's feature to multiply the second argument by two (used only for address computations, branches, and calls). The ALU can also sign/zero extend the second argument to different lengths, which is used by various addressing modes.

Postincrement/predecrement addressing modes (including `[pop]` and `[push]`) write the result of the computation back to the ALU's first register argument, just like instructions that write to registers. For the postincrement case, the ALU result is written to the register, while the value read out from it is used as the address.

There are more options that make sense for `src` operands than `dest` operands. This fact is used to differentiate between the `sext(r8)` and `zext(r8)` source forms; `zext(r8)` is `sext(r8)` with the `d` bit set to 1, which would normally mark it as a destination. (Except for the `test` instruction, which also uses the `d` bit to distinguish it from the `cmp` instruction, but retains `sext(r8)` behavior.)

`swap` instructions `swap reg1, reg2` instructions work like `mov reg1, reg2` instructions, but feed the second register file port's input with the output from the first.

`swap mem, reg` instructions use a special trick to be able to start reading out the register data immediately and start shifting in the new data later, when it arrrives. First, for 16 bit swap instructions, the two register ports are connected as one big shift register, to avoid having to switch between lower and upper halves of a register pair; reading and writing would not agree on when to switch.

The write request is started as soon as possible after sending the read request, and the ALU is allowed to shift out bits from the register file to feed as write payload, but the ALU does not start counting cycles to finish the operation. The bits shifted into the target register at this point may be garbage. When the read response payload arrives, the ALU starts counting cycles. The payload bits are shifted into the target register and the instruction is finished. This may partially overlap with the cycles used to read out the original register contents.

This setup allows the register file to take care of read response data without sending a write message in response (it has already been sent), which means that the scheduler doesn't have to block the prefetcher while waiting for the read response data.

`shift` instructions `shift` instructions rely on the register file's ability to shift/rotate values in place, rotating right 2 bits per cycle. For this reason, all shift operations operate in place on register values. Some complications arise since

- the only supported shift step is 2 bits (how to handle odd shift counts?),
- the only supported shift direction is right shift,
- the sign bit (top bit) of the registers in the register file is not readily available, but needed by `sar`.

For the 16 bit case, both register file ports are connected together to form a 16 bit shift register.

To work witin the restrictions:

- Odd shift amounts are handled by the ror1 stage, which needs 5/9 cycles to rotate an 8/16 bit value right one step by rotating it right 5/9 cycles in steps of 2 bits, through a 1 bit delay.
- `sar` records the sign bit at the end of the ror1 stage.
- Left shifts and rotates are implemented using right rotation with `8 - shift_count` or `16 - shift_count` number of steps.

Also:

- `shr` and `sar` instructions replace the topmost bits with zeros/the sign during the final rotate stage (and the final ror1 cycle, for odd shifts).
- `shl` replaces bits with zeros during the `ror1` stage - the bit positions to be cleared are exactly those that will not be rotated through the lsbs during the final rotate stage.
- The shift count for `rol` / `shl` is negated while loading it into the immediate register during the data stage.
  - `rol reg, imm4` must be encoded a a `ror` instruction, with a negated shift count.

`mul` instructions `mul` instructions use the immediate register for two purposes:

- The bottom half, which we will call the `factor` register in this context, holds the 8 bit unsigned factor from the second operand.
- The top half, which we will call the `partial` register in this context, holds a partial sum.

For `mul reg, imm6` instructions, the `imm6` factor is loaded into the `factor` register when loading the instruction. For `mul reg, src` instructions, the data stage loads the `factor` register, in the same way as `shift reg, src` instructions load the same register with the shift count.

A `mul` instruction multiplies an 8/16 bit destination register with the 8 bit `factor` register, producing a 16/24 bit product. Every cycle, it consumes 2 bits from the original value of the destination register and produces two bits of the product.

Consider multiplying an 8 bit factor `dcba` with an 8 bit value `hgfe`, where each letter denotes 2 bits. The needed computation can be written as

```
 dcba    * h
  dcba   * g
   dcba  * f
+   dcba * e
--------
```

The basic algorithm starts by calculating `dcba * e`, the first term in the sum, forming the current partial sum. This is the only term that affects the bottom 2 bits of the result, which can be output as the first 2 result bits. The remaining bits are shifted 2 steps right to align with the next level. In the second step, `dcba * f` is added to the partial sum from the last step, producing a new partial sum, and everything proceeds as before, etc.

The basic algorithm still requires the multiplication of an 8 bit factor by a 2 bit number every cycle. Multiplication by 0 and 1 are easy, and multiplication by 2 is just a left shift, but multiplication by 3 would require an extra adder. Instead, the multiplier multiplies by -1. This produces the same result for the lowest 2 bits. At the same time, it sets a carry to add one to the incoming 2 bit value during the next cycle. This results in the need to multiply by a number between 0 and 4, but 4 can be handled in a similar way 3, in this case: multiply by zero, set a carry for the next step.

In total, the multiplier requires

- 9 bits of scratch space for the partial sum (the `partial` register and a sign bit, reusing the ALUs temporary sign bit (not the sign flag)),
- an 11 bit adder,
- a one step shifter, and provisions to multiply the factor by zero or invert its bits.

How to test

TODO

External hardware

Requires the RP2040 microcontroller on the Tiny Tapeout 7 demo board or similar to serve as RAM emulator, adhering to the memory interface protocol described above.

