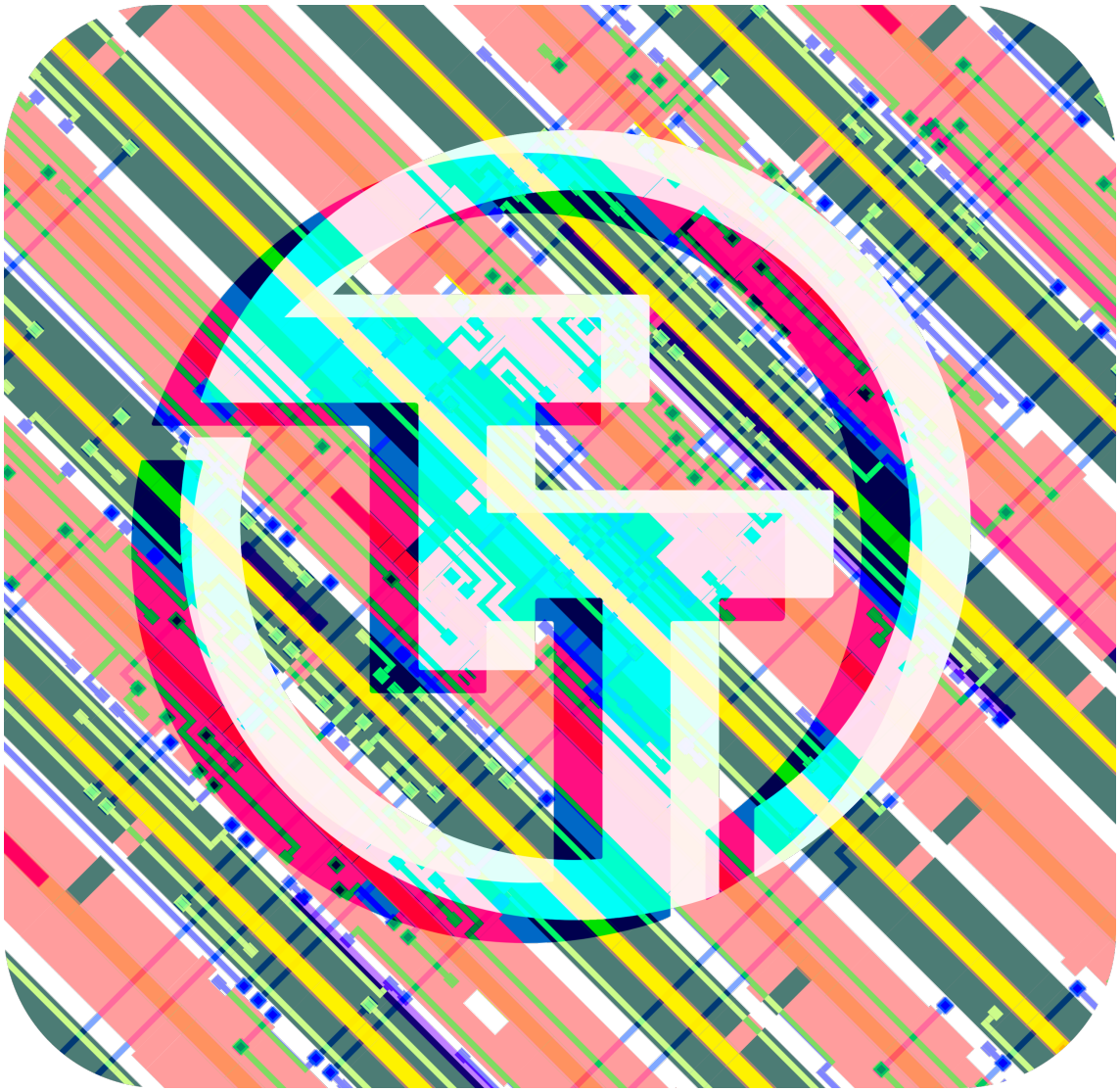


Tiny Tapeout IHP 0p2 Datasheet

Project Repository

https://github.com/TinyTapeout/tinytapeout-ihp-0p2

November 12, 2024

Contents

Chip map

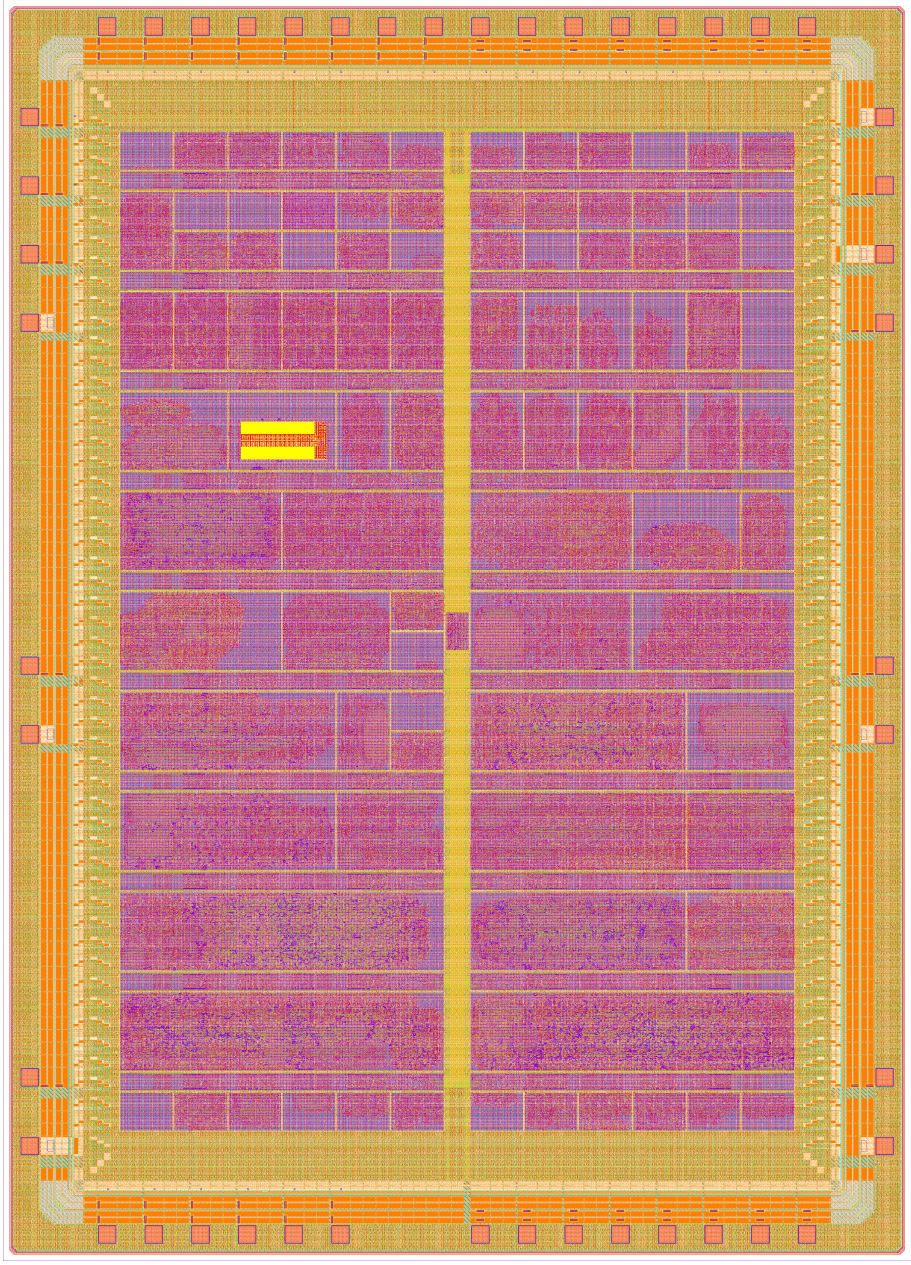


Figure 1: GDS render

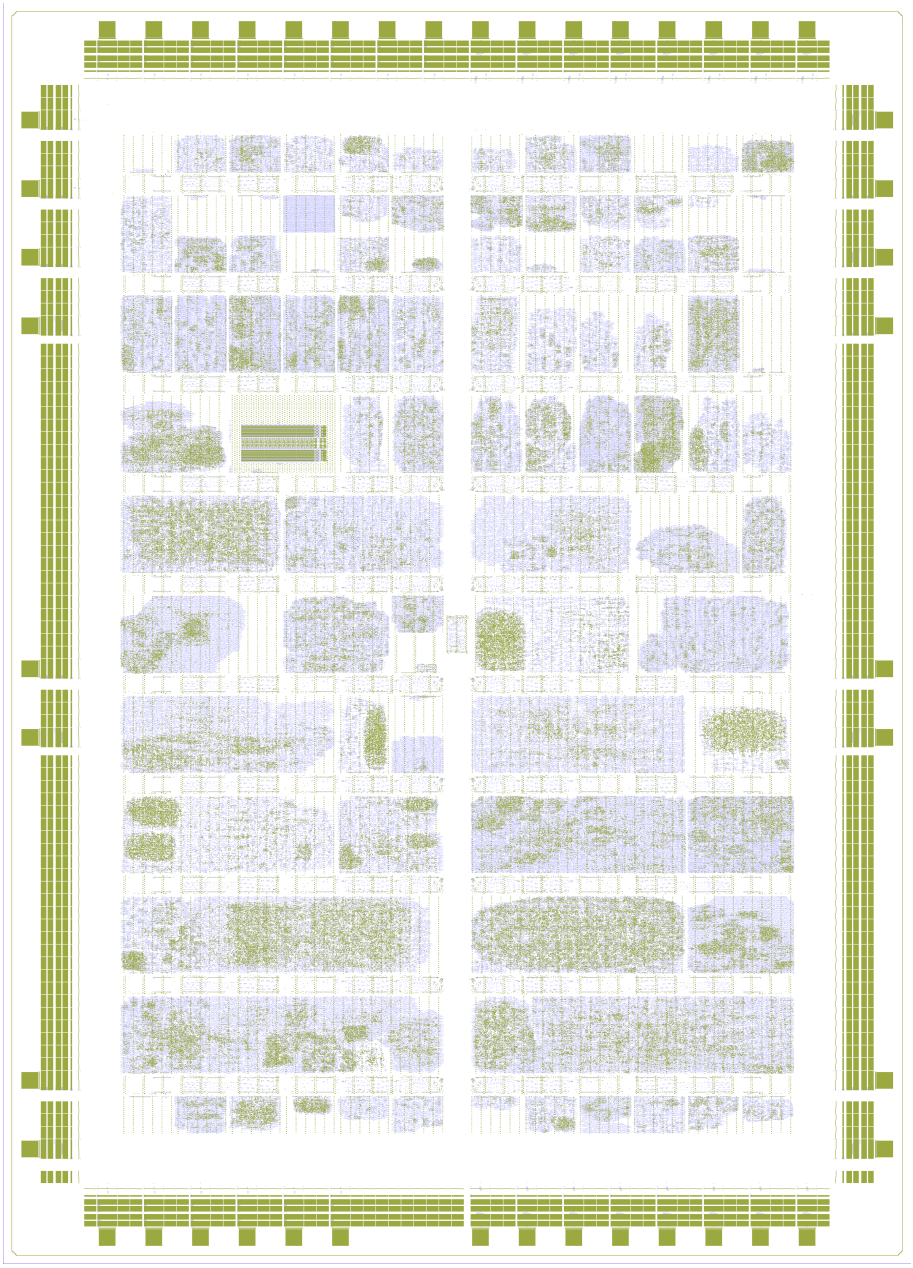


Figure 2: Logic density (local interconnect layer)

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: `&quot;TT\xFA\xBB&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	addr1	data1	
2	addr2	data2	
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 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	uo_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 (`uo_out`).
2. Set `rst_n` high and `sel` low and observe that the bidirectional pins (`uio_in`) are output on the output pins (`uo_out`).
3. Set `sel` high and observe that the counter is output on both the output pins (`uo_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_a1	output1 / counter1	in_b1 / counter1
2	in_a2	output2 / counter2	in_b2 / counter2
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]

VGA Mandelbrot [4]

- Author: Mike Bell
- Description: Mandelbrot on VGA, racing the beam
- GitHub repository
- HDL project
- Mux address: 4
- Extra docs
- Clock: 100000000 Hz

How it works

The Mandelbrot fractal is computed "racing the beam" and displayed through the TinyVGA Pmod.

One iteration of the computation is done every clock cycle, and a maximum iteration depth of 16 iterations is used. The design is clocked at 100MHz, allowing four clock cycles per 25MHz pixel clock. This means one value is computed every 4 pixels, giving a result like this:

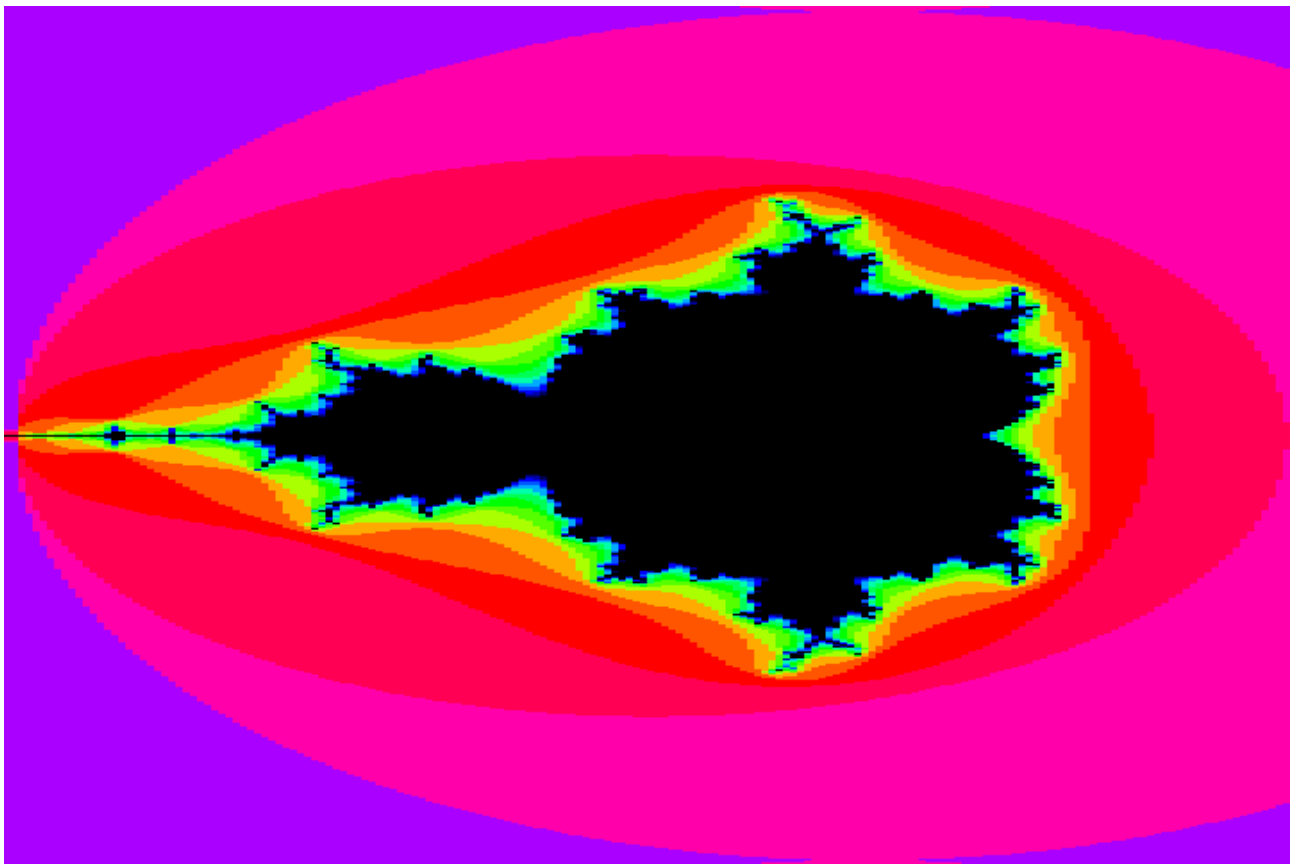


Figure 3: The Mandelbrot set

The computation uses 16-bit fixed point arithmetic. The multiplications are approximated to save area, giving a possible error in the least significant bit. This gives at least 14-bit accuracy on each iteration.

The output image is at a 720x480 resolution (180x480 Mandelbrot pixels).

How to test

Provide a 100MHz clock.

The image position and zoom can be configured using the input and bidir pins.

in[2:0] control the configuration to set, and {io[7:0], in[7:3]} specify a signed value when setting a register.

These values should only be updated during vsync.

Ctrl	Value
0	Enable demo mode (Zooms in and out repeatedly)
1	Set X coordinate for top-left of screen to value / 2^10
2	Set Y coordinate for top-left of screen to value / 2^11
3	No action
4	Set X increment per column to value[9:0] / 2^13
5	Set Y increment per column to value[9:0] / 2^13
6	Set X increment per row to value[7:0] / 2^13
7	Set Y increment per row to value[7:0] / 2^13

Note there are 180 columns and 480 rows displayed.

External hardware

Tiny VGA Pmod in the output socket.

Pinout

#	Input	Output	Bidirectional
0	Ctrl 0	R1	Input 5
1	Ctrl 1	G1	Input 6
2	Ctrl 2	B1	Input 7
3	Input 0	vsync	Input 8

#	Input	Output	Bidirectional
4	Input 1	R[0]	Input 9
5	Input 2	G[0]	Input 10
6	Input 3	B[0]	Input 11
7	Input 4	hsync	Input 12

FazyRV-ExoTiny [10]

- Author: Meinhard Kissich
- Description: A minimal SoC based on FazyRV that uses external QSPI ROM and RAM.
- GitHub repository
- HDL project
- Mux address: 10
- Extra docs
- Clock: 50000000 Hz

How it works

This TinyTapeout implements a System-on-Chip (SoC) design based on the FazyRV RISC-V core. Documentation on the SoC can be found in github.com/meiniKi/FazyRV-ExoTiny. For details on the FazyRV core, please refer to github.com/meiniKi/FazyRV.

Features

- Instantiates FazyRV with a chunk size of 2 bits.
- Uses external instruction memory (QSPI ROM) and external data memory (QSPI RAM).
- Provides 6 memory-mapped general-purpose outputs and 7 inputs.
- Provides an SPI peripheral with programmable CPOL and a buffer of up to 4 bytes.

Pinout Overview The overview shows the pinout for the TinyTapeout Demo PCB. A detailed description of the pins is given below.

Block Diagram The block diagram outlines the on-chip peripherals and related addresses.

How to test

Once the design is enabled and released from reset, it first enables Quad Mode in the RAM. The Wishbone accesses are converted into QSPI transfers to exchange data. The first read from ROM (boot address: `0x00000000`) enabled Continuous Mode to reduce the latency. To get started, you can flash the demo firmware in `FazyRV-ExoTiny/demo`. See the repo for more information.

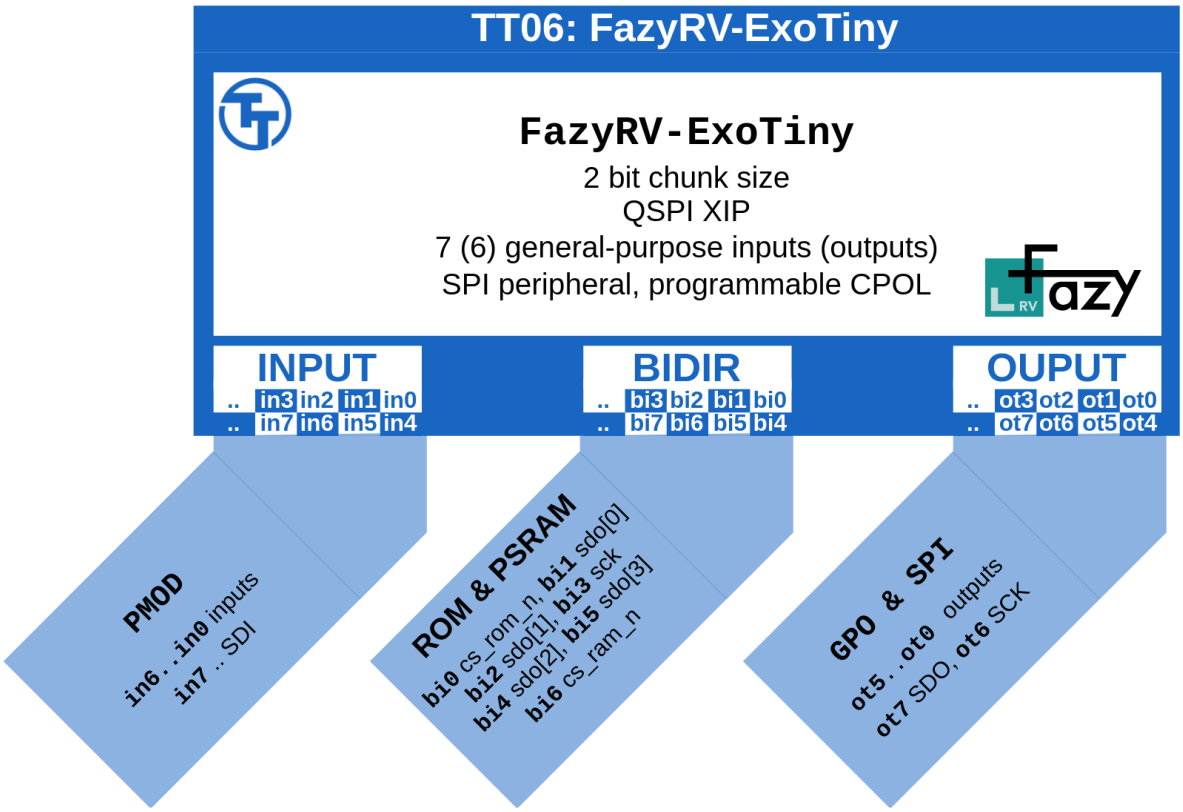


Figure 4: Pinout overview

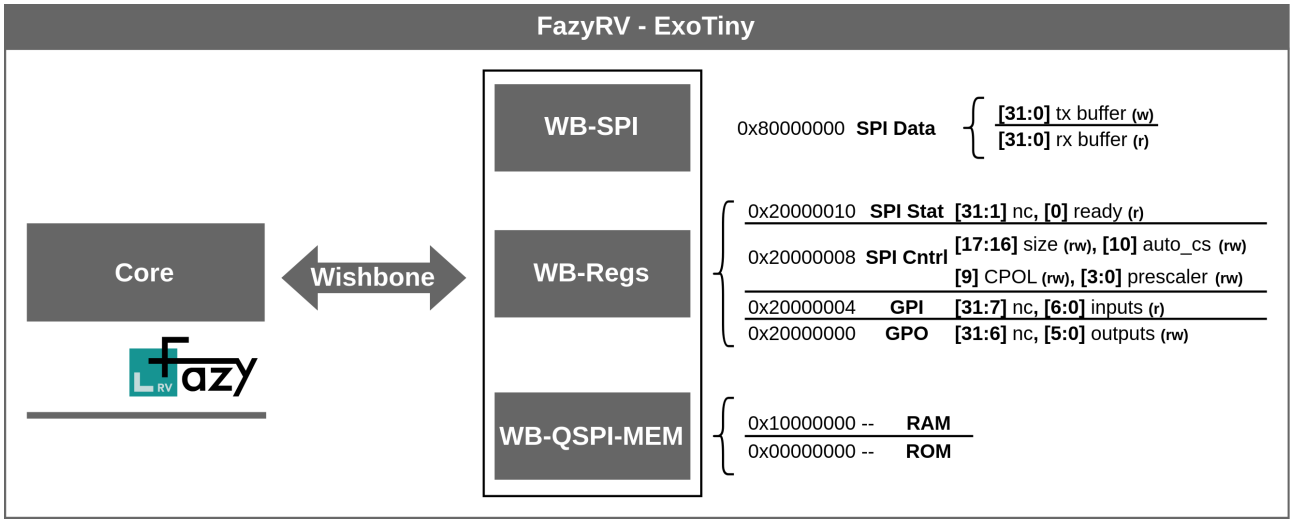


Figure 5: Block diagram

Important: `rst_n` is not synchronized. Make sure it is released sufficient hold time after the rising clock edge and sufficient setup time before the falling edge. Do not release reset while `clk` is low. The design appears to be on the edge of implementability. An additional dff breaks convergence.

External hardware

- QSPI ROM: W25Q128JV or compatible
- QSPI RAM: APS6404L-3SQR or compatible

The design uses external ROM (Flash) and external RAM. All bus accesses in these regions are converted to QSPI transfers to read data from the ROM or to read/write data from/to the RAM, respectively. Alternatively, you can synthesize a model in an FPGA and attach it to the BIDIR PMOD header.

Pinout

#	Input	Output	Bidirectional
0	General purpose input (GPI) 0.	General purpose output (GPO) 0.	QSPI ROM chip select (low active).
1	General purpose input (GPI) 1.	General purpose output (GPO) 1.	QSPI ROM/RAM SDO[0].
2	General purpose input (GPI) 2.	General purpose output (GPO) 2.	QSPI ROM/RAM SDO1.
3	General purpose input (GPI) 3.	General purpose output (GPO) 3.	QSPI ROM/RAM SCK.
4	General purpose input (GPI) 4.	General purpose output (GPO) 4.	QSPI ROM/RAM SDO2.

#	Input	Output	Bidirectional
5	General purpose input (GPI) 5.	General purpose output (GPO) 5.	QSPI ROM/RAM SDO[3].
6	General purpose input (GPI) 6.	(User) SPI SCK.	QSPI RAM chip select (low active).
7	(User) SPI SDI.	(User) SPI SDO.	NC.

8 bit RSA encryption [32]

- Author: Caio Alonso da Costa
- Description: 8 bit RSA encryption coprocessor with SPI interface
- GitHub repository
- HDL project
- Mux address: 32
- Extra docs
- Clock: 50000000 Hz

How it works

This project consists of an 8-bit RSA verilog design that implements the RSA (https://en.wikipedia.org/wiki/RSA_(cryptosystem)) encryption/decryption scheme with an 8-bit private/public key size.

The design implements modular exponentiation (https://en.wikipedia.org/wiki/Modular_exponen through a series of Montgomery modular multiplication (https://en.wikipedia.org/wiki/Montgom to encrypt/decrypt a message using an 8-bit key.

Due to I/O constraints, a SPI slave peripheral has been created to load/read data into/from the design.

The SPI Slave peripheral implementation supports all 4 SPI mode of operations (CPOL is configurable through ui2 and CPHA is configurable through ui[3]), 8 Configurable (Read/Write) 8-bit registers and 8 Status (Read only) 8-bit registers.

The RP2040 SPI1 peripheral shall be used to communicate with the RSA core. Configure RP2040 SPI1 peripheral to GPIOs 24 to 27.

SPI Limitations:

- 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]
1	Configurable Read/Write register 1 - bit1 Stop, bit[0] Start - Rising edge detector to trigger encryption/decryption
2	Configurable Read/Write register 2 - Plain text [7:0]
3	Configurable Read/Write register [3] - E [7:0]
4	Configurable Read/Write register [4] - M [7:0]
5	Configurable Read/Write register [5] - Montgomery Constant [7:0]
6	Configurable Read/Write register [6]
7	Configurable Read/Write register [7] - Spare [7:0] - Connected to 7-segment Display
8	Status Read Only register [0] - bit[0] IRQ - Encryption/decryption completed
9	Status Read Only register 1 - Fixed data 8'hC4
10	Status Read Only register 2 - Fixed data 8'h10
11	Status Read Only register [3] - Fixed data 8'hDE
12	Status Read Only register [4] - Fixed data 8'hAD
13	Status Read Only register [5] - Fixed data 8'h00
14	Status Read Only register [6] - Encrypted/Decrypted data [7:0]
15	Status Read Only register [7] - Fixed data 8'hFF

Connection

RP2040 SPI Master <–SPI–> SPI_WRAPPER <–regaccess–> User logic (RSA)

- 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    |
```

- 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    |
```

How to test

Key generation example:

1. Choose two large prime numbers p and q : p = 7, q = 13
2. Compute n = p * q : n = 91
3. Compute Euller totient function (n) = (p - 1) * (q - 1) : (n) = 72
4. Choose an integer e such that 1 < e < (n) and gcd(e, (n)) = 1: e = 11
5. Determine d as d e^(−1) (mod (n)); that is, d is the modular multiplicative inverse of e modulo (n) : d = 59

Private key {e, n} = {11, 91}

Public key {d, n} = {59, 91}

The plain text is limited to a number in the interval [0:91[, as per this example. Since the design uses the Montgomery mutiplication, a Montgomery Constant shall be used to map the plain text into the Montgomery integer domain.

6. Compute Montgomery constant (fixed value that depends only on the value of p and q and the max-key lenght of the RSA core implementation).

Const = (2 ** (2 * hwbits)) mod n, where hwbits = (8 (RSA max key-lenght core support) + 2).

Const = (2 ** (2 * (8+2))) mod 91 = 74

Now, use SPI Master peripheral in RP2040 to start communication on SPI interface towards this design. Remember to configure the SPI mode in the RP2040 accordingly.

Steps for start an/a encryption/decryption process:

1. Write any value between 0 and n-1 to the configurable Read/Write register 2 - Plain text [7:0]: Value suggestes: 12
2. Write to configurable Read/Write register [3] the value of e: 11
3. Write to configurable Read/Write register [4] the value of n: 91
4. Write to configurable Read/Write register [5] the value of const: 74
5. Write to configurable Read/Write register 1 the value 1 - (Trigger the start encryption command).
6. Wait for rising edge of the IRQ output.
7. Read the Status Read Only register [6] - Encrypted data. Valeu expected: 38.

12 $\hat{}$ 11 mod 91 = 743008370688 mod 91 = 38

https://github.com/calonso88/tt09_rsa/blob/main/test/test.py implements a self-checking test that verify the encrypted data produced by the RSA core against the predicted values produced locally on the test. The test randomize the elements for key generation and the plain text. All derived values needed for the encryption/decryption are calculated locally in the test through helper functions.

External hardware

Not required.

Pinout

#	Input	Output	Bidirectional
0	gpio_start	spare[0]	irq
1	gpio_stop	spare1	ui[4]
2	cpol	spare2	ui[5]
3	cpha	spare[3]	spi_miso
4		spare[4]	spi_cs_n
5		spare[5]	spi_clk
6		spare[6]	spi_mosi
7		spare[7]	ui[7]

Retro Console [37]

- Author: Toivo Henningsson
- Description: 8½ bit retro console with sprite and tile graphics + synth
- GitHub repository
- HDL project
- Mux address: 37
- Extra docs
- Clock: 50350000 Hz

Overview

AnemoneGrafx-8 is a retro console containing

- a PPU for VGA graphics output
- an analog emulation polysynth for sound output

The console is designed to work together with the RP2040 microcontroller on the Tiny Tapeout 06 Demo Board, the RP2040 providing

- RAM emulation,
- connections to the outside world for the console (except VGA output),
- the CPU to drive the console.

Features:

- PPU:
  - 320x240 @60 fps VGA output (actually 640x480 @60 fps VGA)
    - Some lower resolutions are also supported, useful if the design can not be clocked at the target 50.35 MHz
  - 16 color palette, choosing from 256 possible colors
  - Two independently scrolling tile planes
    - 8x8 pixel tiles
    - color mode selectable per tile:
      - 2 bits per pixel, using one of 15 subpalettes per tile
      - 4 bits per pixel, halved horizontal resolution
  - 64 simultaneous sprites (more can be displayed at once with some Copper tricks)
    - mode selectable per sprite:
      - 16x8, 2 bits per pixel using one of 15 subpalettes per sprite

            - 8x8, 4 bits per pixel
        * up to 4 sprites can be loaded and overlapping at the same pixel
            - more sprites can be visible on same scan line as long as they are not too cramped together
    - Simple Copper-like function for register control synchronized to pixel timing
        * write PPU registers
        * wait for x/y coordinate

- AnemoneSynth:
    - 16 bit 96 kHz output
    - 4 voices, each with
        * Two oscillators
            - sweepable frequency
            - noise option
        * Three waveform generators with 8 waveforms: sawtooth/triangle/2 bit sawtooth/2 bit triangle/square wave/pulse wave with 37.5% / 25% / 12.5% duty cycle
        * 2nd order low pass filter
            - sweepable volume, cutoff frequency, and resonance

The console is designed to be clocked at 50.35 MHz, twice the pixel clock of 25.175 MHz used for VGAmode 640x480 @60 fps. (The frequency does not have to be terribly precise though, and there are ways to clock the console considerably slower and still get a useful output.)

Contents:

Design rationale

The design target was

- PPU with 2 bpp graphics, with

  - VGA output at 640x480 @60 fps, doubled from PPU output at 320x240 @60 fps,
  - 2 planes of 8x8 pixel tiles,
  - at least 8 sprites per scan line.
- Four voice analog emulation synthesizer with each voice in the style of the monosynth https://github.com/toivoh/tt05-synth.

Design considerations:

- On chip memory takes a lot of area, maybe 1 tile per 64 bytes
  - 8 kB of video RAM for the PPU would be infeasible on-chip
  - 192+80 bits per voice would need a lot of space
- Solution: Use the RP2040 microcontroller (with 264 kB of RAM) on the Tiny Tapeout demo board as a RAM emulator
  - Store only what is necessary on chip, use higher bandwidth to reduce needed on-chip storage
    * Let PPU render the same scan line contents twice to double pixels vertically, instead of trying to do it once and store the results
- Limited number of pins ==> use serial interface(s) for RAM emulation
- PPU needs predictable memory access latency, but only reading
  - I was able to implement a RP2040 solution that uses PIO (programmable IO) and DMA but not CPU
    * Gives fixed read latency, RP2040 can add extra latency to reach suitable delay
  - PPU designed assuming data arrives just in time to calculate address 4 reads later
- Synth uses context switching to keep track of state of only one voice at a time
  - Needs some bandwidth, but low/fixed latency is less important
  - Use synchronous serial interface with start bit and TX/RX FIFOs to allow RP2040 CPU to service the interface
- Sizing:
  - PPU
    * 16 bit address ==> might as well read 16 bit data words
    * 8 bits/pixel read bandwidth needed for two tile planes with (16 bit) tile map and 2 bpp graphics

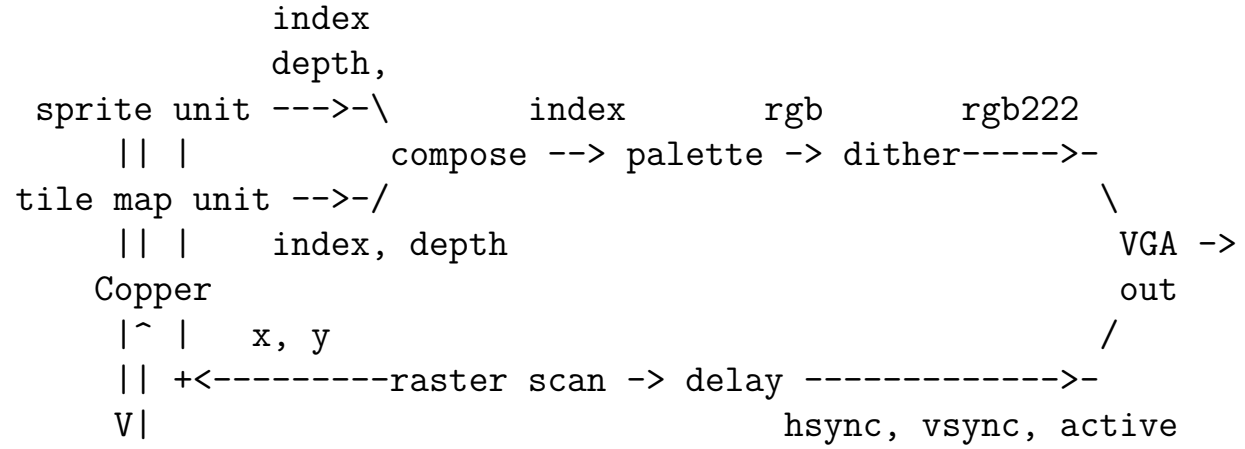
        * 16 bits/pixel read bandwidth gives space to read in new sprites during scan line
            · keeping track of only 4 sprites at a time to reduce on-chip storage
        * Overhead to keep track of each sprite means that it might as well use 32 bits of pixel data per scan line
        * Palette registers take a lot of space; limit to 16 palette colors
    - Synth:
        * Context switching cannot be overlapped with processing
        * 3x 20 bits of extra on-chip buffers allow producing four voice output samples between context switches, keeping down context switching time

How it works

The console consists of two parts:

- The PPU generates a stream of pixels that can be output as a VGA signals
    - based on tile graphics, map, and sprite data read from memory, and the contents of the palette registers.
- The synth generates a stream of samples by
    - context switching between voices at a rate of 96 kHz
        * producing four 96 kHz sample contributions from each voice in one go and adding to internal buffers
    - outputting each 96 kHz sample once it has received contributions from each voice

PPU



```
---> read unit --->
data           addr
```

The PPU is composed of a number of components, including:

- The *sprite unit* reads and buffers sprite data and sprite pixels, outputting color index and depth for the topmost sprite pixel
- The *tile map unit* reads and buffers tile map and tile pixel data, outputting color index and depth for the topmost tile map pixel
- The *Copper* reads an instruction stream of PPU register write, wait for x/y, and jump instructions, and updates PPU registers accordingly
- The *read unit* prioritizes read requests to graphics memory between the sprite unit, tile map unit, and Copper, and keeps track of the queue of reads that have been sent but the data has not yet been received

The PPU uses 4 clock cycles to generate each pixel, which is duplicated into two VGA pixels of two cycles each. (The two VGA pixels can be different due to dithering.)

Many of the registers and memories in the PPU are implemented as shift registers for compactness.

The read unit The read unit transmits a sequence of 16 bit addresses, and expects to recieve the corresponding 16 bit data word after a fixed delay. In this way, it can address a 128 kB address space. The delay is set so that the tile map unit can request tile map data, and receive it just in time to use it to request pixel data four pixels later. The read unit transmits 4 address bits per cycle through the `addr_out` pins, and recieves 4 data bits per cycle through the `data_in` pins, completing one 16 bit read every *serial cycle*, which corresponds to one pixel or four clock cycles.

The tile map unit has the highest priority, followed by the Copper, and finally the sprite unit, which is expected to have enough buffering to be able to wait for access. The tile map unit will only make accesses on every other serial cycle on average, and the Copper at most once every 6 serial cycles (or every 2 in fast mode), but they can both be disabled/paused for parts of the frame to give more bandwidth to the sprite unit.

The tile map unit The tile map unit handles two independently scrolling tile planes, each composed of 8x8 pixel tiles. The two planes get read priority on alternating serial cycles. Each plane sends a read every four serial cycles, alternating between reading tile map data and the corresponding pixel data for the scan line. The pixel data for each plane (16 bits) is stored in a shift register and gradually shifted out until the register can be quickly refilled. The sequencing of the refill operation is adjusted to provide one extra pixel of delay in case the pixel data arrives one pixel early (as it might have to do since the plane only gets read priority every other cycle).

The sprite unit The sprite unit is the most complex part of the PPU. It works with a list of 64 sprites, and has 4 sprite buffers that can hold sprite data for the current scan line. Once the final x coordinate of a sprite has passed, the corresponding sprite buffer can be reused to load a new sprite on the same scan line, as long as there is time to load the data before it should be displayed.

Sprite data is stored in memory in two structures:

- The sorted buffer
- The object attribute buffer

The sorted buffer must list all sprites to be displayed, sorted from left to right, with y coordinate and index. (16 bits/sprite) The object attribute buffer contains all other object attributes: coordinates (only 3 lowest bits of y needed), palette, graphic tile, etc. (32 bits/sprite)

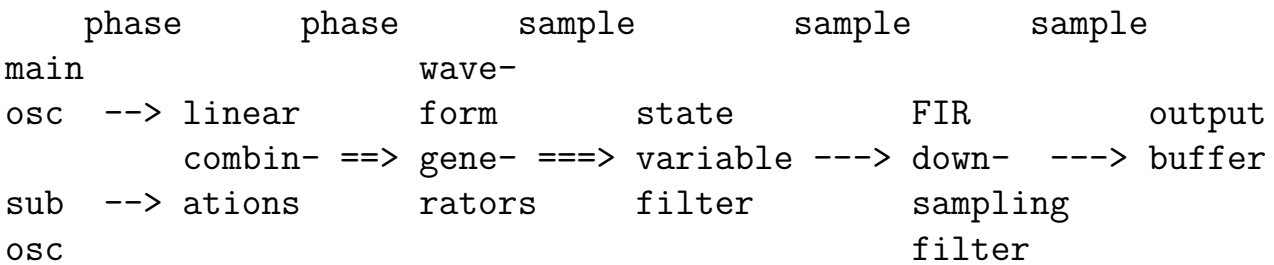
Sprite processing proceeds in three steps, each with its own buffers and head/tail pointers:

- Scan the sorted list to find sprites that overlap the current y coordinate (in order of increasing x value), store them into the id buffer (4 entries)
- Load object attributes for sprites in the id buffer, store in a sprite buffer and free the id buffer entry (4 sprite buffers)
- Load sprite pixels for sprites in the sprite buffers

Each succeeding step has higher priority to access memory, but will only be activated when the preceeding step can feed it with input data.

Pixel data for each sprite buffer is stored in a 32 bit shift register, and gradually shifted out as needed. If sprite pixels are loaded after the sprite should start to be displayed, the shift register will catch up as fast as it can before starting to provide pixels that can be displayed. This will cause the leftmost pixels of the sprite to disappear (or all of them, if too many sprites are crowded too close).

AnemoneSynth



AnemoneSynth does ANalog EMulation ONE voice at a time: it has 4 voices, but there is only memory for one voice at a time. The synth makes frequent context switches between the voices to be able to produce an output signal that contains the sum of the outputs.

Each voice contributes four 96 kHz time steps worth of data to the output buffer before being switched out for the next. As soon as all voices have contibuted to an output buffer entry, it is fed to the output, and the space is reused for a new entry. The voices are processed in a staggered fashion: First voice 0 contributes to output sample 0-3, finalizing output sample 0, then voice 1 contributes to output sample 1-4, finalizing output sample 1, etc...

The synth is nominally sampled at 3072 kHz to produce output samples at a rate of 96 kHz. The high sample rate is used so that the main oscillator can always produce an output that is exactly periodic with a period corresponding to the oscillator frequency, while maintaing good frequency resolution ($<$ 1.18 cents at up to 3 kHz). The 32x downsampling is done with a 96 tap FIR filter, so that each input sample contributes to three output samples. The FIR filter is optimized to minimize aliasing in the 0 - 20 kHz range after the 96 kHz output has been downsampled to 48 kHz with a good external antialiasing filter, assuming that the input is a sawtooth wave of 3 kHz or less.

To reduce computations, most of the samples that a voice would feed into the FIR filter are zeros. Usually, the voice steps eight 3072 kHz samples at a time, adding a single nonzero sample. Seen from this perspective, each voice is sampled at 384 kHz. This is just enough so that the state variable filter appears completely open when the cutoff frequency is set to the maximum.

To maintain frequency resolution, the main oscillator can periodically take a step of a single 3072 kHz sample, to pad out the period to the correct length. This results in advancing the state variable filter an eigth of the usual time step, and sending an output sample with an eigth of the usual amplitude through the FIR filter. The sub-oscillator does not have the same independent frequency resolution at the 3 highest octaves since it does not control the small steps, but is often used at a much lower frequency, and can often sync up harmonically with the main oscillator.

The state variable filter is implemented using the same ideas as described and used in https://github.com/toivoh/tt05-synth, using a shift-adder for the main computations. The shift-adder is also time shared with the FIR filter; each FIR coefficient is stored as a sum / difference of powers of two (the FIR table was optimized to keep down the number of such terms). The shift-adder saturates the result if it would overflow, which allows to overdrive the filter.

Each oscillator uses a phase of 10 bits, forming a sawtooth wave. A clock divider is used to get the desired octave. To get the desired period, the phase sometimes needs

to stay on the same value for two steps. To choose which steps, the phase value is bit reversed and compared to the mantissa of the oscillator's period value (the exponent controls the clock divider). This way, only a single additional bit is needed to keep track of the oscillator state beyond the current phase value.

Each time a voice is switched in, five sweep values are read from memory to decide if the two oscillator periods and 3 control periods for the state variable filters (see https://github.com/toivoh/tt05-synth) should be incremented or decremented. A similar approach is used as for the oscillator update above, with a clock divider for the exponent part of the sweep rate, and bit reversing the swept value to decide whether to take a small or a big step when one should be taken.