Pinout

#	Input	Output	Bidirectional
0	rx_in[0]	tx_out[0]	
1	rx_in[1]	tx_out[1]	
2		tx_fetch	
3		tx_jump	
4			
5			
6			
7			

RLE Video Player [969]

- Author: Mike Bell
- Description: Reads run length encoded data from QSPI flash, displays on VGA
- GitHub repository
- HDL project
- Mux address: 969
- Extra docs
- Clock: 25175000 Hz

How it works

A 6bpp run length encoded image or video is read from a W25Q128JV or similar QSPI flash, and output to 640x480 VGA.

This is perfect for displaying the Bad Apple music video.

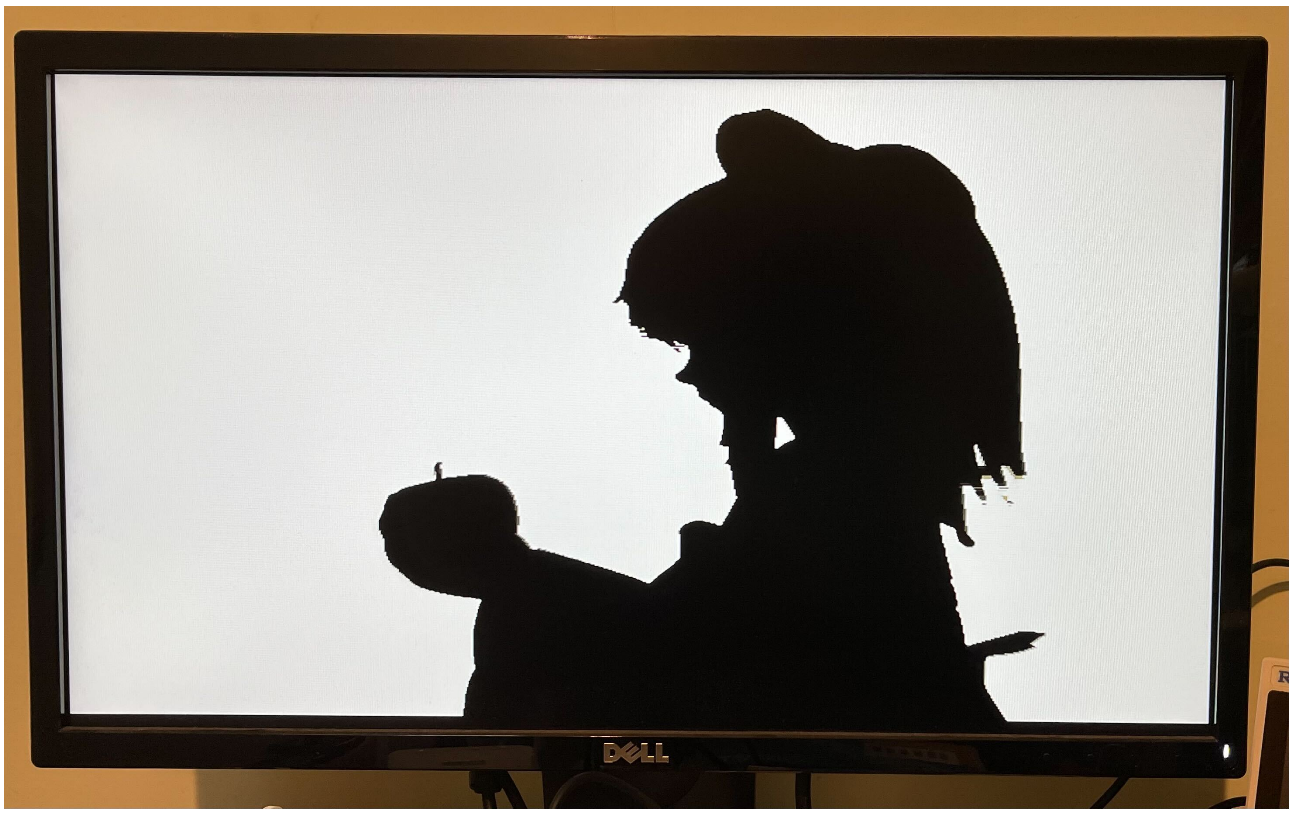


Figure 45: A frame from Bad Apple, rendered by the FPGA version of this design

Run Length Encoding The encoding uses 16-bit words. Most words are a run length in the top 10 bits, and a colour in the bottom 6 bits. A run must come to the end at the end of each row.

A row can be repeated by encoding a word `0xF800` + number of repeats at the end of a row.

A run must be at least 2 pixels, and any group of 3 consecutive runs within a row must be at least 24 pixels, otherwise the data buffer will empty. This could definitely be improved!

If input 3 is high, each frame is repeated once, so playback is 30Hz instead of 60Hz.

The data is read starting at address 0. The special word `0xFFC0` causes the player to stop and restart from address 0 at the beginning of the next frame, restarting the video. This could also be used to display a still image.

How to test

Create a RLE binary file (docs/scripts to do this TBD) and load onto the flash. The pinout matches the QSPI PMOD. Connect that to the bidi pins. Note the flash must support the h6B Fast Read Quad Output command, with 8 dummy cycles between address and data.

Connect the Tiny VGA PMOD to the output pins.

Inputs 2-0 set the read latency for the SPI in half clock cycles, it's likely that will need to be set to 2 (set input 1 high and inputs 0 and 2 low). This latency depends on the total round trip time through the mux and out to the flash and back. Valid values are 1 to 4.

Run with a 25MHz clock (or ideally 25.175MHz).