IO interfaces

AnemoneGrafx-8 has four IO interfaces:

- VGA output `uo` / `(R1, G1, B1, vsync, R0, G0, B0, hsync)`
- Read-only memory interface `(addr_out[3:0], data_in[3:0])` for the PPU
- TX/RX interface `(tx_out[1:0], rx_in[1:0])` for the synth, system control, and vblank events
  - `rx_in[1:0] = uio[7:6]` can be remapped to `rx_in_alt[1:0] = ui[5:4]` to free up `uio[7:6]` for use as outputs
- Additional video outputs `(Gm1_active_out, RBm1_pixelclk_out)`. Can output either
  - Additional lower RGB bits to avoid having to dither the VGA output
  - Active display signal and pixel clock, useful for e g HDMI output

The pins also have additional functions:

- `data_in[0]` is sampled into `cfg[0]` as long as `rst_n` is high, to choose the pin configuration:
  - `cfg[0] = 0`: `uio[7:6]` is used to input `rx_in[1:0]`,
  - `cfg[0] = 1`: `uio[7:6]` is used to output `{RBm1_pixelclk_out, Gm1_active_out}`, `rx_in_alt[1:0]` is used for RX input.
- When the PPU is in reset (due to `rst_n`=0 or `ppu_en`=0), the `addr_out` pins loop back the values from `data_in`, delayed by two register stages. This should be useful to set up the correct latency for the PPU RAM interface.

VGA output The VGA output follows the Tiny VGA pinout, giving two bits per channel. The PPU works with 8 bit color:

```
        Bits  2    1     0
Channel
red           | R1 | R0 | RBm1 |
green         | G1 | G0 | Gm1  |
blue          | B1 | B0 | RBm1 |
```

where the least significant bit it is identical between the red and blue channel. By default, dithering is used to reduce the output to 6 bit color (two bits per channel). Dithering can be disabled (using `dither_en=0` in the `ppu_ctrl` register), and the low order color bits `{RBm1, Gm1}` can be output on `{RBm1_pixelclk_out, Gm1_active_out}` (using `rgb332_out=1` in the `ppu_ctrl` register and `cfg[0]=1`).

The other output option for `(Gm1_active_out, RBm1_pixelclk_out)` is to output the `active` and `pixelclk` signals: (using `rgb332_out=0` in the `ppu_ctrl` register and `cfg[0]=1`)

- `active` is high when the current RGB output pixel is in the active display area.
- `pixelclk` has one period per VGA pixel (two clock cycles), and is high during the second clock cycle that the VGA pixel is valid.

Read-only memory interface The PPU uses the read-only memory interface to read video RAM. The interface handles only reads, but video RAM may be updated by means external to the console (and needs to, to make the output image change!).

Each read sends a 16 bit word address and receives the 16 bit word at that address as data, allowing the PPU to access 128 kB of data. A read occurs during one *serial cycle*, or 4 clock cycles. As soon as one serial cycle is finished, the next one begins.

The address `addr[15:0]` for one read is sent during the serial cycle in order of lowest bits to highest:

```
addr_out[3:0] = addr[3:0]   // cycle 0
addr_out[3:0] = addr[7:4]   // cycle 1
addr_out[3:0] = addr[11:8]  // cycle 2
addr_out[3:0] = addr[15:12] // cycle 3
```

The corresponding `data[15:0]` should be sent in the same order to `data_out[3:0]` with a specific delay that is approximately three serial cycles (TODO: describe the exact delay needed!). The `data_in` to `addr_out` loopback function has been provided to help calibrate the required data delay.

To respond correctly to reads requests, one must know when a serial cycle starts. This accomplished by an initial synchronization step:

- After reset, `addr_pins` start at zero.
- During the first serial cycle, a fixed address of `0x8421` is transmitted, and the corresponding data is discarded.

TX/RX interface The TX/RX interface is used to send a number of types messages and responses, mostly for use by the synth. It uses start bits to allow each side to initiate a message when appropriate; subsequent bits are sent on subsequent clock cycles. The `tx_out` and `rx_in` pins are expected to remain low when no messages are sent.

The `tx_out[1:0]` pins are used for messages from the console:

- a message is initiated with one cycle of `tx_out[1:0] = 1` (low bit set, high bit clear),
- 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 bits to highest.

The `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,
- during the following 8 cycles, a 16 bit payload is sent through `rx_in[1:0]`, from lowest bits to highest.

TX message types:

- 0: Context switch: Store payload into state vector, return the replaced state value with RX header=1, increment state pointer.
- 1: Sample out: Payload is the next output sample from the synth, 16 bits signed.
- 2: Read: Payload is address, return corresponding data with RX header=2.
- 3: Vblank event. Payload should be ignored.

RX message types:

- 1: Context switch response with data.

- 2: Read response with data.
- 3: Write register. Top byte of payload is register address, bottom is data value.

Available registers:

- 0: `sample_credits` (initial value 1)
- 1: `sbio_credits` (initial value 1)
- 2: `ppu_ctrl` (initial value `0b01011`)

The function of the registers is documented in the respective sections.

Using the PPU

The PPU is almost completely controlled through the contents of VRAM (video RAM). The Copper is restarted when a new frame begins, and starts to read instructions at address `0xfffe`. The Copper should be used to set up the PPU registers for the new frame before the active area starts, and is the only thing that can write PPU registers. The PPU registers in turn control the display of tile planes and sprites.

PPU registers There are 32 PPU registers, which control different aspects of the PPU's operation. Each register contains up to 9 bits. The registers are laid out as follows:

Address	Category	Bits 8	7	6	5	4	3	2	1	0
0 - 15	pal0-pal15	r2	r1	rb0	g2	g1	g0	b2	b1	X
16	scroll	scroll_x0								
17	.	X	scroll_y0							
18	.	scroll_x1								
19	.	X	scroll_y1							
20	copper_ctrl	cmp_x								
21	.	cmp_y								
22	.	jump_low								
23	.	jump_high								
24	base_addr	base_sorted								
25	.	base_oam								
26	.	base_map1				base_map0				X
27	.	X			b_tile_s	b_tile_p				X
28	gfxmode1	r_xe_hsync				r_x0_fp				
29	gfxmode2	vpol	hpol	vsel		r_x0_bp				
30	gfxmode3	xe_active								
31	displaymask	X		lspr	lpl1	lpl0	dspr	dpl1	dpl0	

where X means that the bit(s) in question are don't care.

Initial values:

- The gfxmode registers are initialized to 320x240 output (640x480 VGA output; pixels are always doubled in both directions before VGA output).
- The displaymask register is initialized to load and display sprites as well as both tile planes (initial value 0b111111).
- The other registers, except in the copper_ctrl category, need to be initialized after reset.

Each PPU register is described in the appropriate section:

- Palette (pal0-pal15)
- Tile planes (scroll, base_map0, base_map1, b_tile_p, lpl0, lpl1, dpl0, dpl1)
- Sprites (base_sorted, base_oam, b_tile_s, lspr, dspr)
- Copper (copper_ctrl)
- Graphics mode gfxmode1-gfxmode3)

Palette The PPU has a palette of 16 colors, each specified by 8 bits, which map to a 9 bit RGB333 color according to

```
        Bits  2    1     0
Channel
red           | r2 | r1 | rb0 |
green         | g2 | g1 | g0  |
blue          | b2 | b1 | rb0 |
```

where the least significant bit is shared between the red and blue channels. Each palette color is set by writing the corresponding palN register. The serial cycle when a palette color register is written, it will be used as the current output pixel if the raster sweep is inside the active display area.

Tile and sprite graphics typically can 2 or 4 bits per pixel. They have a 4 bit pal attribute that specifies the *subpalette*, or mapping from tile pixels to palette colors:

```
pal           Color 0     Color 1     Color 2     Color 3
value         (unless
               transparent)
    0              0           1           2           3
    4              4           5           6           7
```

```
 8              8              9             10             11
12             12             13             14             15

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

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

 3              8             12              1              5
 7              9             13              2              6
11             10             14              3              7

15             ---------- 16 color mode ------------
```

Color 0 is always transparent unless the `always_opaque` bit of the sprite/tile is set. If no tile or sprite covers a given pixel, palette color 0 is used the background color.

4 color and 16 color mode uses different graphic tile formats (see below). In 16 color mode, each pixel selects a palette directly, but index 0 is still transparent unless `always_opaque=1`.

Tile graphic format Tile plane and sprite graphics are both based on 16 byte graphic tiles, storing 8 rows of pixels with each row as a separate 16 bit word, from top to bottom:

- 2 bit/pixel tiles are 8x8 pixels
- 4 bit/pixel tiles are 4x8 pixels (strecthed to 8x8 pixels when used in tile planes)

The format of each line is

```
          15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
2 bpp    |  p7 |  p6 |  p5 |  p4 |  p3 |  p2 |  p1 |  p0 |
4 bpp    |     p3    |     p2    |     p1    |     p0    |
```

where `p0` is the leftmost pixel, then comes `p1`, etc.

Tile planes The PPU supports two independently scrolling tile planes, where plane 0 is in front of plane 1. Four `display_mask` bits control the behavior of the tile planes:

- When `dpl0` (`dpl1`) is cleared, plane 0 (1) is not displayed.
- When `lpl0` (`lpl1`) is cleared, no data for plane 0 (1) is loaded.

If a plane is not to be displayed, its `lplN` bit can be cleared to free up more read bandwidth for the sprites and Copper. The plane's `lplN` bit should be set at least 16 pixels before it should be displayed, to avoid visual artifacts.

The tile planes are drawn based on three VRAM regions with starting addresses controlled by PPU registers:

```
plane_tiles_base = b_tile_p  << 14
map0_base        = base_map0 << 12
map1_base        = base_map1 << 12
```

The `scroll_x` and `scroll_y` registers for each plane are added to the raster position of the current pixel to find the pixel position in the corresponding tile map to display. The raster position is x=128, y=255 for the bottom right corner of the screen, increasing to the right and decreasing going up.

The tile map for each plane is 64x64 tiles, and is stored row by row. Each map entry is 16 bits:

```
  15 - 12        11          10   -   0
| pal     | always_opaque | tile_index |
```

where the tile is read from word address

```
tile_addr = plane_tiles_base + (tile_index << 3)
```

and `pal` and `always_opaque` work as described in the Palette section.

Sprites Each sprite can be 16x8 pixels (4 color) or 8x8 pixels (16 color). The PPU supports up to 64 simultaneous sprites in OAM, but only 4 can overlap at the same time. More than 64 sprites can be displayed in a single frame by using the Copper to change base addresses mid frame.

Once a sprite is done for the scan line, the PPU can load a new sprite into the same slot, to display later on the same scan line, but it takes a number of pixels (partially depending on how much memory traffic is used by the tile planes and the Copper.) Five reads are needed to load a new sprite (1 for the sorted list, 2 for OAM, 2 for the pixels). More may be needed to skip through the sorted list, but the PPU can scan ahead to gather the next 4 sprite hits on the scan line. The pixel reads are dependent on the OAM reads, which are dependent on the sorted list reads. With both tile planes active (and the Copper inactive), a bandwidth of 8 bits/pixel is available to read in new sprites. With `5*16 = 80` bits/sprite, a new sprite can be loaded every 10 pixels on average (5 pixels if the tile planes are inactive).

Two `display_mask` bits control the behavior of the sprite display:

- When `dspr` is cleared, no sprites are displayed.
- When `lspr` is cleared, no data for sprites is loaded.

It will take some time after `lspr` is set before new sprites are completely loaded and can be displayed. Sprites start loading for a new scan line as soon as the active display part of the previous scan line is finished.

Sprites are drawn based on three VRAM regions with starting addresses controlled by PPU registers:

```
sprite_tiles_base = b_tile_s    << 14
sorted_base       = base_sorted <<  6
oam_base          = base_oam    <<  7
```

Sprites are described by two lists, each with 64 entries:

- The *sorted list* lists sprites sorted horizontally.
- *Object Attribute Memory* (OAM) defines most properties for the sprites.

To display sprites correctly, they must be listed in the sorted list in order of increasing x coordinate, starting from `sorted_base`. Each entry in the sorted list is 16 a bit word with contents

```
  15   14   13 - 8   7 - 0
| m1 | m0 |  index |   y   |
```

where

- m0 (m1) hides the sprite on even (odd) scan lines if it is set, (each output pixel is displayed on two VGA scan lines)
- index is the sprite's index in OAM,
- y is the sprite's y coordinate.

If there are less than 64 sprites to be displayed, the remaining sorted entries should be masked by setting m0 and m1, or moving the sprite to a y coordinate where it is not displayed. If there are more sprites than can be displayed in the same area, m0 can be set to mask some and m1 to mask others, showing them on alternating scan lines.

For each sprite, OAM contains two 16 bit words attr_y and attr_x, which define most of the sprite's properties. attr_y for sprite 0 is stored first, followed by attr_x, then attr_y for sprite 1, etc... The contents are

```
attr_y:  15 14   13   -   4   3   2 - 0
       |   X   | tile_index | X | ylsb3 |

attr_x:  15 - 12        11          10 - 9   8 - 0
       |   pal   | always_opaque |  depth |   x   |
```

where

- the sprite's graphics are fetched from the two consecutive graphic tiles starting at sprite_tiles_base + (tile_index << 4),
- ylsb3 is the lowest 3 bits of the sprite's y coordinate,
- pal and always_opaque work as described in the Palette section,
- depth specifies the sprite's depth relative to the tile planes,
- x is the sprite's x coordinate.

If several visible sprites overlap, the lowest numbered sprite with an opaque pixel wins. The depth value then decides whether the winning sprite is displayed in front of the tile planes:

- 0: In front of both tile planes.
- 1: Behind plane 0, in front of plane 1.
- 2: Behind both tile planes.
- 3: Not displayed.

A sprite with a `depth` value of 3 will block sprites with higher index from being displayed in the same location. If a sprite should not be displayed but does not need to block other sprites in this manner, omit it from the sorted list instead.

The sprite x coordinate value starts at 128 at the left side of the screen, and increases to the right. The sprite y cooridnate value starts at 255 at the bottom of the screen and decreases going upward.

Copper The Copper executes simple instructions, which can

- write to PPU registers,
- wait until a given raster position is reached,
- jump to continue Copper execution at a different VRAM location, or
- halt the Copper until the beginning of the next frame.

The Copper is restarted each time a new frame begins, just after the last active pixel of the previous frame has been displayed. It always starts at VRAM location `0xfffe`, with `fast_mode = 0`. Placing a jump instruction at `0xfffe`–`0xffff` allows to quickly switch between prepared lists of Copper instructions, and to choose where they should be placed in VRAM.

Each Copper instruction is 16 bits:

```
  15 - 7 |     6       5 - 0
|  data  | fast_mode |  addr |
```

where

- `data` is the data to be written to a PPU register,
- `fast_mode` enables the Copper to run 3 times as fast, but is incompatible with waiting and jumping,
- `addr` specifies the PPU register to be written (see PPU registers).

The Copper halts if it receives an instruction with `addr = 0xb111111`, otherwise it writes `data` to the PPU register given by `addr`, if one exists.

The `copper_ctrl` PPU registers have specific effects on the Copper:

Compare registers Writing a value to cmp_x or cmp_y causes the Copper to delay the next write until the current raster x/y position is $>=$ the specified compare value.

For each scan line, the raster position for x goes through the following phases in order

Phase	x raster positions		
front porch	24 + r_x0_fp	to	31
horizontal sync	32	to	32 + r_xe_hsync
back porch	96 + r_x0_bp	to	127
active	128	to	xe_active

yielding increasing raster positions for x.

The raster position for y counts as zero until the active region starts in the y direction. Then, the compare value is 512 - 2*screen_height + y where y is the number of scan lines since the start of the active region in the y direction.

Jumps Usually, the Copper loads instructions from consecutive addresses. A sequence of two instructions is needed to execute a jump:

- First, write the low byte of the jump address to jump_low.
- Then, write the high byte of the jump address to jump_high. The jump is executed.

There should be no writes to cmp_x or cmp_y between these two instructions, as the same register is used to store the compare value and the low byte of the jump address while waiting for the write to jump_high.

Fast mode Whenever an instruction arrives at the Copper, the value of fast_mode in the instruction overwrites the current value. When fast_mode = 0, the Copper does not start to read a new instruction until the previous one has finished. This allows waiting for compare values and jumping to work as intended. When fast_mode = 1, the Copper can send a new read every other serial cycle (unless blocked by reads from the tile planes, which have higher priority), queuing up several reads before the instruction data from the first one arrives. This can allow the Copper to work up to 3 times as fast, and works as intended as long as no writes are done to the copper_ctrl registers.

The fast_mode bit

- Should be set to zero

  - at least three instructions before a write to any of the `copper_ctrl` registers,
  - for instructions that follow a write to `cmp_x` or `cmp_y`.

- Can be set to one by an instruction that writes to `jump_high` (but not the other `copper_ctrl` registers) unless it needs to be zero due to any of the above.

Graphics mode registers The `gfxmode` registers control the timing of the VGA raster scan. The horizontal timing can be changed in fine grained steps, while the vertical timing supports 3 options.

The intention of the `gfxmode` registers is to support output in video modes

VGA mode	Frame rate	PPU output mode at full PPU clock rate
640x480	60 fps	320x240
640x400	70 fps	320x200
640x350	70 fps	320x175

These VGA modes are all based on a pixel clock of 25.175 MHz, which can be achieved if the console is clocked at twice the pixel clock, or 50.35 MHz. (VGA monitors should be quite tolerant of deviations around this frequency, 50.4 MHz should be fine and can be achieved with the RP2040 PLL.)

The intention is also to support reduced horizontal PPU resolution while generating a VGA signal according to one of the above VGA modes, in case the console has to be clocked at a lower frequency. This will lower the output frequency that can be achieved by the synth as well.

Vertical timing The `vsel` bits select between vertical timing options:

vsel	VGA lines	PPU output height		recommended polarity
0	480	240		vpol=1, hpol=1
1	64	32	test mode (not VGA)	-
2	400	200		vpol=0, hpol=1
3	350	175		vpol=1, hpol=0

The `hpol` and `vpol` bits control the horizontal and vertical sync polarity (0=positive, 1=negative). Original VGA monitors may use these to distinguish between modes; more modern monitors should be able to detect the mode from the timing.

Horizontal timing Possible horizontal timings include

PPU output width	VGA pixels /PPU pixel	PPU clock	gfxmode1	gfxmode2	gfxmode3
320	2	50.35 MHz	0x0178	0x0188	0x01bf
212	3	33.57 MHz	0x00f9	0x0190	0x0153
208	3	33.57 MHz	0x00f8	0x018d	0x014f
160	4	25.175 MHz	0x00bc	0x0194	0x011f

where the `vsel`, `hpol`, and `vpol` bits have been set to 480 line mode, but can be easily changed by updating the `gfxmode2` value. The 208 width mode is a tweak on the 212 width mode to fit a whole number of tiles (26) in the horizontal direction. These modes have been designed to stretch a PPU pixel horizontally into 2, 3, or 4 VGA pixels; other modes are possible with other settings.

The "PPU clock" column lists the recommended clock frequency to feed the console in order to achieve the 60 fps (`vsel=0`) or 70 fps (`vsel=2` or `vsel=3`). In practice, VGA monitors seem quite tolerant of timing variations, and might, e g, accept a 640x480 BGA signal at down to 2/3 of the expected clock rate.

The `gfxmode` registers control the horizontal parameters timing according to

```
active:      xe_active - 127  PPU pixels
front porch: 8  - r_x0_fp     PPU pixels
hsync:        1 + r_xe_hsync  PPU pixels
back porch:  32 - r_x0_bp     PPU pixels
```

where `xe_active` must be $>=$ 128.

The `ppu_ctrl` register The `ppu_ctrl` register controls some additional aspects of the PPU. It can be written through the TX/RX interface.

The contents are

```
       4           3           2        1     0
| rgb332_out | dither_en | vblank_int | X | ppu_en |
```

with initial value `0b01011`. Functions:

- The PPU is kept in reset as long as `ppu_en=0`.

- When `vblank_int=1`, the PPU sends a vblank message (TX header=3) on the TX channel whenever a frame is done.
- The `dither_en` bit controls dithering:
  - when `dither_en=1`, the PPU applies dithering to the output pixels,
  - when `dither_en=0`, `{R1, R0}`, `{G1, G0}`, `{B1, B0}` just contain the top 2 bits of each color channel.
- The `rgb332_out` bit controls what is output on the `Gm1_active_out` and `RBm1_pixelclk_out` pins, when they are configured as outputs:
  - when `rgb332_out=1`, the bottom bit of `G` and `RB` is output (combine with `dither_en=0` to get the whole `RGB332` output)
  - when `rgb332_out=0`, the pixel clock and active signal are output instead.

Using AnemoneSynth

AnemoneSynth has four identical voices, each with

- two oscillators (main and sub-),
- three waveform generators,
- a second order filter.

The synth is designed for an output sample rate of `output_freq` = 96 kHz (higher sample rates are used in intermediate steps), which should be achievable if the console is clocked at close to the target frequency of 50.35 MHz. The user can reduce `output_freq` by requesting output samples less frequently.

The hardware processes one voice at a time, and periodically performs a context switch through the TX/RX interface to write the state of the active voice out to RAM and read in the state of the next voice to make active. The voice state can be divided into dynamic state (updated by the synth) and parameters (not updated by the synth).

The periods of the two oscillators, as well as three control periods for the filter, are part of the dynamic state. These periods are continuously updated according to the voice's sweep parameters, which can specify a certain rate of rise or fall, or a replacement value. Sweep parameters are not stored in the voice state, but are read from RAM as needed to update the periods. Envelopes can be realized by changing sweep parameters over time. The behavior of a voice is controlled through its parameters and its sweeps.

Voice state The voice state consists of twelve 16 bit words, or 192 bits:

bit address	bit width	name	
0	1	delayed_s	
1	2	delayed_p	
3	3	fir_offset_low	
6	10	phase[0]	main oscillator phase
16	10	phase[1]	sub-oscillator phase
26	6	running_counter	
32	20	y	filter state (output)
52	20	v	filter state
72	14	float_period[0]	main oscillator period
86	14	float_period[1]	sub-oscillator period
100	10	mod[0]	control period 0
110	10	mod[1]	control period 1
120	10	mod[2]	control period 2
130	5	lfsr_extra	
135	1	ringmod_state	
136	13	wf_params[0]	waveform 0 parameters
149	13	wf_params[1]	waveform 1 parameters
162	13	wf_params[2]	waveform 2 parameters
175	13	voice_params	voice parameters
188	4	unused	

The parameter part of the state begins at `wf_params[0]`. There are three sets of waveform parameters `wf_params`, each consisting of 13 bits:

bit address	bit width	name	default
0	3	wf	
3	2	phase0_shl	0
5	2	phase1_shl	0
7	2	phase_comb	0/1/2 for waveform 0/1/2
9	2	wfvol	0
11	1	wfsign	0
12	1	ringmod	0

The default values should be seen as a suggestion of an initial point to start from when experimenting with parameters settings. There is no hardware mechanism to set these values as defaults.

The voice parameters `voice_params` also consist of 13 bits:

bit address	bit width	name	default
0	1	lfsr_en	0
1	2	filter_mode	0
3	3	bpf_en	0
6	1	hardsync	0
7	4	hardsync_phase	0
11	2	vol	0

Frequency representation Frequencies are represented by periods in a simple floating point format, with 4 bits for the octave and 10 or 6 bits for the mantissa:

```
{oct[3:0], mantissa[9:0]} = float_period[i]  // for oscillator periods
{oct[3:0], mantissa[5:0]} = mod[i]           // for control periods
```

The period value can be calculated as

```
osc_period[i] = (1024 + mantissa) << oct     // for oscillator periods
mod_period[i] =   (64 + mantissa) << oct     // for control periods
```

except that `oct=15` corresponds to an infinite period, or a frequecy of zero. The oscillator frequencies are given by

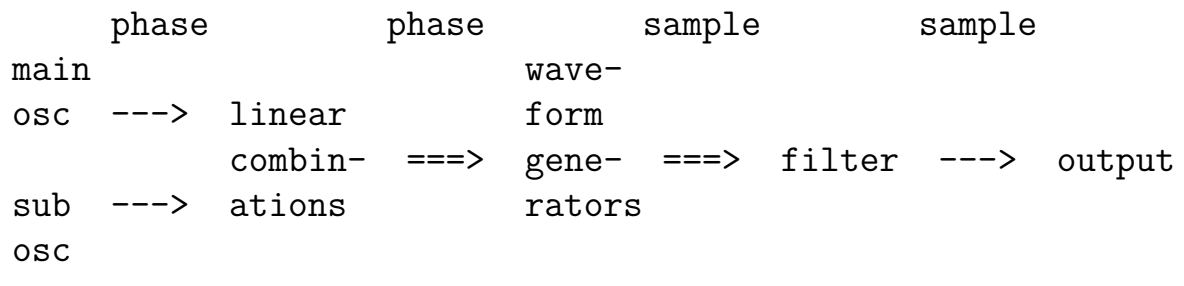
```
osc_freq[i] = output_freq * 32 / osc_period[i]
```

so at `output_freq = 96 kHz`, the highest achievable oscillator frequency is 3 kHz (and the lowest is a bit below 0.1 Hz). The control frequencies are given by

```
mod_freq[i] = output_freq * 256 / mod_period[i]
```

The floating point representation for the periods helps keep the same relative frequency resolution over all octaves. It also means that a linear sweep of the floating point period representation will sound very much like an exponential sweep of the frequency, which is similar to the linear-to-exponential conversion used by most analog synths.

Signal path



The signal path starts at the two oscillators, which feed 3 waveform generators. Each waveform generator can be fed with a different linear combination of oscillator phases. The waveforms are fed into the filter. Finally, the output of the filter is summed for all the voices to create the synth's output signal.

Oscillators The main and sub-oscillators are both sawtooth oscillators. When we talk about phase, it refers to such a sawtooth value, increasing at a constant rate over the period, and wrapping once per period. The sub-oscillator can produce noise instead by setting `lfsr_en=1`. (TODO: Describe noise frequency dependence on `osc_period[1]`.)

Each voice is nominally sampled at 32 `*` `output_freq`, with 32 subsamples per output sample. Most of the time, it advances by 8 subsamples at a time, but occasionally by a single subsample, which is used to improve the frequency resolution at the three highest octaves, and avoid aliasing. The choice of when to step by 8 subsamples and when to step by 1 is controlled by the main oscillator, which means that the sub-oscillator has less independent frequency resolution for the 3 highest octaves (1 bit less when its `oct=2`, 2 bits less when its `oct=1`, and 3 bits less when its `oct=0`). The sub-oscillator will often be at a much lower frequency than the main oscillator.

It is possible to combine the output of the oscillators in different ways to derive new frequencies, but if possible, the main oscillator's frequency should be set to the voice's intended pitch, (or the pitch divided by an integer that is as small as possible), to allow the synth's supersampling to produce the best results and to avoid aliasing artifacts, especially at high pitches. If the voice's output signal is periodic with the main oscillator's period, there should be very little aliasing artifacts. If the output waveform varies slowly when the voice output is chopped up into periods equal to the main oscillator period, there should still be little aliasing.

The sub-oscillator can be hard-synced to the main oscillator by setting `hardsync=1`. When enabled, the (10 bit) phase of the sub-oscillator resets to `hardsync_phase` `&lt;&lt;` 6 whenever the main oscillator completes a period.

Combining the oscillators The phase_comb, phase0_shl and phase1_shl parameters of each waveform specify how to calculate the waveform generator's input phase from the oscillator phases, with phase_comb selecting between four modes:

```
phase_comb    Waveform generator input phase
         0    (main << phase0_shl) + (sub << phase1_shl)
         1    (main << phase0_shl) - (sub << phase1_shl)
         2    (main << phase0_shl)
         3                          (sub << phase1_shl)
```

A good starting point is to set phase_comb to 0 for one waveform, 1 for one, and 2 for one, leaving the other waveform parameters the same. Combined with a sub-oscillator at around a 1/1000 of the main oscillator frequency, this creates a detuning effect. Higher frequency compared to the main oscillator gives more detuning.

Waveform generator The wf parameter selects between 8 waveforms:

```
wf    Waveform
 0    sawtooth wave
 1    sawtooth wave, 2 bit
 2    triangle wave
 3    triangle wave, 2 bit
 4    square wave
 5    pulse wave, 37.5% duty cycle
 6    pulse wave, 25%   duty cycle
 7    pulse wave, 12.5% duty cycle
```

All waveforms have a zero average level. The peak-to-peak amplitude of the pulse waves is half that of the other waveforms.

The waveform amplitude is multiplied by 2^-wfvol before feeding into the filter. If wfsign=1, it is inverted. If wfvol=3, wfsign=1, the waveform is silenced.

If ringmod=1, the waveform is inverted when the output of the previous waveform generator is negative (before the effects of wfvol, wfsign, and ringmod have been applied, waveform 2 is previous to waveform 0).

Filter The output from each waveform generator is fed into the filter. The filter_mode parameters selects the filter type:

filter_mode	Filter type
0	2nd order filter
1	2nd order filter, transposed
2	2nd order filter, two volumes, default damping
3	Two cascaded 1st order low pass filters

The meaning of the mod states depends on the filter mode:

filter_mode	mod_freq[0]	mod_freq[1]	mod_freq[2]
0	cutoff	fdamp	fvol
1	cutoff	fdamp	fvol
2	cutoff	fvol2	fvol
3	cutoff	cutoff2	fvol

(see Frequency representation for the definition of mod_freq).

The transposed filter mode 1 is expected to be a bit noisier than the default mode 0, and have somewhat different overdrive behavior. The bpf_en[i] parameter can be used in filter modes 1 and 3 to change the point where waveform i feeds into the filter:

- For filter_mode=1, bpf_en[i]=1 makes the filter behave as a band pass filter for that waveform.
- For filter_mode=3, bpf_en[i]=1 feeds the waveform straight into the second low pass filter.

The volume feeding into the filter is generally given by

```
gain = fvol / cutoff
```

but for filter_mode=3,

- fvol2 is used instead of fvol for waveform 1,
- cutoff2 is used instead of cutoff when bpf_en[i]=1.

It is possible to overdrive the filter, which will saturate. This can be a desirable effect.

For filter modes 0-2, the filter cutoff frequency is given by

```
cutoff_freq = cutoff / (2*pi)
```

Filter mode 3 uses two cascaded 1st order low pass filters, the first with cutoff frequency given by `cutoff` and the second by `cutoff2` (TODO: check).

Filter modes 0 and 1 implement resonant filters, the resonance is given by

```
Q = cutoff / f_damp
```

where the resonance can start to be noticeable when `Q` becomes > 1. The resonance for filter mode 2 is fixed at `Q=1`.

Output The filter output from each voice is multiplied by `2^(-vol)` and the contributions are added together to from the synth's output.

Sweeps Each voice has five sweep values, which can be used to sweep the oscillator and control frequencies gradually up or down, or set them to new values without interfering with synth's state updates.

Each sweep value is a 16 bit word. A voice will periodically send read messages (TX header $= 2$) to read its sweep values, with

```
address = (voice_index << 3) + sweep_index
```

where `voice_index` goes from 0 to 3 and `sweep_index` describes the target of the sweep value:

```
sweep_index   target
          0   float_period[0]
          1   float_period[1]
          2   mod[0]
          3   mod[1]
          4   mod[2]
```

The sweep value can have two formats:

```
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
| X | 0 | replacement value                                     |
| X | 1 | X       |sign| oct          | mantissa                |
```

In the first case, the target value is simply replaced. For `mod` targets, the lowest four bits of the replacement value are discarded.

In the second case, the target is incremented (`sign=0`) or decremented (`sign=1`) at a rate that is described by `oct` and `mantissa`, which follow the same kind of simple floating point format as is used for `mod` values. The maximum rate that the target can be incremented or decremented by one is `output_freq / 2`, achieved when `oct` and `mantissa` are zero. In general, the sweep rate is

```
sweep_rate = 32 * output_freq / ((64 + mantissa) << oct)
```

Sweeping will never cause the target value to wrap around, but may cause it to stop a single step short of the extreme value. When `oct=15`, sweeping is disabled. This can be accomplished by setting the sweep value to all ones.

Power chords and other frequency combinations A single voice can be set up to produce power chords, e g, letting the 3 waveform generators produce outputs in precise or approximate frequency ratio 2 : 3 : 4 or 3 : 4 : 6. It is usually preferable that the frequency ratios are not exact, to get some detuning.

For the 2 : 3 : 4 case, assume that the sub-oscillator frequency is set to roughly half the main oscillator frequency, with the sub-oscillator frequency representing one frequency unit. The desired frequncies can be achieved in different ways, e g:

```
Frequency   Combination              Computation
        2    main                    2          = 2
        2   (main<<1) - (sub<<1)     2*2 - 1*2  = 2
        3    main     +  sub         2   + 1    = 3
        3   (main<<1) -  sub         2*2 - 1    = 3
        4   (main<<1)                2*2        = 4
        4    main     + (sub<<1)     2   + 1*2  = 4
```

The way that a frequency is achieved matters when the sub-oscillator is not at exactly half the main oscillator frequency. A mix such as `main, main*2 - sub, main + sub*2` will produce three independent frequencies with some detuned upwards and some downwards (since different signs for the sub-oscillator are used).

For the 3 : 4 : 6 case, assume that the sub-oscillator frequency is set to roughly one fourth of the main oscillator frequency, with the sub-oscillator frequency still representing one frequency unit. In this case, the desired frequncies can, e g, be achieved as

```
Frequency   Combination              Computation
        3    main      -  sub        4   - 1    = 3
        4    main                   4          = 4
        4   (main<<1) - (sub<<2)    4*2 - 1*4 = 4
        6    main      + (sub<<1)    4   + 1*2 = 6
        6   (main<<1) - (sub<<1)    4*2 - 1*2 = 6
```

A mix such as `main - sub, main, main + 2*sub` might be good.

Power chords work well with overdriving the filter. Filter mode 3 might sometimes be useful, overdriving the first low pass filter but allowing the second to take out some of the high end.

These are just some examples of how the `phase_comb`, `phase0_shl`, and `phase1_shl` parameters can be used to produce waveforms with different frequencies within the same voice.

Context switching To perform a context switch, the synth

- Sends a sequence of 12 context switch messages on the TX channel, with TX header=0 and payload equal to the each of its twelve 16 bit state words in turn.
- Receives a sequence of 12 context switch responses on the RX channel, with RX header=1 and payload equal to the replacement value for the corresponding state word.

These sequences can and should overlap (for time efficiency); as soon as a context switch message has been sent on the TX channel, a corresponding response can be sent on the RX channel. The synth can send several context switch messages before receiving any response, as long as the `sbio_credits` register has been set appropriately (see below).

One way to implement the context switch response mechanism is as a swap operation. A state buffer of 3 x 12 sixteen bit words is needed, and a pointer into it. Each time a context switch message arrives,

- the old value at `buffer[pointer]` is sent back in a context switch response message,
- the new state value is assigned to `buffer[pointer]`,
- `pointer` is incremented, and wrapped around if it reaches the end of the buffer.

Only 3 x 12 words of state are needed here because the final 12 state words are stored in the synth at any time. Which buffer entries correspond to the state of which voice will shift over time.

Another way to implement the context switch response is to keep a state buffer of 4 x 12 words, with each 12 word section dedicated to the state of a specific voice. This is probably easier to work with. Separate read and write pointers are used, with `read_pointer` initialized 12 steps (one voice) ahead of `write_pointer`. Each time a context switch message arrives,

- the value at `buffer[read_pointer]` is sent back in a context switch response message,
- the new state value is assigned to `buffer[write_pointer]`,
- both pointers are incremented, and wrapped around if they reach the end of the buffer.

In this case, since the same voice state is always kept at the same buffer position, the parameter part of the state does not need to be written to the buffer when a context switch message arrives.

Changing voice parameters The sweep parameters can be changed at any time, since they are just read by the synth. More care is needed to update voice parameters that are part of the state, since it is periodically being switched in and out.

The easiest way to change voice parameters is probably if the second scheme described for context switching above is used, and the parameter part of the state received from the synth during context switching is ignored. Then, the parameters can be changed at any time in the corresponding position in the state buffer, and will be read into the synth as needed when context switching into the voice.

Dynamic state can also be changed between the time it is switched out and in again, but more care is needed.

The synth begins with the state of the first voice in its on-chip state, but the state is uninitialized. During the first context switch, the write data can be ignored to throw away this uninitialized state, making the voice read its state from the state buffer the next time.

Credit mechanisms The *sample credits* mechanism lets the user limit the rate at which the synth produces samples. The two bit `sample_credits` register is initialized to one at reset, and decremented each time a new sample is finished and sent as a message (with TX header=1). When `sample_credits` is zero, the synth pauses at some point before sending the next sample. When the user is ready to receive more samples, it should write a nonzero value to the `sample_credits` register. By writing a value that is larger than one, the synth can continue processing also after sending a sample.

The synth tries to limit the number of outstanding messages that have not received a reply, so as not to overload the receive FIFO in the RP2040 (or whoever receives the messages). Each context switch and read message (TX header = 0 or 2) expects a single reply (RX header = 1 or 2). A counter for outstanding messages is increased whenever a message of the former type is sent, and decreased whenever a message of the latter type is received. No credited messages will be sent as long as the outstanding counter equals the value in the `sbio_credits` register; the synth will wait for a credited response first to decrease the number of outstanding messages.

Sample out and vblank messages do not expect a response and do not increase the outstanding counter, but should be infrequent enough that it is enough to reserve one extra space for each in the receive FIFO. Write register messages (RX header=3) do not affect the outstanding message counter and can be sent to the synth at any time.

How to test

A RAM emulator program for RP2040 is needed to test the console (TODO: publish source code). The RAM emulator code can be modified to update VRAM to test the PPU, and update synth parameters to test the synth. The RAM emulator could also receive commands to do these things over the RP2040's USB-UART.

Testing the PPU A Pmod is needed for VGA output, see below.

Write Copper instructions to VRAM to initialize the PPU registers that don't have predefined initial values (see PPU registers). Set up tile planes, sprites, or both, the `displaymask` register can be used to disable tile planes or sprites if they are not used. TODO: example (in the RAM emulator code?)

Testing AnemoneSynth Means of sound output is TBD, see below.

Disable all sweeps (set the sweep parameters to all ones) and set the voice parameters to the default values described in the Voice state section. Set

- the main oscillator frequency to the desired pitch,
- `float_period[1] = float_period[0] + (10 &lt;&lt; 10)`,
- `mod[0]` and `mod[1]` to twice the main oscillator frequency,
- `mod[2] = mod[0] + (2 &lt;&lt; 6)`.

TODO: example (in the RAM emulator code?)

External hardware

A Pmod for VGA is needed for video output, that can accept VGA output according to https://tinytapeout.com/specs/pinouts/#vga-output. Means of sound output is TBD. The RP2040 receives the sound samples and could output them in different ways depending on programming. The pins `ui[7:4]` (or at least `ui[7:6]`, depending on pin configuration) have been left unused in the design so that the RP2040 can drive them to output sound. Supporting a Pmod for I2S would be one possibility.

Pinout

#	Input	Output	Bidirectional
0	data_in[0]	R1	addr_out[0]
1	data_in1	G1	addr_out1
2	data_in2	B1	addr_out2
3	data_in[3]	vsync	addr_out[3]
4	rx_alt_in[0]	R0	tx_out[0]
5	rx_alt_in1	G0	tx_out1
6		B0	rx_in[0] / Gm1_active_out
7		hsync	rx_in1 / RBm1_pixelclk_out

Chess [43]

- Author: Hannah Ravensloft
- Description: chess move generator
- GitHub repository
- HDL project
- Mux address: 43
- 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

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

2048 sliding tile puzzle game (VGA) [68]

- Author: Uri Shaked
- Description: Slide numbered tiles on a grid to combine them to create a tile with the number 2048.
- GitHub repository
- HDL project
- Mux address: 68
- Extra docs
- Clock: 0 Hz

How it works

2048 is a single-player sliding tile puzzle video game. Your goal is to slide numbered tiles on a grid to combine them and create a tile with the number 2048. The game is won when a tile with the number 2048 appears on the board, hence the name of the game. The game is lost when the board is full and no more moves can be made.

The game is played on a 4x4 grid, with numbered tiles that slide when a player moves them using `ui_in` pins. The game starts with two tiles with the number 2 on the board. The player can move the tiles in four directions: up, down, left, and right. When the player moves the tiles in a direction, the tiles slide as far as they can in that direction until they hit the edge of the board or another tile. If two tiles with the same number collide, they merge into a single tile with the sum of the two numbers. The resulting tile cannot merge with another tile again in the same move.

How to test

Use the `ui_in` pins to move the tiles on the board:

`ui_in` pin	Direction
0	Up
1	Down
2	Left
3	Right

After resetting the game, you will see a jumping "2048" animation on the screen. Press any of the `ui_in[3:0]` pins to start the game. The game will start with two tiles with the number 2 on the board. Use the `ui_in` pins to move the tiles in the desired direction. The game will end when the board is full and no more moves can be made.

The game offers two color themes: modern and retro. You can switch between the two themes by setting `ui_in[6]`.

Setting `ui_in[7]` to 1 will enter unit test mode. In this mode, the game displays a colorful rectangle on the top of the screen, and accepts debug commands on the `uio` pins. Check out the test bench for more information.

External hardware

TinyVGA PMOD

Pinout

#	Input	Output	Bidirectional
0	btn_up	R1	debug_cmd
1	btn_down	G1	debug_cmd
2	btn_left	B1	debug_cmd
3	btn_right	VSync	debug_cmd
4		R0	debug_data
5		G0	debug_data
6	retro_colors	B0	debug_data
7	debug_mode	HSync	debug_data

1bit_am_sdr [74]

- Author: James Sharp
- Description: 1bit AM software defined radio
- GitHub repository
- HDL project
- Mux address: 74
- Extra docs
- Clock: 50000000 Hz

How it works

This project is a Software Defined Radio pipeline for AM radio reception written in verilog. It works using an external comparator as a 1-bit ADC frontend which is oversampled and decimated 4096 times to give an extra 6 bits of precision. It is based on this Hackaday Project: https://hackaday.io/project/170916-fpga-3-r-1-c-mw-and-sw-sdr-receiver by Alberto Garlassi.

Although this is a fully digital core, but there are plans to make an analog frontend circuit as an analog design in future Tiny Tapeouts, so both designs would be hooked up together to create a radio with few external components.

Also, this core is very big - 3x2 Tiny Tapeout tiles (@ 64% utilisation). An area of future development could be to simplify the demodulation pipeline to reduce gate count.

How to test

You need to connect an external comparator and RC network. You will probably need a simple RF amplifier as well. See below for more information.

The core has a SPI interface for setting the demodulation frequency and gain. It consists of a single 32-bit shift register. It has the following format:-

Bits 31 - 30	Bits 29 - 26	Bits 25 - 0
Unused	Gain	NCO Phase incr.

The gain can take on the following values:

"Gain" value	Actual Gain
0	x1

"Gain" value	Actual Gain
1	x2
2	x4
3	x8
4	x16
5 - 7	x32

If the gain is set too high, the demodulated signal will wrap and sound distorted, so adjust the gain down to the minimum needed to hear the station clearly.

The "NCO Phase increment" is the value that is added to the NCO phase every clock cycle. Use the following python code to calculate the value to write, based on the desired carrier frequency:

```
hex(int((1<<26) * <carrier frequency> / <chip clock frequency>))
```

E.g., for 936kHz (ABC Radio national Hobart) at 50MHz clock frequency, it would be:

```
> hex(int((1<<26) * 936000 / 50000000))
'0x132b55'
```

External hardware

- External comparator
- Resistor bias network
- RC network
- External SPI microcontroller to set station
- RF amplifier

Pinout

#	Input	Output	Bidirectional
0	COMP_IN	COMP_OUT	
1	SPI_MOSI	PWM	
2	SPI_SCK		
3	SPI_CSb		
4			

#	Input	Output	Bidirectional
5			
6			
7			

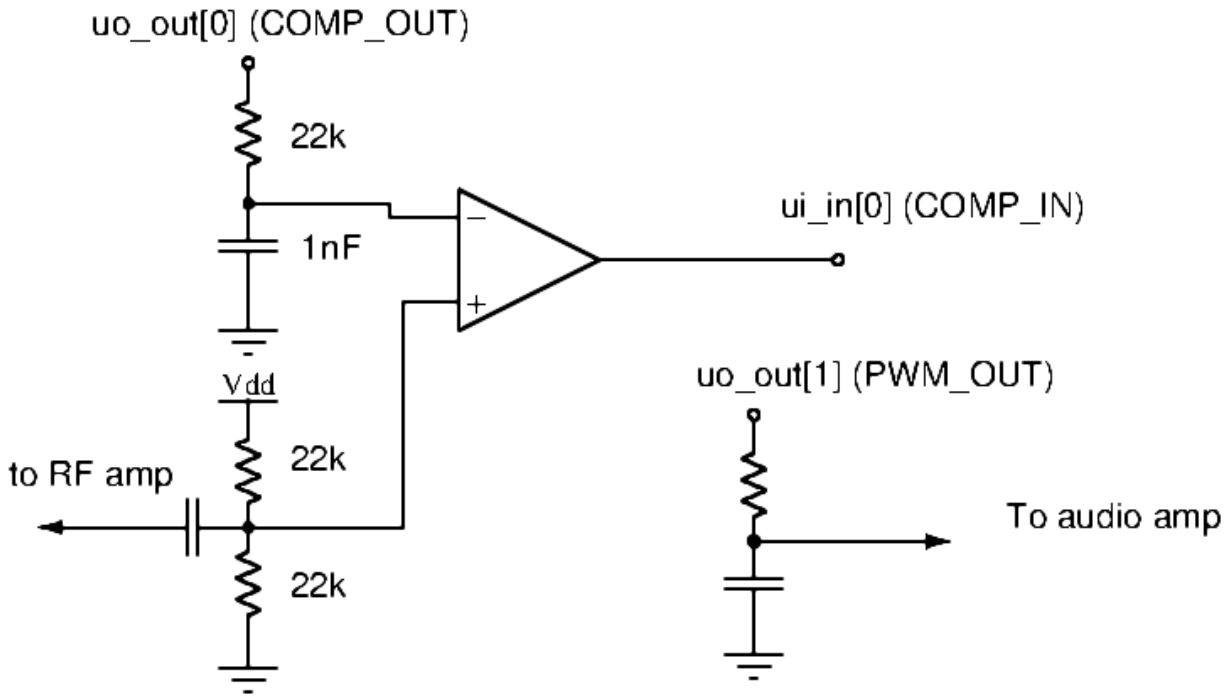


Figure 6: Schematic diagram of external circuitry

Conway's Game of Life on UART and VGA [101]

- Author: Ciro Cattuto
- Description: A simulation of Conways' Game of Life visualized to an ANSI terminal over UART and to VGA
- GitHub repository
- HDL project
- Mux address: 101
- 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:

- `&lt;space&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). And/or TinyVGA PMOD for VGA output.

Pinout

#	Input	Output	Bidirectional
0	simulation start		hsync
1	simulation randomization		B[0]
2			G[0]
3	UART RX		R[0]
4		UART TX	vsync
5			B1
6			G1
7			R1

mulmul [105]

- Author: JJ Wong
- Description: Small 4-bit vector multiplication engine
- GitHub repository
- HDL project
- Mux address: 105
- Extra docs
- Clock: 0 Hz

How it works

Write the registers and vector length and accumulator value (optional) into the chip's registers using the read and write opcodes, then run the system with the run opcode. The vectors will be multiplied and summed together in two clock cycles and output an 8-bit word.

Input words are 4 bits wide. Write the length of the 4-bit vectors you want to multiply into address 0. The vectors should be in words 1-32. Word 1 will be multiplied by word 17, etc. The result will be accumulated into words 33-34 (8 bits).

How to test

You can run the testbench tests in the test dir.

External hardware

Will be programmed by RP2040. No other external hardware.

Pinout

#	Input	Output	Bidirectional
0	addr[0]	out[0]	data[0]
1	addr1	out1	data1
2	addr2	out2	data2
3	addr[3]	out[3]	data[3]
4	addr[4]	out[4]	state[0]
5	addr[5]	out[5]	state1
6	op[0]	out[6]	

#	Input	Output	Bidirectional
7	op1	out[7]	

ROTFPGA v2a [107]

- Author: htfab
- Description: A reconfigurable logic circuit made of identical rotatable tiles
- GitHub repository
- HDL project
- Mux address: 107
- Extra docs
- Clock: 10000000 Hz

How it works

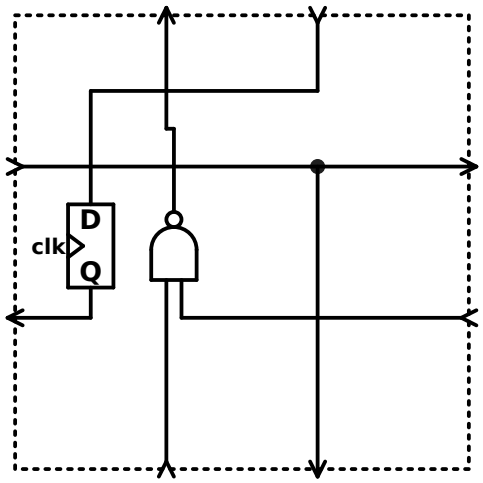


Figure 7: Logic tile

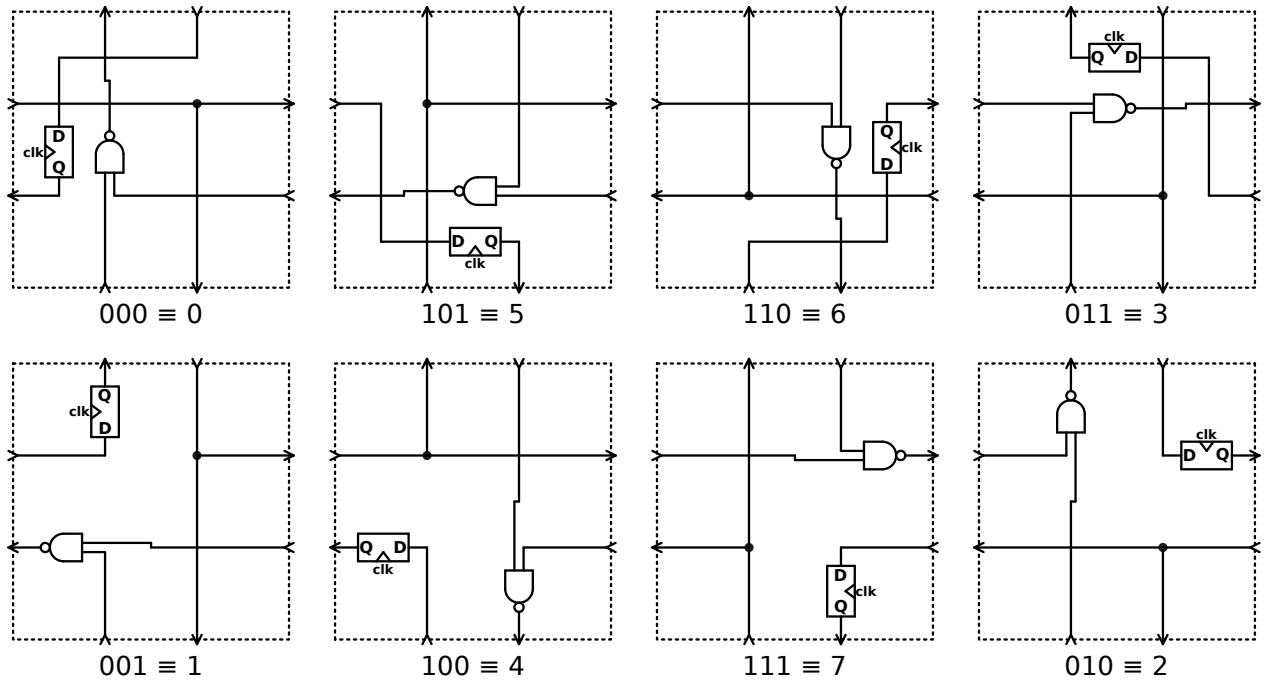
A reconfigurable logic circuit built from identical copies of the tile above containing a NAND gate, a D flip-flop and a buffer, with each tile individually rotated or reflected as described by the FPGA configuration. Port of the original ROTFPGA from Caravel to TinyTapeout.

Porting the design required a 50-fold decrease in chip area which was achieved using a combination of cutting corners, heavy optimization and a few design changes. In particular:

- The FPGA was reduced from 24×24 to 8×8 tiles. There are 8 inputs and 8 outputs instead of 12 each.
- To compensate for smaller size, tiles can also be mirrored in addition to rotation.
- Tiles (being the most repeated part of the design) were rewritten as hand-optimized gate-level Verilog.

- Each tile only contains 1 flip-flop (the one exposed to the user). Configuration is now stored in latches.
- Configuration and reset are performed using a routing-efficient scan chain, so the design is no longer routing constrained. This allows standard cells to be placed with $>80\%$ density.
- Openlane and its components are 2 years more mature, hardening the same HDL more efficiently.

Configuration Each tile can be configured in 8 possible orientations. Bits 0, 1 and 2 correspond to a diagonal, horizontal and vertical flip respectively. Any rotation or reflection can be described as a combination:



(The bottom row looks somewhat different, but we just rearranged the wires so that the inputs and outputs line up with the unmirrored tiles.)

Tiles are arranged in an 8×8 grid:

- Top, bottom, left and right inputs and outputs are connected to the tile in the respective direction.
- Tiles mostly wrap around, e.g. the bottom output of a cell in the last line connects to the top input of the cell in the first line.
- As an exception to the wrapping rules, left inputs in the first column correspond to chip inputs and right outputs in the last column correspond to chip outputs.
- There is a scan chain meandering through all the tiles, visiting lines from top to bottom and within each line going from left to right.

This is a 4×4 model of the tile grid, showing regular i/o as black and the scan chain as grey:

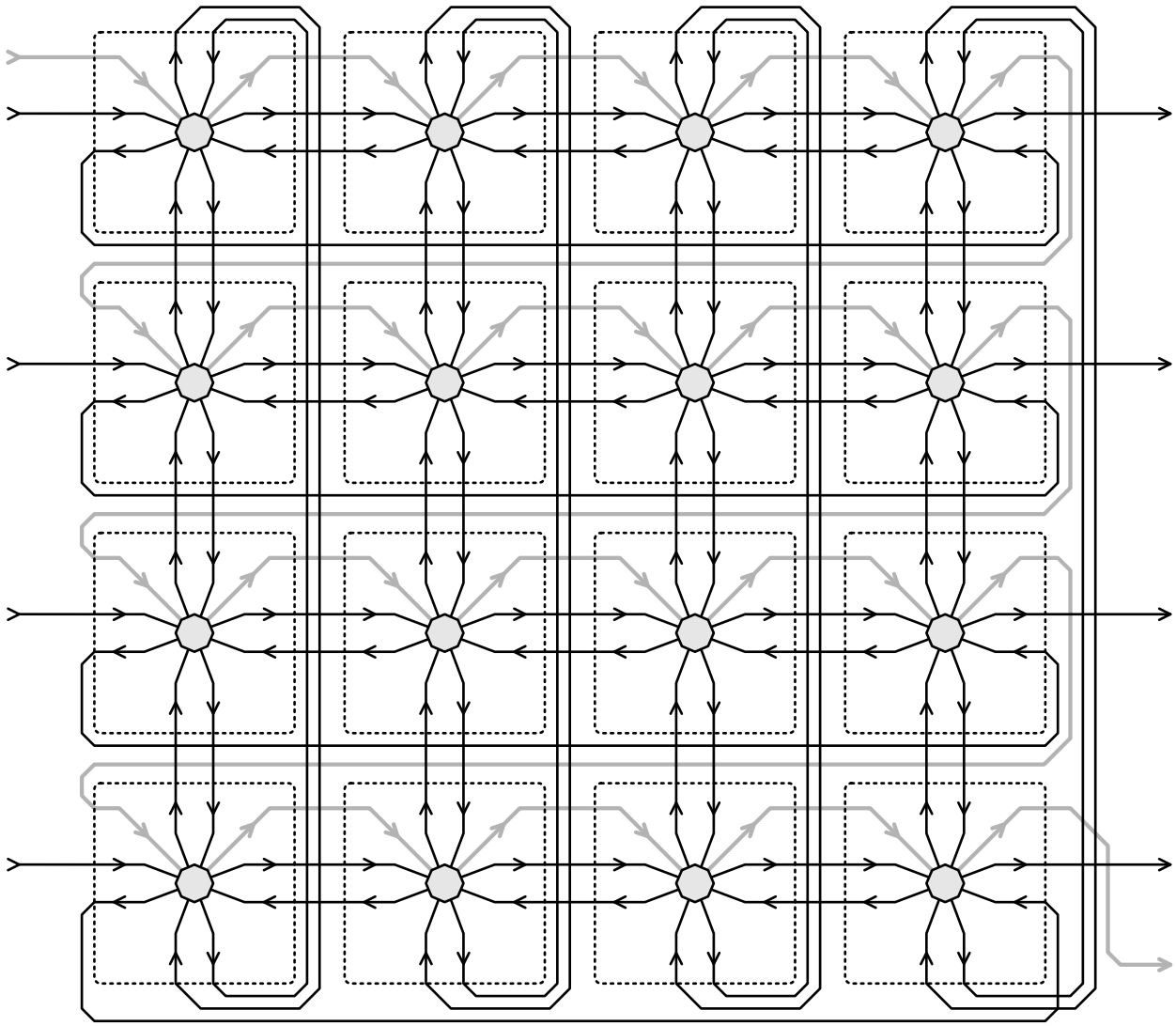


Figure 8: Grid model

When the *scan enable* input is 0, the FPGA operates normally and each tile sets its flip-flop to the input it receives from one of the neighboring tiles according to its current rotation/reflection. When *scan enable* is 1, it sets the flip-flop to the value received through the scan chain instead. This allows us to set the initial state of each flip-flop and also to query their state later for debugging. With some extra machinery it also allows us to change the rotations/reflections.

When the 2-bit *configuration* input is is 01, each cell updates its *vertical flip* bit to the current value of its flip-flop. Similarly, for 10 it sets the *horizontal flip* and for 11 it sets the *diagonal flip*. When *configuration* is 00, all three flip bits are latched and the orientation doesn't change.

One can thus configure the FPGA by sending the sequence of all *diagonal flip* bits

through the scan chain, then setting *configuration* to 11 and back to 00, then sending all *horizontal flip* bits, setting *configuration* to 10 and back to 00, and finally sending the *vertical flip* bits and setting *configuration* to 01 and back to 00.

Note that in order to save space the flip bits are stored in latches, not registers. Changing the *configuration* input from 00 to 11 or vice versa can cause a race condition where it is temporarily 01 or 10, overwriting the horizontal or vertical flip bits. Therefore one should configure the diagonal flips first.

Loop breaker The user design may intentionally or inadvertantly contain combinational loops such as ring oscillators. To help debug such designs, the chip has a loop breaker mechanism using a *loop breaker enable* input as well as a 2-bit *loop breaker class* input.

Tiles are assigned to loop breaker classes:

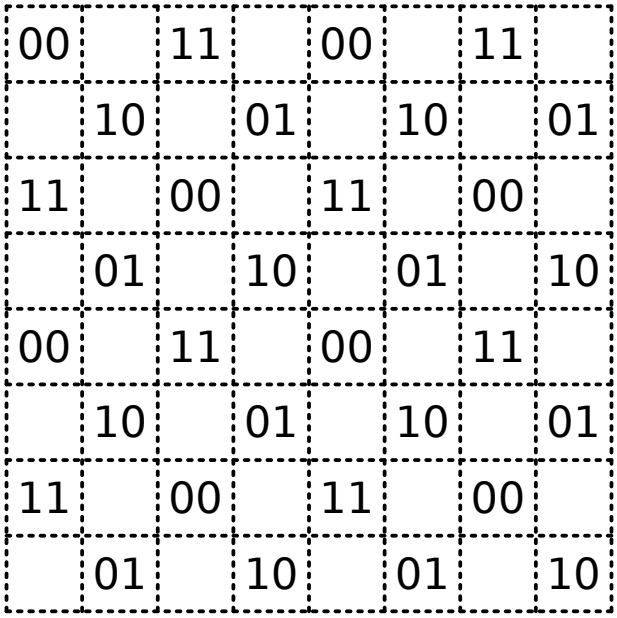


Figure 9: Loop breaker tile classes

The loop breaker latches a tile output if and only if the following conditions are all met:

- The *loop breaker enable* input is 1.
- The current tile has a non-empty class that is different from the *loop breaker class* input.
- The output doesn't come from the tile's flip-flop.

The loop breaker has the following properties:

- If *loop breaker enable* is 1 and *loop breaker class* is constant, there are no combinational loops running. If we also pause the clock, the circuit keeps a steady state.
- If *loop breaker enable* is 1 and we cycle *loop breaker class* through all possible values repeatedly while the clock is paused, everything will eventually propagate. If we also assume that the design has no race conditions, it will behave in the same way as if *loop breaker enable* was 0.

Reset Setting the *active-low reset* input to 0 has the following effect:

- Override *scan enable* to 1, *scan chain* input to 0 and disengage the latches for vertical, horizontal and diagonal flips. When kept low for 64 clock cycles this will reset the state and configuration in every tile.
- Override *loop breaker enable* to 1 and *loop breaker class* to 00. This ensures that we play nice with other designs on TinyTapeout and keep a steady state while our design is not selected.

Pin mapping Input pins:

- `clk` provides a clock signal for the flip-flops
- `rst_n` is the *active-low reset* described above
- `ui_in[7:0]` are passed to the leftmost column of tiles as inputs from the left

Output pins:

- `uo_out[7:0]` come from the rightwards output of the rightmost column of tiles

Bidirectional pins:

- `uio_in[0]` is the *scan enable* input
- `uio_in[1]` is the *scan chain* input
- `uio_in[3:2]` are the *configuration* input bits
- `uio_in[4]` is the *loop breaker enable* input
- `uio_in[6:5]` are the *loop breaker class* input bits
- `uio_out[7]` is the *scan chain* output

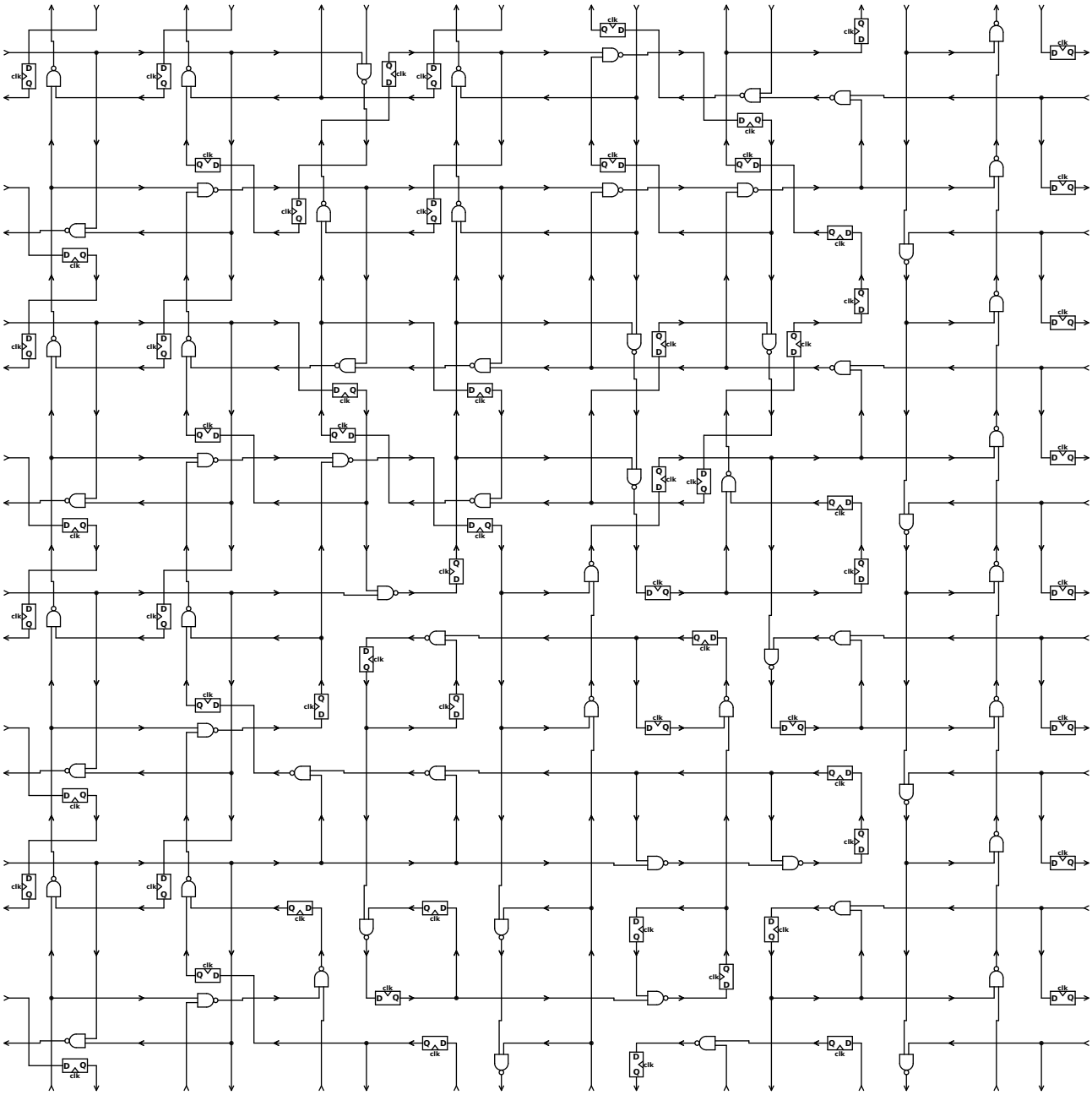


Figure 10: Diagram corresponding to `fpga_config` in `test.py`

How to test

Follow the test suite the `test` directory. It sets up the FPGA with the following two configurations and runs a battery of tests on each.

Test configuration 1 used for upload, download, single-step and propagation tests:

Test configuration 2 used for testing the loop breaker with manual and automatic cycles:

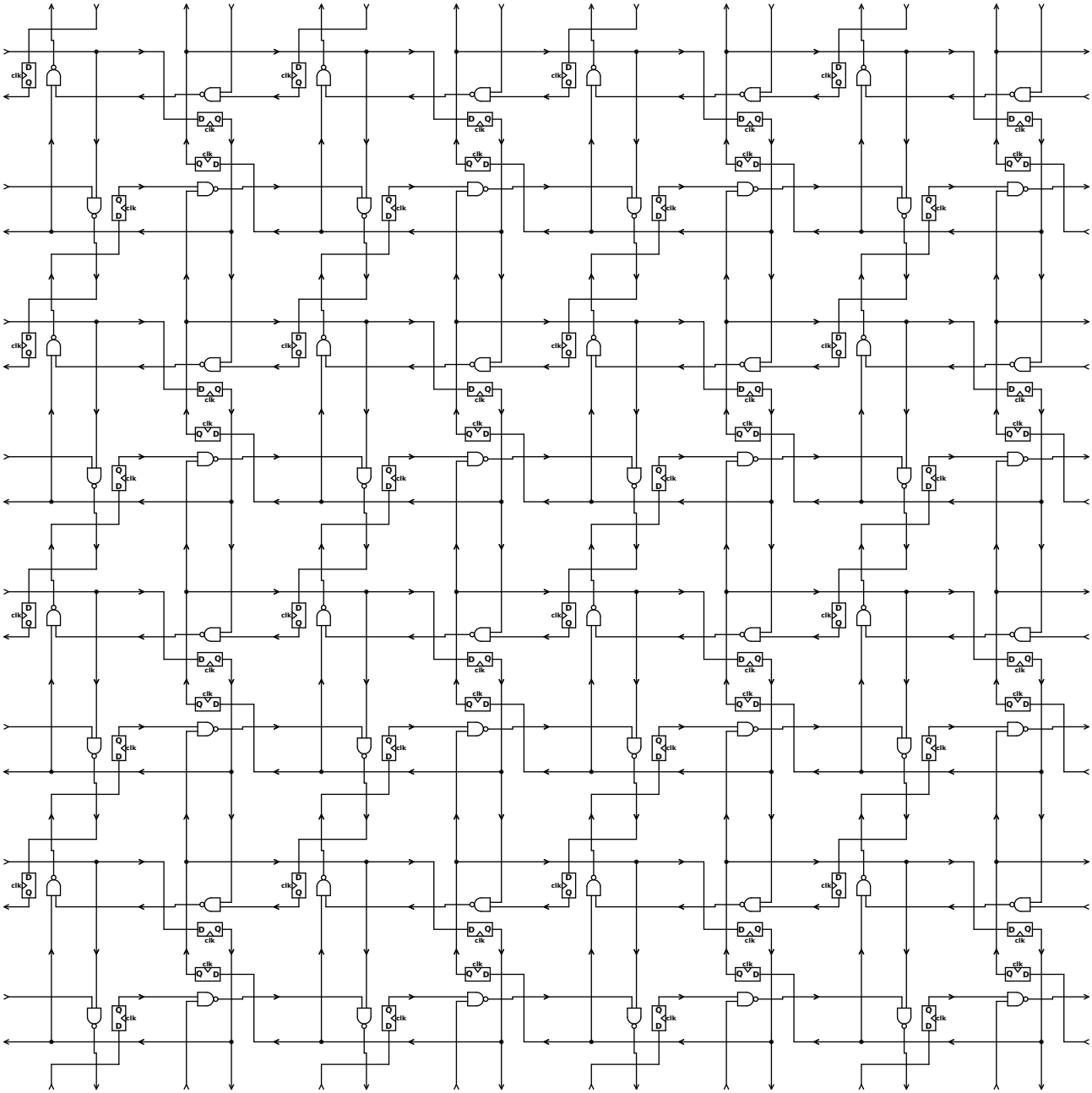


Figure 11: Diagram corresponding to `cfg` from the loop breaker test in `test.py`

External hardware

None

Pinout

#	Input	Output	Bidirectional
0	tile(0,0) left in	tile(7,0) right out	*scan enable* input
1	tile(0,1) left in	tile(7,1) right out	*scan chain* input
2	tile(0,2) left in	tile(7,2) right out	*configuration* input bit 0
3	tile(0,3) left in	tile(7,3) right out	*configuration* input bit 1
4	tile(0,4) left in	tile(7,4) right out	*loop breaker enable* input
5	tile(0,5) left in	tile(7,5) right out	*loop breaker class* input bit 0
6	tile(0,6) left in	tile(7,6) right out	*loop breaker class* input bit 1
7	tile(0,7) left in	tile(7,7) right out	*scan chain* output

simon_cipher [128]

- Author: Simon Cipher
- Description: Bitserial implementation of Simon-128
- GitHub repository
- HDL project
- Mux address: 128
- Extra docs
- Clock: 0 Hz

How it works

This is a bitserial implementation of the SIMON Block Cipher. SIMON is a 128-bit block cipher, see The SIMON and SPECK families of Lightweight Block Ciphers. A bit-serial implementation exchanges throughput for area, thereby creating a compact cipher that is dominated by flip-flops and multiplexer cells. However, the overal design size becomes minimal. A detailed description of the bitserial implementation technique for SIMON is available in SIMON Says, Break the Area Records for Symmetric Key Block Ciphers on FPGAs .

Cell	Count
flip-flop	281
mux	588
other logic	199
TOTAL	1068

The design uses a 3-bit input and a 2-bit output, in addition to clock and reset.

Port	Function
ui[0]	Bitserial Data Input
ui[7:6]	Control Word
uo[0]	Bitserial Data Output
uo[7]	Data Output Valid

The data input is asserted by the control word, and must be valid when the control word indicates a plaintext-loading or key-loading operation.

The data output is asserted by the valid bit, and should be ignored when the data valid bit is 0. The output ciphertext is produced in 128 consecutive clock cycles.

The 2-bit control word defines the operation of the cipher. The LSB is a debug bit study to key-loading process and to verify that the key register was correctly loaded.

Control	Function
00	Idle
01	Load 128-bit plaintext
10	Load 128-bit key (see LIMITATIONS)
11	Encrypt and return ciphertext

LIMITATIONS

This design forces the key bits to 0 upon loading, so that the effective key value of the cipher is always hardcoded to 00000000_00000000_00000000_00000000. This disables the use of the design as a cipher, yet it still demonstrates how a bit-serial architecture can be designed.

How to test

Study the testbench for example test vectors.

External hardware

No external hardware is needed for this project.

Pinout

#	Input	Output	Bidirectional
0	ui_in[0]	uo_out[0]	
1	ui_in1	uo_out1	
2	ui_in2	uo_out2	
3	ui_in[3]	uo_out[3]	
4	ui_in[4]	uo_out[4]	
5	ui_in[5]	uo_out[5]	
6	ui_in[6]	uo_out[6]	
7	ui_in[7]	uo_out[7]	

VGA Screensaver with Tiny Tapeout Logo [130]

- Author: Uri Shaked
- Description: Tiny Tapeout Logo bouncing around the screen (640x480, TinyVGA Pmod)
- GitHub repository
- HDL project
- Mux address: 130
- Extra docs
- Clock: 0 Hz

How it works

Displays a bouncing Tiny Tapeout logo on the screen, with animated color gradient.

Figure 12: 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,
- `solid_color` (ui_in1) to use a solid color instead of an animated gradient.

External hardware

TinyVGA PMOD

Pinout

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

KianV RISC-V RV32E Baremetal SoC [138]

- Author: Dipl.-Ing. Hirosh Dabui
- Description: A baremetal RISC-V RV32E ASIC with audio, spi, uart
- GitHub repository
- HDL project
- Mux address: 138
- 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		led1	cs1 psram
7		led2	always high

ROTFPGA v2b [161]

- Author: htfab
- Description: A reconfigurable logic circuit made of identical rotatable tiles
- GitHub repository
- HDL project
- Mux address: 161
- Extra docs
- Clock: 10000000 Hz

How it works

This design is a minor modification of ROTFPGA v2a, intended as a "control group" for testing latches on IHP. If ROTFPGA v2b works but v2a doesn't, it indicates an issue with latches. Otherwise it might be a problem with the design itself.

Most of the documentation carries over from ROTFPGA v2a and is not repeated here. The differences are:

- Latches are simulated using flip-flops
- Some inputs are combined to make room for two extra inputs

Simulation of latches Latches are replaced with flip-flops that operate on the "latch clock" whereas original flip-flops are modified to act on the "flop-flop clock".

In practice the "latch clock" and the "flip-flop clock" are gated versions of `clk`, enabled by `in_l_gate` and `in_ff_gate` respectively.

Input reshuffling To add `in_l_gate` and `in_ff_gate` to the inputs, the number of existing inputs had to be reduced. Since `in_cfg` is typically only used when `in_se` is high and `in_lbc` is typically only used when `in_se` is low, they were combined into `in_cfg_lbc`.

How to test

The changes above were incorporated into the test suite. Every clock tick in the original test was replaced by 50 "latch clocks" followed by a single "latch and flip-flop clock" and then by 50 more "latch clocks".

External hardware

None

Pinout

#	Input	Output	Bidirectional
0	tile(0,0) left in	tile(7,0) right out	*scan enable* input
1	tile(0,1) left in	tile(7,1) right out	*scan chain* input
2	tile(0,2) left in	tile(7,2) right out	*configuration* / *loop breaker class* input bit 0
3	tile(0,3) left in	tile(7,3) right out	*configuration* / *loop breaker class* input bit 1
4	tile(0,4) left in	tile(7,4) right out	*loop breaker enable* input
5	tile(0,5) left in	tile(7,5) right out	clock gating for flip-flops
6	tile(0,6) left in	tile(7,6) right out	clock gating for simulated latches
7	tile(0,7) left in	tile(7,7) right out	*scan chain* output

Asynchronous Multiplier [163]

- Author: Tommy Thorn
- Description: An asynchronous multiplier
- GitHub repository
- HDL project
- Mux address: 163
- Extra docs
- Clock: 50000000 Hz

How it works

This design emits a sequence of `r = x^2+x`, for x=0,1,2,... on the outputs using the handshake protocol (tie ack to req to get free running sequence). Well, in truth, we use 26-bits of internal precision, but we only have 15-bits for outputs, we what is actually emitted is `r ^ (r &gt;&gt; 15)`.

The very naive algorithm (with the body unrolled once) is

```
x = 0
loop:
  x = x + 1
  a = b = c = x
  while b != 0:
    if (b & 1) == 1:
      c += a
    a *= 2
    b /= 2
    if (b & 1) == 1:
      c += a
    a *= 2
    b /= 2
  output (c)
```

which was hand translated (roughly following Introduction to Asynchronous Circuit Design) into a token flow graph:

Note, I use a simpler, less expensive, construction for the conditional iteration as having independent control-flow for the trivial condition is overkill.

The graph was realized using four-phase bundled data. Alas, I'm still working on the timing analysis, so the inserted delays are (hopefully) way oversized.

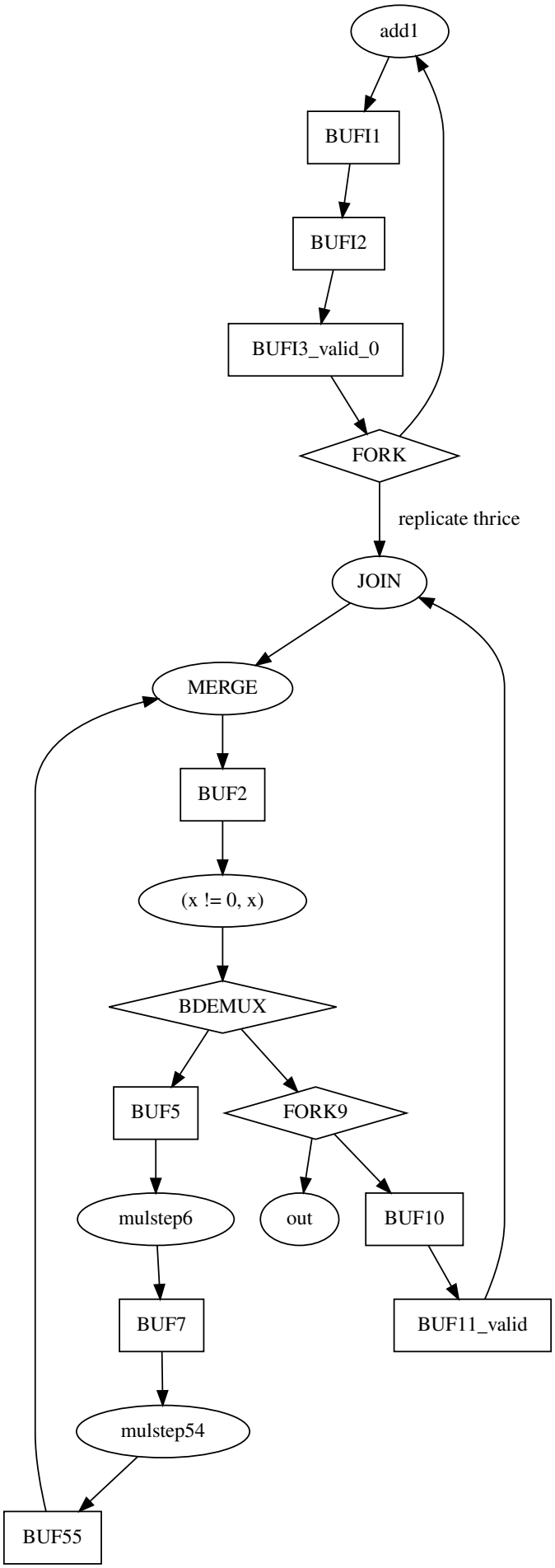


Figure 13: token-flow graph

How to test

The data is presented using the standard 4-phase (RTZ) protocol (idle, Req, Req+Ack, Ack, idle, ...). To get a continuous stream, simply tie ack to req. The values expected are 0, 2, 6, ..., x(x+1)

External hardware

A logic analyzer is convenient to pick up the values on the outputs, but default RP2040 works fine.

Pinout

#	Input	Output	Bidirectional
0	ack	req	result_7
1		result_0	result_8
2		result_1	result_9
3		result_2	result_10
4		result_3	result_11
5		result_4	result_12
6		result_5	result_13
7		result_6	result_14

SRAM (1024x8) test [167]

- Author: Uri Shaked
- Description: Tests the foundry provided SRAM macro
- GitHub repository
- HDL project
- Mux address: 167
- Extra docs
- Clock: 0 Hz

How it works

This is a 1 kbyte SRAM controller module. It allows reading or writing a single byte at a time.

There are 10 address lines, 8 data lines, and 1 write enable line.

To read a byte, set the write enable line (wen) to 0, and the data lines (dout[7:0]) will be set to the value of the byte at the address specified by the address lines (addr[5:0]).

To write a byte, set the write enable line (wen) to 1, and set the data lines (din[7:0]) to the desired value. Writing is only possible when the bank_sel line is 0.

The lower 6 address bits (addr[5:0]) are exposed as input pins.

The upper 4 address lines are stored in the address_bank register. To change the upper address bits, set the bank_sel line to 1, and set the data lines (addr[9:6] / uio[3:0]) to the desired value.

How to test

1. Set addr[5:0] to the desired address, set din[7:0] to the desired value, set wen to 1, and set bank_sel to 0, then pulse the clock line. The value at the specified address should be updated to the value of din[7:0].
2. Set addr[5:0] to the desired address, set wen to 0, and set bank_sel to 0, then pulse the clock line. The value at the specified address should be output on dout[7:0].
3. Set addr[9:6] to the desired value, set bank_sel to 1, then pulse the clock line. The upper address bits should be updated to the value of addr[9:6].

External hardware

None

Pinout

#	Input	Output	Bidirectional
0	addr[0]	dout[0]	din[0]/addr[6]
1	addr1	dout1	din1/addr[7]
2	addr2	dout2	din2/addr[8]
3	addr[3]	dout[3]	din[3]/addr[9]
4	addr[4]	dout[4]	din[4]
5	addr[5]	dout[5]	din[5]
6	bank_sel	dout[6]	din[6]
7	wen	dout[7]	din[7]

Zilog Z80 [171]

- 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: 171
- 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

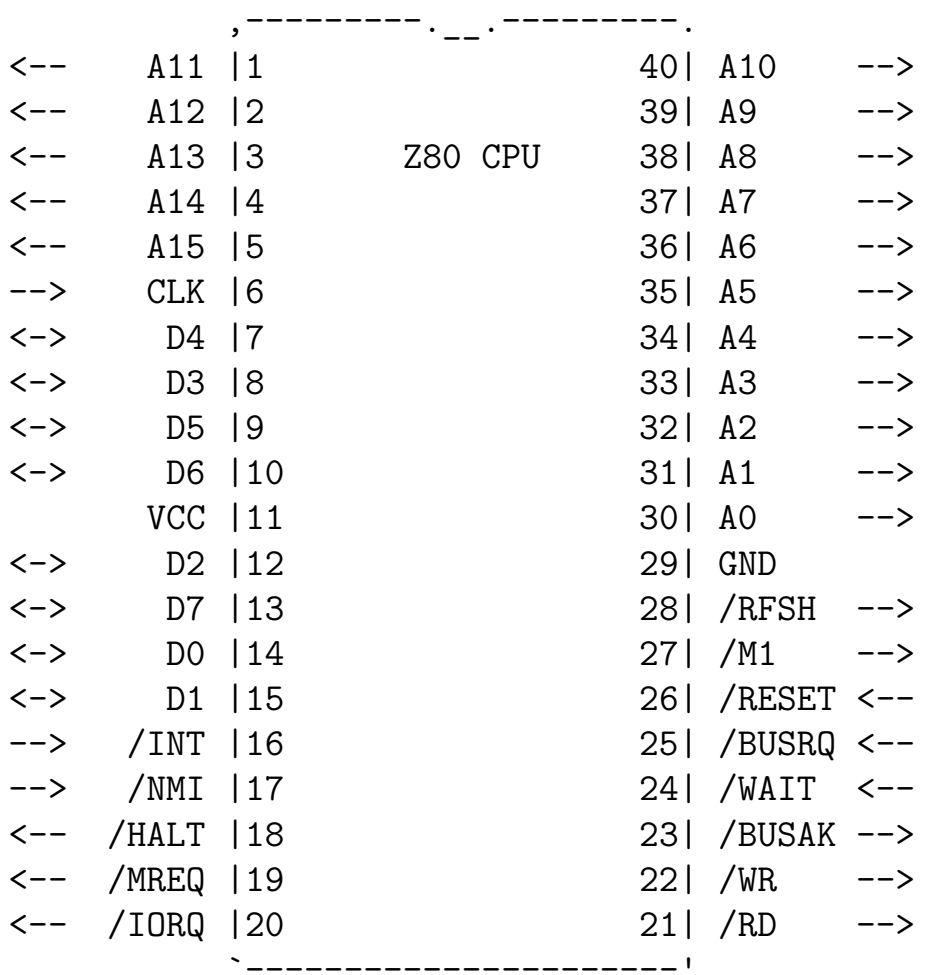
- nMOS original frequency 4MHz. CMOS frequency up to 20 MHz. This tapeout on 130 nm is expected to support frequency up to 50 MHz.
- 158 instructions including support for Intel 8080A instruction set as a subset.
- Two sets of 6 general-purpose reigsters which may be used as either 8-bit or 16-bit register 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)

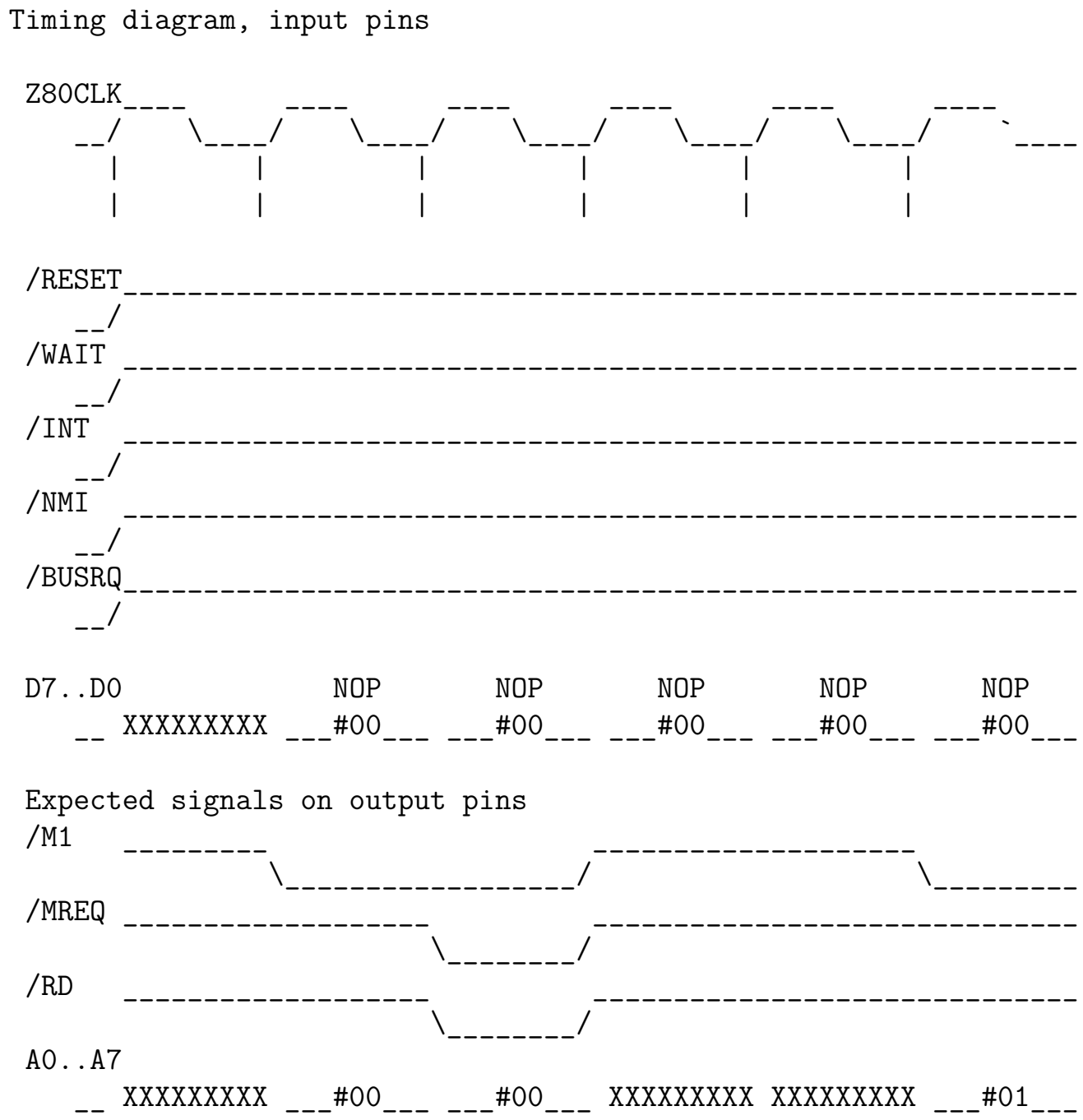
Z80 Pinout



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.



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

Minilogix [198]

- Author: Harald Pretl
- Description: A configurable 8b in, 8b out logic block with optional feedback
- GitHub repository
- HDL project
- Mux address: 198
- Extra docs
- Clock: 20000000 Hz

How it works

A programmable 8b input, 8b output freely programmable logic block with optional internal feedback. This can serve many purposes, once an FPGA-style configuration SW is available.

How to test

- Load the logic block in serial mode.
- Test the logic functionality by applying different digital inputs.

External hardware

Just a way to set digital inputs is needed, plus a way to check the digital outputs.