External hardware

- QSPI PMOD
- Tiny VGA PMOD

Pinout

#	Input	Output	Bidirectional
0	SPI latency[0]	R[1]	CS
1	SPI latency[1]	G[1]	SD0
2	SPI latency[2]	B[1]	SD1
3	30Hz select	vsync	SCK
4		R[0]	SD2

#	Input	Output	Bidirectional
5		G[0]	SD3
6		B[0]	Unused CS
7		hsync	Unused CS

secret L [971]

- Author: stuart childs
- Description: basic first test - secret code on input displays a letter as output
- GitHub repository
- Wokwi project
- Mux address: 971
- Extra docs
- Clock: 0 Hz

How it works

Uses a pair of inverters and a pair of AND gates

How to test

Use the DIP switches to input a secret code - then the display will show a letter. Basic test in order to learn more about the TT process

External hardware

Not needed

Pinout

#	Input	Output	Bidirectional
0	input0		
1	input1		
2	input2		
3			
4			
5			
6			
7			

Send Receive [973]

- Author: Michael Ogata, Nelson Hastings, Samson Melamed
- Description: A dual use project where you can set a project to either a sender or receiver and send bits
- GitHub repository
- HDL project
- Mux address: 973
- Extra docs
- Clock: 0 Hz

How it works

Send Recieve Project Description

A simple 4-value send/receive chip

This project realizes a very simple 4-value sender and receiver chip. The mode of the chip is determined by input 7: **sender low, receiver high**. Currently, the protocol only supports 1 sender and 1 receiver.

Sender Mode Sender mode is enabled when **Input[7]** is set low. In sender mode, the chip looks for a rising value on bits 0:4 of the input bus. This can be achieved through the use of a 4-button PMOD module, or by toggling the corresponding DIP switch on the carrier board.

Sender Mode Input:

#	Input	Description
0	input[0]	bit[0] to send
1	input[1]	bit[1] to send
2	input[2]	bit[2] to send
3	input[3]	bit[3] to send
4	unused	N/A
5	unused	N/A
6	unused	N/A
7	input[7]	Sender/Receiver indicator

To send data, the sender pulls **Output[0]** high and relays the corresponding input bit (i) on **Output[i+1]** (Note: currently only one bit can be sent at a time). The

Sender also sets **Output[7]** to indicate to the connected chip that it should operate in receiver mode.

Finally, The sender keeps **Output[0]** high long enough to prevent the receiver's debounce filtering from masking the signal before setting it low again.

Sender Mode Output:

#	Output	Description
0	Transmit	Signal receiver to read data
1	x_bit[0]	Transmit bit[0]
2	x_bit[1]	Transmit bit[1]
3	x_bit[2]	Transmit bit[2]
4	x_bit[3]	Transmit bit[3]
5	unused	N/A
6	unused	N/A
7	Mode_of_connected_chip	Sender/Receiver indicator

Receiver Mode Receiver mode is enabled when **Input[7]** is set high. In receiver mode, the chip listens on **Input[0:5]**. It waits until the positive edge of **Input[0]** before registering an input.

Receiver Mode Input:

#	Input	Description
0	Read	Read input bits
1	input[0]	recv_ bit[0]
2	input[1]	recv_ bit[1]
3	input[2]	recv_ bit[2]
4	input[3]	recv_ bit[3]
5	unused	N/A
6	unused	N/A
7	input[7]	Sender/Receiver indicator

The chip utilizes the carrier chip's 7 segment display to echo a value corresponding to the input data:

Input[1:4]	Output Value	Display Value
1000	0000_0110	1
0100	0101_1011	2
0010	0110_0110	4

Input[1:4]	Output Value	Display Value
0001	0111_1111	8

The receiver initially displays zero (on the 7-segment display) until it receives the read signal from the sender. The receiver then displays the value received or an E when the value is not 1, 2, 4, and 8. The last value received, or E is displayed until the receives gets the signal to read the next value.

How to test

To connect two chips, configure them as follows:

- Sender
  - Inputs: (Choose One)
    - None (i.e. the chip will read data from the carrier chip's DIP switches)
    - 6-pin, 4-button PMOD keypad plugged into **Input[0-5]** (top row)
  - Outputs:
    - Connect all 12 output pins to the inputs on the receiver
- Receiver
  - Inputs
    - Power, ground, and all 8 data pins from the sender must be connected to the receiver's **corresponding** inputs. This requires the use of a specially crafted cross-over cable to ensure the correct connections.

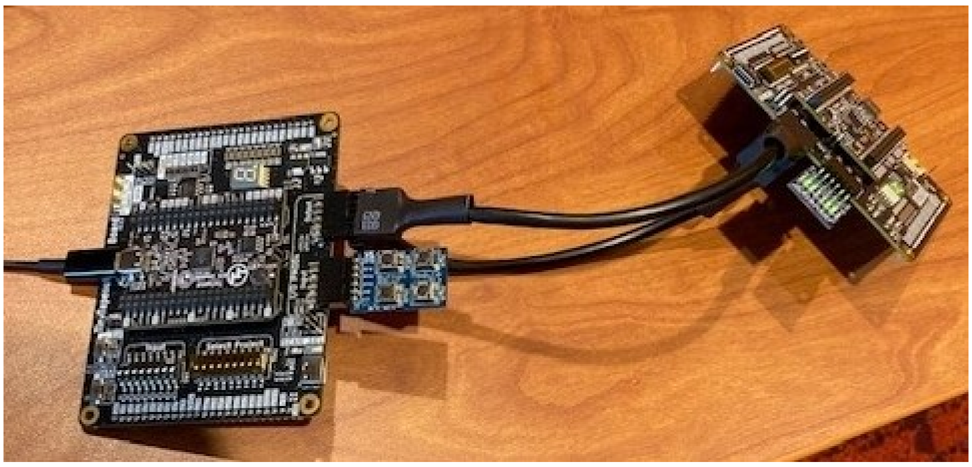


Figure 46: picture of two chips connected as sender-receiver

External hardware

- PMOD 4 button keypad

Pinout

#	Input	Output	Bidirectional
0	Input[0]/Message Ready	Transmit/LCD Display Value[0]	
1	Input[1]/recv_bit[0]	input[0]/LCD Display Value[1]	
2	Input[2]/recv_bit[1]	Input[1]/LCD Display Value[2]	
3	Input[3]/recv_bit[2]	Input[2]/LCD Display Value[3]	
4	unused/recv_bit[3]	Input[3]/LCD Display Value[4]	
5	unused	unused/LCD Display Value[5]	
6	unused	unused/LCD Display Value[6]	
7	Sender/Receiver Mode Select (0/1)	Mode of connected chip//LCD Display Value[7]	

RISCV32I with spi wrapper [974]

- Author: Devin Macy
- Description: A mostly support riscv32i CPU with a bare bones spi wrapper allowing you to upload programs to instruction memory and echo information sent through SPI
- GitHub repository
- HDL project
- Mux address: 974
- Extra docs
- Clock: 50000000 Hz

How it works

My project is a 5-stage riscv32i cpu core supporting most of the bare bones instruction set. The instructions that are not supported are any instructions dealing with csr's, harts, memory fences, or modes of operation. The cpu core has 16 words (64 bytes) of instruction memory and registers and 8 words (32 bytes) of data memory.

For programmability, a spi wrapper has been added that starts in boot mode, requiring you to upload a program and entering echo mode before the cpu can do anything. When the exit boot command is given to the spi it will enter echo mode, releasing the cpu from reset, and repeat back any byte given to it as a sort of health check.

In its current state, there is no way to observe the status of the cpu not in simulation, since the spi doesn't hand over control of itself to the cpu to output data. However the passed/failed signals on `uo_out[6]` and `uo_out[7]` respectively, check to see if register 10 is equal to 45. Which you can upload a simple program that adds numbers from 1-10 and stores them into register 10 to test functionality.

Impact This project was developed in part with the Microelectronics Security Training Center (MEST) through the class "ChipCraft: The Art of Chip Design for Non-Experts" deveoped and taught by Efabless and Redwood EDA. The class is designed for non-experts in the field and covers the entire microelectronics ecosystem including RTL design, Verification, GDS generation, tape-out process, and fabrication.

During the class, you get hands on experience and learn concepts of microelectronics design by designing a calculator and prototyping on the first time use of the TT FPGA demo board before moving to designing a 5-stage riscv32i cpu using Makerchip and TL-Verilog.

How to test

Have prepared a riscv32i binary of up to 16 instructions here is a pretty good rescource for putting together binaries The cpu has 3 control and status registers (csr's) in data memory, taking up the first 3 entries and giving the user 5 words to work with.

data memory	csr usage
`dmem[0]`	32-bit cycle counter
`dmem[1]`	spi wrapper csr - bits 7-0 is the last byte recieved and bit 8 is a recieved byte valid signal. the rest are unused
`dmem[2]`	the last 32-bit instruction written to instruction memory. populated when command 0xC5 is issued

Connect the spi signals `SCLK`, `CS`, `MOSI`, and `MISO` to their respective pins. The spi commands are as follows:

command	code	description
load `ll_byte`	0xC0	load bits 7-0 of the instruction word
load `lh_byte`	0xC1	load bits 15-8 of the instruction word
load `hl_byte`	0xC2	load bits 23-16 of the instruction word
load `hh_byte`	0xC3	load bits 31-24 of the instruction word
load `imem_addr`	0xC4	load what 3-bit address you want to write the previously built instruction word to
write to `imem[imem_addr]`	0xC5	write built instruction to address loaded into imem_addr
exit boot mode	0xC6	enter echo mode, echoing back any byte given afterwards
re-enter boot mode	0xC7	re-enter boot mode, holding cpu in reset
command error	default	invalid command was given, throw cmd_error on `uo_out[5]` high requiring a reset to clear

*The spi commands require 2-bytes per command, even if the command doesnt use the second byte

**The current mode can be observed on `uo_out[4]`. Mode is low when in boot and high when in echo

Toggle `CS` high then low after power on. Program the cpu through spi by following a cmd,data,cmd,data cadence loading all the bytes of the instruction word, then loading the address to write to, then writing to imem until finish writing your program to instruction memory. Send the exit boot command to release the cpu and observe the results.