Pinout

#	Input	Output	Bidirectional
0	data in0	data out0	load enable
1	data in1	data out1	load clk
2	data in2	data out2	load data
3	data in3	data out3	
4	data in4	data out4	
5	data in5	data out5	dbg out0
6	data in6	data out6	dbg out1
7	data in7	data out7	dbg out2

Experiment Number Six: Laplace LUT [202]

- Author: Paul Hansel
- Description: ASCII ROM encoding the LaTeX characters needed to typeset the Laplace transforms of a few specialized functions.
- GitHub repository
- HDL project
- Mux address: 202
- Extra docs
- Clock: 10000000 Hz

How it works

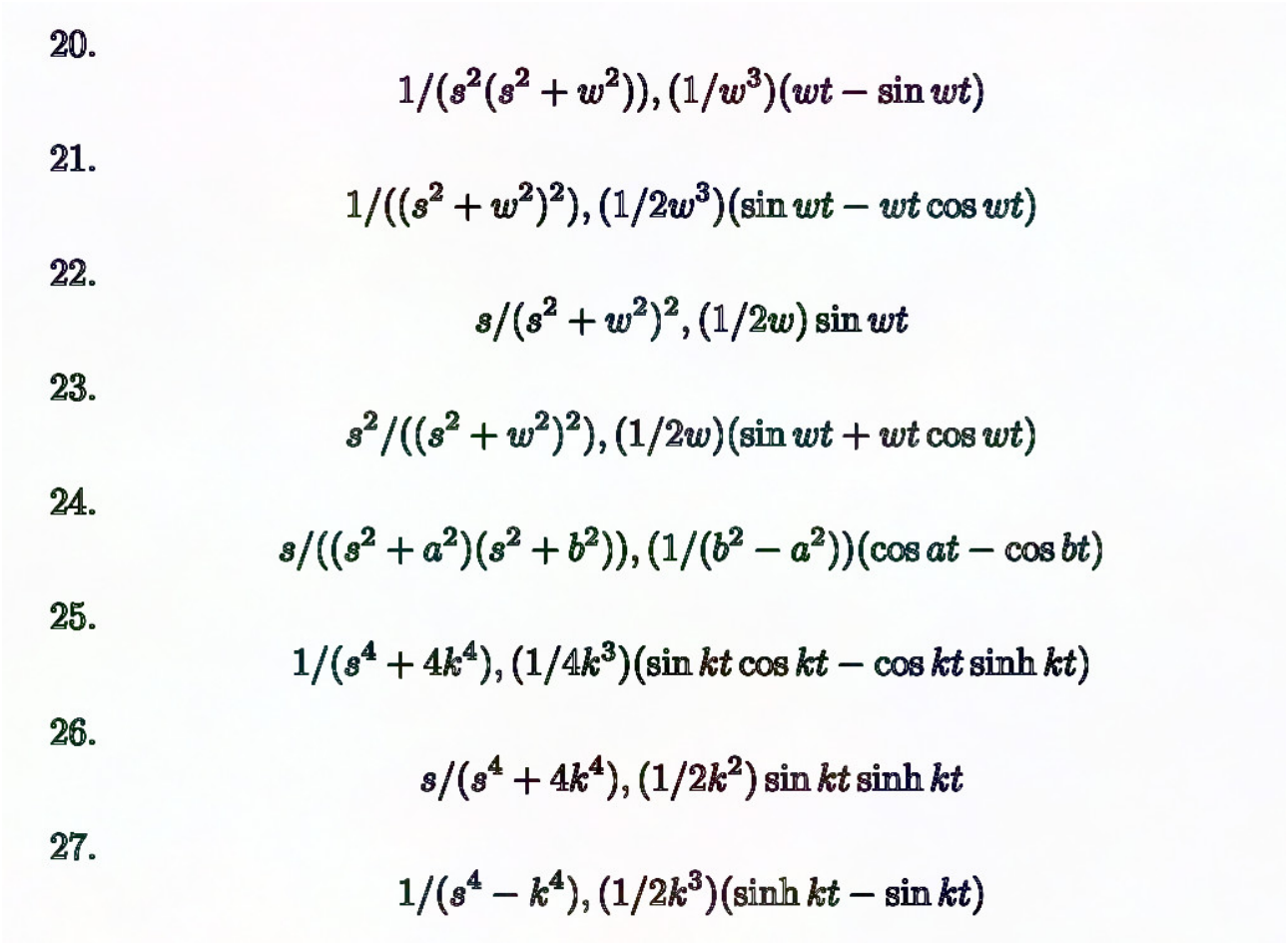


Figure 14: LaTeX screenshot

This project provides an ASCII encoding of the LaTeX code to typeset a few dozen Laplace transforms of common functions. When the user sets the lower ui_in pins to a number, asserts reset and then asserts ui_in 6 high, the project will begin clocking out the transform char-by-char, with uio_out showing F(s) = L{f(t)} and uo_out showing

f(t) itself. If either one is shorter than the other for a particular transform, empty space characters are appended.

It uses two different address spaces to do this: mem_addr, which maps each pair of concatenated ASCII characters (function, transformed function) from all transforms back-to-back as 16-bit values to a linear 10-bit address space, and pointer_addr, which maps the concatenated start address and length of each row (within mem_addr space) as 20-bit values to that row's line number in an 8-bit address space (with only 6 bits used).

The read-only Verilog containing the actual ASCII data is generated by a python script that reads the LaTeX source directly. Verification is achieved in the same way.

How to test

Program a number onto ui_in[5:0] between 0 and 46. Toggle reset_n (high/low/high), then toggle ui_in[6] high to start printing. Watch uo_out and uio_out for the resulting ASCII characters.

The input address bus accepts a number (0-46) corresponding to an arbitrary Laplace tranform encoding; it must be set before asserting start. The active-high character output enable signal must be high to start or continue character output. The clock divider disable input must be high to run at full speed or low to run at 1 character per 5x10^7 clocks.

External hardware

Switches or jumpers to 3V3 or 0V will be needed to set the increment on ui_in. It may be helpful to install LEDs on each of the LHS_x and RHS_x outputs to observe the output.

Pinout

#	Input	Output	Bidirectional
0	Address bit 0	RHS_BIT_0	LHS_BIT_0
1	Address bit 1	RHS_BIT_1	LHS_BIT_1
2	Address bit 2	RHS_BIT_2	LHS_BIT_2
3	Address bit 3	RHS_BIT_3	LHS_BIT_3
4	Address bit 4	RHS_BIT_4	LHS_BIT_4
5	Address bit 5	RHS_BIT_5	LHS_BIT_5
6	Character output enable	RHS_BIT_6	LHS_BIT_6

#	Input	Output	Bidirectional
7	Clock divider disable	RHS_BIT_7	LHS_BIT_7

VGA Pong with NES Controllers [225]

- Author: Brandon S. Ramos
- Description: Pong using 2 NES Controllers with a VGA display
- GitHub repository
- HDL project
- Mux address: 225
- Extra docs
- Clock: 25175000 Hz

How it works

This project is designed to play Pong with two players using NES controllers which output to a VGA compatible monitor.

How to test

You will need two NES controllers which will take in 3 wires (not including power and ground). Hook up the connections as shown in the bidirectional I/O.

Bidirectional:

1. NES_Controller_Left[0] data
2. NES_Controller_Left1 clock
3. NES_Controller_Left2 latch
4. NES_Controller_Right[0] data
5. NES_Controller_Right1 clock
6. NES_Controller_Right2 latch
7. NC
8. NC You will also need the hook up the output to a VGA breakout board. I created my own using a perfboard and some resistors but you can use the TinyTapeout VGA PMOD, just ensure that you hook up r0,r1 on the VGA PMOD both to r from the output as my design only uses 1 bit for each signal.

Output:

1. h_sync
2. v_sync
3. r
4. g

5. b
6.
7.
8.

External hardware

- VGA PMOD or your own VGA breakout board
- 2 NES controllers
- VGA compatible monitor

Pinout

#	Input	Output	Bidirectional
0		h_sync	NES_Controller_Left[0]
1		v_sync	NES_Controller_Left1
2		r	NES_Controller_Left2
3		g	NES_Controller_Right[0]
4		b	NES_Controller_Right1
5			NES_Controller_Right2
6			
7			

DemoSiine [227]

- Author: SagarDevAchar
- Description: A Wavy and Rainbowy TT08 Demoscene Submission
- GitHub repository
- HDL project
- Mux address: 227
- Extra docs
- Clock: 25000000 Hz

How it works

The project structure is as shown below:

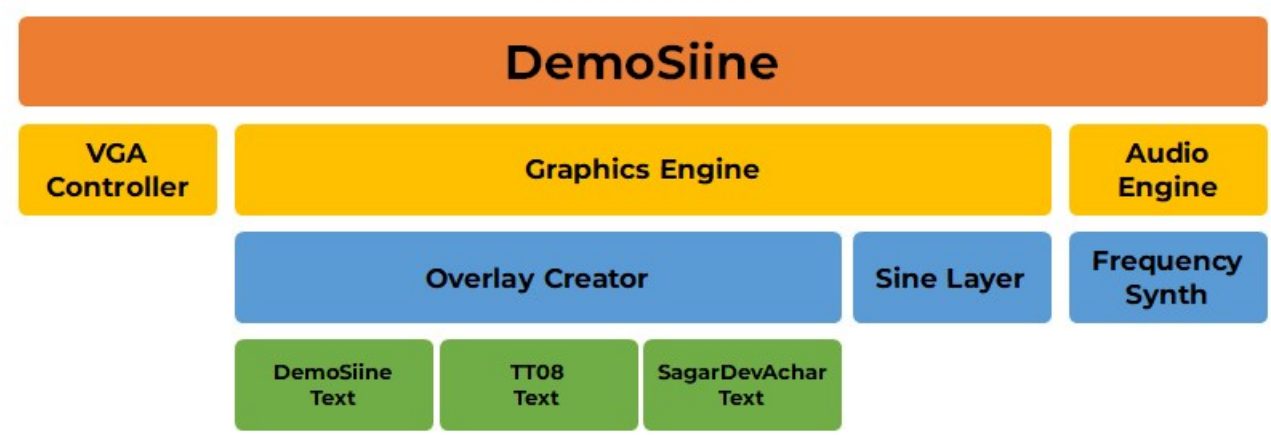


Figure 15: DemoSiine Project Structure

The **Graphics Engine** (driven by the **VGA Controller**, 640x480 @ 60Hz) is an on-demand RGB display pixel generator whose output can be altered using a few input pins. Previews of the different possible display outputs are provided in the last section of this documentation.

The **Audio Engine** drives the **Frequency Synth** to produce a ~28 second looping sound track @ 140 BPM at the output.

External hardware

- Leo's TinyVGA Pmod connected to OUTPUT terminal (`uo_out`)
- Mike's TT Audio Pmod connected to BIDIR terminal (`uio_out`)
- Some switches to the INPUT terminal (`ui_in`)

How to test

- Connect the necessary peripherals
- Provide a 25MHz clock to the top module `tt_um_demosiine_sda`
- Reset the design (if necessary)
- Enjoy the show :)
- Tweak the inputs to customize your show!

Input Configurations

The design takes in 8 digital inputs from the INPUT terminal to modify the on-screen graphics (and audio) to create funky visual effects. All inputs are expected to be LOW to render the output as shown in the default preview as shown below.

The effect of each input pin is presented in the table below:

Input Pin	Parameter	When LOW	When HIGH
`ui_in[7]`	Audio State	Play	Pause
`ui_in[6]`	Animation State	Run	Stop
`ui_in[5]`	Background Style	Black	Rolling RGB
`ui_in[4]`	Overlay Style	Cycle RGB	Rolling RGB
`ui_in[3]`	Overlay State	Enabled	Disabled
`ui_in[2]`	Big Sine State	Enabled	Disabled
`ui_in[1]`	Little Sine State	Enabled	Disabled
`ui_in[0]`	Colour Inversion	Normal	Negative

Previews

Provided below are a some of my favourite previews generated from DemoSiine along with the INPUT configuration which generated them:

Pinout

#	Input	Output	Bidirectional
0	Frame Positive / Negative	Video Red MSB	
1	Enable / Disable Little Sine Layer	Video Green MSB	
2	Enable / Disable Big Sine Layer	Video Blue MSB	
3	Enable / Disable Overlay	Video V-Sync	
4	Toggle Overlay Style	Video Red LSB	

#	Input	Output	Bidirectional
5	Toggle Background Style	Video Green LSB	
6	Run / Stop Animation	Video Blue LSB	
7	Play / Pause Audio	Video H-Sync	Audio Output

Figure 16: DemoSiine Default Video Output Preview

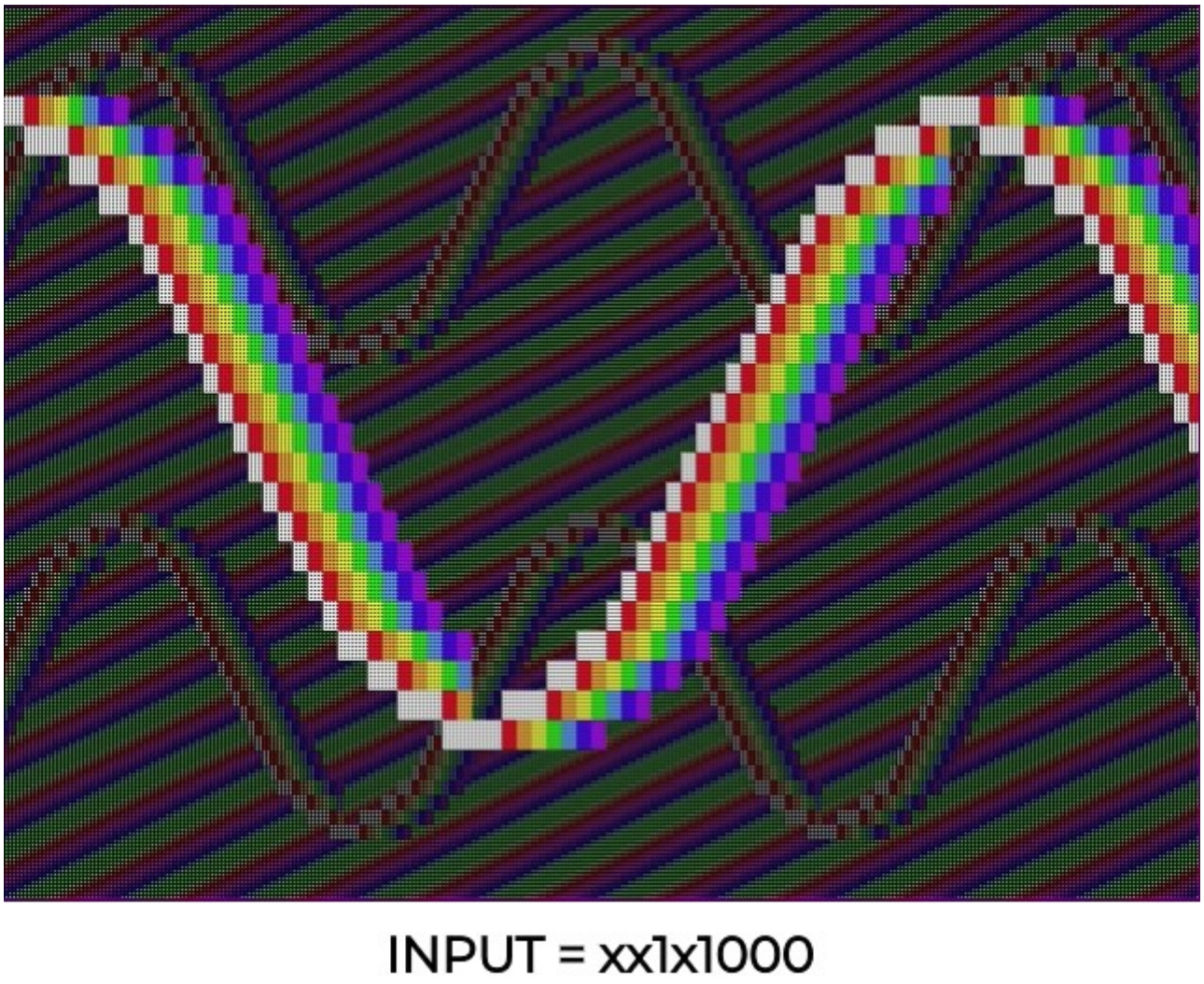


Figure 17: DemoSiine Video Output Preview 2

Figure 18: DemoSiine Video Output Preview 3

Figure 19: DemoSiine Video Output Preview 4

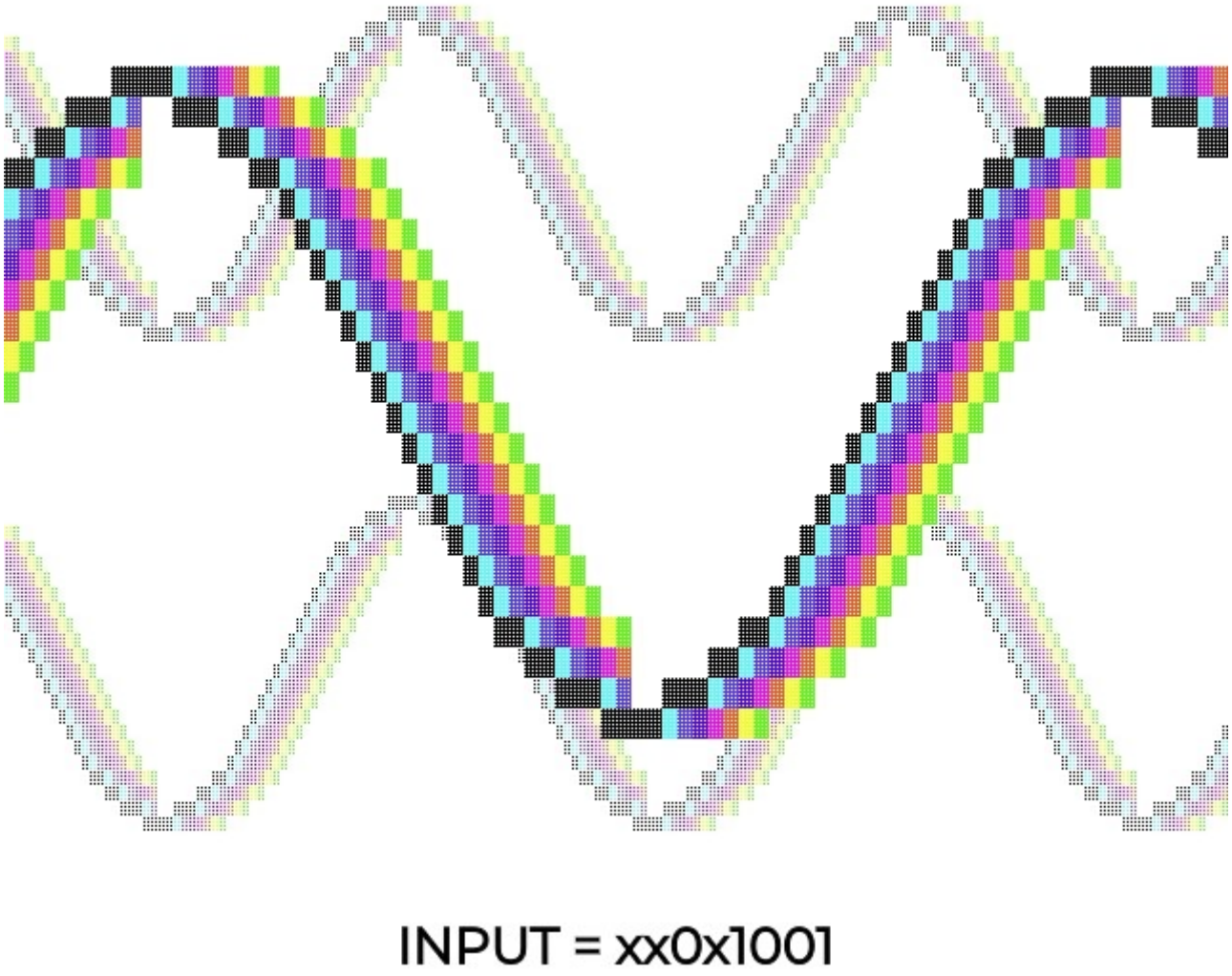


Figure 20: DemoSiine Video Output Preview 5

Figure 21: DemoSiine Video Output Preview 6

Rounding error [229]

- Author: Edwin Török
- Description: Competition entry
- GitHub repository
- HDL project
- Mux address: 229
- Extra docs
- Clock: 25250000 Hz

How it works

Idea This started out as an attempt to implement a ray tracer in 2 TT tiles. However, there isn't enough room for a proper one, precision has to be limited, which leads to unavoidable rounding errors.

So embrace rounding errors, and make them the primary feature!

The end result doesn't resemble a 3D scene, or a sphere, or in fact not even a properly rounded circle, but it has rounding errors! And that is the goal of this project now!

HardCaml The RTL was written using HardCaml, an OCaml DSL that emits Verilog. For convenience the generated Verilog is committed into the source tree, so no additional tools are needed.

I used registers with asynchronous reset, in theory it should be better for an area constrained design.

VGA signal generation

ModeLine VGA signal timing is described in "3. DMT Video Timing Parameter Definitions" in "VESA Display Monitor Timing Standard Version 1.0, Rev. 13", and is implemented in `src/generator/modeline.ml`. Examples on how to implement them on an FPGA are available in several places.

The code supports several resolutions, however to conserve area for the demo I've chosen only 640x480@59.94Hz, which has negative `hsync`/`vsync` polarities. This resolution would need a 25.175 MHz pixel clock, however that can't be produced exactly by the TT08 board, it can only approximate it using a PWM. Therefore, the design is configured to run at the nearest frequency that can be exactly generated:

25.25 MHz, which should be within the 0.5% acceptable by the standard. The ModeLine implemented is: `ModeLine &quot;640x480_59.94&quot; 25.175 640 656 752 800 480 490 492 525 -hsync -vsync`. (This has 59.94 refresh rate and not 60Hz due to the standard preferring NTSC and its 1.001 adjustment).

The design itself runs off the VGA pixel clock, as I didn't want to deal with potential clock domain crossing issues.

Counters There are 2 counters: one for H, and one for V synchronization pulses. When the H counter overflows it enables and increments the V counter for 1 cycle. This is implemented in `generator/vga.ml`, together with waveform expectation tests.

Both H and V counters start out in the visible area for convenience (we can directly use these counters as x/y coordinates, without needing to perform arithmetic in the circuit), then blank the colour signals for the duration of the front porch, synchronization signal and back porch. Although the monitor would recognize the hsync+vsync low as the start of a frame, this is equivalent, but offset by a few clocks.

R, G, B colours The demo supports 2-bit colours, and as usual these would be sRGB colours, not a linear scale. So we define an internal table indexed by 3 bits representing a linear RGB value, mapping to the sRGB bits.

A register is used for the output, both to avoid logic glitches becoming visible to the monitor, and to provide a reg to reg path that `OpenSTA` can use to compute setup/hold times.

Generating the colours When test mode is used (pin `ui[0]` set to `1`) the design outputs vertical colour bars with a white-black-white border. This doesn't have rounding errors, everything is sharp.

In normal mode (pin `ui[0]` is `0`) the "rounding error graphics" is rendered, see below.

Ray marching For an explanation of how ray marching works, see this ray marching tutorial. The "scene" is represented using signed distance functions. The "eye" Z coordinate is animated between 3.5 and 4.5 in 256 steps, where each frame is one step.

CORDIC Fixed point arithmetic with 9 bits of precision is used in the HDL, with the exponent tracked by the generator code to reduce register width (though this is not as good as tracking it in hardware, but that'd require more area). Vector normalization is implemented using the CORDIC implementation provided by HardCaml, configured to use 10 bits, and a limited number of iterations (4) to fit into the desired area. This works by rotating the vector until its angle is 0, and then rotating a second unit vector to match the rotation of the original. Or equivalently transform the original from rectangular to polar coordinates, overwrite the length with 1, and convert back from polar to rectangular. CORDIC is defined for 2D in the library, and I define a 3D wrapper based on rectangular to spheric coordinate conversions, although there would be ways to directly compute a 3D version of CORDIC, that is not implemented here.

This is implemented in `src/vecmath`.

GLSL ES "emulation" The low level operations are wrapped by a higher level embedded DSL that allows writing code quite similar to GLSL ES, with a very small number of operators: arithmetic (+, -, *, /), comparison (==, `&lt;&gt;`), `abs`, `min`, `max`, `clamp`, `length`, `distance`, `dot`, `normalize`, `reflect`.

Unfortunately the full renderer didn't fit into 2 tiles, so had to comment out quite a lot of the "GLSL" code (only 1 step of ray marching, no clamping, very simple gradient approximation), what is remaining does not resemble a sphere, or in fact it doesn't even look 3D.

OpenLane configuration The target density had to be increased to 98% to fit, and the setup slack margin setting had to be increased, see config.json. There are max slew and max fanout violations at 100C and 1.6V, but that shouldn't prevent the design from working at 25C and 1.8V.

The design was simulated using both tt-vgaviz and vgasim, although had to adjust the modeline for vgasim to recognize the standard one. A simple cocotb test which checked vsync/hsync generation was added post submission.

Simulating There is a `src/sim/vgasim.ml`, which generates a `demo.v` compatible with vgasim, this uses a different resolution though. `vgasim` has to be called with `-g 640x480`, and `videomode.h` needs to be edited to use `480 490 492 525` (don't know why it wants 521, that doesn't seem to be the standard timing).

Alternatively the `cocotb` test in `test/` can be run with `make -B WAVES=1`, and then tt-vgaviz can be used: `tt-vgaviz tb.vcd` (actually in FST format).

How to test

Configuration

- Provide a 25.25 MHz clock on the `clk` pin (RP2040 should be able to provide this with no jitter). Or if you can try 25.175 MHz instead, but this will have some jitter. YMMV.
- Power the design with at least 1.8V

Main demo

- Set pin `ui[0]` to 0 to run the default demo.
- Reset the design
- You should see circles moving slowly and large rounding errors:

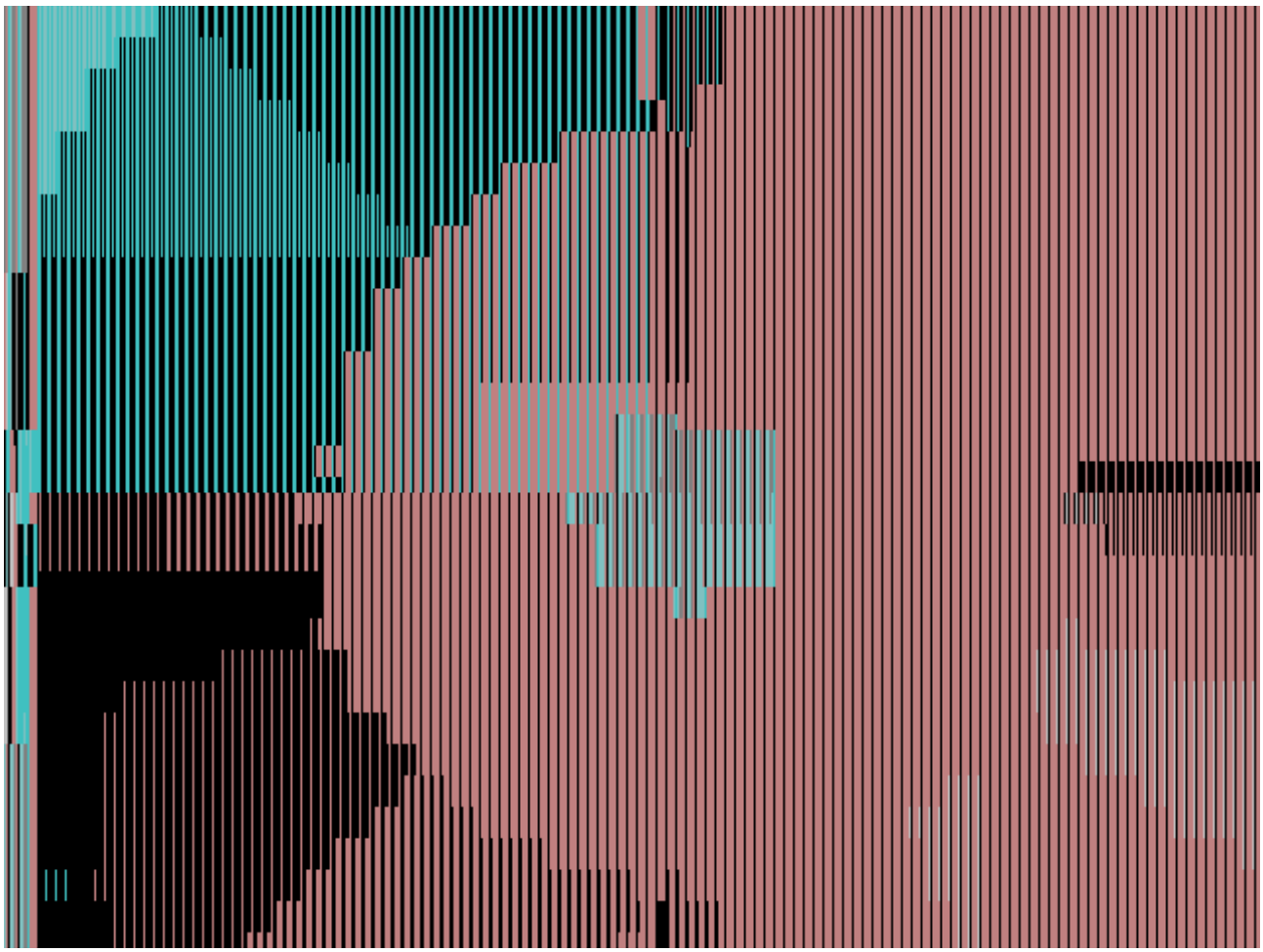
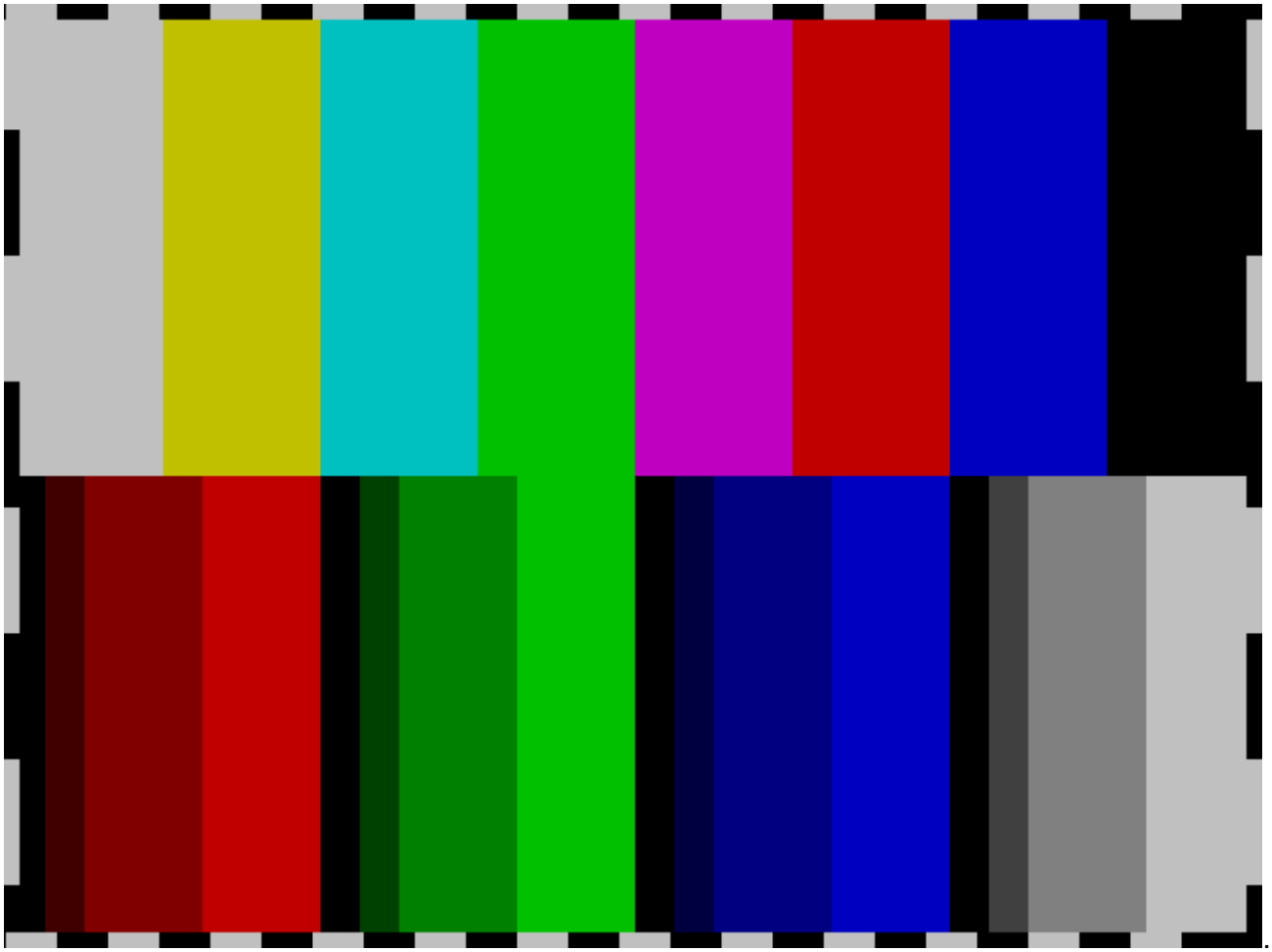


Figure 22: circles

Test mode

- Set pin `ui[0]` to 1 to show a test image with color bars.
- Reset the design again if desired
- You should see:



External hardware

Connect according to the Demoscene rules

- VGA output using Leo's VGA PMOD on pins `uo[0-7]`, connected to a monitor supporting 640x480 resolution.

Pinout

#	Input	Output	Bidirectional
0	test mode (0=no, 1=yes)	r1	
1		g1	

#	Input	Output	Bidirectional
2		b1	
3		vsync	
4		r0	
5		g0	
6		b0	
7		hsync	PWM output

VGA Pride [231]

- Author: Rebecca G. Bettencourt
- Description: A VGA demo for showing pride flags
- GitHub repository
- HDL project
- Mux address: 231
- Extra docs
- Clock: 0 Hz

How it works

Displays pride flags on the screen.

To add another flag, create a flag.v file and add it to `src/flag_index.v`, `test/Makefile`, and `info.yaml`, using the existing flags as examples.

How to test

Connect to a VGA monitor. Set the following inputs to change the displayed flag:

- `ui_in[7]` to display the first flag
- `ui_in[6]` to display the next flag
- `ui_in[5]` to display the previous flag
- `ui_in[4]` to display the flag whose index is on `uio_in`

Index	Flag
0	Rainbow flag, 6 stripes
1	Rainbow flag, 7 stripes
2	Rainbow flag, 8 stripes
3	Rainbow flag, 9 stripes
4	Philadelphia rainbow flag
5	Progress rainbow flag
6	Progress rainbow flag 2021 version
7	Trans pride flag
8	Abrosexual pride flag
9	Aceflux pride flag
10	Aegosexual pride flag
11	Agender pride flag
12	Androgyne pride flag
13	Androsexual pride flag

Index	Flag
14	Aporagender pride flag
15	Aroace pride flag
16	Aroflux pride flag
17	Aromantic pride flag
18	Asexual pride flag
19	Aspec pride flag
20	Bigender pride flag (pink purple white purple blue)
21	Bigender pride flag (blue white purple white pink)
22	Bigender pride flag (pink yellow white purple blue)
23	Bisexual pride flag
24	Ceterosexual pride flag
25	Demiandrogyne pride flag (pink purple blue)
26	Demiandrogyne pride flag (green white green)
27	Demiboy pride flag
28	Demifluid pride flag
29	Demiflux pride flag
30	Demigender pride flag
31	Demigirl pride flag
32	Demiromantic pride flag
33	Demisexual pride flag
34	Disability rights flag (gold silver bronze tricolor)
35	Disability rainbow flag
36	Gender-neutral pride flag
37	Genderfluid pride flag
38	Genderflux pride flag
39	Genderqueer pride flag
40	Greygender pride flag
41	Greysexual pride flag
42	Gynosexual pride flag
43	Intersex pride flag (purple circle)
44	Intersex pride flag (blue/pink gradient)
45	Thislesbianlife lesbian pride flag (pink and red)
46	Sadlesbeandisaster lesbian pride flag, 7 stripes (orange and pink)
47	Sadlesbeandisaster lesbian pride flag, 5 stripes (orange and pink)
48	Lydiandragon lesbian pride flag (violet crocus dill rose)
49	Maya Kern lesbian pride flag (violet rose crocus dill)
50	RebeccaRGB femme lesbian pride flag (violet lavender pink rose)
51	Littleender pride flag
52	Maverique pride flag
53	Leonis Ignis MLM pride flag (brown and blue)

Index	Flag
54	Vincian MLM pride flag, 7 stripes (green and blue)
55	Vincian MLM pride flag, 5 stripes (green and blue)
56	Vincian MLM pride flag (light blue and light green)
57	Multigender pride flag
58	Multisexual pride flag
59	Neptunic pride flag
60	Neutrois pride flag
61	Nonbinary pride flag
62	Objectum pride flag
63	Omnisexual pride flag
64	Pangender pride flag
65	Pansexual pride flag
66	Polyamory pride flag (blue, red, black with yellow pi)
67	Polyamory pride flag (blue, magenta, purple with yellow heart)
68	Polygender pride flag
69	Polysexual pride flag
70	Pomosexual pride flag
71	Proculsexual pride flag
72	IBM PS/2 pride flag
73	Queer pride flag
74	Trains pride flag (*Train Landscape*, Ellsworth Kelly, 1953)
75	Transfeminine pride flag
76	Transmasculine pride flag
77	Transneutral pride flag
78	Trigender pride flag
79	Unlabeled pride flag
80	Uranic pride flag
81	Voidpunk pride flag

External hardware

TinyVGA PMOD

Pinout

#	Input	Output	Bidirectional
0	address mode	R1	A0
1		G1	A1

#	Input	Output	Bidirectional
2		B1	A2
3		VSync	A3
4	set	R0	A4
5	prev	G0	A5
6	next	B0	A6
7	reset	HSync	A7

VGA Nyan Cat [233]

- Author: Andy Sloane
- Description: Displays the classic nyan.cat animation
- GitHub repository
- HDL project
- Mux address: 233
- Extra docs
- Clock: 25175000 Hz

VGA nyan cat

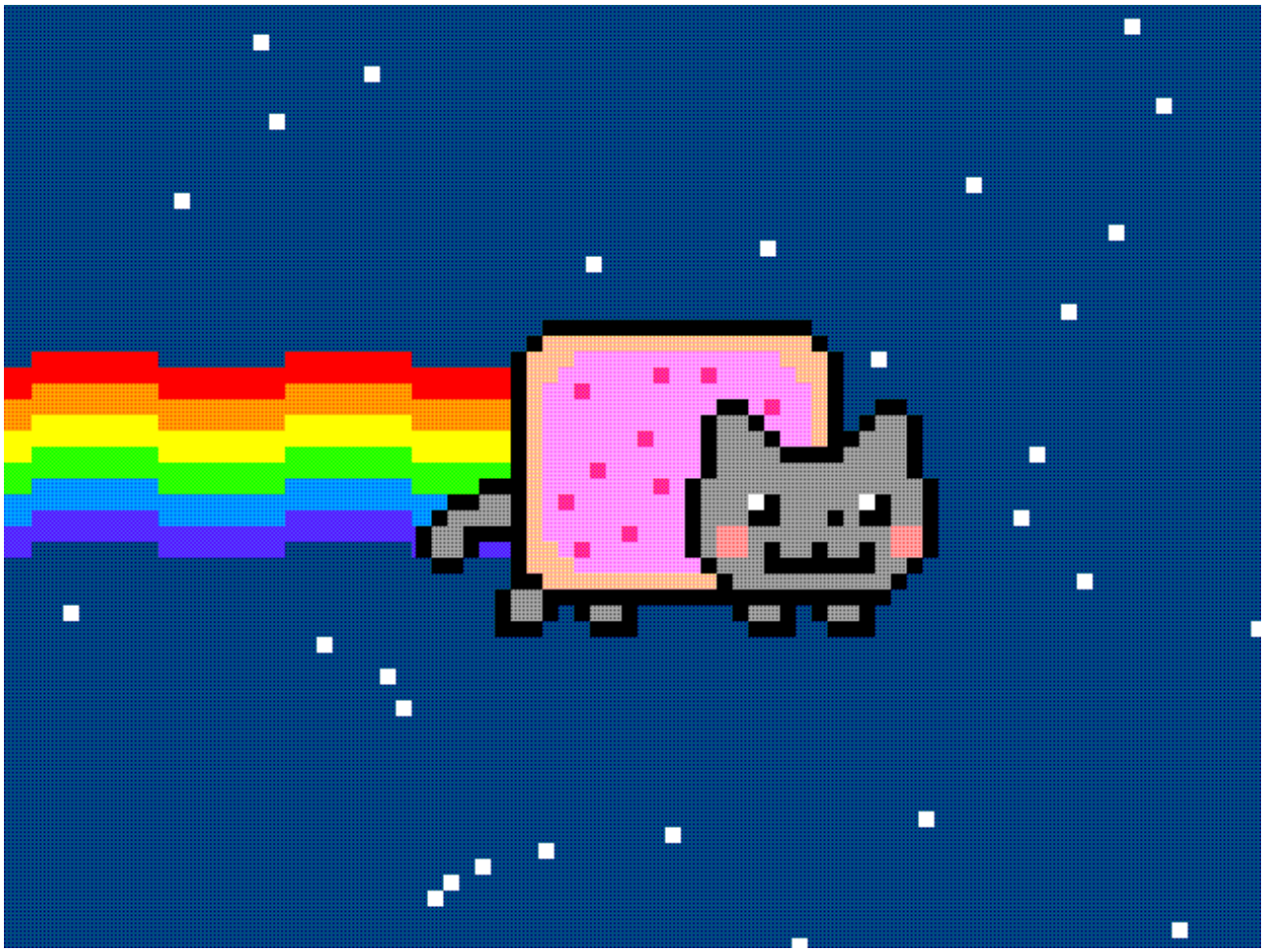


Figure 23: nyancat preview

How it works Outputs nyancat on VGA with music!

Colors and animation are all from the original nyan.cat site, using a 2x2 Bayer dithering matrix which inverts on alternate frames for better color rendition on the Tiny VGA Pmod.

Sound is generated from a MIDI file, split into melody and bass parts. Melody and bass are each square waves mixed with a simple exponential decay envelope, which is then fed to a low-pass filter and then a sigma-delta DAC.

This was designed to fit into 1 tile, and it *almost* did – the cells take up about 93% of 1 tile, but detailed routing doesn't finish. With the deadline approaching I was forced to grow it to 1x2, so I threw in a little easter egg.

How to test Set clock to 25.175MHz or thereabouts, give reset pulse, and enjoy

External hardware TinyVGA Pmod for video on o[7:0].

1-bit sound on io[7], compatible with Tiny Tapeout Audio Pmod, or any basic ~20kHz RC filter on io7 to an amplifier will work.

Pinout

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

Flame demo [235]

- Author: Konrad Beckmann & Linus Mårtensson
- Description: Flame demo
- GitHub repository
- HDL project
- Mux address: 235
- Extra docs
- Clock: 25000000 Hz

Flame - Konrad & Linus tinytapeout08 demo compo entry

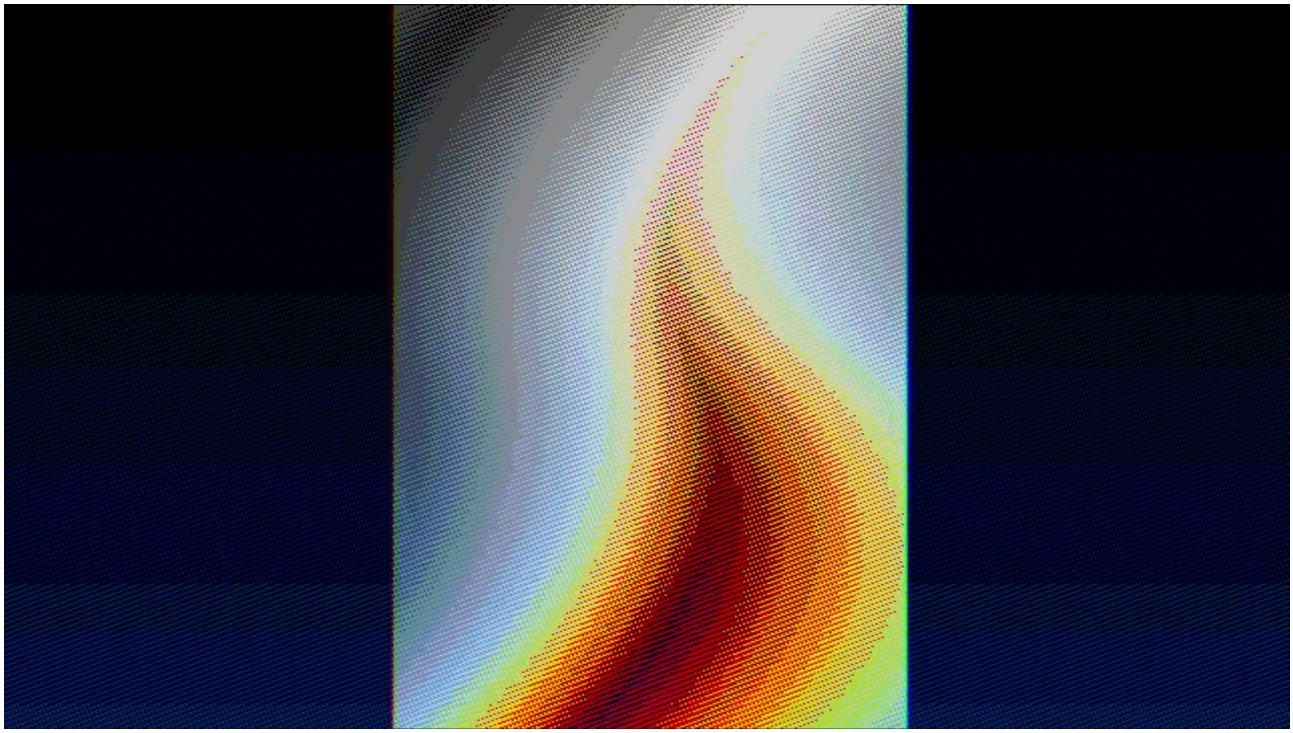


Figure 24: preview

How it works It shows a flame and plays audio. The VGA output is standard 640x480@60Hz, audio is simple 1 bit PWM.