Here is an example (in verilog syntax) of a buffer used to program the instruction word `addi x10, x0, 45` into address 5 then echo 0xAA

```
buff[14*8] = {8'hc0, 8'h13, 8'hc1, 8'h05, 8'hc2, 8'hd0, 8'hc3, 8'h02,
              8'hc4, 8'h05, 8'hc5, 8'hxx, 8'hc6, 8'haa}
```

Transmitted MSB of the buffer first, and in words is

```
buff[14*8] = {load, ll_byte, load, lh_byte, load, hl_byte, load, hh_byte,
              load, imem_addr, write imem[imem_addr], dont care, exit boo
```

You will need to program more than 5 instructions to really see any results, since it is a 5-stage pipeline and takes 5-cycles to see a write-back to the register file. An example program that adds up numbers 1-10 and stores them into register 10, in assembly, is as follows:

```
.text
         reset:
            ADD x10, x0, x0            # Initialize r10 (a0) to 0
            ADD x14, x10, x0           # Initialize sum register a4 with
            ADDI x12, x10, 10          # Store count of 10 in register a
            ADD x13, x10, x0           # Initialize intermediate sum reg
         loop:
            ADD x14, x13, x14          # Incremental addition
            ADDI x13, x13, 1           # Increment count register by 1
            BLT x13, x12, loop         # If a3 is less than a2, branch t
         done:
            ADD x10, x14, x0           # Store final result to register
            JAL x1, done               # Infinite loop storing result to
```

External hardware

male-to-male connectors and a arduino

Pinout

#	Input	Output	Bidirectional
0	sclk		
1	cs		
2	mosi		
3		miso	
4		mode	
5		cmd_error	
6		passed	
7		failed	

ALU 74181 [975]

- Author: Caio Alonso da Costa
- Description: SPI peripheral and an 8-bit ALU implemented with 2x 4-bit slice arithmetic logic unit 74181
- GitHub repository
- HDL project
- Mux address: 975
- Extra docs
- Clock: 50000000 Hz

How it works

Replica of the famous 4-bit slice arithmetic logic unit (ALU). https://en.wikipedia.org/wiki/74181

The project instantiate two times the replica of the 74818 to perform mathematical and logical operations on 8 bit words.

A multiplex is used to taps different parts of the user logic and map them to the 7 segment display to support debug.

Due to I/O constraints, a SPI slave peripheral has been created to load/read data into the design.

SPI Slave peripheral implementation supports all 4 SPI modes of operation. 8 Configurable (Read/Write) registers. 8 Status (Read only) registers.

RP2040 SPI1 is used to communicate with the device. Map SPI1 IOs to GPIOs 24 to 27.

Limitations on SPI:

- Single register access per SPI transaction.
- SPI transaction is limited to 16 bits transfer at a time (Addr + Data). Please refer to Protocol for timing diagrams.
- Design tested for 8 configuration registers + 8 status registers.
- Even though the number of configuration registers and status registers is configurable, design only supports equal number of configuration and status registers for now.
- Writes targeting Read Only address are dropped, i.e., no configuration registers gets updated.

Address Space:

Address	Type of register
0	Configurable Read/Write register [0] - Data A (8 bits)
1	Configurable Read/Write register [1] - Data B (8 bits)
2	Configurable Read/Write register [2] - {c_in, M, S3, S2, S1, S0} [5:0] (6 bits)
3	Configurable Read/Write register [3] - Select for 7 segment display [2:0] (3 bits)
4	Configurable Read/Write register [4]
5	Configurable Read/Write register [5]
6	Configurable Read/Write register [6]
7	Configurable Read/Write register [7]
8	Status Read Only register [0] - Data F (8 bits)
9	Status Read Only register [1] - {c_out0, equal0, p0, g0, c_out1, equal1, p1, g1} [7:0] (8 bits)
10	Status Read Only register [2] - Output of debug Multiplexer [3:0] (4 bits) and Zeros [7:4] (4 bits)
11	Status Read Only register [3] - Output of bin_to_7seg_decoder (8 bits)
12	Status Read Only register [4] - Fixed data 8'hC4 (8 bits)
13	Status Read Only register [5] - Fixed data 8'h10 (8 bits)
14	Status Read Only register [6] - Fixed data 8'h66 (8 bits)
15	Status Read Only register [7] - Output of bin_to_7seg_decoder delayed by 1 clock cycle (8 bits)

Connection

RP2040 SPI Master <–SPI–> SPI_WRAPPER <–regaccess–> User logic

- SPI: MOSI MISO SCLK CS
- regaccess: config_regs (used to drive/control user logic), status_regs (used to read/monitor user logic)

Protocol

SPI settings

- Address Bits = 4 and Databits = 8, MSB First
- Tested SPI frequency: spi_clk <= clk / 20

SPI commands

- Write data cmd = 0x80+addr, addr = 0 ~ 7

```
Bit:         | <15>      <14>         <13>         <12>        <11>
MOSI:        |   1   | Don't Care | Don't Care | Don't Care | addr[3] |
MISO:        |   0   |     0      |     0      |     0      |    0    |
CS:       1  |   0   |     0      |     0      |     0      |    0    |
```

Wavedrom for Write data transfer:

```
{signal: [
  {name: 'spi_cs', wave: '10................1'},
  {name: 'spi_clk', wave: '0.P..............P0'},
  {name: 'spi_mosi', wave: 'z1x..3333444444444zz', data: ['Addr[3]', 'Addr
  {name: 'spi_miso', wave: '0..................'}],
head: {text:
  ['tspan',
    ['tspan', {class:'error h3'}, 'Write transfer '],
  ]
},
config: { hscale: 2 },
}
```

- Read data cmd = 0x00+addr, addr = 0 ~ 15

```
Bit:       | <15>      <14>         <13>         <12>        <11>
MOSI:      |   0   | Don't Care | Don't Care | Don't Care | addr[3] | ad
MISO:      |   0   |     0      |     0      |     0      |    0    |
CS:    1   |   0   |     0      |     0      |     0      |    0    |
```

Wavedrom for Read data transfer:

```
{signal: [
  {name: 'spi_cs', wave: '10................1'},
  {name: 'spi_clk', wave: '0.P..............P0'},
  {name: 'spi_mosi', wave: 'z0x..3333xxxxxxxxzz', data: ['Addr[3]', 'Addr
  {name: 'spi_miso', wave: 'z0.......33333333zz', data: ['Data[7]', 'Data
head: {text:
  ['tspan',
    ['tspan', {class:'error h3'}, 'Read transfer '],
  ]
},
config: { hscale: 2 },
}
```

How to test

Use SPI1 Master peripheral in RP2040 to start communication on SPI interface towards this design. Remember to configure the SPI mode using the switches in DIP switch (if you'd like to have CPOL=1 and CPHA=1). Alternatively, don't use the DIP switches and use the RP2040 GPIOs to configure the SPI mode in the desired mode.

Example code to initialize SPI in REPL:

```
spi_miso = tt.pins.pin_uio3
spi_cs = tt.pins.pin_uio4
spi_clk = tt.pins.pin_uio5
spi_mosi = tt.pins.pin_uio6
spi_miso.init(spi_miso.IN, spi_miso.PULL_DOWN)
spi_cs.init(spi_cs.OUT)
spi_clk.init(spi_clk.OUT)
spi_mosi.init(spi_mosi.OUT)
spi = machine.SoftSPI(baudrate=10000, polarity=0, phase=0, bits=8, firstb
spi_cs(1)
```

Example code to Write to Addres[0] Data 0xA5:

```
spi_cs(0); spi.write(b'\x80\xa5'); spi_cs(1)
```

Example code to Read from Addres[12]:

```
spi_cs(0); spi.write(b'\x0C'); spi.read(1); spi_cs(1)
```

External hardware

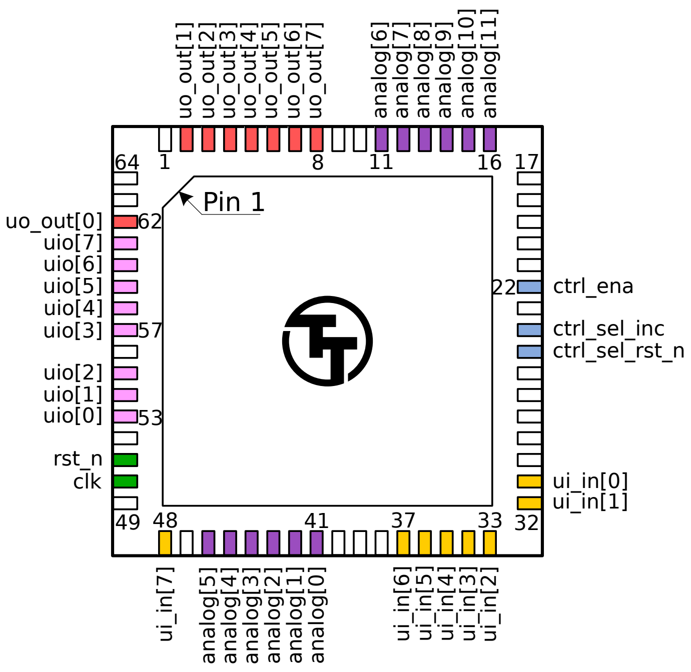
Not required.

Pinout

#	Input	Output	Bidirectional
0	cpol	decod_reg[0]	
1	cpha	decod_reg[1]	
2		decod_reg[2]	
3		decod_reg[3]	spi_miso
4		decod_reg[4]	spi_cs_n
5		decod_reg[5]	spi_clk
6		decod_reg[6]	spi_mosi
7		decod_reg[7]	

Pinout

The chip is packaged in a 64-pin QFN package. The pinout is shown below.



Bottom View

Figure 47: Pinout

Note: you will receive the chip mounted on a breakout board. The pinout is provided for advanced users, as most users will not need to solder the chip directly.

The Tiny Tapeout Multiplexer

Overview

The Tiny Tapeout Multiplexer distributes a single set of user IOs to multiple user designs. It is the backbone of the Tiny Tapeout chip.

It has the following features:

- 10 dedicated inputs
- 8 dedicated outputs
- 8 bidirectional IOs
- Supports up to 512 user designs (32 mux units, each with up to 16 designs)
- Designs can have different sizes. The basic unit is a called a tile, and each design can occupy up to 16 tiles.

Operation

The multiplexer consists of three main units:

1. The controller - used to set the address of the active design
2. The spine - a bus that connects the controller with all the mux units
3. Mux units - connect the spine to individual user designs

The Controller

The mux controller has 3 inputs lines:

Input	Description
`ena`	Sent as-is (buffered) to the downstream mux units
`sel_rst_n`	Resets the internal address counter to 0 (active low)
`sel_inc`	Increments the internal address counter by 1

It outputs the address of the currently selected design on the `si_sel` port of the spine (see below).