How to test Run clock at 25MHz, connect VGA and sound Pmods, and give it a reset pulse.

External hardware Follows the democompo hardware rules:

TinyVGA Pmod for video on o[7:0].

1-bit sound on io[7], compatible with Tiny Tapeout Audio Pmod, or any basic ~20kHz RC filter on io7 to an amplifier will work.

Pinout

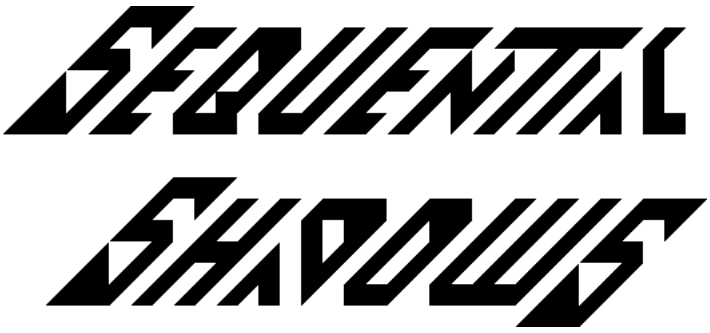
#	Input	Output	Bidirectional
0	ui_in[0]	ui_out[0]	uio_out[0]
1	ui_in1	ui_out1	uio_out1
2	ui_in2	ui_out2	uio_out2
3	ui_in[3]	ui_out[3]	uio_out[3]
4	ui_in[4]	ui_out[4]	uio_out[4]
5	ui_in[5]	ui_out[5]	uio_out[5]
6	ui_in[6]	ui_out[6]	uio_out[6]
7	ui_in[7]	ui_out[7]	uio_out[7]

Sequential Shadows Deluxe [TT08 demo competition] [258]

- Author: Toivo Henningsson
- Description: My contribution to the TT08 demo competition, extended version
- GitHub repository
- HDL project
- Mux address: 258
- Extra docs
- Clock: 50400000 Hz

Intro

Curly / Medieval presents



my contribution to the Tiny Tapeout 8 demo competition. Code, graphics, and music by Curly (Toivo Henningsson) of Medieval.

This is the deluxe version, with Pmod VGA RGB444 output support and a few changes from the original, in 2x2 tiles compared to the original's 1x2.

The demo can be seen at https://youtu.be/pkiTu3iLA_U (captured from a Verilator simulation).

How it works

See the documentation for the original version: https://github.com/toivoh/tt08-demo/blob/main/docs/info.md / Tiny Tapeout 8 project [770]. The deluxe version adds some tweaks such as a shadow beneath the logo, and credits.

How to test

Plug in a TinyVGA compatible Pmod on the TT08 demo board's out Pmod. Plug in a Pmod compatible with Mike's audio Pmod on the TT08 demo board's bidir Pmod. Set all inputs to zero to get the default behavior. **Warning: The default behavior includes some flashing ligts.** Set `v_bass_off` and `v_drums_off` (keep `ui_in` at 3 instead of 0) to remove flashing. The demo starts directly after reset.

This demo is best viewed with the monitor rotated 90 degrees, with the left side facing down.

Inputs There is no guarantee that changing the inputs after reset is released works as intended, but it probably does. Some of the inputs provide options on how the demo is run:

- `v_bass_off`: Setting this high reduces flashing when the audio visualizer is on by turning off the bass.
- `v_drums_off`: Setting this high reduces flashing when the audio visualizer is on by turning off the drums.
- `v_bass_low`: Setting this high keeps the bass at its default octave even when the audio visualizer is on, which increases flashing.
- `pause`: While this is high, the demo is paused and the sound is turned off. Can probably be used to start the demo paused.
- `step_frame`: While this is high, the the demo advances one frame per cycle. Used for testing.
- `rgb444_mode`: Setting this high sets the output to RGB444 mode instead of the default RGB222
- `pmod_vga_pinout`: Setting this high enables the alternative Pmod VGA pinout.
  - The `t_` outputs are used when `pmod_vga_pinout` is low. This fits the TinyVGA Pmod pinout. (`p_` only outputs are not driven.)
  - The `p_` outputs are used when `pmod_vga_pinout` is high. This fits the Pmod VGA pinout.
- `logo_shadow_off`: When high, removes the logo's shadow (like in the non-deluxe version).

If using A Pmod VGA as output, you can set `rgb444_mode` to increase the color depth, or leave it unset to get the original RGB222 experience. Please try both: which to prefer is a matter of taste.

For the demo competition, only use a Pmod VGA if you have one and can get sound output while using it. If using Pmod VGA, set `pmod_vga_pinout`, and you can set

`rgb444_mode` as well. Don't set any other inputs. If using TinyVGA for output, set all inputs to zero.

External hardware

This project needs

- either
  - a TinyVGA VGA Pmod.
  - Mike's audio Pmod.
- or a Pmod VGA
  - There is no ready option to output the audio in this case, but it's still present on the same pins, so you may be able to get it out with some creative wiring, and e g feed it to Mike's audio Pmod.

The choice of pinout is controlled by the `pmod_vga_pinout` input.

Pinout

#	Input	Output	Bidirectional
0	v_bass_off	t_R1 / p_R0	p_G0
1	v_drums_off	t_G1 / p_R1	p_G1
2	v_bass_low	t_B1 / p_R2	p_G2
3	pause	t_vsync / p_R3	p_G3
4	rgb444_mode	t_R0 / p_B0	p_hsync
5	pmod_vga_pinout	t_G0 / p_B1	p_vsync
6	logo_shadow_off	t_B0 / p_B2	audio_out_n
7	step_frame	t_hsync / p_B3	audio_out

No Time For Squares, IHP edition [266]

- Author: Tommy Thorn
- Description: It's a 12-hour clock, drawn with triangles rendered by a race-the-beam triangle render
- GitHub repository
- HDL project
- Mux address: 266
- Extra docs
- Clock: 31500000 Hz

How it works

The main part is a state machine that incrementally rasterizes three triangles based on their edge equations while generating the timing for VGA. Every frame the second, minute, and hour are updated which are indexed into tables to get the edge equations corresponding to the moving hands.

How to test

Hook up the TinyVGA to the output (not bidir) port and that to a VGA monitor. Marvel at the display. Assert input pin 6 and 7 to move the minute and hour hand respectively.

External hardware

TinyVGA is required, otherwise this is a pretty boring design.

Pinout

#	Input	Output	Bidirectional
0	debugsel0	R1	
1	debugsel1	G1	
2	debugsel2	B1	
3	debugsel3	vsync	
4	unused0	R0	
5	unused1	G0	
6	minute, advance minute	B0	

#	Input	Output	Bidirectional
7	hour, advance hour	hsync	

Simon's Caterpillar [289]

- Author: htfab
- Description: Port of Caterpillar Logic to Simon Says PMOD
- GitHub repository
- HDL project
- Mux address: 289
- Extra docs
- Clock: 50000 Hz

How it works

Simon's Caterpillar is a re-implementation of the game Caterpillar Logic by Fuks Michael targeting Tiny Tapeout with the Simon Says PMOD.

The game consists of 20 levels. Each level has a secret rule that is valid for certain sequences of colors. For instance, if the rule is "contains exactly two yellow tokens" then blue-yellow-green-yellow is a valid sequence and yellow-red-blue is an invalid one.

A new level starts in exploration mode. You can ask an unlimited number of questions where you learn whether a particular sequence is valid or not. Once you know the rule you can activate challenge mode. Now the roles are reversed and the game asks you 15 questions. If you can answer all of them correctly, you advance to the next level.

How to test

Set the clock to 50 kHz. Activate and reset the project. The 7-segment display should indicate level 1 and only the blue led should light up. You are in exploration mode.

Exploration mode A sequence of up to 7 colors can be typed into the buffer with short presses of the buttons. The leds indicate the sequence status in real time:

- red: sequence is invalid
- green: sequence is valid
- blue: buffer is empty
- yellow: buffer is full

(The empty sequence is neither valid nor invalid.)

Further operations are available as long button presses or a combination of two buttons:

- long-press red: clear buffer
- long-press yellow: erase last color from buffer ("backspace")
- long-press blue: show buffer contents (as a series of led flashes)
- long-press green: activate challenge mode
- short-press green & yellow: show a random valid sequence (and load into buffer)
- short-press red & blue: show a random invalid sequence (and load into buffer)
- short-press blue & yellow: switch to next level
- short-press red & green: switch to previous level
- short-press green & blue: toggle sound

Challenge mode A sequence of up to 6 colors is shown as a series of led flashes. Press the green or red button to mark it as valid or invalid respectively.

Each correct answer adds a notch (turns on a new segment on the 7-segment display). After the 15th one the next level is loaded. An incorrect answer switches back to exploration mode.

Other keys and combinations:

- short-press or long-press blue: repeat the current question
- short-press red & yellow: switch back to exploration mode
- short-press blue & yellow: add a notch
- short-press red & green: remove a notch
- short-press green & blue: toggle sound

External hardware

Simon Says PMOD

Pinout

#	Input	Output	Bidirectional
0	red button	red led	segment A
1	green button	green led	segment B
2	blue button	yellow led	segment C
3	yellow button	blue led	segment D
4	display polarity	speaker	segment E
5		digit 1	segment F
6		digit 2	segment G
7			

TT08 Pachelbel's Canon demo [291]

- Author: Mike Bell
- Description: Tiny Tapeout visuals with the classic Canon soundtrack
- GitHub repository
- HDL project
- Mux address: 291
- Extra docs
- Clock: 36000000 Hz

How it works

The project plays Pachelbel's Canon along with some fun visuals.

How to test

Set the inputs to 0, clock at 36MHz.

External hardware

Tiny Tapeout Audio Pmod in the bidir Tiny VGA Pmod in the output

Pinout

#	Input	Output	Bidirectional
0		R1	
1		G1	
2		B1	
3		vsync	
4		R[0]	
5		G[0]	
6		B[0]	
7		hsync	PWM output

Demo by a1k0n [293]

- Author: Andy Sloane
- Description: Tiny Tapeout demo competition entry
- GitHub repository
- HDL project
- Mux address: 293
- Extra docs
- Clock: 48000000 Hz

a1k0n's tinytapeout08 demo compo entry

Figure 25: preview

How it works It's a standalone VGA+sound demo that fits in two tiles; you'll just have to see. The demo is short, looping after about 25 seconds.

This was developed with a 48MHz clock, so it's in a funky VGA video mode – it's standard 640x480@60Hz VGA timing and 4:3 aspect ratio, but with 1220 horizontal pixels instead of 640. All graphics are dithered down to RGB222 with a Bayer matrix which alternates each frame. Because of the dithering and the weird resolution, it looks best on a real CRT, but any VGA monitor ought to work.

Sound is generated using a 16-bit sigma-delta DAC on io7 from an internal 3-channel synth (triangle, noise, and square waves).

Sines and cosines are generated by an old HAKMEM trick which generates a slightly off-center circle but that doesn't matter in this application:

```
cos_new = cos - (sin>>k)
sin_new = sin + (cos_new(!)>>k)
```

The plane is rendered by doing a bit-by-bit non-restoring division of the y coordinate during the horizontal blanking interval to find a fixed point reciprocal, which is then used as an x increment for the plane u coordinate. As a drastic simplification, the plane v coordinate is *also* the x increment value (when you do the math, it turns out they are proportional).

Starfield is generated by an LFSR that increments every line which provides an x-offset and speed for each star by picking out individual bits of the LFSR state.

The "TT08" logo uses the outline of an old demo font, but the actual coloring is procedural as it would take too much combinational logic to reproduce exactly.

Soundtrack is a riff on "Crooner" by Drax/Vibrants, composed as a bunch of text in a Python script with limitations on song structure and octave range. Kick drum and bass share the triangle channel, lead arpeggios on square, and hihat noise.

I'm not super happy about the "programmer colors" everywhere, but I ran out of room trying to add palettes.

How to test Run clock at 48MHz, connect VGA and sound Pmods, and give it a reset pulse (falling edge).

External hardware Follows the democompo hardware rules:

TinyVGA Pmod for video on o[7:0].

1-bit sound on io[7], compatible with Tiny Tapeout Audio Pmod, or any basic ~20kHz RC filter on io7 to an amplifier will work.

Pinout

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

VGA Drop (audio/visual demo) [295]

- Author: ReJ aka Renaldas Zioma, eriQue aka Erik Hemming, Matthias Kampa
- Description: Tiny 8 part Megademo! TBL[Nesnausk]SonikClique
- GitHub repository
- HDL project
- Mux address: 295
- Extra docs
- Clock: 25200000 Hz

How it works

VGA signal generator

How to test

We are learning how VGA and Sky130 works here

External hardware

VGA PMOD

Pinout

#	Input	Output	Bidirectional
0		R1	Audio (PWM)
1		G1	Audio (PWM)
2		B1	Audio (PWM)
3		VSYNC	Audio (PWM)
4		R0	Audio (PWM)
5		G0	Audio (PWM)
6		B0	Audio (PWM)
7		HSYNC	Audio (PWM)

Warp [297]

- Author: sylefeb
- Description: Demo on TinyTapeout? Let's do something!
- GitHub repository
- HDL project
- Mux address: 297
- Extra docs
- Clock: 25000000 Hz

Warp

> Please make sure to watch the demo for a few minutes as various effects play out before it loops. At start it waits for a few seconds to ensure VGA sync is achieved.

How it works

> *But does it work?*

Preface This demo is written in Silice, my HDL. Here is the actual source. Silice now fully support TinyTapeout as a build target.

Graphics The core effect is a classical tunnel effect ; however this is normally done with a "huge" pre-computed table having one entry per-pixel. So I thought it'd be challenging and fun to do it while racing the beam! Plus, I really like this effect.

There are several tricks at play: a shallow CORDIC pipeline to compute an *atan* and *length*, and a few precomputed $1/x$ distances to interpolate between – these form keypoint rings along the tunnel. All the effects are then obtained by combining multiple layers in various ways (like a *tunnel effect processor* which registers can be configured for various effects).

The demo uses a lot of dithering (ordered Bayer dithering) given the output is RGB 2-2-2. All computations are grayscale and the RGB lense effect is obtained by delaying the grayscale values using the tunnel distance in R and B.

I also tried to make the logo interesting by deviating from a classical pixelated look. It is composed of tiles, either full or triangular, with a comparator and a bit of logic to do all four possible triangles.

The tunnel viewpoint change is obtained simply by shifting the tunnel center. I was surprised that a simple translation gives such a convincing effect (almost as if the viewpoint was rotating).

The 'blue-orange' tunnel effect is obtained through temporal dithering, one frame being the standard tunnel, the other the rotated tunnel. This gets combined with the RGB lense distortion, achieving the final look.

Audio I am no musician, so making a soundtrack was a challenge for me, but that's something I've always wanted to try. In the end it was a very enjoyable part of the design, and I was surprised at how compact this can be made, the soundtrack using perhaps around 10% of the entire design.

I tried to make a track that matches the spirit and rhythm of the graphics. It is what is is, but I'm happy that there's sound at all!

How to test Plug the VGA+audio PMODs to the board and run. Maybe it works?

Simulation of both audio and video can run on an ECPIX5, with the Diligent VGA PMOD on ports 0,1 and an I2S audio PMOD on port 2 (upper row). The audio also runs on an ULX3S using its DAC (but no video in this case).

External hardware

- VGA PMOD
- Audio PMOD

See https://tinytapeout.com/competitions/demoscene/

Pinout

#	Input	Output	Bidirectional
0		R1	
1		G1	
2		B1	
3		VS	
4		R0	
5		G0	
6		B0	
7		HS	Audio

Bouncy Capsule [299]

- Author: htfab
- Description: Demoscene project featuring... well, a bouncy capsule
- GitHub repository
- HDL project
- Mux address: 299
- Extra docs
- Clock: 25000000 Hz

How it works

This is an entry to the Tiny Tapeout demoscene competition

How to test

- Attach the standard PMODs
- Run the clock at 25 (or 25.175) MHz
- Reset the design
- Sit back and enjoy
- Optionally change the input switches

External hardware

- Tiny VGA PMOD
- TT Audio PMOD (or MuseLab's Audio PMOD)

Pinout

#	Input	Output	Bidirectional
0	Pause kinematics	Tiny VGA R1	PDM audio out
1	Reset kinematics	Tiny VGA G1	PDM audio out
2	Mute sound	Tiny VGA B1	PDM audio out
3	Kill sound	Tiny VGA VSync	PDM audio out
4	Hide background	Ting VGA R0	PDM audio out
5	Hide text	Tiny VGA G0	PDM audio out
6	Lock colors	Tiny VGA B0	PDM audio out
7	No re-orientation	Tiny VGA HSync	PDM audio out

raybox-zero TTIHP0p2 edition [326]

- Author: algofoogle (Anton Maurovic)
- Description: TTIHP0p2 experimental resub of 'simple VGA ray caster game demo' from TT07
- GitHub repository
- HDL project
- Mux address: 326
- Extra docs
- Clock: 25175000 Hz

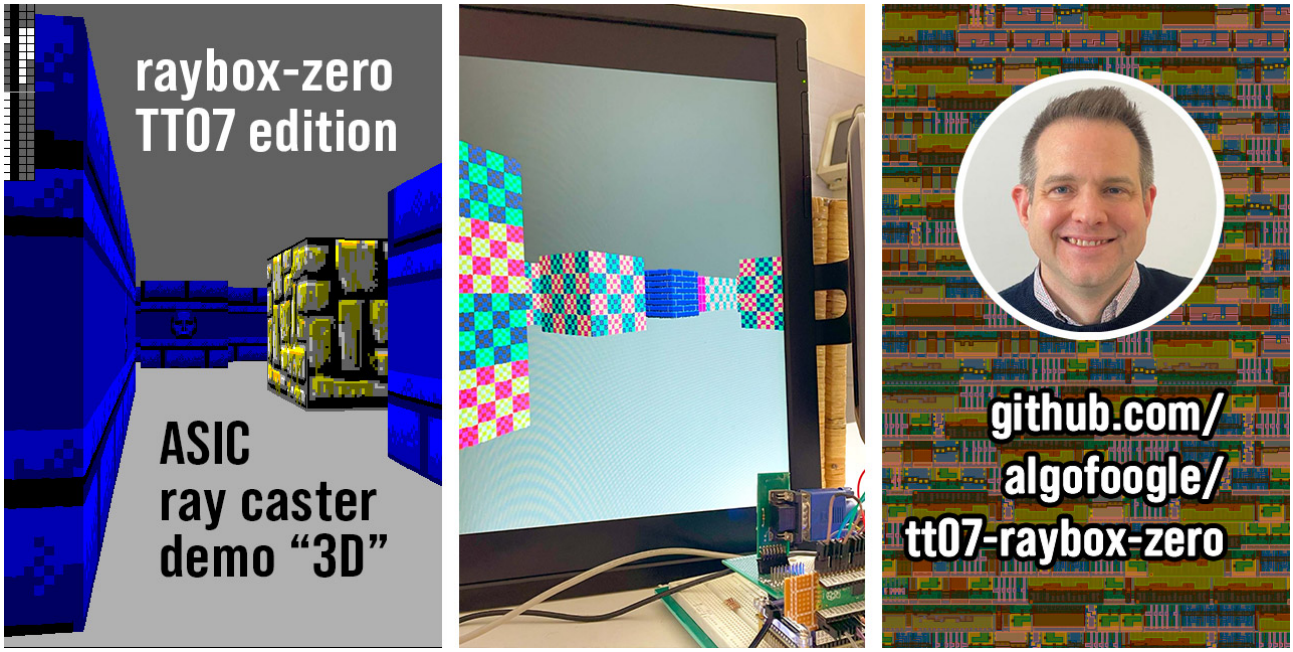


Figure 26: TT07 raybox-zero showing 3D views in simulation and on an FPGA

How it works

This resubmission of tt07-raybox-zero on TTIHP0p2 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.

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.

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	red1	Out: tex_csb
1	SPI in: pov_mosi	green1	Out: tex_sclk
2	SPI in: pov_ss_n	blue1	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

VGA donut [330]

- Author: Andy Sloane
- Description: Renders a 3D torus on a VGA display
- GitHub repository
- HDL project
- Mux address: 330
- Extra docs
- Clock: 48000000 Hz

VGA Donut

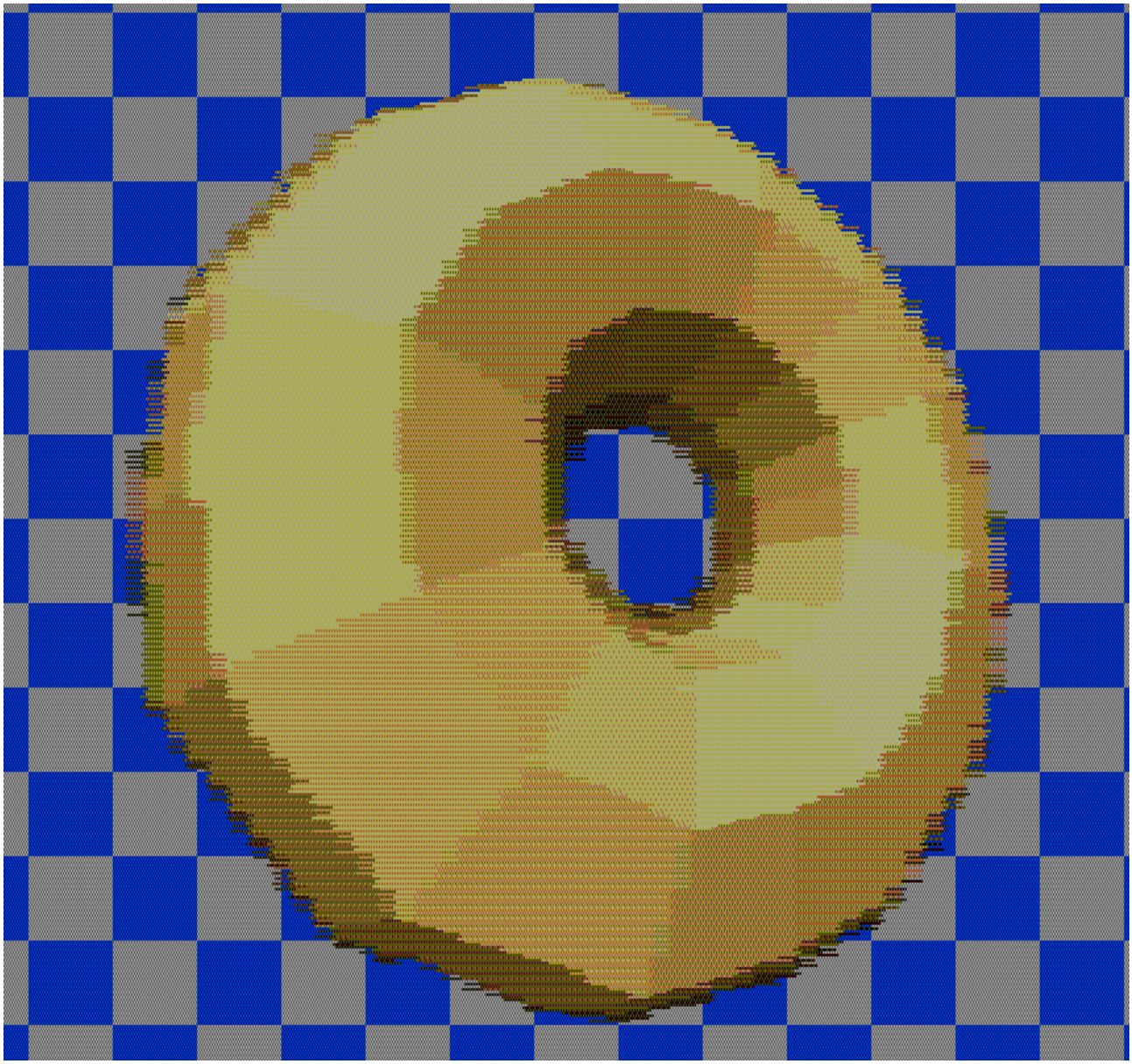


Figure 27: preview

How it works Renders a faceted donut to a VGA monitor.

Like my other demo on tt08, this runs in a weird VGA resolution: 1220x480, but still 4:3 aspect ratio like 640x480.

Interestingly, it is not actually rendering any polygons; this is sphere traced (AKA raymarched), using a CORDIC unit to calculate the distance between a point and the surface of the torus. But, because we don't have much time (we're racing the VGA beam!), we do just two or three CORDIC iterations, which causes the donut surface to actually become polyhedral. This trick was accidentally discovered by Bruno Levy while playing with a C version of my original donut code and I had to try it out in Verilog – so here we are.

The reason it has such low horizontal resolution is because it's doing 16 ray marching steps per "pixel", with five CORDIC iterations unrolled into one clock cycle (three iterations for the major axis, and two for the minor axis).

In order to fit this into 2x2 TinyTapeout tiles, a lot of sacrifices were made; for one, it doesn't have a multiplier so the ray steps are by approximate orders of magnitude. New donut "pixels" are rendered every 16 clock cycles, so the demo makes heavy use of dithering in both space and time – the video looks much better than the screenshot above.

How to test Connect VGA Pmod to output, set clock to 48MHz, and give it a reset pulse.

External hardware TinyVGA Pmod for video on o[7:0].

Pinout

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

maddihp [353]

- Author: Jonny Edwards
- Description: a multi use multi-hit dot product accelerator V2
- GitHub repository
- HDL project
- Mux address: 353
- Extra docs
- Clock: 0 Hz

How it works

This is a simple circuit to calculate:

- a vector dot product ie the sum of `w_i*x_i` where `i` can be anything up to about 40 (`insn=2`)
- Minimum of a list of data (`insn=0`)
- Maximum of a list of data (`insn=1`)

It has been designed as a coprocessor. The data is first added by setting `load=1` and then supplying the data for the dot product the `index` and `data`. Each set is a `w`,`x` pair. Its a 4 bit system and runs when `run=1` and needs at least 16 clock cycles produce the answer. The answer is 12 bit value.

How to test

I've tested this using a verilator simulation included below - I like the cpp workbench for this. The testing has been mainly for numerical stability.

External hardware

I intend for this to be driven by the RP2040 and to work as a "coprocessor" for vector calculations Other.

#	Input	Output	Bidirectional

Pinout

#	Input	Output	Bidirectional
0	index[0]	out[0]	out[8]
1	index1	out1	out[9]
2	index2	out2	out[10]
3	index[3]	out[3]	out[11]
4	data[0]	out[4]	instruction [0]
5	data1	out[5]	instruction 1
6	data2	out[6]	load
7	data[3]	out[7]	run

Multimode Modem [355]

- Author: Joerdson Silva
- Description: Performs digital modulation and demodulation in amplitude, frequency and phase schemes.
- GitHub repository
- HDL project
- Mux address: 355
- Extra docs
- Clock: 10000000 Hz

How it works

The multimode modem uses a clock signal to generate digitized signals over time, in sinusoidal format. From this digitized sinusoid, the modulation process is applied using different methods for each scheme, implemented through specific internal blocks to perform modulations ASK (switching the amplitude of the sine wave), FSK (switching the frequency of the sine wave through a digital signal modulator) and PSK (phase coding). In the demodulation stage, these three modulation schemes are analyzed to recover the original information, manifesting as '0' or '1' values that reflect the data signal already restored after the process.

Inputs and Outputs

The multimode modem has the following inputs and outputs: | Type | Function | Size | |——|———|——| | Input | clk | 1 bit | | Input | rst_n | 1 bit | | Input | sel | 2 bits | | Output | mod_out | 7 bits | | Output | demod_out | 1 bit |

How to test

Apply a clock of 10 MHz. Next, apply a "1" logic level "reset" signal to synchronize the modem system and then make the "reset" signal a "0" logic level. Then select the type of modulation to be used, according to the sequence below. After selecting the modulation type, the modulated signal is expressed at the "mod_out" output and the demodulated signal at the "demod_out" output.

- Sel = "01" <= ASK modulation and demodulation
- Sel = "10" <= FSK modulation and demodulation
- Sel = "11" <= PSK modulation and demodulation

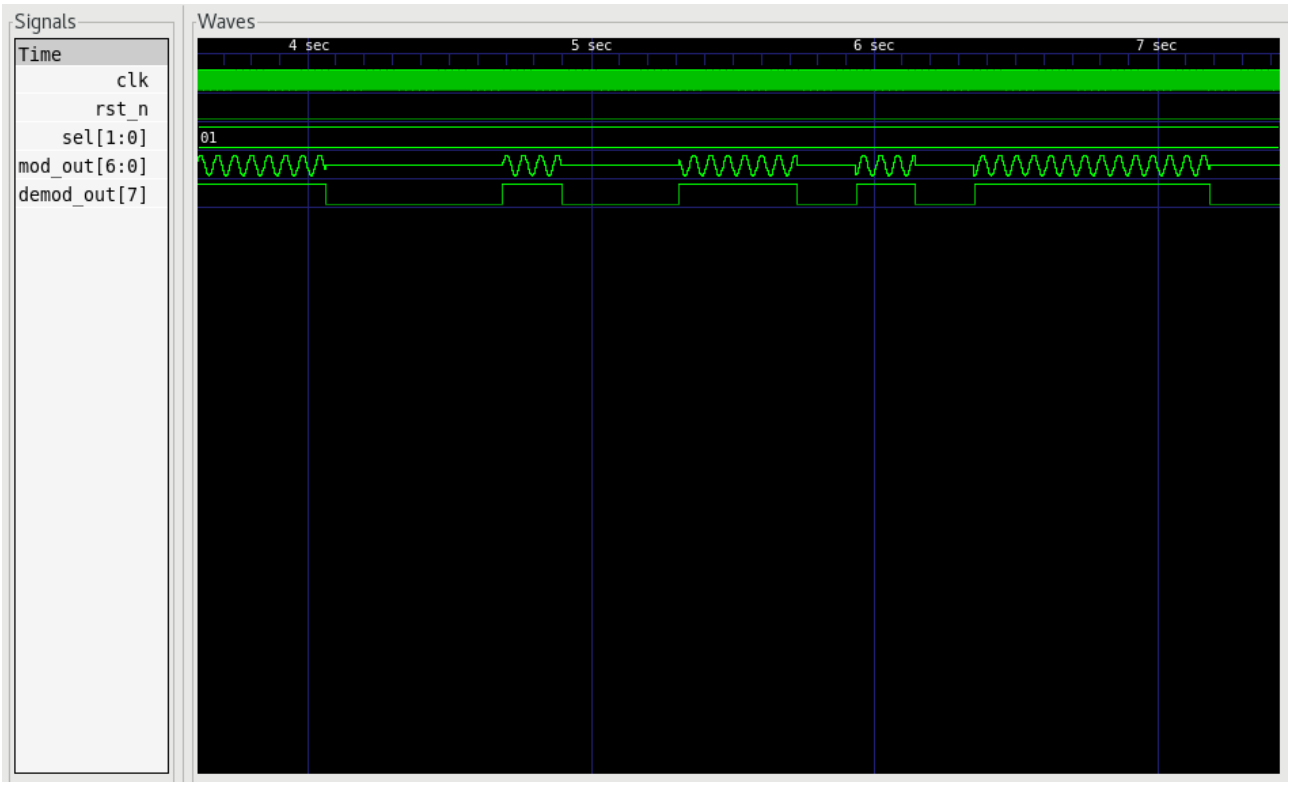


Figure 28: 01

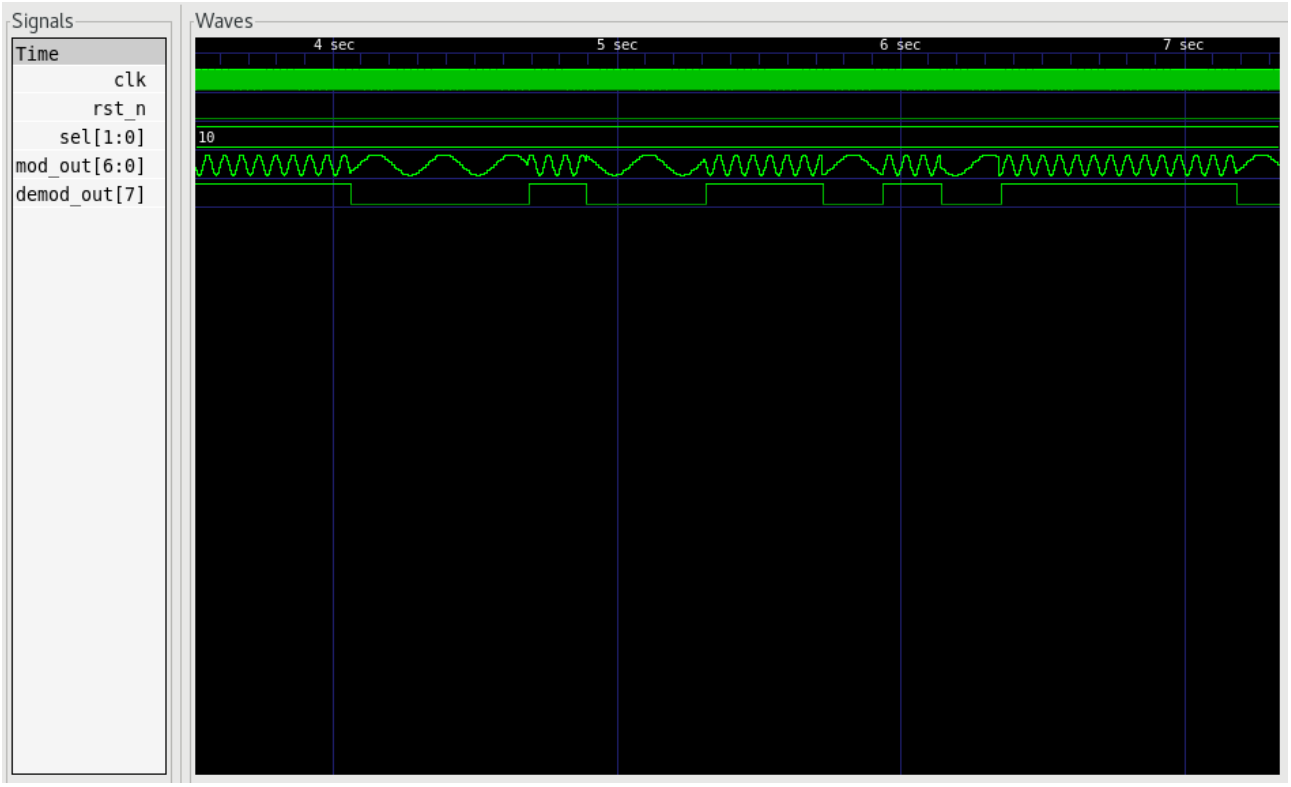


Figure 29: 10

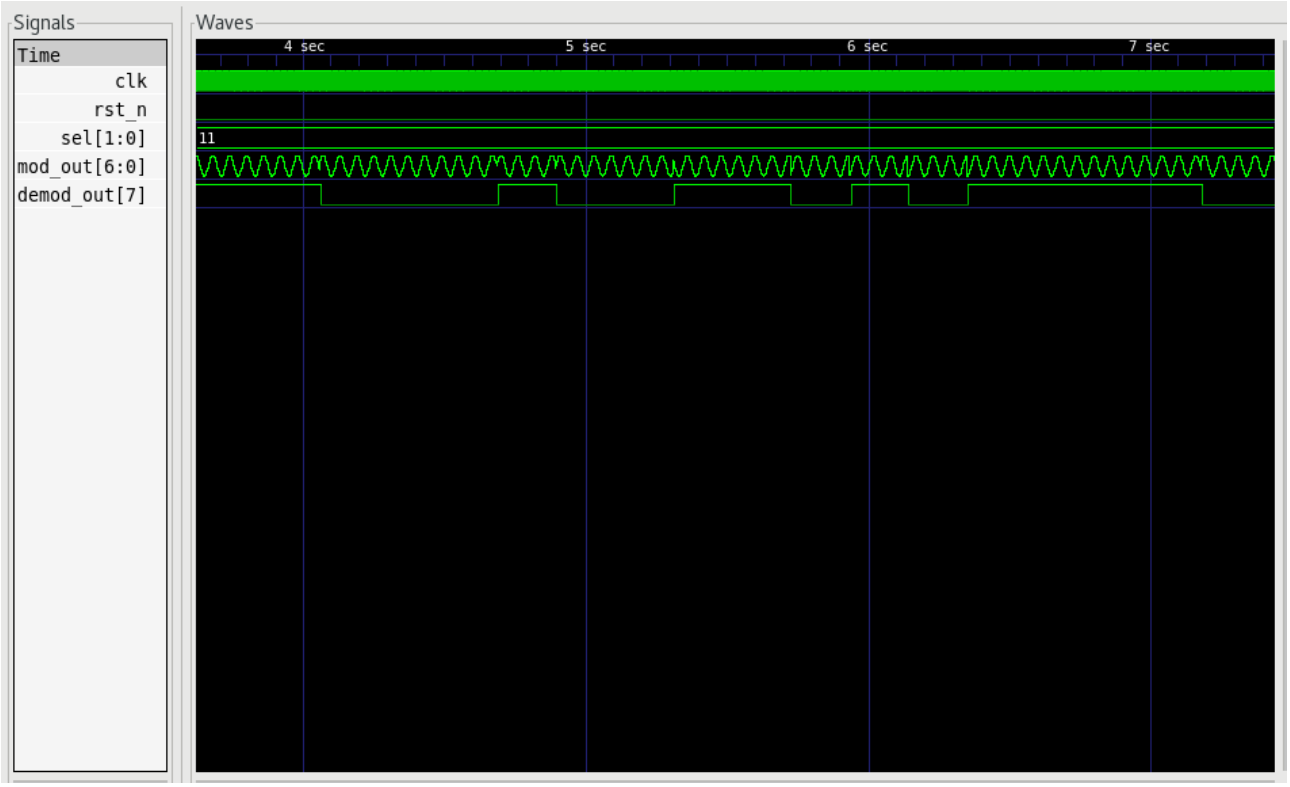


Figure 30: 11

External hardware

Analog Discovery 2.

Pinout

#	Input	Output	Bidirectional
0	selection_0	modulation_out_0	
1	selection_1	modulation_out_1	
2		modulation_out_2	
3		modulation_out_3	
4		modulation_out_4	
5		modulation_out_5	
6		modulation_out_6	
7		demodulation_out	

Frequency Counter SSD1306 OLED [357]

- Author: Pawel Sitarz (embelon)
- Description: Simple Frequency Counter displaying result on SSD1306 SPI OLED
- GitHub repository
- HDL project
- Mux address: 357
- Extra docs
- Clock: 1000000 Hz

How it works

Project measures frequency on ui[0] input by counting pulses during 100ms periods. Measured frequency is then displayed on graphical 128x32 pixels OLED display in form of emulated 7-segment display.

How to test

Internal logic needs 1MHz clock (to be generated by RPi Pico)

- Connect PMOD OLED display to see measurement
- Connect unknown frequency signal to be measured to ui[0]

External hardware

Freqquency is displayed on 128x32 OLED display with SSD1306 controller: PMOD OLED

Pinout

#	Input	Output	Bidirectional
0	clk_x - measured frequency input	OLED nRST	
1		OLED nVBAT	
2		OLED nVDC	
3		OLED nCS	
4		OLED Data/Command	
5		OLED CLK	
6		OLED Data Out	

#	Input	Output	Bidirectional
7			

I2C BERT [359]

- Author: Darryl Miles
- Description: I2C Bit Error Rate Test
- GitHub repository
- HDL project
- Mux address: 359
- Extra docs
- Clock: 10000000 Hz

How it works

Documentation is up with asciidoc on https://github.com/dlmiles/tt05-i2c-bert

Issue synchronous reset, ensure interface inputs are set to zero. Power-on-reset configuration is possible via the input pins, see documentation.

This design is an I2C peripheral that implements an 8-bit ALU over I2C. The purpose of the ALU is to allow pattern testing to occur and read back the accumulated result.

There are a few clocking modes, the default uses SCL pin as per the standard.

Connection to I2C interface:

- uio2 = SDA (should be direct to RP2040 pin with capable mode)
- uio[3] = SCL (shouid be direct to RP2040 pin with capable mode)

When in open-drain mode the standard pull-up resistor is in the order of 4k7 to 10k and no more than 400pF capacitance on lines. Higher speeds my require attention to those metrics for your setup. The project is peripheral only and does not drive SCL. So open-drain or push-pull can be used by the controller no matter the mode setup in this project.

Power-on-reset configuration (set all zero for standard mode):

- ui_in1 sets CLOCKMUX to use divider
- ui_in2 sets PUSHPULL I2C bus mode (by default open-drain is in use)
- ui_in[3] activates DIV12 divider setup on reset (default is 10Mhz for 10Khz)
- {uio_in[7:0], ui_in[7:4]} is the DIV12 value to use

The design is based around a high-speed clock, at default speed of 10MHz.

Other than the default divider setup for CLOCKMUX mode there is no restriction upon the system clock used, other than trying to operate at low ratios of system-clock:SCL. The design has been simulated from "3:1" upto 1000000:1. Maybe lower than 3:1 is possible.

How to test

RP2040 code is expected to be provided to conduct testing based on simulation expectations.

External hardware

I2C Controller/RP2040

Pinout

#	Input	Output	Bidirectional
0	i2cSampleDivisor bit0	segment a	
1	i2cSampleDivisor bit1	segment b	
2		segment c	I2C SCL (bidi) old
3		segment d	I2C SDA (bidi)
4		segment e	I2C SCL (bidi) new
5		segment f	
6		segment g	
7	7seg or accm	dot	powerOnSense (out)

Collatz conjecture brute-forcer [361]

- Author: Vytautas Šaltenis
- Description: Runs a Collatz sequence calculation for a given number
- GitHub repository
- HDL project
- Mux address: 361
- Extra docs
- Clock: 0 Hz

How it works

The module takes a (large) integer number `N` as an input and computes the Collatz sequence until it reaches 1. When it does, it allows reading back two numbers:

1) The orbit length (i.e. the number of steps it took to reach 1)
2) The highest recorded value of the upper 16 bits of the 144-bit internal iterator

The latter number is an indicator for good candidates for computing path records. The non-zero upper bits indicate that the highest iterator value `Mx(N)` is in the range of the previous path records and should be recomputed in the full offline. (Holding on to the entire 144 bits of `Mx(N)` number would be more obvious, but this almost doubles the footprint of the design, hence, this optimisation).

How to test

The module can be in 2 states: IO and COMPUTE. After reset, the chip will be in IO mode. Since the input is intended to be much larger than the available pins, the input number is uploaded one byte at a time, increasing the address of where in the internal 144-bit-wide register that byte should be stored.

Same for reading the output, except that the output numbers are limited to 16-bits each, so it takes much fewer operations to read them.

The full loop of computations works like this:

1) Set input (see below)
2) Pull `start compute` pin to high. The chip will start computations and will pull `compute busy indicator` pin to high
3) Keep reading `compute busy indicator` pin until it gets low again
4) Read the output (see below)

Writing input:

1) Set `write enable` pin to low
2) Wait at least one cycle
3) Expose your input byte to `input0-7`
4) Expose the target address for that byte to `address0-4`
5) Wait at least one cycle
6) Set `write enable` pin to high

Reading output:

1) Set `orbit/max select` pin to low
2) Set `address0-4` to 0
3) Read low byte of orbit length from `output0-7`
4) Set `address0-4` to 1
5) Read high byte of orbit length from `output0-7`
6) Set `orbit/max select` pin to high
7) Repeat steps 2-5 to read the upper `Mx(N)` bits

Pinout

#	Input	Output	Bidirectional
0	input0	output0	address0
1	input1	output1	address1
2	input2	output2	address2
3	input3	output3	address3
4	input4	output4	address4
5	input5	output5	orbit/max select
6	input6	output6	start compute
7	input7	output7	write enable or compute busy indicator

Power gating test (1x2) [363]

- Author: htfab
- Description: Placeholder for a power gated test design (preliminary work on supporting power gated designs on later IHP shuttles)
- GitHub repository
- HDL project
- Mux address: 363
- Extra docs
- Clock: 0 Hz

How it works

To be filled later

How to test

To be filled later

External hardware

None

Pinout

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

Goldcrest RISC-V [394]

- Author: Felix Roithmayr, Lucas Klemmer, Daniel Große
- Description: A microcoded RISC-V based on SUBLEQ
- GitHub repository
- HDL project
- Mux address: 394
- Extra docs
- Clock: 20000000 Hz

How it works

A Microcoded RISC-V processor, based on subleq.