For instance, to select the design at address 12, you need to pulse `sel_rst_n` low, and then pulse `sel_inc` 12 times:

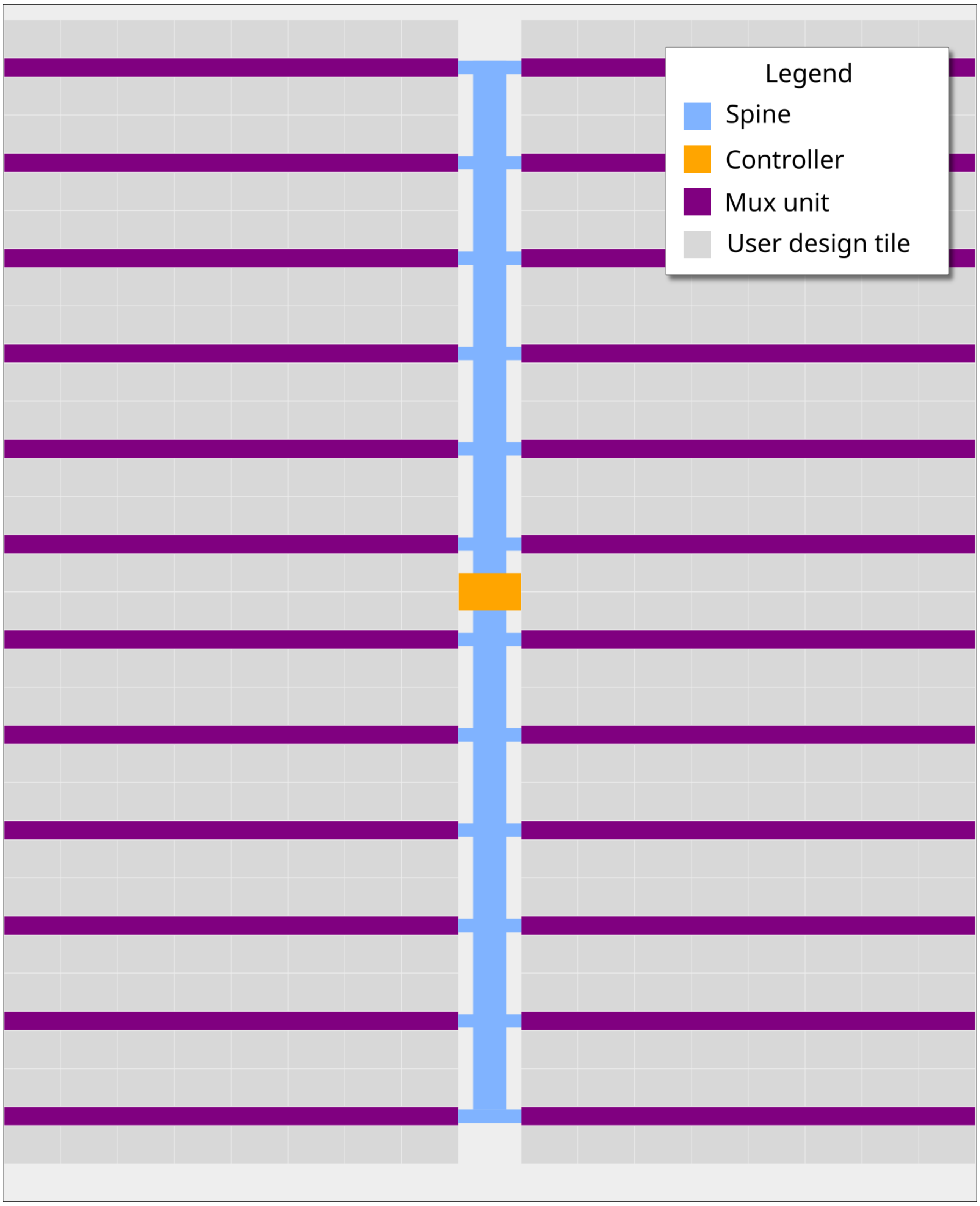


Figure 48: Mux Diagram

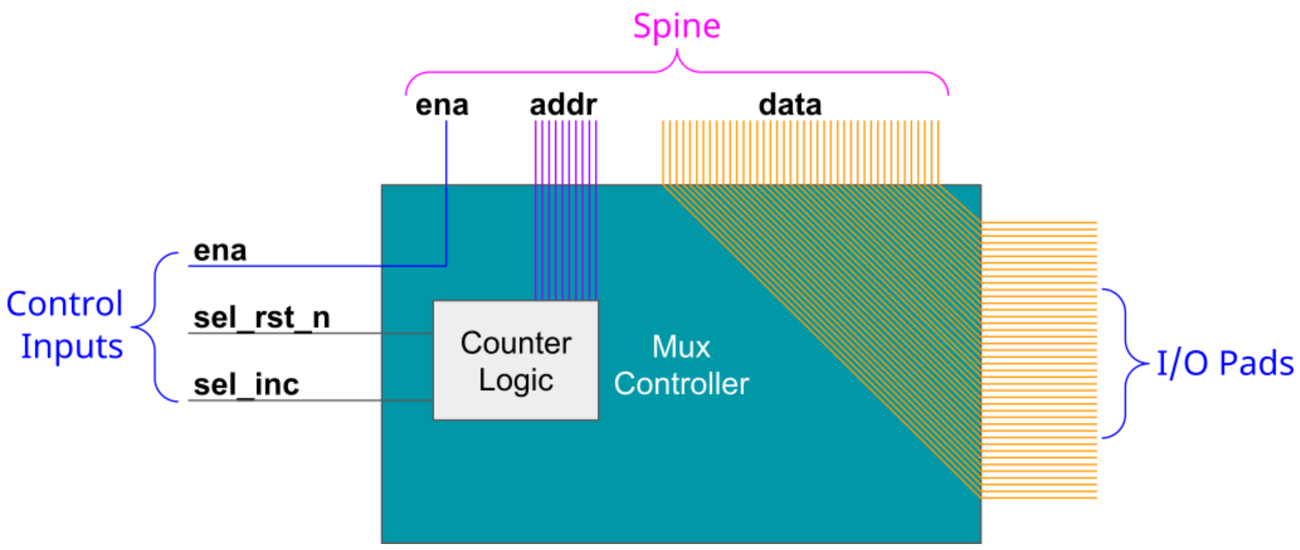


Figure 49: Mux Controller Diagram

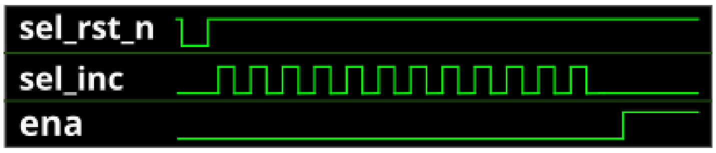


Figure 50: Mux signals for activating the design at address 12

Internally, the controller is just a chain of 10 D flip-flops. The `sel_inc` signal is connected to the clock of the first flip-flop, and the output of each flip-flop is connected to the clock of the next flip-flop. The `sel_rst_n` signal is connected to the reset of all flip-flops.

The following Wokwi projects demonstrates this setup: https://wokwi.com/projects/36434780766 It contains an Arduino Nano that decodes the currently selected mux address and displays it on a 7-segment display. Click on the button labeled `RST_N` to reset the counter, and click on the button labeled `INC` to increment the counter.

The Spine

The controller and all the muxes are connected together through the spine. The spine has the following signals going on it:

From controller to mux:

- `si_ena` - the `ena` input
- `si_sel` - selected design address (10 bits)
- `ui_in` - user clock, user `rst_n`, user inputs (10 bits)
- `uio_in` - bidirectional I/O inputs (8 bits)

From mux to controller:

- `uo_out` - User outputs (8 bits)
- `uio_oe` - Bidirectional I/O output enable (8 bits)
- `uio_out` - Bidirectional I/O outputs (8 bits)

The only signal which is actually generated by the controller is `si_sel` (using `sel_rst_n` and `sel_inc`, as explained above). The other signals are just going through from/to the chip IO pads.

The Multiplexer (The Mux)

Each mux branch is connected to up to 16 designs. It also has 5 bits of hard-coded address (each unit gets assigned a different address, based on its position on the die).

The mux implements the following logic:

If `si_ena` is 1, and `si_sel` matches the mux address, we know the mux is active. Then, it activates the specific user design port that matches the remaining bits of `si_sel`.

For the active design:

- `clk`, `rst_n`, `ui_in`, `uio_in` are connected to the respective pins coming from the spine (through a buffer)