How to test

Can't test it yet, they will follow soon.

External hardware

Two SPI memory chips, one for ROM one for RAM, as well as a UART output, 4 GPIO input pins, 4 GPIO output pins and support for 3 further SPI devices.

Pinout

#	Input	Output	Bidirectional
0	UART rx	UART tx	Other SPI MOSI
1	External SPI ROM MISO	External SPI ROM SCK	Other SPI CS1
2	External SPI RAM MISO	External SPI ROM MOSI	Other SPI CS2
3	Other SPI MISO	External SPI ROM CS	Other SPI CS3
4	GPIO in 0	External SPI RAM SCK	GPIO out 0
5	GPIO in 1	External SPI RAM MOSI	GPIO out 1

#	Input	Output	Bidirectional
6	GPIO in 2	External SPI RAM CS	GPIO out 2
7	GPIO in 3	Other SPI SCK	GPIO out 3

Transmit UART [417]

- Author: Tom Keddie
- Description: Simple UART transmitting strings
- GitHub repository
- HDL project
- Mux address: 417
- Extra docs
- Clock: 115200 Hz

How it works

The clock is used to generate and async serial stream from a fixed set of strings.

How to test

Apply a clock to match the desired baud rate and hook one of the active outputs to an async serial input.

External hardware

Aync serial port on a system to read the data

Pinout

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

DJ8 8-bit CPU [419]

- Author: DaveX
- Description: DJ8 8-bit CPU with parallel Flash / RAM interface
- GitHub repository
- HDL project
- Mux address: 419
- Extra docs
- Clock: 13760000 Hz

How it works

DJ8 is a 8-bit CPU featuring:

- 8 x 8-bit register file
- 3-4 cycles per instruction
- 15-bit external address bus
- 8-bit external data bus
- Built-in 256-bytes demo ROM with 2 demos

Thanks to its external parallel bus, it could be connected to parallel flash or RAM and can run at full speed without (de)serialization overhead.

Sample assembly code could be found in test bench and demo ROM.

Previous implementations:

- TT07 DJ8 8-bit CPU w/ DAC - Verilog, Mixed-signal, 8-bit DAC
- TT06 DJ8 8-bit CPU - VHDL

Memory Map

From	To	Description
0x0000	0x7fff	External memory
0x8000	0xffff	Internal Test ROM (256 bytes, mirrored)

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] $\hat{}$ 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 TTIHP 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

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).

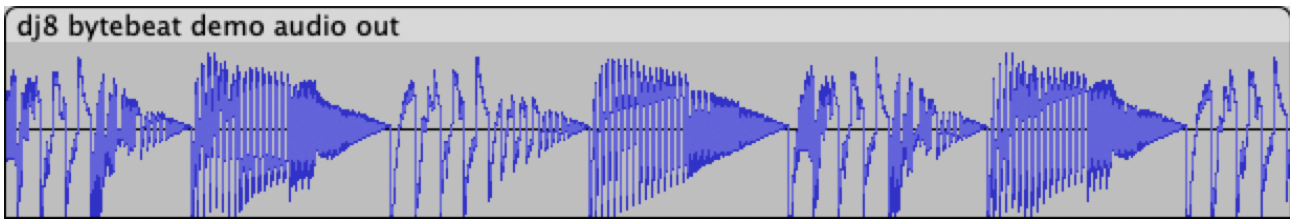
Demo 1: Rotating LED indicator

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 a speaker to uo[4] or use Tiny Tapeout Simon Says PMOD. Play with the DIP switches to change the loop settings.

It is highly recommended to add a simple low-pass RC filter on the speaker line to filter out the buzzing 8kHz carrier. Ideal cut-off frequency between 3kHz and 8kHz, TBD.

Set SW1 and/or SW2 at reset time to adjust speed in case the design doesn't run at 14MHz.

External hardware

- No external hardware for Demo 1
- Speaker 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

PILIPINASLASALLE [421]

- Author: Alexander Co Abad and Dino Dominic Ligutan
- Description: 7-seg Display for PILIPINASLASALLE
- GitHub repository
- HDL project
- Mux address: 421
- Extra docs
- Clock: 0 Hz

How it works

Based from https://wokwi.com/projects/341279123277087315

On power-up, the 7-segment display should display the text PILIPINASLASALLE one at a time per clock cycle. The "dp" output toggles every clock cycle.

How to test

Default mode: Set the clock input to a low frequency such as 1 Hz to see the text transition per clock cycle.

Manual mode: Set the input pin 7 to HIGH and toggle input pins 0-3. The character displayed for each input combination should be according to the table above.

External hardware

7-segment display

Pinout

#	Input	Output	Bidirectional
0	BCD Bit 3 (A)	segment a	
1	BCD Bit 2 (A)	segment b	
2	BCD Bit 1 (A)	segment c	
3	BCD Bit 0 (A)	segment d	
4		segment e	
5		segment f	
6		segment g	

#	Input	Output	Bidirectional
7	Manual Input Mode	segment dp	

RLE Video Player [423]

- Author: Mike Bell
- Description: Reads run length encoded data from QSPI flash, displays on VGA
- GitHub repository
- HDL project
- Mux address: 423
- 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 31: 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]	R1	CS
1	SPI latency1	G1	SD0
2	SPI latency2	B1	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

VGA Experiments in Tennis [425]

- Author: Tom Keddie
- Description: Simple Game
- GitHub repository
- HDL project
- Mux address: 425
- Extra docs
- Clock: 25175000 Hz

How it works

VGA game using paddles attached to input.

How to test

Attach VGA pmod and connect to monitor. Use the inputs to move the paddles

External Hardware

Digilent VGA PMOD or mole99 vga pmod. Buttons to play game on in0-in3

Pinout

#	Input	Output	Bidirectional
0	left paddle up	r1/r0 (mole99/digilent)	g0
1	left paddle down	g1/r1 (mole99/digilent)	g1
2	right paddle up	b1/r2 (mole99/digilent)	g2
3	right paddle down	vsync/r3 (mole99/digilent)	g3
4	score reset	r0/b0 (mole99/digilent)	hsync
5	Speed LSB	g0/b1 (mole99/digilent)	vsync

#	Input	Output	Bidirectional
6	Speed MSB	b0/b2 (mole99/digilent)	tied low
7	pmod sel (high=mole99, low=digilent)	hsync/b3 (mole99/digilent)	tied low

Gray scale and Sobel filter [427]

- Author: Diana Natali Maldonado Ramirez
- Description: Grayscale and Sobel filter.
- GitHub repository
- HDL project
- Mux address: 427
- Extra docs
- Clock: 10000000 Hz

How it works

This project performs grayscale conversion and Sobel filtering with the aim of detecting edges in an image.

Below is a block diagram of the implementation:

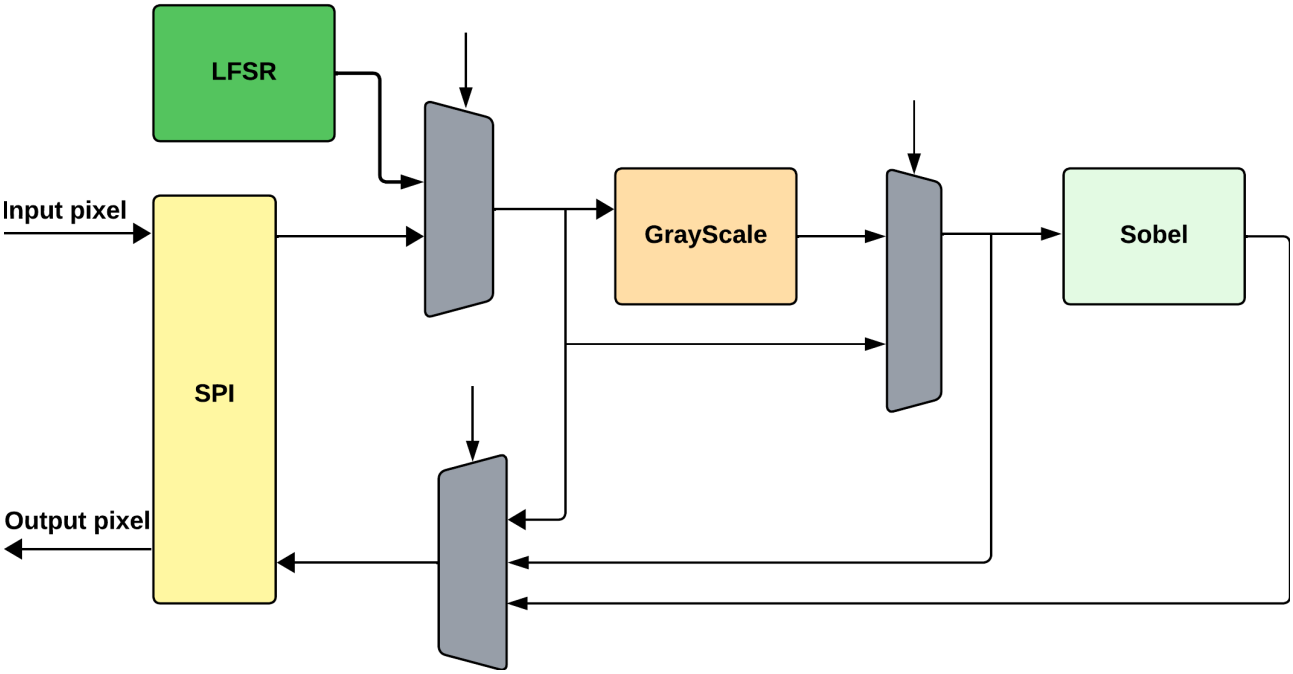


Figure 32: arc

How to test

It is necessary for the pixels to be sent via an SPI protocol; for this purpose, the input `ui_in[2:0]` is designated as follows:

- `ui_in[0]` → SPI Clock

- `ui_in[1]` → Chip Select
- `ui_in[2]` → Input Pixel

As shown in the previous image, there are some processing options:

1. Bypass → Returns the input pixel unprocessed.
2. Grayscale → Returns the pixel converted to grayscale, so it is recommended that the input pixel be RGB.
3. Sobel → Returns the edge detection corresponding to the input pixel, so it is recommended that the input pixel be grayscale.
4. Grayscale + Sobel → Returns the edge detection of the input pixel by performing both grayscale processing and the Sobel filter, so it is recommended that the input pixel be RGB.

To select one of the processing options, the input `ui_in[4:3]` is designated as follows:

- `ui_in[4:3]` = 00 → Grayscale + Sobel
- `ui_in[4:3]` = 01 → Sobel
- `ui_in[4:3]` = 10 → Grayscale
- `ui_in[4:3]` = 11 → Bypass

To perform the Sobel filter processing, it must be enabled according to the selected processing. This can be enabled or disabled as needed through the input `ui_in[5]`, where 1 enables and 0 disables.

The result of the processing corresponds to the output `uo_out[0]`.

There is also a functionality for the input to the different processing options to come from an internal LFSR block; for this purpose, the pins `uio_in[3:2]` are dedicated for input.

External hardware

Any device that allows sending and receiving data via an SPI protocol, like a Raspberry Pi.

#	Input	Output	Bidirectional

Pinout

#	Input	Output	Bidirectional
0	spi_sck_i	spi_sdo_o	LFSR_enable_i
1	spi_cs_i	lfsr_done	seed_stop_i
2	spi_sdi_i	ena	lfsr_en_i
3	select_process_i[0]	output_px[0]	
4	select_process_i1	output_px1	
5	start_sobel_i	output_px2	
6		output_px[3]	
7		output_px[4]	

Game of Life 8x32 (siLife) [454]

- Author: Uri Shaked
- Description: Silicon implementation of Conway's Game of Life with LED Dot Matrix Output
- GitHub repository
- HDL project
- Mux address: 454
- Extra docs
- Clock: 10000000 Hz

How it works

It is a silicon implementation of Conway's Game of Life. The game is played on a 8x32 grid, and the rules are as follows:

- Any live cell with fewer than two live neighbours dies, as if by underpopulation.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overpopulation.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

How to test

Demo mode: The demo mode loads a pre-defined game into the grid and advances it automatically. To enter the demo mode, `wr_en` high while reseting the design (`rst_n` low). Use the `pattern_sel` inputs to select the desired demo pattern. Set `en` to 1 to automatically advance one generation every 0.4 seconds (assuming a 10MHz clock). To pause the game, set `en` to 0.

Manual mode: Load the initial grid row by row. Each row is loaded by selecting the row number (using the `row_sel[4:0]` inputs), setting the `cell_in[7:0]` inputs to the desired state, and pulsing the `wr_en` input.

Once the grid is loaded, set the `en` input to 1 to start the game. The game will advance one step in each clock cycle. To pause the game, set the `en` input to 0.

To view the current state of the grid, set the `row_sel[4:0]` inputs to the desired row number, `max7219_en` to 0, and read the `cell_out[7:0]` outputs.

Alternatively, set `max7129_en` to 1 to display the grid on a MAX7219 LED Matrix (FC-16 module).

External Hardware

MAX7219 LED Matrix (FC-16 module)

Pinout

#	Input	Output	Bidirectional
0	row_sel[0] / pattern_sel	cell_out[0] / max7129_cs	cell_in[0]
1	row_sel1	cell_out1 / max7129_clk	cell_in1
2	rol_sel2	cell_out2 / max7129_din	cell_in2
3	rol_sel[3]	cell_out[3]	cell_in[3]
4	rol_sel[4]	cell_out[4]	cell_in[4]
5	max7129_en	cell_out[5]	cell_in[5]
6	en	cell_out[6]	cell_in[6]
7	wr_en	cell_out[7]	cell_in[7]

TinyQV Risc-V SoC [458]

- Author: Michael Bell
- Description: A Risc-V SoC for Tiny Tapeout
- GitHub repository
- HDL project
- Mux address: 458
- Extra docs
- Clock: 64000000 Hz

How it works

TinyQV is a small Risc-V SoC, implementing the RV32EC instruction set plus the Zcb and Zicond extensions, with a couple of caveats:

- Addresses are 28-bits
- Program addresses are 24-bits
- gp is hardcoded to 0x1000400, tp is hardcoded to 0x8000000.

Instructions are read using QSPI from Flash, and a QSPI PSRAM is used for memory. The QSPI clock and data lines are shared between the flash and the RAM, so only one can be accessed simultaneously.

Code can only be executed from flash. Data can be read from flash and RAM, and written to RAM.

The SoC includes a UART and an SPI controller.

Address map

Address range	Device
0x0000000 - 0x0FFFFFF	Flash
0x1000000 - 0x17FFFFF	RAM A
0x1800000 - 0x1FFFFFF	RAM B
0x8000000 - 0x8000007	GPIO
0x8000010 - 0x800001F	UART
0x8000020 - 0x8000027	SPI
0x8000028 - 0x800002B	PWM

GPIO

Register	Address	Description
OUT	0x8000000 (W)	Control out0-7, if the corresponding bit in SEL is high
OUT	0x8000000 (R)	Reads the current state of out0-7
IN	0x8000004 (R)	Reads the current state of in0-7
SEL	0x800000C (R/W)	Bits 0-7 enable general purpose output on the corresponding bit on out0-7. Bit 8 enables PWM output on out7, bit 9 enables PWM output on io7.

UART

Register	Address	Description
DATA	0x8000010 (W)	Transmits the byte
DATA	0x8000010 (R)	Reads any received byte
STATUS	0x8000014 (R)	Bit 0 indicates whether the UART TX is busy, bytes should not be written to the data register while this bit is set. Bit 1 indicates whether a received byte is available to be read.

Debug UART (Transmit only)

Register	Address	Description
DATA	0x8000018 (W)	Transmits the byte
STATUS	0x800001C (R)	Bit 0 indicates whether the UART TX is busy, bytes should not be written to the data register while this bit is set.

SPI

Register	Address	Description
DATA	0x8000020 (W)	Transmits the byte in bits 7-0, bit 8 is set if this is the last byte of the transaction, bit 9 controls Data/Command on out3
DATA	0x8000020 (R)	Reads the last received byte
CONFIG	0x8000024 (W)	The low 2 bits set the clock divisor for the SPI clock to 2*(value + 1), bit 2 adds half a cycle to the read latency when set
STATUS	0x8000024 (R)	Bit 0 indicates whether the SPI is busy, bytes should not be written or read from the data register while this bit is set.

How to test

Load an image into flash and then select the design.

Reset the design as follows:

- Set rst_n high and then low to ensure the design sees a falling edge of rst_n. The bidirectional IOs are all set to inputs while rst_n is low.
- Program the flash and leave flash in continuous read mode, and the PSRAMs in QPI mode
- Drive all the QSPI CS high and set SD2:SD0 to the read latency of the QSPI flash and PSRAM in cycles.
- Clock at least 8 times and stop with clock high
- Release all the QSPI lines
- Set rst_n high
- Set clock low
- Start clocking normally

Based on the observed latencies from tt3p5 testing, at the target 64MHz clock a read latency of 2 or 3 is likely required. The maximum supported latency is currently 3, but should get up to 5 to have a chance at running at faster clock speeds.

The above should all be handled by some MicroPython scripts for the RP2040 on the TT demo PC.

Build programs using the riscv32-unknown-elf toolchain and the tinyQV-sdk, some examples are here.

External hardware

The design is intended to be used with this QSPI PMOD on the bidirectional PMOD. This has a 16MB flash and 2 8MB RAMs.

The UART is on the correct pins to be used with the hardware UART on the RP2040 on the demo board.

The SPI controller is intended to make it easy to drive an ST7789 LCD display (more details to be added).

It may be useful to have buttons to use on the GPIO inputs.

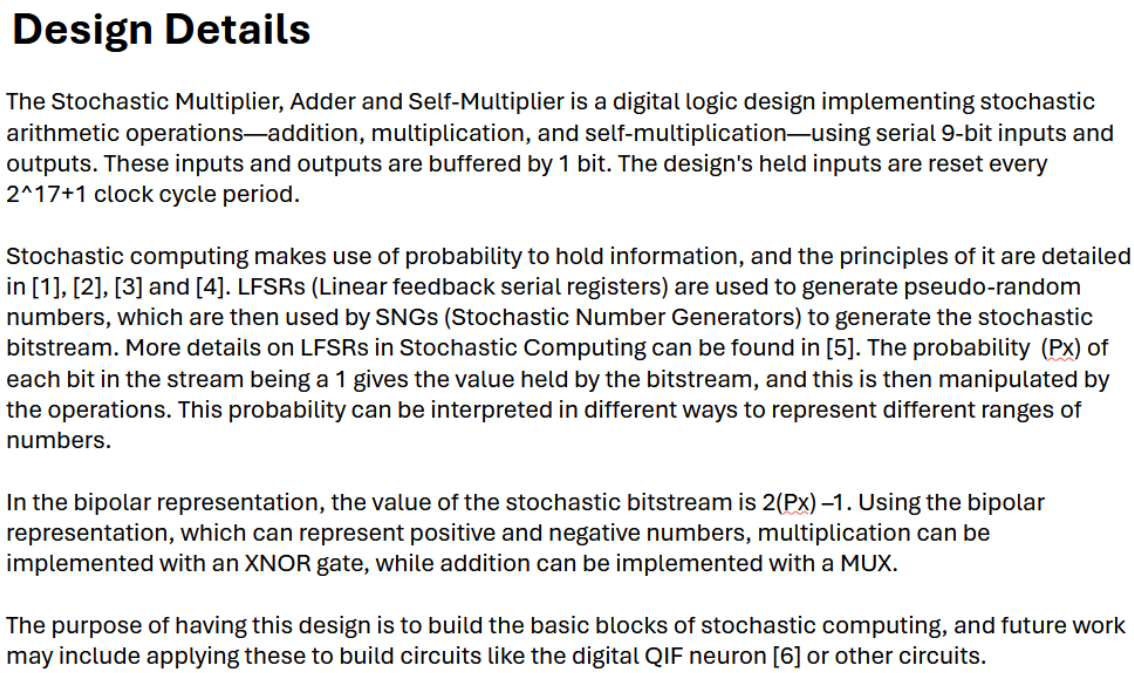
Pinout

#	Input	Output	Bidirectional
0	Interrupt 0	UART TX	Flash CS
1	Interrupt 1	UART RTS	SD0
2	SPI MISO	SPI DC	SD1
3	GP in3	SPI MOSI	SCK
4	GP in4	SPI CS	SD2
5	GP in5	SPI SCK	SD3
6	GP in6	Debug UART TX	RAM A CS
7	UART RX	Debug signal / PWM	RAM B CS / PWM

Stochastic Multiplier, Adder and Self-Multiplier [481]

- Author: Ciecen Lestari, Chih-Kuan Ho, David Parent
- Description: Multiplier, Adder and Self-Multiplier using stochastic computing
- GitHub repository
- HDL project
- Mux address: 481
- Extra docs
- Clock: 100 Hz

How it works



Design Details

The Stochastic Multiplier, Adder and Self-Multiplier is a digital logic design implementing stochastic arithmetic operations—addition, multiplication, and self-multiplication—using serial 9-bit inputs and outputs. These inputs and outputs are buffered by 1 bit. The design's held inputs are reset every 2^17+1 clock cycle period.

Stochastic computing makes use of probability to hold information, and the principles of it are detailed in [1], [2], [3] and [4]. LFSRs (Linear feedback serial registers) are used to generate pseudo-random numbers, which are then used by SNGs (Stochastic Number Generators) to generate the stochastic bitstream. More details on LFSRs in Stochastic Computing can be found in [5]. The probability (Px) of each bit in the stream being a 1 gives the value held by the bitstream, and this is then manipulated by the operations. This probability can be interpreted in different ways to represent different ranges of numbers.

In the bipolar representation, the value of the stochastic bitstream is 2(Px) –1. Using the bipolar representation, which can represent positive and negative numbers, multiplication can be implemented with an XNOR gate, while addition can be implemented with a MUX.

The purpose of having this design is to build the basic blocks of stochastic computing, and future work may include applying these to build circuits like the digital QIF neuron [6] or other circuits.

Figure 33: image

REFERENCES USED

General Stochastic Computing Design:

1 A. Alaghi, W. Qian, and J. P. Hayes, "The Promise and Challenge of Stochastic Computing," IEEE Trans. Comput. Aided Des. Integr. Circuits Syst., vol. 37, no. 8, pp. 1515–1531, Aug. 2018, doi: 10.1109/TCAD.2017.2778107.

2 B. R. Gaines, "Stochastic computing," in Proceedings of the April 18-20, 1967, spring joint computer conference, in AFIPS '67 (Spring). New York, NY, USA: Association for Computing Machinery, Apr. 1967, pp. 149–156. doi: 10.1145/1465482.1465505.

Pins Utilization

Input Pins:
ui_in[0] for serial input of 9bit (+1 bit buffer) probability with 1 bit buffer.
ui_in[1] for serial input of 9bit (+1 bit buffer) probability with 1 bit buffer.

Output Pins:
uo_out[0] for serial output of 9bit (+1 bit buffer) probability result of multiplier.
uo_out[1] for serial output of 9bit (+1 bit buffer) probability result of adder.
uo_out[2] for serial output of 9bit (+1 bit buffer) probability result of self-multiplier.
uo_out[3] signals the inner reset of the clk_counter of the module (not rst_n).

Figure 34: image

Multiplier: Input 0 * Input 1

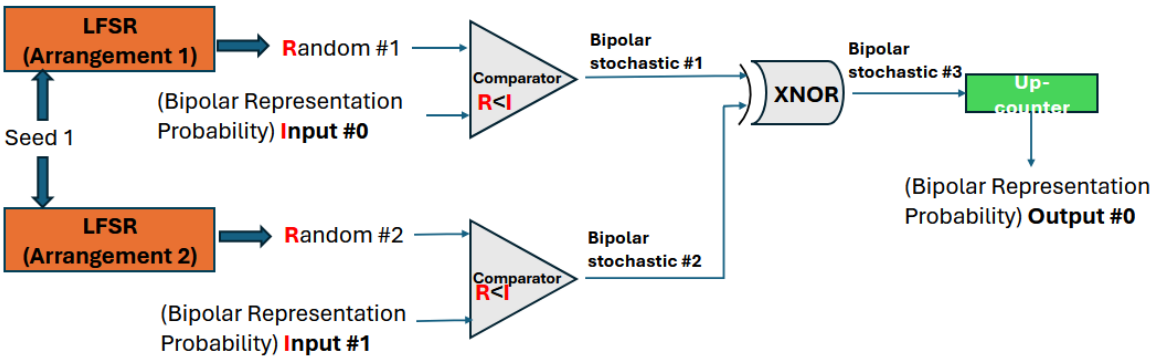


Figure 35: image

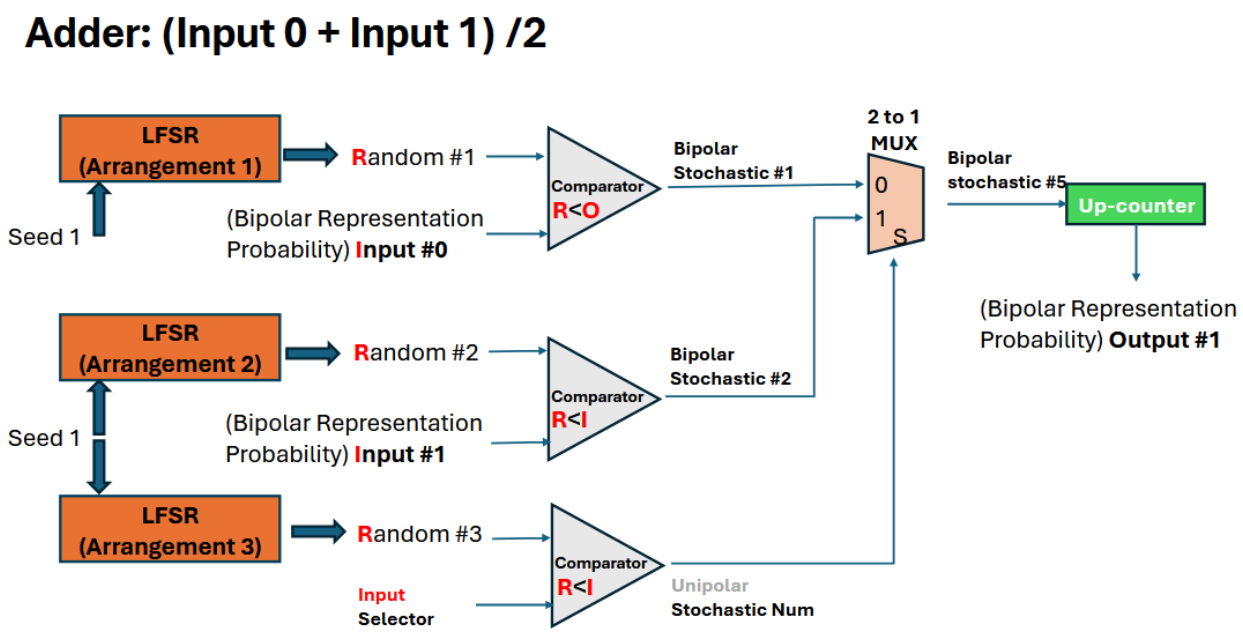


Figure 36: image

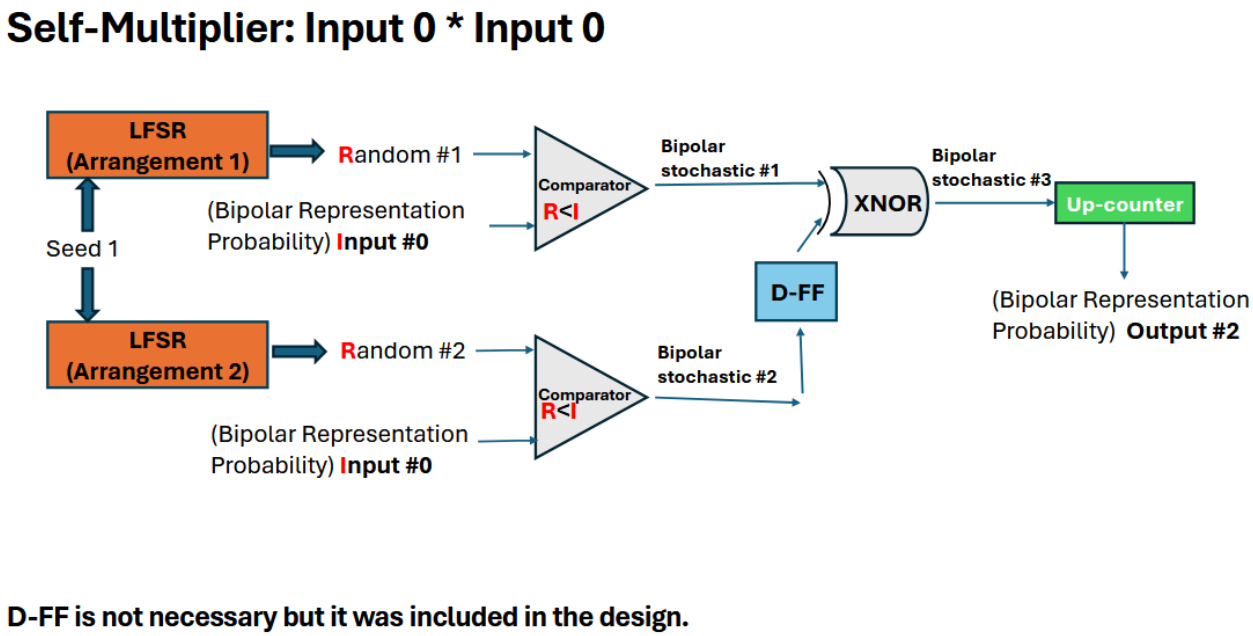


Figure 37: image

[3] Gross, W. J., & Gaudet, V. C. (Eds.). (2019). Stochastic Computing: Techniques and Applications (1st ed. 2019.). Springer International Publishing. https://doi.org/10.1007/978-3-030-03730-7

[4] Qian, W. (2011). Digital yet deliberately random: Synthesizing logical computation on stochastic bit streams (Order No. 3466985). Available from ProQuest Dissertations & Theses Global: The Sciences and Engineering Collection. (885872145). Retrieved from http://search.proquest.com.libaccess.sjlibrary.org/dissertations-theses/digital-yet-deliberately-random-synthesizing/docview/885872145/se-2

LFSR Design in Stochastic Computing:

[5] Jason H. Anderson, Yuko Hara-Azumi, and Shigeru Yamashita. 2016. Effect of LFSR seeding, scrambling and feedback polynomial on stochastic computing accuracy. In Proceedings of the 2016 Conference on Design, Automation & Test in Europe (DATE '16). EDA Consortium, San Jose, CA, USA, 1550–1555. https://dl.acm.org/doi/abs/10.5555/2971808.2972171

Digital QIF neuron:

[6] 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 2 repeating streams of 9 bits (+1 bit buffer) that represent the numbers to be multiplied/added. The self multiplier only processes input from the 1st stream. Read the serial output result, which is also 9bits (+1 bit buffer).

External hardware

ADALM2000

#	Input	Output	Bidirectional

Pinout

#	Input	Output	Bidirectional
0	serial_input_1	serial_output_mul	
1	serial_input_2	serial_output_add	
2		serial_output_smul	
3		clk_counter_reset	
4			
5			
6			
7			

8 Bit Digital QIF [483]

- Author: David Parent
- Description: The circuit will spike when the input is positive. It will reset when the signal exceeds a predetermined value
- GitHub repository
- HDL project
- Mux address: 483
- Extra docs
- Clock: 0 Hz

How it works

8 bit QIF

How to test

Sne in 8 bit 2's complement positive number, read out spikes

External hardware

adlam 2000

Pinout

#	Input	Output	Bidirectional
0	B0	AS0	
1	B1	S1	
2	B2	S2	
3	B3	S3	
4	B4	S4	
5	B5	S5	
6	B6	S6	
7	B7	S7	

CEJMU Beers and Adders [485]

- Author: Prof. Dr.-Ing. Matthias Jung, Philipp Wetzstein, Derek Christ, Jonathan Hager
- Description: Several projects to show in lectures. Includes a simple state-machine, a decoder and two 24 bit adders. Refer to documentation for details
- GitHub repository
- HDL project
- Mux address: 485
- Extra docs
- Clock: 12000000 Hz

How it works

The goal of our design is to be able to show different RTL designs on a real chip in our lectures. Therefore, an internal multiplexer selects different projects. The multiplexer is controlled by uio_in[1:0]. The following designs can be selected:

- state machine that models a vending machine
- decoder to attach the vending machine to a coin acceptor
- 24 bit Ripple Carry Adder
- 24 bit Carry Lookahead Adder

How to test

- 00: A state machine, which models a vending machine. This state machine outputs 1, if 1.50€ have been fed into it. Inputs are taken from ui_in[1:0] with the following meaning: 00 = 0€ (nothing changes), 01 = 0.50€, 10 = 1€, 11 = undefined
- 01: A module that decodes pulses coming from a coin acceptor into coin ids. The number of pulses is equivalent with the decoded id. With a second instance of the vending machine automaton, this module makes it possible to physically insert coins into the machine.
- 10: Ripple Carry Adder with 24 bit input and 25 bit output
- 11: Carry Lookahead Adder with 24 bit input and 25 bit output

Since we only have 8 bit input and output, an internal logic is responsible for taking the inputs in 8 bit chunks and outputting the results in 8 bit chunks. This logic can be used as follows:

1. Select the adder you want to use: uio_in[1:0] == 10 (RCA) or 11 (CLA)

2. Reset the chip for at least one cycle
3. ui_in[7:0] should now be assigned a[23:16]
4. Wait for one cycle, repeat with a[15:8], a[7:0]
5. Repeat with b[23:16], b[15:8], b[7:0]
6. The inputs are now read into the design and will be send to the adders by asserting uio_in2 to 1 (this is done to have a reference signal when measuring)
7. If you are ready to read the outputs, set uio_in[3] to 1 and wait one cycle
8. z[23:16] can now be read from uo_out
9. Wait one cycle, z[15:8] can now be read
10. Repeat for z[7:0]

Note that the overflows of both adders are always brought out to uio_out[7:6] to allow measurements. A reset upon changing the design is required to ensure valid results

External hardware

No external hardware is strictly required. Since the goal of both adders is to measure the difference in execution speed, an oscilloscope is helpful. The decoder for the coin acceptor was designed for the HX-916

Pinout

#	Input	Output	Bidirectional
0	Multiplexed to all designs (refer to documentation for details)	Multiplexed from all designs (refer to documentation for details)	Select design (input)
1	...	...	Select design (input)
2	...	...	start_calc
3	...	...	output_result
4	...	...	unused
5	...	...	unused
6	...	...	overflow bit of RCA (output)
7	...	...	overflow bit of CLA (output)

Classic 8-bit era Programmable Sound Generator SN76489 [487]

- Author: ReJ aka Renaldas Zioma
- Description: The SN76489 Digital Complex Sound Generator (DCSG) is a programmable sound generator chip from Texas Instruments.
- GitHub repository
- HDL project
- Mux address: 487
- Extra docs
- Clock: 4000000 Hz

How it works

This Verilog implementation is a replica of the classical **SN76489** programmable sound generator. With roughly a 1400 logic gates this design fits on a **single tile** of the TinyTapeout.

The goals of this project

1. closely replicate the behavior and eventually the complete **design of the original** SN76489
2. provide a readable and well documented code for educational and hardware **preservation** purposes
3. leverage the **modern fabrication** process

A significant effort was put into a thorough **test suite** for regression testing and validation against the original chip behavior.

The future work

The next step is to incorporate analog elements into the design to match the original SN76489 - DAC for each channel and an analog OpAmp for channel summation.

Chip technical capabilities

- **3 square wave** tone generators
- **1 noise** generator
- 2 types of noise: *white* and *periodic*
- Capable to produce a range of waves typically from **122 Hz** to **125 kHz**, defined by **10-bit** registers.
- **16** different volume levels

Registers The behavior of the SN76489 is defined by 8 "registers" - 4 x 4 bit volume registers, 3 x 10 bit tone registers and 1 x 3 bit noise configuration register.

Channel	Volume registers	Tone & noise registers
0	Channel #0 attenuation	Tone #0 frequency
1	Channel #1 attenuation	Tone #1 frequency
2	Channel #2 attenuation	Tone #2 frequency
3	Channel #3 attenuation	Noise type and frequency

Square wave tone generators Square waves are produced by counting down the 10-bit counters. Each time the counter reaches the 0 it is reloaded with the corresponding value from the configuration register and the output bit of the channel is flipped producing square waves.

Noise generator Noise is produced with 15-bit Linear-feedback Shift Register (LFSR) that flips the output bit pseudo randomly. The shift rate of the LFSR register is controller either by one of the 3 hardcoded power-of-two dividers or output from the channel #2 tone generator is used.

Attenuation Each of the four SN76489 channels have dedicated attenuation modules. The SN76489 has 16 steps of attenuation, each step is 2 dB and maximum possible attenuation is 28 dB. Note that the attenuation definition is the opposite of volume / loudness. Attenuation of 0 means maximum volume.

Finally, all the 4 attenuated signals are summed up and are sent to the output pin of the chip.

Historical use of the SN76489

The SN76489 family of programmable sound generators was introduced by Texas Instruments in 1980. Variants of the SN76489 were used in a number of home computers, game consoles and arcade boards:

- home computers: TI-99/4, BBC Micro, IBM PCjr, Sega SC-3000, Tandy 1000
- game consoles: ColecoVision, Sega SG-1000, Sega Master System, Game Gear, Neo Geo Pocket and Sega Genesis
- arcade machines by Sega & Konami and would usually include 2 or 4 SN76489 chips

The SN76489 chip family competed with the similar General Instrument AY-3-8910.

The original pinout of the SN76489AN

```
              ,--._.--.
       D5  -->|1    16|<-- VCC
       D6  -->|2    15|<-- D4
       D7  -->|3    14|<-- CLOCK
    ready* <--|4    13|<-- D3
      /WE  -->|5    12|<-- D2
      /ce* -->|6    11|<-- D1
 AUDIO OUT <--|7    10|<-- D0
       GND ---|8     9|    not connected*
              `-------'
       * -- omitted from this Verilog implementation
```

Difference from the original hardware

This Verilog implementation is a completely digital and synchronous design that differs from the original SN76489 design which incorporated analog parts.

Audio signal output While the original chip had integrated OpAmp to sum generated channels in analog fashion, this implementation does digital signal summation and digital output. The module provides two alternative outputs for the generated audio signal:

1. digital 8-bit audio output suitable for external Digital to Analog Converter (DAC)
2. pseudo analog output through Pulse Width Modulation (PWM)

Separate 4 channel output Outputs of all 4 channels are exposed along with the master output. This allows to validate and mix signals externally. In contrast the original chip was limited to a single audio output pin due to the PDIP-16 package.

No DC offset This implementation produces output 0/1 waveforms without DC offset.

No /CE and READY pins Chip enable control pin **/CE** is omitted in this design for simplicity. The behavior is the same as if **/CE** is tied *low* and the chip is considered always enabled.

Unlike the original SN76489 which took 32 cycles to update registers, this implementation handles register writes in a single cycle and chip behaves as always **READY**.

Synchronous reset and single phase clock The original design employed 2 phases of the clock for the operation of the registers. The original chip had no reset pin and would wake up to a random state.

To make it easier to synthesize and test on FPGAs this implementation uses single clock phase and synchronous reset for registers.

A configurable clock divider was introduced in this implementation.

1. the original SN76489 with the master clock internally divided by 16. This classical chip was intended for PAL and NTSC frequencies. However in BBC Micro 4 MHz clock was employed.
2. SN94624/SN76494 variants without internal clock divider. These chips were intended for use with 250 to 500 KHz clocks.
3. high frequency clock configuration for TinyTapeout, suitable for a range between 25 MHz and 50 Mhz. In this configuration the master clock is internally divided by 128.

The reverse engineered SN76489

This implementation is based on the results from these reverse engineering efforts:

1. Annotations and analysis of a decapped SN76489A chip.
2. Reverse engineered schematics based on a decapped VDP chip from Sega Mega Drive which included a SN76496 variant.

How to test

Summary of commands to communicate with the chip

The SN76489 is programmed by updating its internal registers via the data bus. Below is a short summary of the communication protocol of SN76489. Please consult SN76489 Technical Manual for more information.

Command	Description	Parameters
`1cc0ffff`	Set tone fine frequency	`f` - 4 low bits, `c` - channel #
`00ffffff`	Follow up with coarse frequency	`f` - 6 high bits
`11100bff`	Set noise type and frequency	`b` - white/periodic, `f` - frequency control
`1cc1aaaa`	Set channel attenuation	`a` - 4 bit attenuation, `c` - channel #

NF1	NF0	Noise frequency control
0	0	Clock divided by 512
0	1	Clock divided by 1024
1	0	Clock divided by 2048
1	1	Use channel #2 tone frequency

Write to SN76489 Hold **/WE** low once data bus pins are set to the desired values. Pull **/WE** high before setting different value on the data bus.

Note frequency

Use the following formula to calculate the 10-bit period value for a particular note :

$$toneperiod_{cycles} = clock_{frequency}/(32_{cycles} * note_{frequency})$$

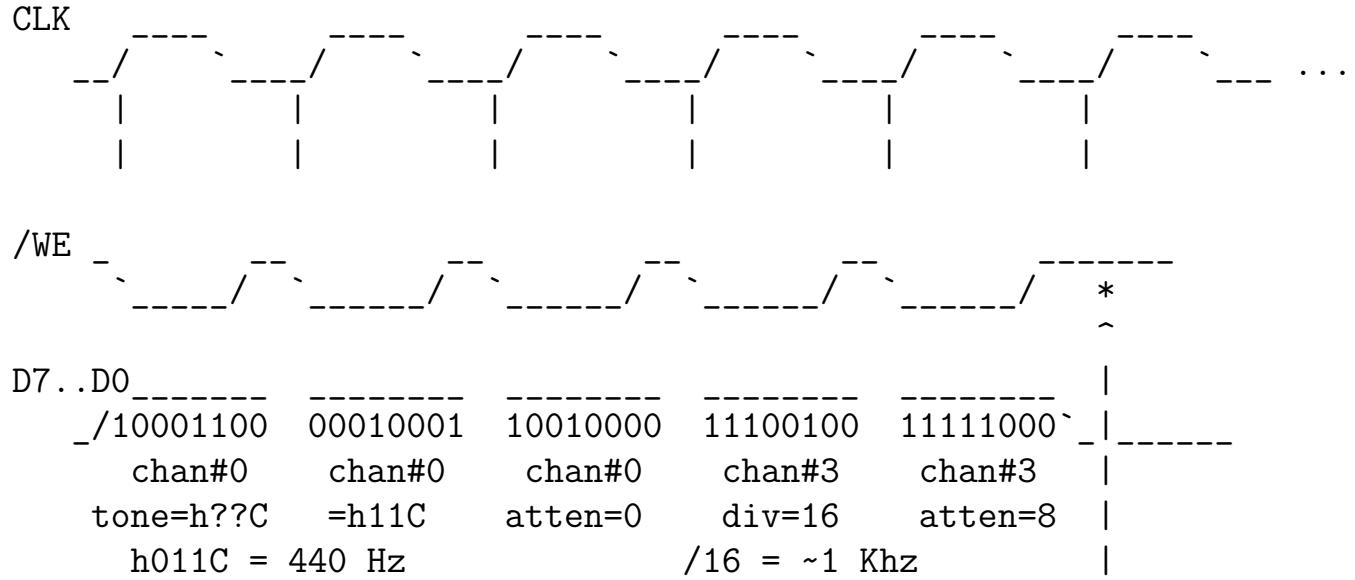
For example 10-bit value that plays 440 Hz note on a chip clocked at 4 MHz would be:

$$toneperiod_{cycles} = 4000000Hz/(32_{cycles} * 440Hz) = 284 = 11C_{hex}$$

An example to play a note accompanied with a lower volume noise

/WE	D7	D6/5	D4..D0	Explanation
0	1	00	01100	Set **channel #0** tone low 4-bits to $C_{hex} = 1100_{bin}$
0	0	00	10001	Set **channel #0** tone high 6-bits to $11_{hex} = 010001_{bin}$
0	1	00	10000	Set **channel #0** volume to **100%**, attenuation 4-bits are $0_{dec} = 0000_{bin}$
0	1	11	00100	Set **channel #3** noise type to **white** and divider to **512**
0	1	11	11000	Set **channel #3** noise volume to **50%**, attenuation 4-bits are $8_{dec} = 1000_{bin}$

Timing diagram



```
                                white noise           |
                                                      |
                                                noise restarts
                                            after /WE goes high and
                                        there was a write to noise register
```

Configurable clock divider

Clock divider can be controlled through **SEL0** and **SEL1** control pins and allows to select between 3 chip variants.

SEL1	SEL0	Description	Clock frequency
0	0	SN76489 mode, clock divided by 16	3.5 .. 4.2 MHz
1	1	—//—	3.5 .. 4.2 MHz
0	1	SN76494 mode, no clock divider	250 .. 500 kHZ
1	0	New mode for TT05, clock div. 128	25 .. 50 MHz

SEL1	SEL0	Formula to calculate the 10-bit tone period value for a note
0	0	$clock_{frequency}/(32_{cycles} * note_{frequency})$
1	1	—//—
0	1	$clock_{frequency}/(2_{cycles} * note_{frequency})$
1	0	$clock_{frequency}/(256_{cycles} * note_{frequency})$

Some examples of music recorded from the chip simulation

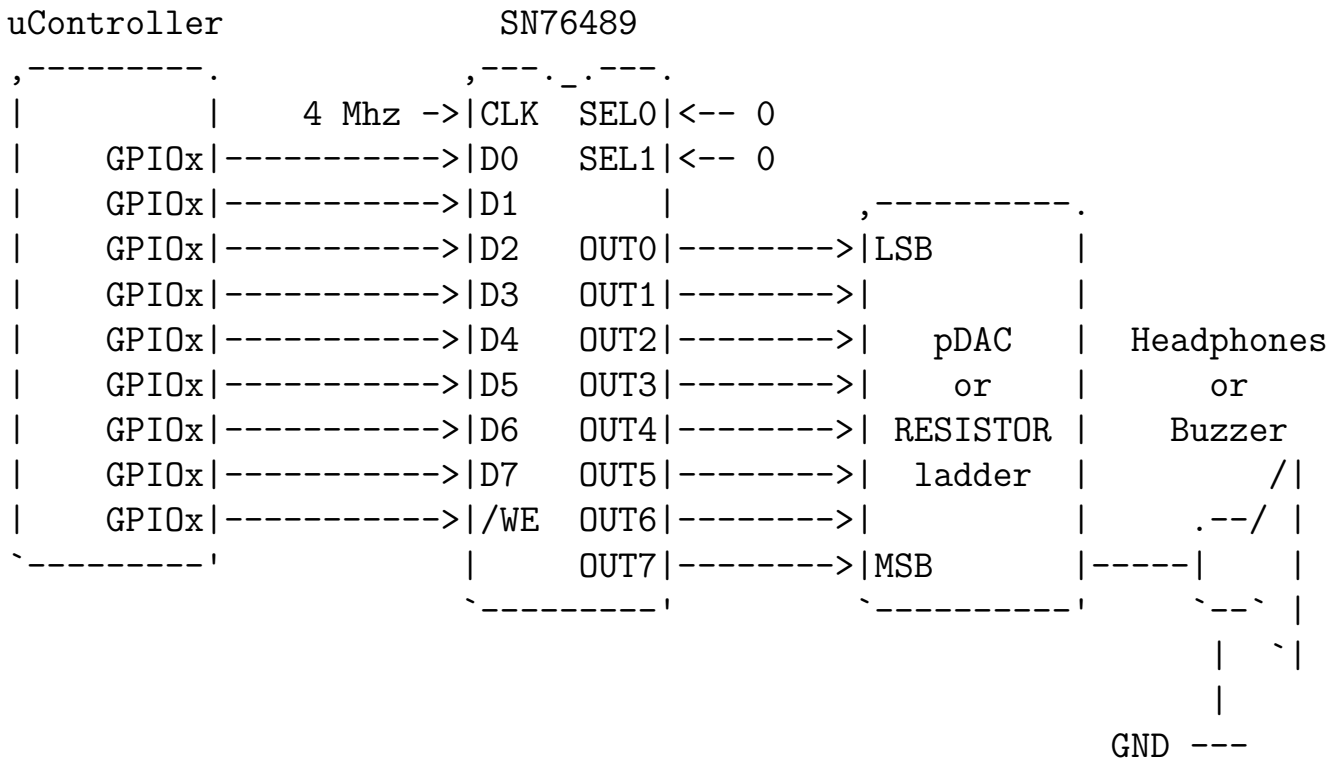
- https://www.youtube.com/watch?v=ghBGasckpSY
- https://www.youtube.com/watch?v=HXLAdA02I-w

External hardware

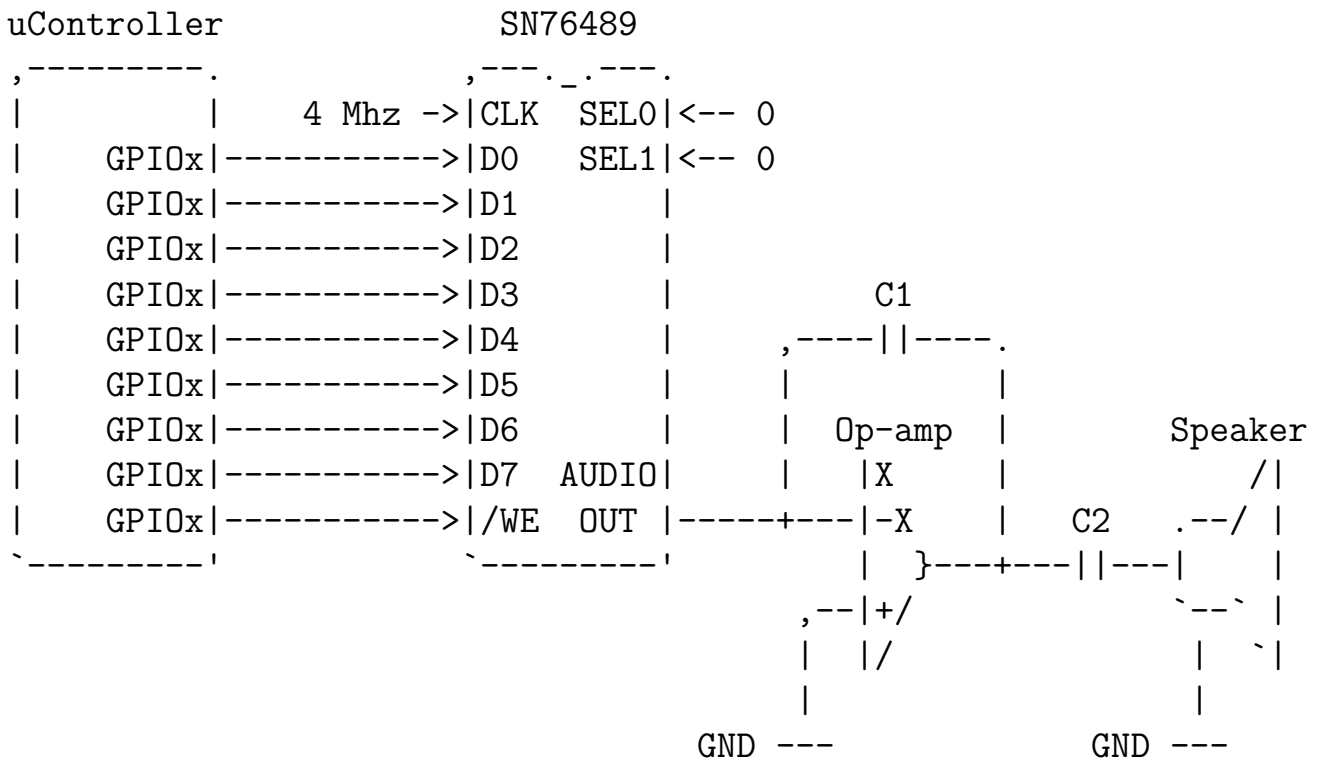
DAC (for ex. Digilent R2R PMOD), RC filter, amplifier, speaker.

The data bus of the SN76489 chip has to be connected to microcontroller and receive a regular stream of commands. The SN76489 produces audio output and has to be connected to a speaker. There are several ways how the overall schematics can be established.

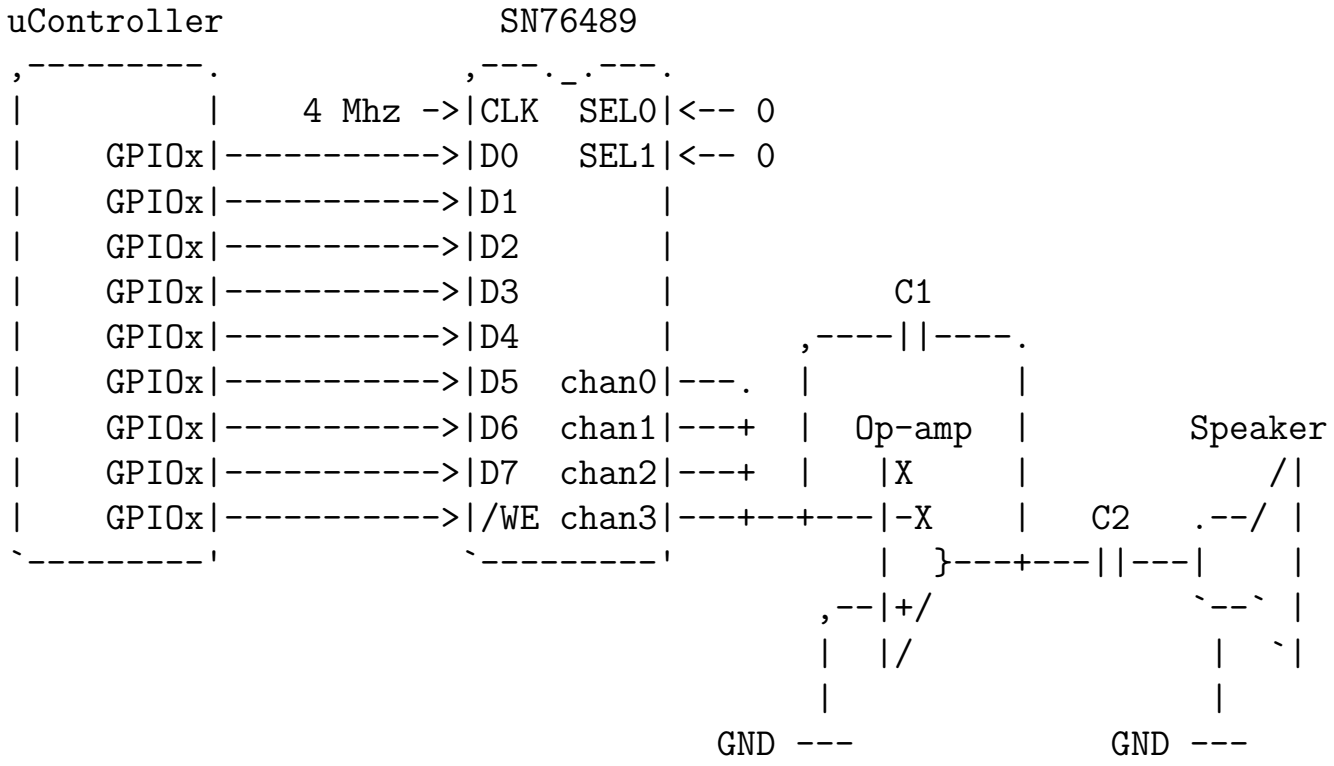
8-bit parallel output via DAC One option is to connect off the shelf data parallel Digital to Analog Converter (DAC) for example Digilent R2R Pmod to the output pins and route the resulting analog audio to piezo speaker or amplifier.



AUDIO OUT through RC filter Another option is to use the Pulse Width Modulated (PWM) AUDIO OUT pin that combines 4 channels with the Resistor-Capacitor based low-pass filter or better the Operation Amplifier (Op-amp) & Capacitor based integrator:



Separate channels through the Op-amp The third option is to externally combine 4 channels with the Operational Amplifier and low-pass filter:



Pinout

#	Input	Output	Bidirectional
0	D0 data bus	digital audio LSB	(in) **/WE** write enable
1	D1 data bus	digital audio	(in) **SEL0** clock divider
2	D2 data bus	digital audio	(in) **SEL1** clock divider
3	D3 data bus	digital audio	(out) channel 0 (PWM)
4	D4 data bus	digital audio	(out) channel 1 (PWM)
5	D5 data bus	digital audio	(out) channel 2 (PWM)
6	D6 data bus	digital audio	(out) channel 3 (PWM)
7	D7 data bus	digital audio MSB	(out) AUDIO OUT master (PWM)

MULDIV unit (8-bit signed/unsigned) [489]

- Author: Darryl Miles
- Description: Combinational Multiply and Divide Unit (signed and unsigned)
- GitHub repository
- HDL project
- Mux address: 489
- Extra docs
- Clock: 0 Hz

Background

Combinational multiply / divider unit (8bit+8bit input)

This is an updated version of the original project that was submitted and manufactured in TT04 (https://github.com/dlmiles/tt04-muldiv4). The previous project was hand crafted in Logisim-Evolution then exported as verilog and integrated into a TT04 project.

This version is the same design, extended to 8-bit wide inputs, but instead of hand crafting the logic gates in a GUI we convert functional blocks into SpinalHDL language constructs. Part of the purpose of this design is to understand the area and timing changes introduced by adding more bits, then to explore alternative topologies.

The goal of the next iteration of this design maybe to introduce a FMA (Fused Multiply Add/Accumulate) function and ALU function to explore if there is some useful composition of these functions (that might be useful in an 8bit CPU/MCU design, or scale to something bigger). The next iteration on from this could explore how to draw the transistors directly (instead of using standard cell library) for such an arrangement, this may result in non-rectangular cells that interlock to improve both area density and timing performance. Or it might go up in smoke... who knows.

How It Works

Due to the limited total IOs available at the external TT interface it is necessary to clock the project and setup UI_IN[0] to load each of the 2 8-bit input registers.

The input side uses latches to capture, which means during the appropiate phase CLK (high) and ADDR state, it alternatively opens/closes, the data is becomes captured into the latches at the CLK NEGEDGE. During the whole time it is open and closed it is providing the data into the appropiate input side of both MUL and DIV units (which are seperate logic modules).

The result becomes immediately available (after propagation and ripple settling time) at the outputs. While the latch it open, maybe artificially by extending duty-cycle of CLK, you should also be able to conduct experiments on modifying input and observing output (when in immediate result mode)

The result output is also multiplexed and has an immediate and registered mode. The immediate mode provides a direct visibility of the MUL/DIV combintational output and should allos timing between input and outputs to be observed. (you need to account for address multiplex of high-low 8bit sides of result). The registered mode capture the result in full at the time of the last ADDR and a CLK posedge. This allows you to change the values for the input side during the next few cycles, while the module ensures to sustain the result value of the last computation at the output. With an appropiate pipeline interleave request and result information to achieve higher throughput.

FIXME

FIXME please check out the original github for any enhanced documentation for this project, potentially improved information nearer PCB+IC delivery (to customer) schedule but also post-production post-physically testing results and information. I hope to produce some kind graphs showing the timing capture and reliability to show and demonstrate the cascade effect. This assumes I have the design correct to allow this to happen, but there are some tricks (like extending CLK on-duty cycle when latches are open) enough to see result capture output.

FIXME provide wavedrom diagram (MULU, MULS, DIVU, DIVS)

FIXME explain IMMediate mode and REGistered mode (to pipeline)

```
FIXME provide blockdiagram of functional units
//     D
//    MUX
//    X Y registers (loaded from multiplexed D)
//     OP -> res flags
//    P P registers
//  DEMUX
//     R
```

FIXME explain architective difference to previous example and considerations why to change.

FIXME explain addressing mode to allow much wider units and potentially uneven input sizes.

Multiplier (signed/unsigned) Method uses Ripple Carry Array as 'high speed multiplier' Setup operation mode bits MULDIV=0 and OPSIGNED(unsigned=0/signed=1) Setup A (multiplier 8-bit) * B (multiplicand 8-bit) Expect result P (product 16-bit)

Divider (signed/unsigned) Method uses Full Adder with Mux as 'combinational restoring array divider algorithm'. Setup operation mode bits MULDIV=1 and OPSIGNED(unsigned=0/signed=1) Setup Dend (dividend 8-bit) / Dsor (divisor 8-bit) Expect result Q (quotient 8-bit) with R (remainder 8-bit)

Divider has error bit indicators that take precedence over any result. If any error bit is set then the output Q and R should be disregarded. When in multiplier mode error bits are muted to 0. No input values can cause an overflow error so the bit is always reset.

How to test

Please check back with the project github main page and the published docs/ directory. There is expected to be some instructions provided around the time the TT05 chips a received (Q4 2024).

At the time of writing receiving a physical chip (from a previous TT edition) back has not occured, so there is no experience on the best way to test this project, so I defer the task of writing this section to a later time.

There should be sufficient instructions here start you own journey.

External hardware

It is expect the RP2040 and a Python REPL should be sufficient test this project.

Thoughts to the future (next iteration)

uio_in[3] might moved to bit4 and DIV0/OVER combined into bit5 This would allow the address the contigious area below. However during a test build of a MULDIV16 version it easily exceeds 1x1, as this stage looking towards making builds with permutations of design/topology and method to generate GDS. So 1x1 is good to achieve this.

The uio_in[3] feature wants to use registered mode to lock result when last address is clocked in this way we can pipeline result and demonstration of what pipelining can do to increase thoughput.

The TB is limited to the 4bit version. Ran out of time to validate registered output and pipeline.

Encapsulate the SpinalHDL Scala netlist generation, and write a yosys JVM module harness (a yosys C++ module that is a JVM thread/process runner, with communication interface, data/ffi API/lifecycle). Then write a yosys plugin that allows it to directly include, use and call for generated data based on parametric details.

Consider emitting a custom cell/macro/GDS_object that yosys can call for, then emit verilog like a regular standard cell module.

Consider modifying OpenROAD/OpenLane to incorporate generated macros directly into other detailed routing environment then have the existing detailed routing work around it as-is.

TODO

Fixup the original logisim schematic labels.

The input re-ordering (which made the SpinalHDL algo easier)

Relabel the P6_EXTND_EN to P7_EXTND_EN the original product index label was a bad choice in retrospect.

Provide the SpinalHDL directory to the project with the sbt project and netlist generation code.

Fill out SpinalHDL unit testing testing.

Test support for SUPPORT_SIGNED=false (try to completely remove nets from output instead of assigning constant False and letting synthesis optimize away)

Implement support for seperate SUPPORT_SIGNED for each input with 3 modes of operation ALWAYS/NEVER/BOTH(like now using control input bit)

Implement and test support for odd-sized inputs, so the width of X and Y or DEND and DSOR can be different sizes.

When input width can be unequal, test out the EOVERFLOW in the divider is wired to the correct port and works in this scenarios.

Provide unit testing for common multipler sizes, obvious byte boundaries but also the sizes common in FPGA DSP primitives.

Pinout

#	Input	Output	Bidirectional
0	Data0 see docs	Result0 see docs	Addr bit0 HI=1/lo=0 mux of Data and Result (input only)
1	Data1 see docs	Result1 see docs	
2	Data2 see docs	Result2 see docs	
3	Data3 see docs	Result3 see docs	Result mux registered=1/immediate=0 (input only)
4	Data4 see docs	Result4 see docs	DIV error overflow (output only)
5	Data5 see docs	Result5 see docs	DIV error divide-by-zero (output only)
6	Data6 see docs	Result6 see docs	OPSIGNED mode (input only)
7	Data7 see docs	Result7 see docs	MULDIV mode (input only)

IHP loopback tile with input skew measurement [491]

- Author: Darryl Miles project from Eric Smith
- Description: Count up to 10, one second at a time.
- GitHub repository
- HDL project
- Mux address: 491
- Extra docs
- Clock: 10000000 Hz

How it works

This project is based on (https://github.com/ericsmi/tt05-loopback-with-skew) but for IHP30.

How to test

Clock the project and modiify the timing and examine FF capture reliabiliy.

External hardware

Skewable clock and data source.

Pinout

#	Input	Output	Bidirectional
0	compare bit 11	segment a	second counter bit 0
1	compare bit 12	segment b	second counter bit 1
2	compare bit 13	segment c	second counter bit 2
3	compare bit 14	segment d	second counter bit 3
4	compare bit 15	segment e	second counter bit 4
5	compare bit 16	segment f	second counter bit 5
6	compare bit 17	segment g	second counter bit 6
7	compare bit 18	dot	second counter bit 7

VGA clock [513]

- Author: Matt Venn
- Description: Shows the time on a VGA screen
- GitHub repository
- HDL project
- Mux address: 513
- 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	

RGB Mixer demo [515]

- Author: Matt Venn
- Description: Zero to ASIC demo project
- GitHub repository
- HDL project
- Mux address: 515
- Extra docs
- Clock: 10000000 Hz

How it works

Debounce the inputs, drive an encoder module, and output a PWM signal for each encoder.

How to test

Twist each encoder and the LEDs attached to the outputs should change in brightness.

By setting the debug port to 0, 1 or 2, the internal value of each encoder is output on the bidirectional outputs.

External hardware

Use 3 digital encoders attached to the first 6 inputs.

Pinout

#	Input	Output	Bidirectional
0	enc0 a	pwm0	encoder bit 0
1	enc0 b	pwm1	encoder bit 1
2	enc1 a	pwm2	encoder bit 2
3	enc1 b		encoder bit 3
4	enc2 a		encoder bit 4
5	enc2 b		encoder bit 5
6	debug bit 0		encoder bit 6
7	debug bit 1		encoder bit 7

Universal Binary to Segment Decoder [517]

- Author: Rebecca G. Bettencourt
- Description: Decodes various binary codes to various segmented displays.
- GitHub repository
- HDL project
- Mux address: 517
- Extra docs
- Clock: 0 Hz

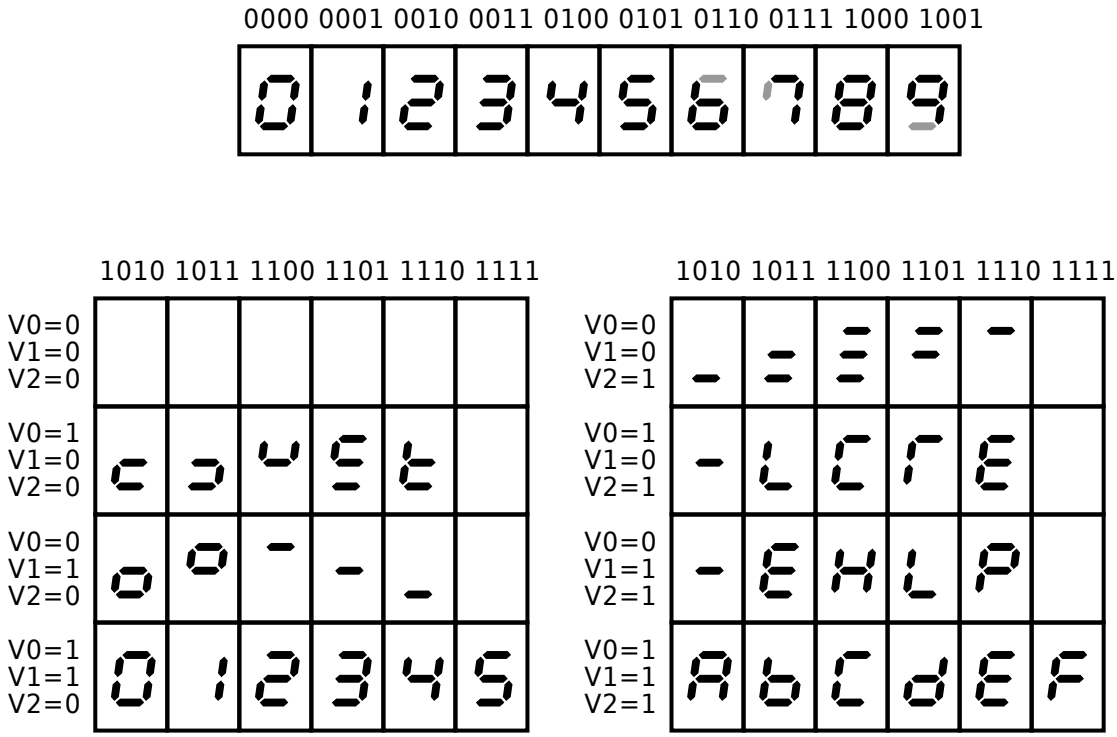
How it works

This project is composed of four modules:

- A BCD to seven segment decoder with a wide variety of options for customizing the appearance of digits
- An ASCII to seven segment decoder with two different "fonts"
- A dual BCD to Cistercian numeral decoder
- A BCV (binary-coded *vigesimal*) to Kaktovik numeral decoder

BCD to seven segment decoder

This mode converts a decimal digit in BCD to its representation on a standard seven segment display. There are inputs that affect the display of the digits 6, 7, and 9, and eight different options for handling out-of-range values. These inputs allow this decoder to match the behavior of just about any other BCD to seven segment decoder, making it *universal*.



The signals used in this mode are:

- **/AL** - Active low. If HIGH, outputs will be HIGH when lit. If LOW, outputs will be LOW when lit.
- **/BI** - Blanking input. If LOW, all segments will be blank regardless of other inputs, including **/LT**.
- **/LT** - Lamp test. When **/BI** is HIGH and **/LT** is LOW, all segments will be lit.
- **/RBI** - Ripple blanking input. If the BCD value is zero and **/RBI** is LOW, all segments will be blank.
- **V0**, **V1**, **V2** - Selects the output when the BCD value is out of range.
- **X6** - When HIGH, the extra segment **a** will be lit on the digit 6.
- **X7** - When HIGH, the extra segment **f** will be lit on the digit 7.
- **X9** - When HIGH, the extra segment **d** will be lit on the digit 9.
- **A**, **B**, **C**, **D** - BCD input from least significant bit **A** to most significant bit **D**.
- **a**, **b**, **c**, **d**, **e**, **f**, **g** - Outputs for a seven segment display.
- **/RBO** - Ripple blanking output. HIGH when BCD value is nonzero or **/RBI** is HIGH.

The pin assignments in this mode are:

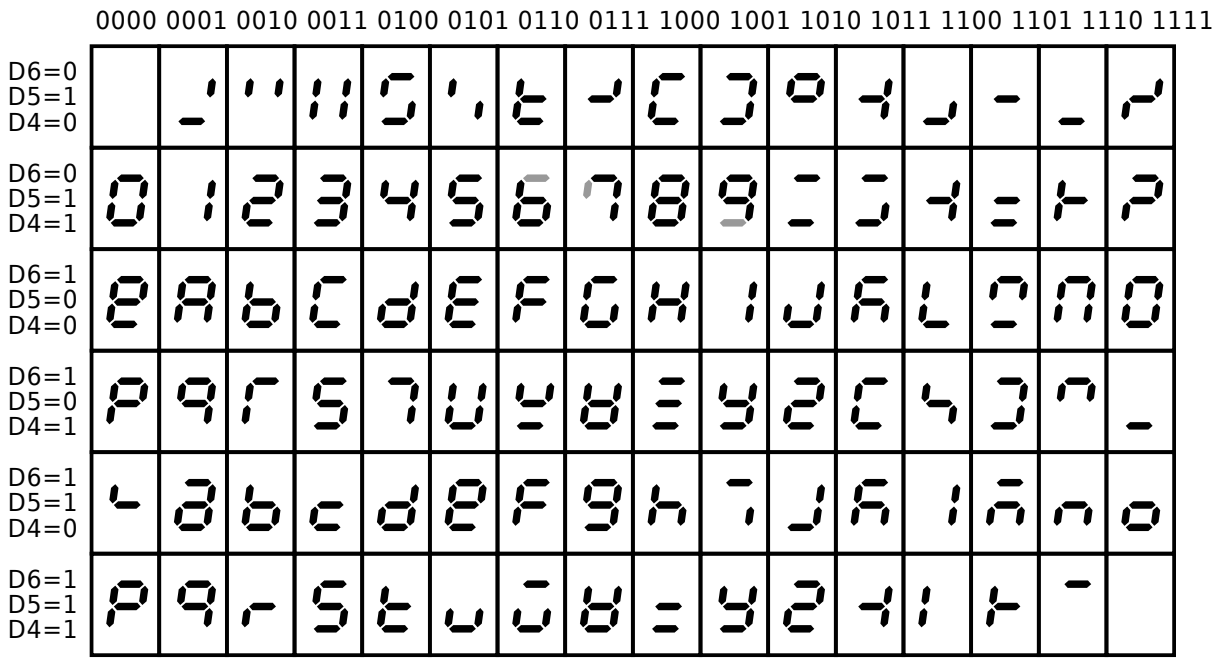
	Dedicated Input	Dedicated Output	Bidirectional
0	A	Segment a	Input - X6

	Dedicated Input	Dedicated Output	Bidirectional
1	B	Segment b	Input - X7
2	C	Segment c	Input - X9
3	D	Segment d	Input - /LT
4	V0	Segment e	Input - /BI
5	V1	Segment f	Input - /AL
6	V2	Segment g	Input - LOW
7	/RBI	/RBO	Input - LOW

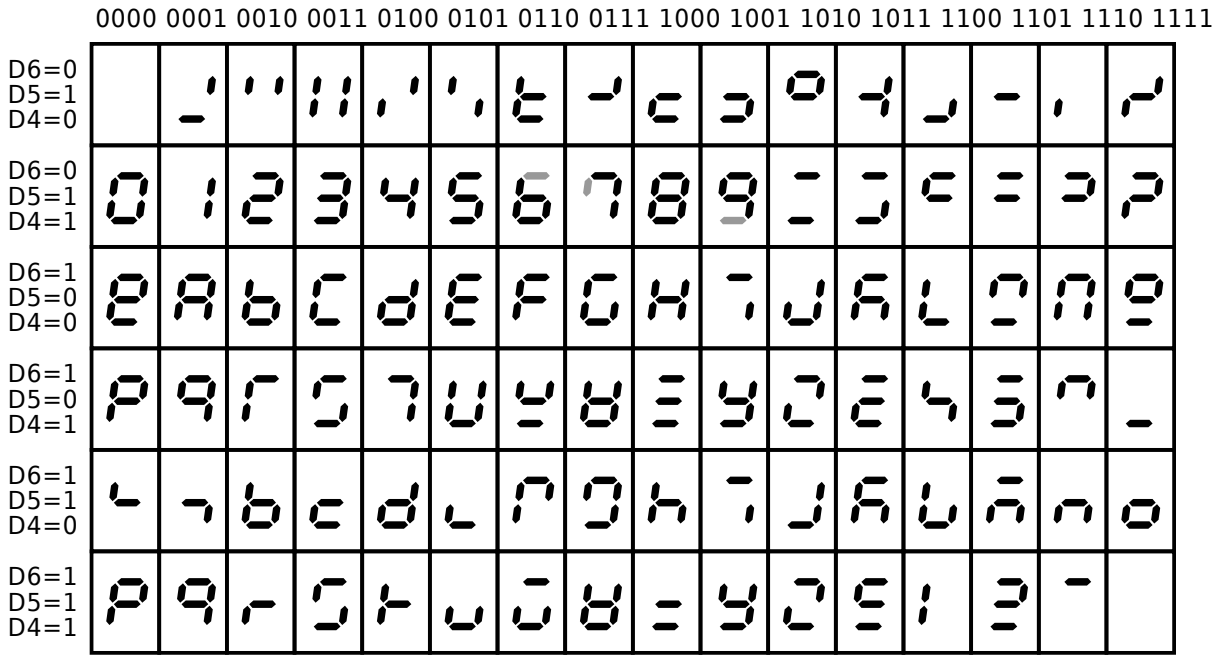
ASCII to seven segment decoder

This mode converts an ASCII character to a representation on a standard seven segment display. Like with the BCD decoder, there are inputs that affect the display of the digits 6, 7, and 9. There are also two choices of "font" and the option to display lowercase letters as uppercase or as lowercase.

FS=0:



FS=1:



The signals used in this mode are:

- **/AL** - Active low. If HIGH, outputs will be HIGH when lit. If LOW, outputs will be LOW when lit.
- **/BI** - Blanking input. If LOW, all segments will be blank regardless of other inputs.
- **FS** - Font select. Selects one of two "fonts."
- **LC** - Lower case. If LOW, lowercase letters will appear as uppercase.
- **X6** - When HIGH, the extra segment **a** will be lit on the digit 6.
- **X7** - When HIGH, the extra segment **f** will be lit on the digit 7.
- **X9** - When HIGH, the extra segment **d** will be lit on the digit 9.
- **D0**...**D6** - ASCII input from least significant bit **D0** to most significant bit **D6**.
- **a**, **b**, **c**, **d**, **e**, **f**, **g** - Outputs for a seven segment display.
- **/LTR** - Letter. LOW when the input is a letter (A...Z or a...z).

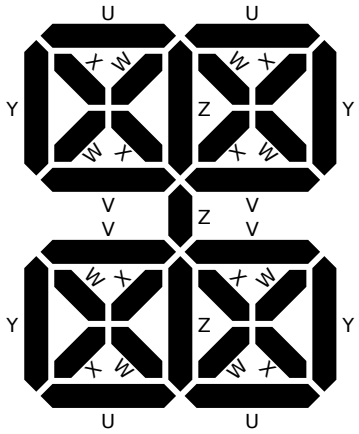
The pin assignments in this mode are:

	Dedicated Input	Dedicated Output	Bidirectional
0	D0	Segment a	Input - X6
1	D1	Segment b	Input - X7
2	D2	Segment c	Input - X9
3	D3	Segment d	Input - FS
4	D4	Segment e	Input - /BI

	Dedicated Input	Dedicated Output	Bidirectional
5	D5	Segment f	Input - /AL
6	D6	Segment g	Input - HIGH
7	LC	/LTR	Input - LOW

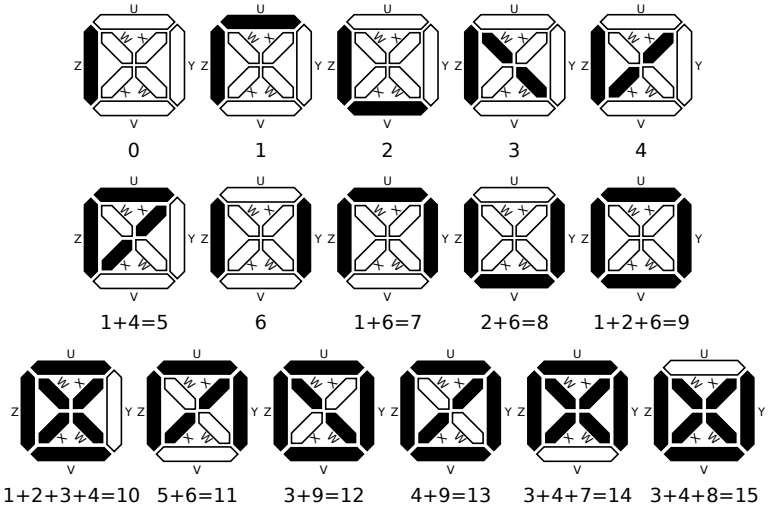
Dual BCD to Cistercian numeral decoder

This mode converts two decimal digits in BCD to their representations on the segmented display for Cistercian numerals shown below.



The patterns produced for each input value are shown below.

Patterns as seen in top right (units) position:



The signals used in this mode are:

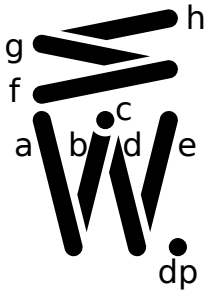
- **/AL** - Active low. If HIGH, outputs will be HIGH when lit. If LOW, outputs will be LOW when lit.
- **/BI** - Blanking input. If LOW, all segments will be blank regardless of other inputs, including **/LT1** and **/LT2**.
- **/LT1** - Lamp test for digit 1. When **/BI** is HIGH and **/LT1** is LOW, all segments for digit 1 will be lit.
- **/LT2** - Lamp test for digit 2. When **/BI** is HIGH and **/LT2** is LOW, all segments for digit 2 will be lit.
- **A1**, **B1**, **C1**, **D1** - BCD input for digit 1 from least significant bit **A1** to most significant bit **D1**.
- **A2**, **B2**, **C2**, **D2** - BCD input for digit 2 from least significant bit **A2** to most significant bit **D2**.
- **U1**, **V1**, **W1**, **X1**, **Y1** - Outputs for digit 1 on a Cistercian segmented display.
- **U2**, **V2**, **W2**, **X2**, **Y2** - Outputs for digit 2 on a Cistercian segmented display.

The pin assignments in this mode are:

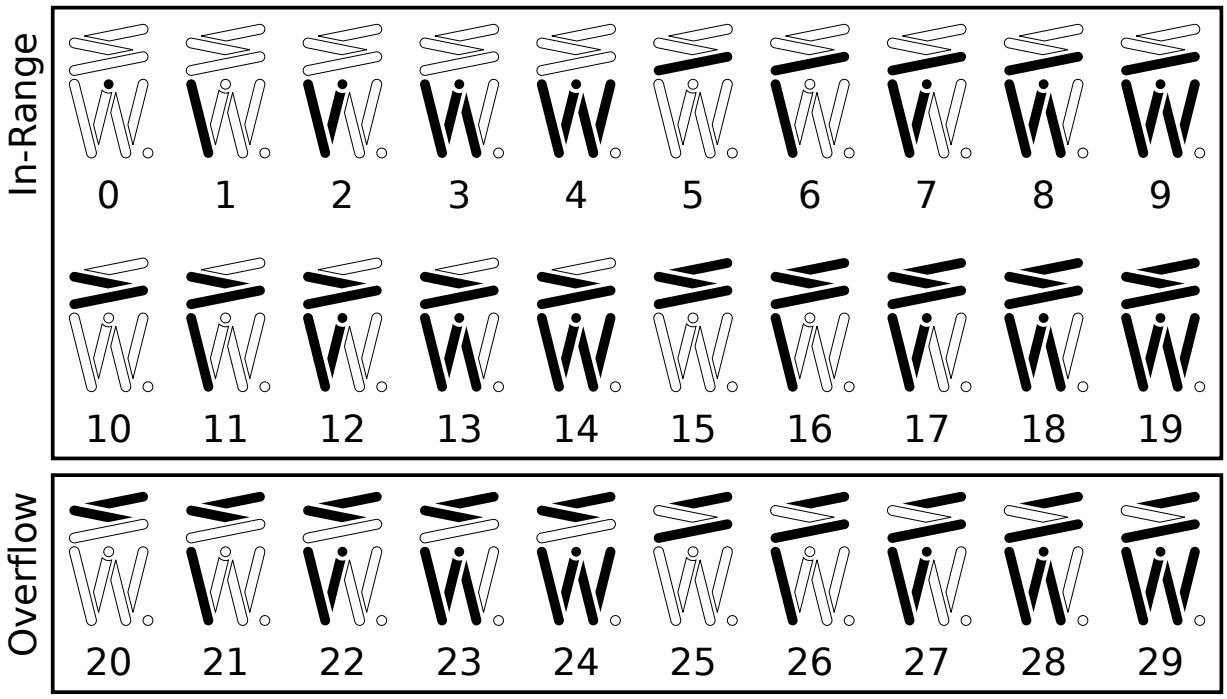
	Dedicated Input	Dedicated Output	Bidirectional
0	A1	Segment U1	Output - Y1
1	B1	Segment U2	Output - Y2
2	C1	Segment V1	Input - /LT1
3	D1	Segment V2	Input - /LT2
4	A2	Segment W1	Input - /BI
5	B2	Segment W2	Input - /AL
6	C2	Segment X1	Input - LOW
7	D2	Segment X2	Input - HIGH

BCV to Kaktovik numeral decoder

This mode converts a *vigesimal* (base 20) digit in BCV (binary-coded vigesimal) to its representation on the segmented display for Kaktovik numerals shown below.



The patterns produced for each input value are shown below.



The signals used in this mode are:

- **/AL** - Active low. If HIGH, outputs will be HIGH when lit. If LOW, outputs will be LOW when lit.
- **/BI** - Blanking input. If LOW, all segments will be blank regardless of other inputs, including **/LT**.
- **/LT** - Lamp test. When **/BI** is HIGH and **/LT** is LOW, all segments will be lit.
- **/RBI** - Ripple blanking input. If the BCV value is zero and **/RBI** is LOW, all segments will be blank.
- **/VBI** - Overflow blanking input. If the BCV value is out of range and **/VBI** is LOW, all segments will be blank.
- **A**, **B**, **C**, **D**, **E** - BCV input from least significant bit **A** to most significant bit **E**.
- **a**, **b**, **c**, **d**, **e**, **f**, **g**, **h** - Outputs for a Kaktovik segmented display.
- **/RBO** - Ripple blanking output. HIGH when BCV value is nonzero or **/RBI** is HIGH.
- **V** - Overflow. HIGH when BCV value is out of range (greater than or equal to 20).

The pin assignments in this mode are:

	Dedicated Input	Dedicated Output	Bidirectional
0	A	Segment a	Output - h
1	B	Segment b	Output - V
2	C	Segment c	

	Dedicated Input	Dedicated Output	Bidirectional
3	D	Segment d	Input - /LT
4	E	Segment e	Input - /BI
5		Segment f	Input - /AL
6	/VBI	Segment g	Input - HIGH
7	/RBI	/RBO	Input - HIGH

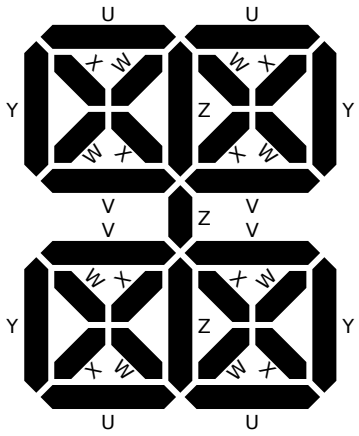
How to test

The `test` directory includes extensive tests for each of the four modules.

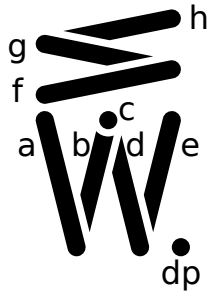
External hardware

For the BCD and ASCII modes, a standard seven-segment display is used.

For the Cistercian mode, a segmented display like the one below is used. There are design files for such a display here.



For the Kaktovik mode, a segmented display like the one below is used. There are design files for such a display here.



Pinout

#	Input	Output	Bidirectional
0	A; D0; A1; A	Segment a; U1; a	X6; X6; Y1; h
1	B; D1; B1; B	Segment b; U2; b	X7; X7; Y2; V
2	C; D2; C1; C	Segment c; V1; c	X9; X9; /LT1; -
3	D; D3; D1; D	Segment d; V2; d	/LT; FS; /LT2; /LT
4	V0; D4; A2; E	Segment e; W1; e	/BI (blanking input)
5	V1; D5; B2; -	Segment f; W2; f	/AL (active low)
6	V2; D6; C2; /VBI	Segment g; X1; g	M0 (mode select)
7	/RBI; LC; D2; /RBI	/RBO; /LTR; X2; /RBO	M1 (mode select)

Hardware UTF Encoder/Decoder [519]

- Author: Rebecca G. Bettencourt
- Description: Converts Unicode code points between UTF-8, UTF-16, and UTF-32.
- GitHub repository
- HDL project
- Mux address: 519
- Extra docs
- Clock: 0 Hz

How it works

This project contains hardware logic to convert between the UTF-8, UTF-16, and UTF-32 encodings for Unicode text.

It will detect and raise an error signal on overlong encodings, out of range code point values, and invalid byte sequences.

(You can optionally disable range checking if you wish to use the original UTF-8 spec that supports values up to 0x7FFFFFFF.)

Basic operation

- In the initial state, all dedicated inputs should be set HIGH.
- At any time, set /RESET (rst_n) LOW and pulse CLK to reset all inputs and outputs to initial state.
- At any time, set /ROUT (input 0) LOW and pulse CLK to seek to the beginning of the output.
- You can set ERRS or /PROPS (input 1) HIGH to get an error status on the dedicated outputs.
- You can set ERRS or /PROPS (input 1) LOW to get character properties on the dedicated outputs.
- You can set CHK (input 2) HIGH to raise an error signal when the code point value is out of range (0x110000).
- You can set CHK (input 2) LOW to ignore out of range code point values and encode/decode values up to 0x7FFFFFFF.
- You can set CBE (input 3) HIGH to specify big endian order for UTF-32 and UTF-16 input and output.
- You can set CBE (input 3) LOW to specify little endian order for UTF-32 and UTF-16 input and output.

Inputting UTF-32

1. Set READ or /WRITE (input 4) LOW.
2. Set /CIO (input 5, character I/O) LOW.
3. Set bidirectional I/O to the first byte of the UTF-32 word and pulse CLK.
4. Set bidirectional I/O to the second byte of the UTF-32 word and pulse CLK.
5. Set bidirectional I/O to the third byte of the UTF-32 word and pulse CLK.
6. Set bidirectional I/O to the fourth byte of the UTF-32 word and pulse CLK.
7. Set /CIO (input 5, character I/O) HIGH.
8. Set READ or /WRITE (input 4) HIGH.
9. If READY (output 0) is HIGH and ERROR (output 5) is LOW, the input and output are both valid.
10. If READY (output 0) is LOW or ERROR (output 5) is HIGH, the input was out of range (0x110000 or, if CHK is LOW, 0x80000000).