- `uo_out`, `uio_oe`, `uio_out` are connected to the respective pins going out to the spine (through a tristate buffer)

For all others, inactive designs (including all designs in inactive muxes):

- `clk`, `rst_n`, `ui_in`, `uio_in` are all tied to zero
- `uo_out`, `uio_oe`, `uio_out` are disconnected from the spine (the tristate buffer output enable is disabled)

Pinout

mprj_io pin	Function	Signal	QFN64 pin
0	Input	ui_in[0]	31
1	Input	ui_in[1]	32
2	Input	ui_in[2]	33
3	Input	ui_in[3]	34
4	Input	ui_in[4]	35

mprj_io pin	Function	Signal	QFN64 pin
5	Input	ui_in[5]	36
6	Input	ui_in[6]	37
7	Analog	analog[0]	41
8	Analog	analog[1]	42
9	Analog	analog[2]	43
10	Analog	analog[3]	44
11	Analog	analog[4]	45
12	Analog	analog[5]	46
13	Input	ui_in[7]	48
14	Input	clk †	50
15	Input	rst_n †	51
16	Bidirectional	uio[0]	53
17	Bidirectional	uio[1]	54
18	Bidirectional	uio[2]	55
19	Bidirectional	uio[3]	57
20	Bidirectional	uio[4]	58
21	Bidirectional	uio[5]	59
22	Bidirectional	uio[6]	60
23	Bidirectional	uio[7]	61
24	Output	uo_out[0]	62
25	Output	uo_out[1]	2
26	Output	uo_out[2]	3
27	Output	uo_out[3]	4
28	Output	uo_out[4]	5
29	Output	uo_out[5]	6
30	Output	uo_out[6]	7
31	Output	uo_out[7]	8
32	Analog	analog[6]	11
33	Analog	analog[7]	12
34	Analog	analog[8]	13
35	Analog	analog[9]	14
36	Analog	analog[10]	15
37	Analog	analog[11]	16
38	Mux Control	ctrl_ena	22
39	Mux Control	ctrl_sel_inc	24
40	Mux Control	ctrl_sel_rst_n	25
41	Reserved	(none)	26
42	Reserved	(none)	27
43	Reserved	(none)	28

† Internally, there's no difference between `clk`, `rst_n`, and `ui_in` pins. They are all just bits in the `pad_ui_in` bus. However, we use different names to make it easier to understand the purpose of each signal.

Sponsored by

efabless.com

Team

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

- Uri Shaked for wokwi development and lots more
- Patrick Deegan for PCBs, software, documentation and lots more
- Sylvain Munaut for help with scan chain improvements
- Mike Thompson and Mitch Bailey for verification expertise
- Tim Edwards and Harald Pretl for ASIC 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
- Jeff and the Efabless Team 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