Inputting UTF-16

1. Set ERRS or /PROPS (input 1) LOW.
2. Set READ or /WRITE (input 4) LOW.
3. Set /UIO (input 6, UTF-16 I/O) LOW.
4. Set bidirectional I/O to the first byte of the first UTF-16 word and pulse CLK.
5. Set bidirectional I/O to the second byte of the first UTF-16 word and pulse CLK.
6. If HIGHCHAR (output 3) is LOW, skip to step 9.
7. Set bidirectional I/O to the first byte of the second UTF-16 word and pulse CLK.
8. Set bidirectional I/O to the second byte of the second UTF-16 word and pulse CLK.
9. Set /UIO (input 6, UTF-16 I/O) HIGH.
10. Set READ or /WRITE (input 4) HIGH.
11. Set ERRS or /PROPS (input 1) HIGH.
12. If READY (output 0) is HIGH and ERROR (output 5) is LOW, the input and output are both valid.
13. If RETRY (output 1) is HIGH, the first word was a high surrogate but the second word was not a low surrogate. The output will be the high surrogate only; the last word will need to be processed again.

Inputting UTF-8

1. Set READ or /WRITE (input 4) LOW.
2. Set /BIO (input 7, byte I/O) LOW.
3. Set bidirectional I/O to the current byte of the UTF-8 sequence and pulse CLK.

4. Repeat step 3 until READY (output 0) or ERROR (output 5) is HIGH.
5. If READY (output 0) is HIGH and ERROR (output 5) is LOW, the input and output are both valid.
6. If RETRY (output 1) is HIGH, the UTF-8 sequence was truncated (not enough continuation bytes). The output will be the truncated sequence only; the last byte will need to be processed again.
7. If INVALID (output 2) is HIGH, the UTF-8 sequence was a single continuation byte or invalid byte (0xFE or 0xFF).
8. If OVERLONG (output 3) is HIGH, the UTF-8 sequence was an overlong encoding.
9. If NONUNI (output 4) is HIGH, the UTF-8 sequence was out of range (0x110000).

Outputting UTF-32

1. Set READ or /WRITE (input 4) HIGH.
2. Set /CIO (input 5, character I/O) LOW.
3. Pulse CLK and read the first byte of the UTF-32 word from the bidirectional I/O.
4. Pulse CLK and read the second byte of the UTF-32 word from the bidirectional I/O.
5. Pulse CLK and read the third byte of the UTF-32 word from the bidirectional I/O.
6. Pulse CLK and read the fourth byte of the UTF-32 word from the bidirectional I/O.
7. Set /CIO (input 5, character I/O) HIGH.
8. If the UTF-32 word is within range, the input and output are both valid.
9. If the UTF-32 word is not within range, then the input was either incomplete or invalid.

Outputting UTF-16

1. Set READ or /WRITE (input 4) HIGH.
2. If UEOF (output 6) is HIGH, then the input was either incomplete or invalid.
3. Set /UIO (input 6, UTF-16 I/O) LOW.
4. Pulse CLK and read the next byte of the UTF-16 sequence from the bidirectional I/O.
5. Repeat step 4 until UEOF (output 6) is HIGH.
6. Set /UIO (input 6, UTF-16 I/O) HIGH.

Outputting UTF-8

1. Set READ or /WRITE (input 4) HIGH.
2. If BEOF (output 7) is HIGH, then the input was either incomplete or invalid.
3. Set /BIO (input 7, byte I/O) LOW.
4. Pulse CLK and read the next byte of the UTF-8 sequence from the bidirectional I/O.
5. Repeat step 4 until BEOF (output 7) is HIGH.
6. Set /BIO (input 7, byte I/O) HIGH.

Error status

When ERRS or /PROPS (input 1) is HIGH, the dedicated outputs will be:

#	Name	Meaning
0	READY	The input and output are complete sequences.
1	RETRY	The previous input was invalid or the start of another sequence and was ignored. Process the output, reset, and try the previous input again.
2	INVALID	The input and output are invalid.
3	OVERLONG	The UTF-8 input was an overlong sequence.
4	NONUNI	The code point value is out of range (0x110000). (This is set independently of the CHK input; the CHK input only changes whether this counts as an error.)
5	ERROR	Equivalent to (RETRY or INVALID or OVERLONG or (NONUNI and CHK)).

If all of these outputs are LOW, the accumulated input is incomplete and more input is required (underflow).

Character properties

When ERRS or /PROPS (input 1) is LOW, the dedicated outputs will be:

#	Name	Meaning
0	NORMAL	The code point value is valid and not a C0 or C1 control character, surrogate, private use character, or noncharacter.
1	CONTROL	The code point value is valid and a C0 or C1 control character (0x00-0x1F or 0x7F-0x9F).
2	SURROGATE	The code point value is valid and a UTF-16 surrogate (0xD800-0xDFFF).
3	HIGHCHAR	The code point value is valid and either a high surrogate (0xD800-0xDBFF) or a non-BMP character (0x10000).
4	PRIVATE	The code point value is valid and either a private use character (0xE000-0xF8FF, 0xF0000) or the high surrogate of a private use character (0xDB80-0xDBFF).
5	NONCHAR	The code point value is valid and a noncharacter (0xFDD0-0xFDEF or the last two code points of any plane).

If all of these outputs are LOW, there is no valid code point in the output.

How to test

The `test.py` file covers a comprehensive set of test cases which are listed in a separate file to avoid bloating the TT09 manual.

External hardware

Any device that needs to process Unicode text.

Pinout

#	Input	Output	Bidirectional
0	/ROUT	READY; NORMAL	I/O LSB
1	ERRS, /PROPS	RETRY; CONTROL	I/O
2	CHK	INVALID; SURROGATE	I/O
3	CBE, /CLE	OVERLONG; HIGHCHAR	I/O
4	READ, /WRITE	NONUNI; PRIVATE	I/O
5	/CIO	ERROR; NONCHAR	I/O
6	/UIO	UEOF	I/O
7	/BIO	BEOF	I/O MSB

Simon Says memory game [521]

- Author: Uri Shaked
- Description: Repeat the sequence of colors and sounds to win the game
- GitHub repository
- HDL project
- Mux address: 521
- Extra docs
- Clock: 50000 Hz

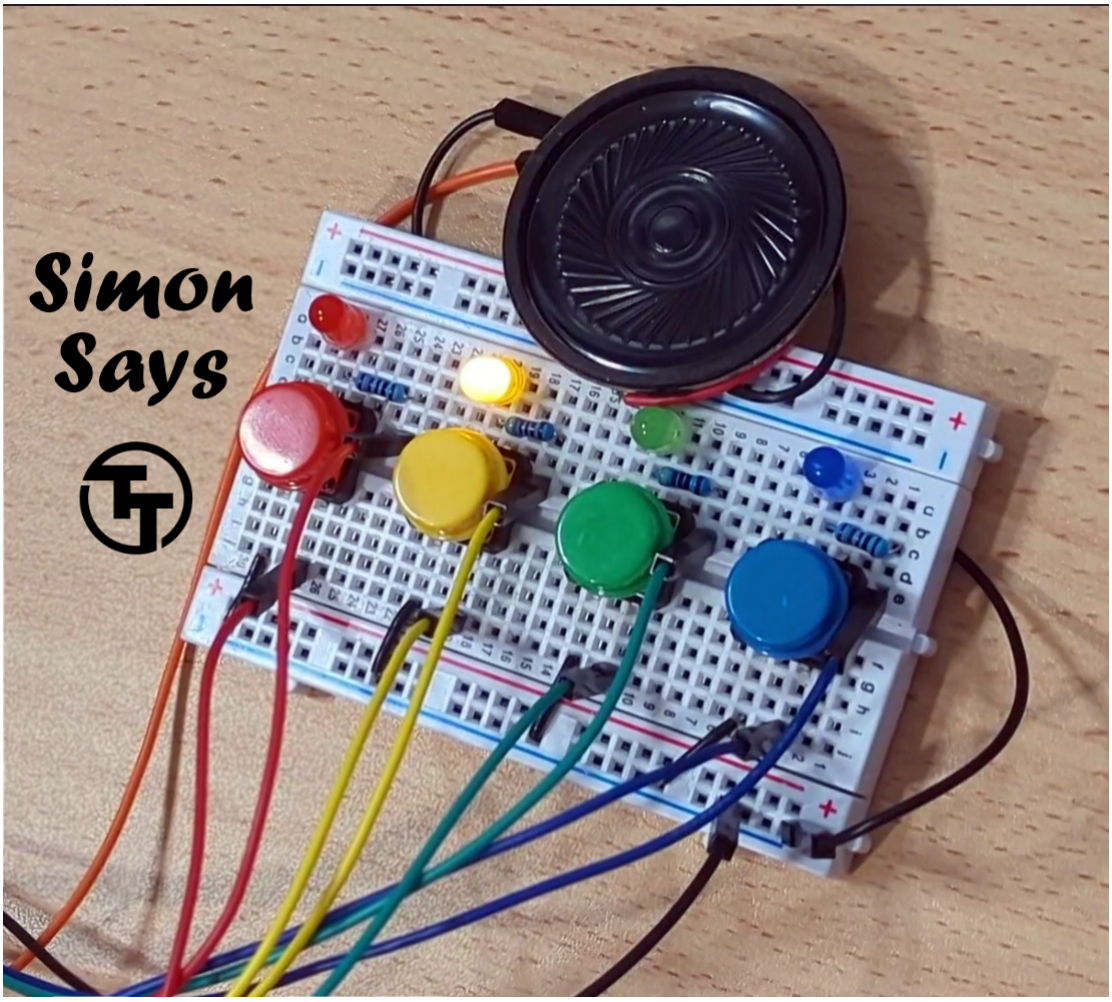


Figure 38: Simon Says Game

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/408757730664700929 (including wiring diagram).

How to test

Use a Simon Says Pmod to test the game.

Provide a 50 KHz clock input, reset the game, and enjoy!

If you don't have the Pmod, you can still connect the hardware manually as follows:

1. Connect the four push buttons to pins `btn1`, `btn2`, `btn3`, and `btn4`. 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.). Don't forget current-limiting resistors!
3. Connect the speaker to the `speaker` pin (optional).
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!

External Hardware

Simon Says Pmod or four push buttons (with pull-down resistors), four LEDs, and optionally a speaker/buzzer and two digit 7-segment display.

Pinout

#	Input	Output	Bidirectional
0	btn1	led1	seg_a
1	btn2	led2	seg_b

#	Input	Output	Bidirectional
2	btn3	led3	seg_c
3	btn4	led4	seg_d
4	seginv	speaker	seg_e
5		dig1	seg_f
6		dig2	seg_g
7			

VC 16-bit CPU [522]

- Author: Paul Campbell
- Description: VC 16-bit CPU - RISV-C cpu
- GitHub repository
- HDL project
- Mux address: 522
- Extra docs
- Clock: 0 Hz

How it works

CPU plus MMU

How to test

needs external RAM system

External hardware

external RAM

Pinout

#	Input	Output	Bidirectional
0	ReadData0	AddressData0	AddressLSB
1	ReadData1	AddressData1	WriteStrobe
2	ReadData2	AddressData2	AddressLatchHi
3	ReadData3	AddressData3	AddressLatchLo
4	ReadData4	AddressData4	unused4
5	ReadData5	AddressData5	unused5
6	ReadData6	AddressData6	unused6
7	ReadData7	AddressData7	InterruptIn

Latch test [523]

- Author: htfab
- Description: Verify that $*DLATCH_N* can be properly techmapped
- GitHub repository
- HDL project
- Mux address: 523
- Extra docs
- Clock: 0 Hz

How it works

Provides a P latch and an N latch to test if they can be hardened correctly.

How to test

The P latch should transparently pass through `P_D` to `P_Q` when `P_E` is high and keep its state when `P_E` is low.

The N latch should transparently pass through `N_D` to `N_Q` when `N_E` is low and keep its state when `N_E` is high.

External hardware

None

Pinout

#	Input	Output	Bidirectional
0	p_e	p_q	
1	p_d	n_q	
2	n_e		
3	n_d		
4			
5			
6			
7			

Classic 8-bit era Programmable Sound Generator AY-3-8913 [544]

- 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
- HDL project
- Mux address: 544
- Extra docs
- Clock: 2000000 Hz

How it works

This Verilog implementation is a replica of the classical **AY-3-8913** programmable sound generator. With roughly a 1500 logic gates this design fits on a **single tile** of the TinyTapeout.

The goals of this project

1. closely replicate the behavior and eventually the complete **design of the original** AY-3-891x with builtin DACs
2. provide a readable and well documented code for educational and hardware **preservation** purposes
3. leverage the **modern fabrication** process

A significant effort was put into a thorough **test suite** for regression testing and validation against the original chip behavior.

Chip technical capabilities

- **3 square wave** tone generators
- A single **white noise** generator
- A single **envelope** generator able to produce 10 different shapes
- Chip is capable to produce a range of waves from a **30 Hz** to **125 kHz**, defined by **12-bit** registers.
- **16** different volume levels

Registers The behavior of the AY-3-891x is defined by 14 registers.

Register	Bits used	Function	Description
0	xxxxxxxx	Channel A Tone	8-bit fine frequency
1	xxxx	—//—	4-bit coarse frequency
2	xxxxxxxx	Channel B Tone	8-bit fine frequency
3	xxxx	—//—	4-bit coarse frequency
4	xxxxxxxx	Channel C Tone	8-bit fine frequency
5	xxxx	—//—	4-bit coarse frequency
6	...xxxxx	Noise	5-bit noise frequency
7	..CBACBA	Mixer	Tone and/or Noise per channel
8	...xxxxx	Channel A Volume	Envelope enable or 4-bit amplitude
9	...xxxxx	Channel B Volume	Envelope enable or 4-bit amplitude
10	...xxxxx	Channel C Volume	Envelope enable or 4-bit amplitude
11	xxxxxxxx	Envelope	8-bit fine frequency
12	xxxxxxxx	—//—	8-bit coarse frequency
13	xxxx	Envelope Shape	4-bit shape control

Square wave tone generators Square waves are produced by counting down the 12-bit counters. Counter counts up from 0. Once the corresponsding register value is reached, counter is reset and the output bit of the channel is flipped producing square waves.

Noise generator Noise is produced with 17-bit Linear-feedback Shift Register (LFSR) that flips the output bit pseudo randomly. The shift rate of the LFSR register is controller by the 5-bit counter.

Envelope The envelope shape is controlled with 4-bit register, but can take only 10 distinct patterns. The speed of the envelope is controlled with 16-bit counter. Only a single envelope is produced that can be shared by any combination of the channels.

Volume Each of the three AY-3-891x channels have dedicated DAC that converts 16 levels of volume to analog output. Volume levels are 3 dB apart in AY-3-891x.

Historical use of the AY-3-891x

The AY-3-891x family of programmable sound generators was introduced by General Instrument in 1978. Soon Yamaha Corporation licensed and released a very similar chip under YM2149 name.

Both variants of the AY-3-891x and YM2149 were broadly used in home computers, game consoles and arcade machines in the early 80ies.

- home computers: Apple II Mockingboard sound card, Amstrad CPC, Atari ST, Oric-1, Sharp X1, MSX, ZX Spectrum 128/+2/+3
- game consoles: Intellivision, Vectrex, Amstrad GX4000
- arcade machines: Frogger, 1942, Spy Hunter and etc.

The AY-3-891x chip family competed with the similar Texas Instruments SN76489.

The original pinout of the AY-3-8913

The **AY-3-8913** was a 24-pin package release of the AY-3-8910 with a number of internal pins left simply unconnected. The goal of AY-3-8913 was to reduce complexity for the designer and reduce the foot print on the PCB. Otherwise the functionality of the chip is identical to AY-3-8910 and AY-3-8912.

```
         ,--._.--.
  GND ---|1    24|<-- /cs*
 BDIR -->|2    23|<--  a8*
  BC1 -->|3    22|<-- /a9*
  DA7 <->|4    21|<-- /RESET
  DA6 <->|5    20|<-- CLOCK
  DA5 <->|6    19|--- GND
  DA4 <->|7    18|--> CHANNEL C OUT
  DA3 <->|8    17|--> CHANNEL A OUT
  DA2 <->|9    16|    not connected
  DA1 <->|10   15|--> CHANNEL B OUT
  DA0 <->|11   14|<-- test*
test* <--|12   13|<-- VCC
         `-------'
  * -- omitted from this Verilog implementation
```

Difference from the original hardware

This Verilog implementation is a completely digital and synchronous design that differs from the original AY-3-8913 design which incorporated internal DACs and analog outputs.

Audio signal output While the original chip had no summation The module provides two alternative outputs for the generated audio signal:

1. digital 8-bit audio output suitable for external Digital to Analog Converter (DAC)
2. pseudo analog output through Pulse Width Modulation (PWM)

Master output channel In contrast to the original chip which had only separate channel outputs, this implementation also provides an optional summation of the channels into a single master output.

No DC offset This implementation produces output 0/1 waveforms without DC offset.

No /A8, A9 and /CS pins The combination of **/A8**, **A9** and **/CS** pins orginially were intended to select a specific sound chip out the larger array of devices connected to the same bus. In this implementation this mechanism is omitted for simplicity, **/A8**, **A9** and **/CS** are considered to be tied **low** and chip behaves as always enabled.

Synchronous reset and single phase clock The original design employed 2 phases of the clock and asynchronous reset mechanism for operation of the registers.

To make it easier to synthesize and test on FPGAs this implementation uses single clock phase and synchronous reset for registers.

The reverse engineered AY-3-891x

This implementation would not be possible without the reverse engineered schematics and analysis based on decapped AY-3-8910 and AY-3-8914 chips.

Explain how your project works

How to test

Summary of commands to communicate with the chip

The AY-3-8913 is programmed by updating its internal registers via the data bus. Below is a short summary of the communication protocol of AY-3-891x. Please consult AY-3-891x Technical Manual for more information.

BDIR	BC1	Bus state description
0	0	Bus is inactive
0	1	(Not implemented)
1	0	Write bus value to the previously latched register #
1	1	Latch bus value as the destination register #

Latch register address First, put the destination register adress on the bus of the chip and latch it by pulling both **BDIR** and **BC1** pins **high**.

Write data to register Put the desired value on the bus of the chip. Pull **BC1** pin **low** while keeping **BDIR** pin **high** to write the value of the bus to the latched register address.

Inactivate bus by pulling both **BDIR** and **BC1** pins **low**.

Register	Format	Description	Parameters
0,2,4	`ffffffff`	A/B/C tone period	`f` - low bits
1,3,5	`0000FFFF`	—//—	`F` - high bits
6	`000fffff`	Noise period	`f` - noise period
7	`00CBAcba`	Noise / tone per channel	`CBA` - noise off, `cba` - tone off
8,9,10	`000Evvvv`	A/B/C volume	`E` - envelope on, `v` - volume level
11	`ffffffff`	Envelope period	`f` - low bits
12	`FFFFFFFF`	—//—	`F` - high bits
13	`0000caAh`	Envelope Shape	`c` - continue, `a` - attack, `A` - alternate, `h` - hold

Note frequency

Use the following formula to calculate the 12-bit period value for a particular note:

$$toneperiod_{cycles} = clock_{frequency}/(16_{cycles} * note_{frequency})$$

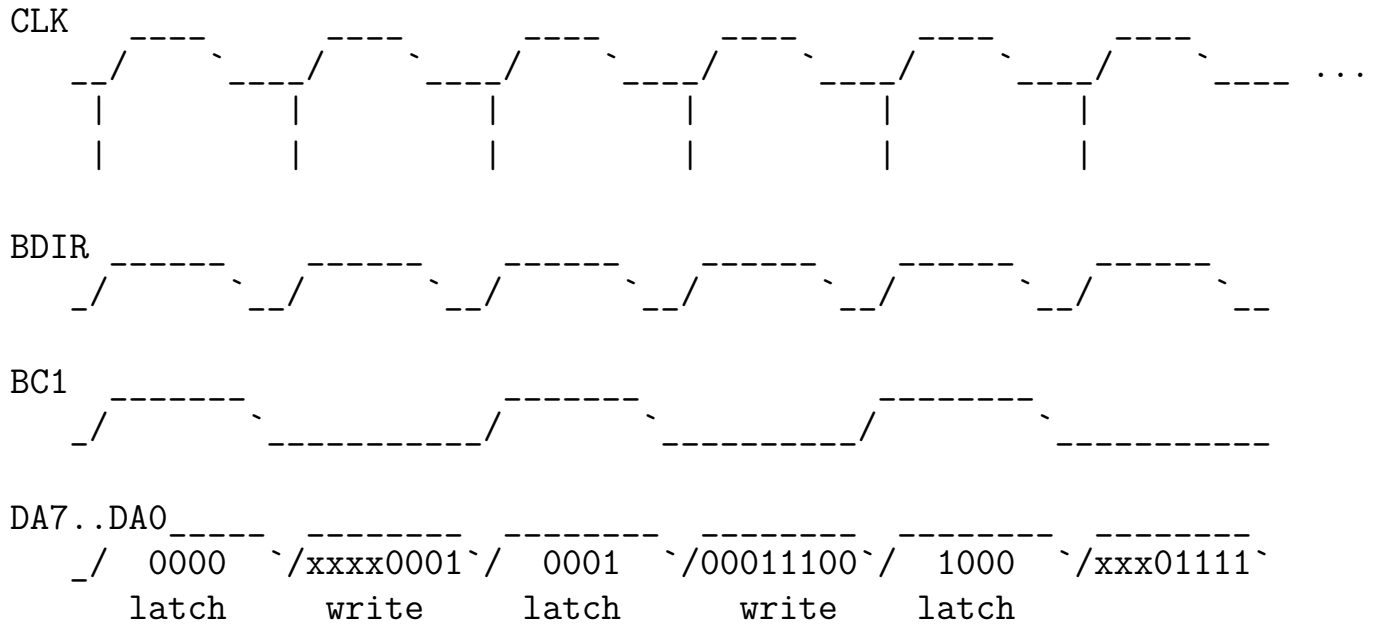
For example 12-bit period that plays 440 Hz note on a chip clocked at 2 MHz would be:

$$toneperiod_{cycles} = 2000000Hz/(16_{cycles} * 440Hz) = 284 = 11C_{hex}$$

An example to play a note at a maximum volume

BDIR	BC1	DA7..DA0	Explanation
1	1	`xxxx0000`	**Latch** tone A coarse register address $0 = 0000_{bin}$
1	0	`xxxx0001`	**Write** high 4-bits of the 440 Hz note $1 = 0001_{bin}$
1	1	`xxxx0001`	**Latch** tone A fine register address $1_{dec} = 0001_{bin}$
1	0	`00011100`	**Write** low 8-bits of the note $1C_{hex} = 00011100_{bin}$
1	1	`xxxx1000`	**Latch** channel A volume register address $8 = 1000_{bin}$
1	0	`xxx01111`	**Write** maximum volume level $15_{dec} = 1111_{bin}$ with the envelope disabled

Timing diagram



Externally configurable clock divider

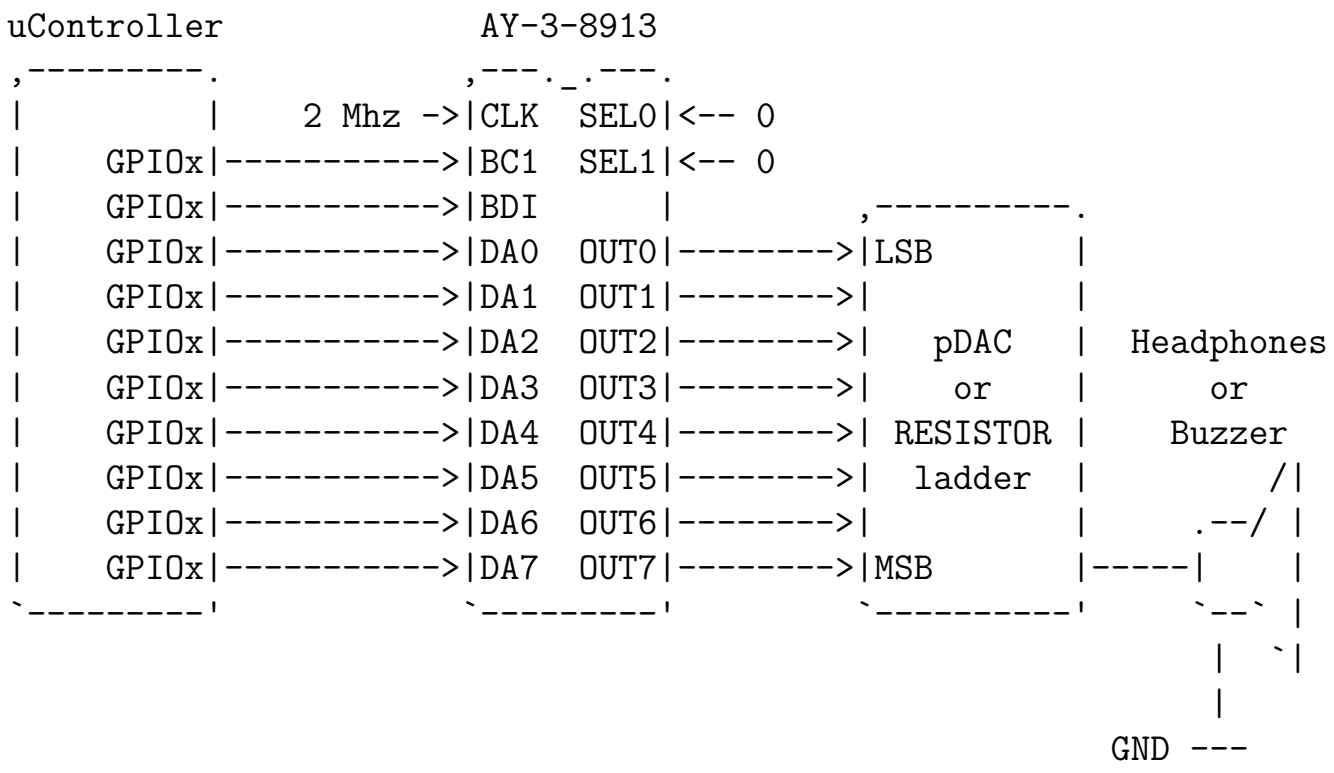
SEL1	SEL0	Description	Clock frequency
0	0	Standard mode, clock divided by 8	1.7 .. 2.0 MHz
1	1	—//—	1.7 .. 2.0 MHz
0	1	New mode for TT05, no clock divider	250 .. 500 kHZ
1	0	New mode for TT05, clock div. 128	25 .. 50 MHz

SEL1	SEL0	Formula to calculate the 12-bit tone period value for a note
0	0	$clock_{frequency}/(16_{cycles} * note_{frequency})$
1	1	—//—
0	1	$clock_{frequency}/(2_{cycles} * note_{frequency})$
1	0	$clock_{frequency}/(128_{cycles} * note_{frequency})$

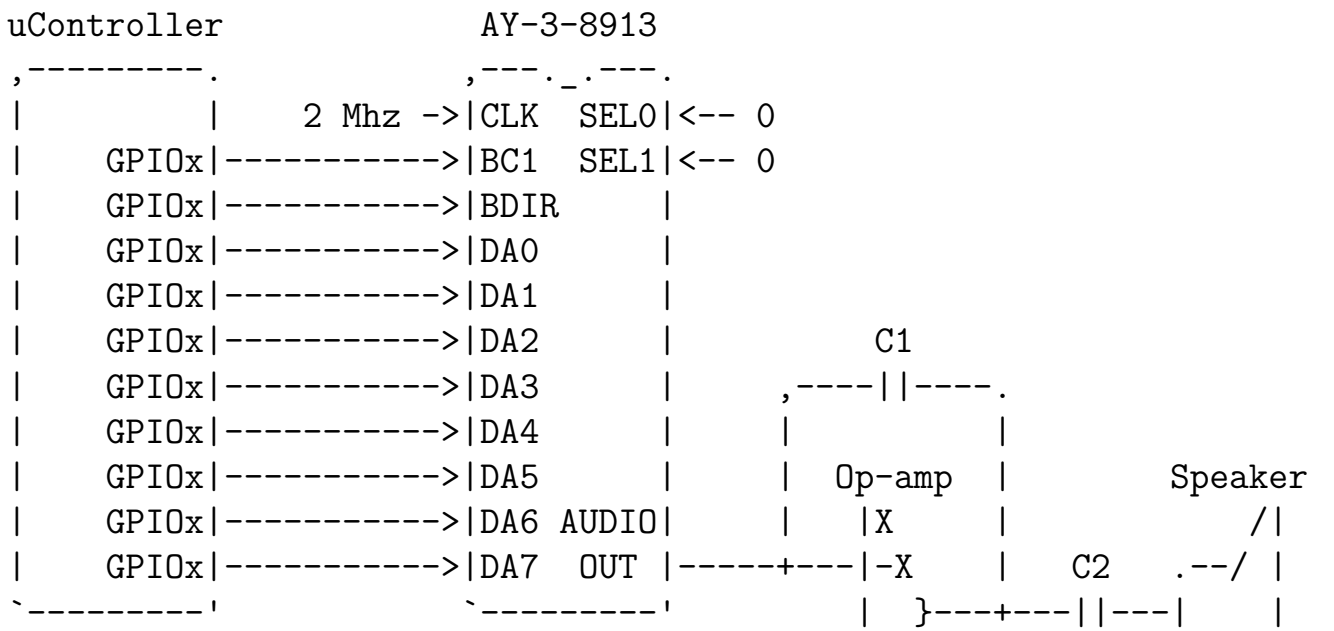
External hardware

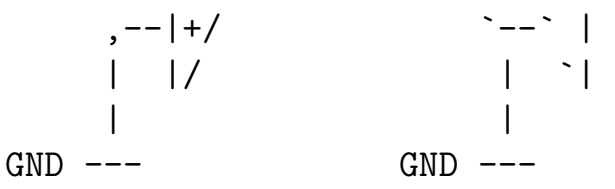
The data bus of the AY-3-8913 chip has to be connected to microcontroller and receive a regular stream of commands. The AY-3-8913 produces audio output and has to be connected to a speaker. There are several ways how the overall schematics can be established.

8-bit parallel output via DAC One option is to connect off the shelf data parallel Digital to Analog Converter (DAC) for example Digilent R2R Pmod to the output pins and route the resulting analog audio to piezo speaker or amplifier.

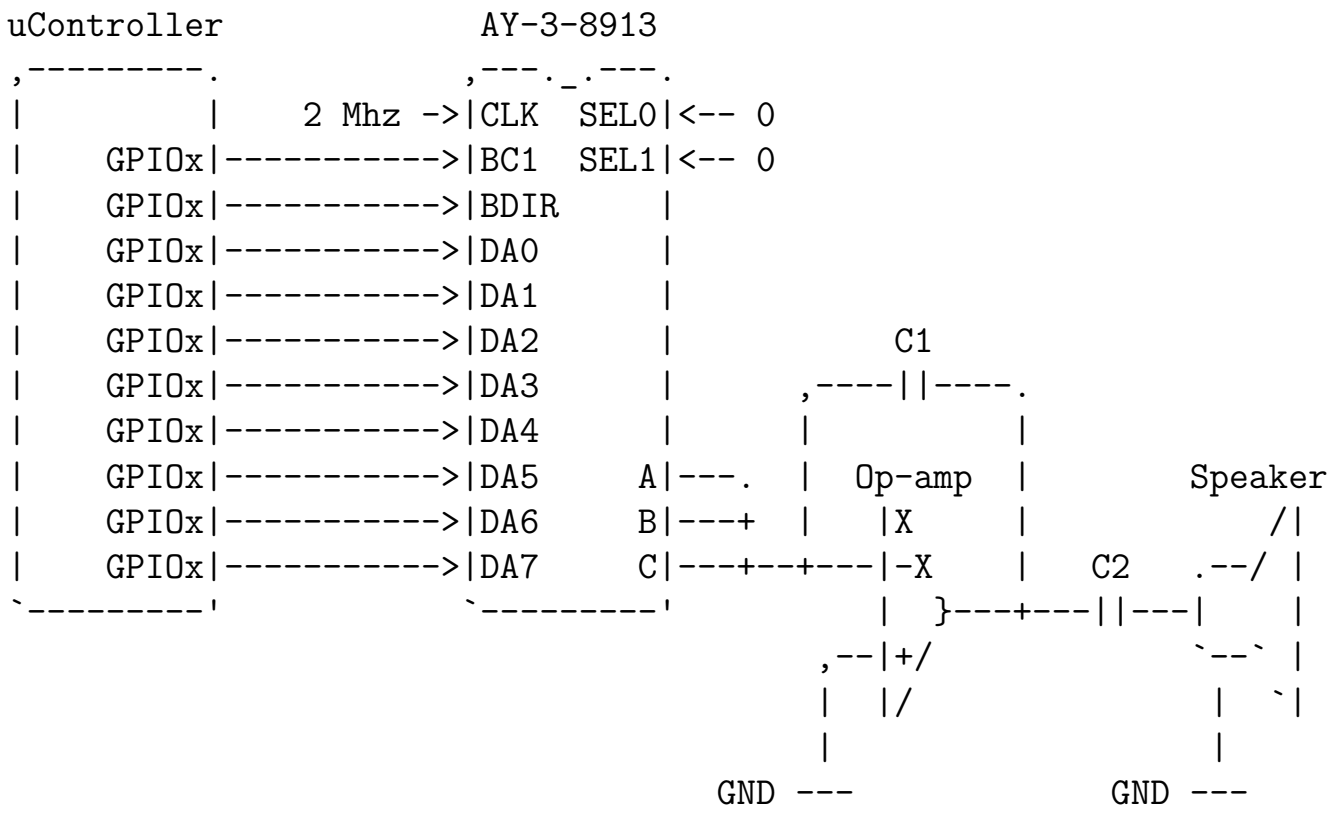


AUDIO OUT through RC filter Another option is to use the Pulse Width Modulated (PWM) AUDIO OUT pin that combines 4 channels with the Resistor-Capacitor based low-pass filter or better the Operation Amplifier (Op-amp) & Capacitor based integrator:





Separate channels through the Op-amp The third option is to externally combine 4 channels with the Operational Amplifier and low-pass filter:



Pinout

#	Input	Output	Bidirectional
0	DA0 - multiplexed data/address bus LSB	audio out (PWM)	(in) **BC1** bus control
1	DA1 - multiplexed data/address bus	digita audio LSB	(in) **BDIR** bus direction

#	Input	Output	Bidirectional
2	DA2 - multiplexed data/address bus	digita audio	(in) **SEL0** clock divider
3	DA3 - multiplexed data/address bus	digita audio	(in) **SEL1** clock divider
4	DA4 - multiplexed data/address bus	digita audio	(out) channel A (PWM)
5	DA5 - multiplexed data/address bus	digita audio	(out) channel B (PWM)
6	DA6 - multiplexed data/address bus	digita audio	(out) channel C (PWM)
7	DA7 - multiplexed data/address bus MSB	digita audio MSB	(out) AUDIO OUT master (PWM)

VGA Scroller [545]

- Author: FavoritoHJS
- Description: Scrolls across a very pixelated cityscape
- GitHub repository
- HDL project
- Mux address: 545
- Extra docs
- Clock: 25000000 Hz

How it works

The terrain is based on an LFSR, using the deterministic randomness of one to generate each layer of the city.

How to test

Set Clock to 25.18MHz, and use a Tiny VGA carrier board for video.

External hardware

This project requires a Tiny VGA carrier board to display video.

Pinout

#	Input	Output	Bidirectional
0		Rh	
1		Gh	
2		Bh	
3		vsync	
4		Rl	
5		Gl	
6		Bl	
7		hsync	

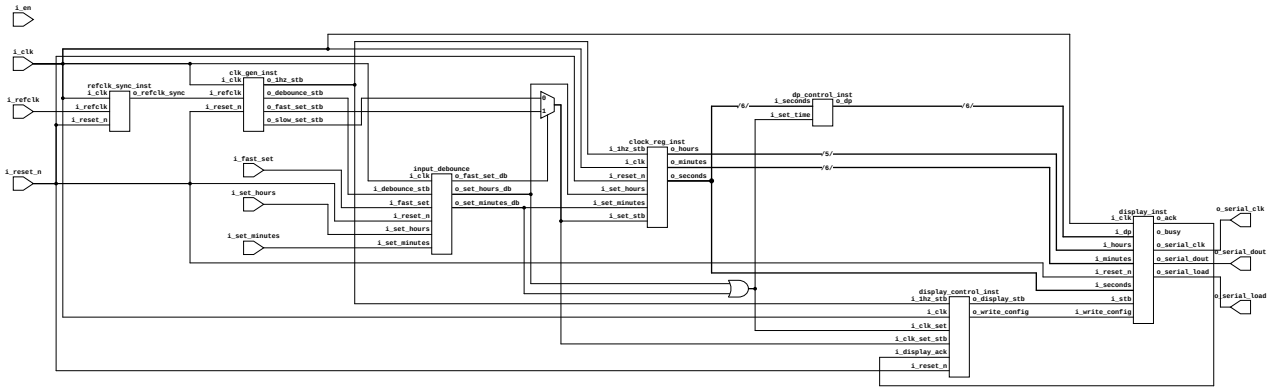
Digital Desk Clock v2.0 [546]

- Author: Sam Ellicott
- Description: 7-Segment Digital Desk Clock for ihp Tapeout
- GitHub repository
- HDL project
- Mux address: 546
- Extra docs
- Clock: 5000000 Hz

How it works

Simple digital clock, displays hours, minutes, and seconds in either a 24h format. Since there are not enough output pins to directly drive a 6x 7-segment displays, the data is shifted out over SPI to a MAX7219 in 7-segment mode. The time can be set using the `hours_set` and `minutes_set` inputs. If `set_fast` is high, then the the hours or minutes will be incremented at a rate of 5Hz, otherwise it will be set at a rate of 2Hz. Note that when setting either the minutes, rolling-over will not affect the hours setting. If both `hours_set` and `minutes_set` are presssed at the same time the seconds will be cleared to zero.

A block diagram of the system is shown below.



How to test

Apply a 5MHz clock to the clock pin and 32.786Khz signal to the refclk pin. Use the `hours_set` and `minutes_set` pins to set the time.

External hardware

Connect the BIDIR PMOD to a MAX7219 7-segment display, For reference Tiny Tape-out SPI

Pinout

#	Input	Output	Bidirectional
0	refclk		Display CS
1			Display MOSI
2	Fast/Slow Set		
3	Set Hours		Display SCK
4	Set Minutes		
5			
6			
7			

Glyph Mode [547]

- Author: James Ross
- Description: Submission for VGA Demoscene
- GitHub repository
- HDL project
- Mux address: 547
- Extra docs
- Clock: 25175000 Hz

How it works

This is a standalone VGA demo that runs with or without input. It will accept two pins `ui_io[0]` and `ui_io[1]` for palette color selection:

`ui_io[1:0]`	Palette
0	Green (default)
1	Red
2	Blue
3	Pride

How to test

Plug into a VGA monitor and select this circuit to test

External hardware

Requires the TinyVGA PMOD

Pinout

#	Input	Output	Bidirectional
0	Palette 0	R1	
1	Palette 1	G1	
2		B1	
3		VSync	
4		R0	

#	Input	Output	Bidirectional
5		G0	
6		B0	
7		HSync	

Giant Ring Oscillator (3853 inverters) [548]

- Author: Uri Shaked
- Description: Configurable ring oscillator with up to 3853 inverters
- GitHub repository
- HDL project
- Mux address: 548
- Extra docs
- Clock: 0 Hz

How it works

A giant, configurable ring oscillator with up to 3853 stages. To enable the ring oscillator, connect one of the output pins to the first input pin (`ring_in` / `ui_in[0]`). Each output pin is connected at a different point in the ring oscillator chain, making it possible to create rings of different lengths:

Pin	Chain length
uo[0]	1
uo1	3
uo2	5
uo[3]	7
uo[4]	11
uo[5]	21
uo[6]	51
uo[7]	101
uio[0]	201
uio1	501
uio2	1001
uio[3]	2001
uio[4]	3001
uio[5]	3853

How to test

Connect one of the output pins (e.g. `uio_out[5]`) to `ring_in`, and measure the output frequency.

External hardware

A scope / logic analyzer to measure the output frequency and the delay between different points in the inverter chain.

Pinout

#	Input	Output	Bidirectional
0	ring_in	len1	len201
1		len3	len501
2		len5	len1001
3		len7	len2001
4		len11	len3001
5		len21	len3853
6		len51	
7		len101	

cfib Demoscene Entry [549]

- Author: Christian Fibich
- Description: Generates VGA video and PWM audio
- GitHub repository
- HDL project
- Mux address: 549
- Extra docs
- Clock: 50000000 Hz

How it works

My entry to the Tinytapeout Demoscene Competition.

It (pseudo-randomly) generates a soundtrack via PWM and displays a waveform via VGA.

How to test

Connect VGA and PWM Pmod.

Then just apply clock and (asynchronous) reset.

External hardware

The project uses:

- Tiny VGA Pmod via `uo_out[7:0]` (https://github.com/mole99/tiny-vga)
- Mike's audio Pmod via `uio_out[7]` (https://github.com/MichaelBell/tt-audio-pmod)

Pinout

#	Input	Output	Bidirectional
0		r1	
1		g1	
2		b1	
3		vsync	
4		r[0]	

#	Input	Output	Bidirectional
5		g[0]	
6		b[0]	
7		hsync	pwm

DDR throughput and flop aperature test [550]

- Author: Darryl Miles project from Eric Smith
- Description: Grab data on every edge of clock with varying pos pulse width
- GitHub repository
- HDL project
- Mux address: 550
- Extra docs
- Clock: 0 Hz

How it works

Badly probably.

Use a positive edge detector on the clock and its compliment. Or together those dectors to get 2 positive pulses per period or a 2x clock. Vary clk 2x pos pulse width by varying number of inv per detect.

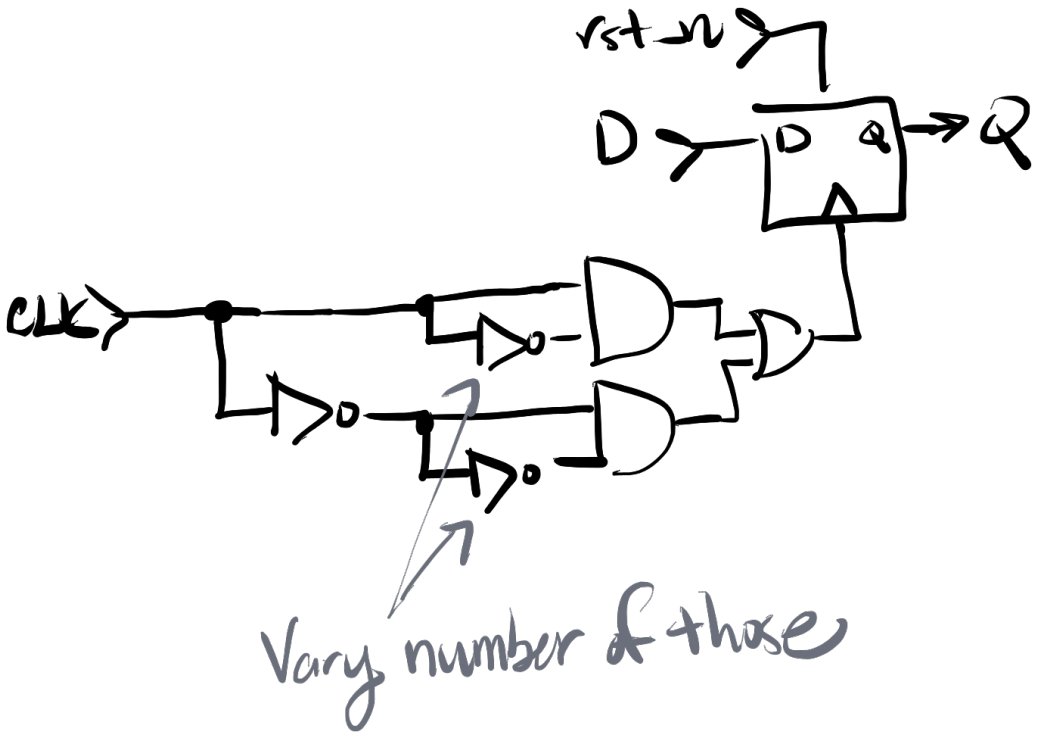


Figure 39: Concept Diagram

How to test

Carefully.

External hardware

Analog Discovery 3

Pinout

#	Input	Output	Bidirectional
0	pulse = 1 inv	q for pulse = 1 inv	
1	pulse = 3 inv	q for pulse = 3 inv	
2	pulse = 5 inv	q for pulse = 5 inv	
3	pulse = 7 inv	q for pulse = 7 inv	
4		q for normal flop	
5		1	
6		1	
7		clk	

TTIHP VGA FUN! [551]

- Author: algofoogle (Anton Maurovic) + htfab
- Description: The digital part of TT08 VGA FUN! with a simple sequencer loop
- GitHub repository
- HDL project
- Mux address: 551
- Extra docs
- Clock: 0 Hz

How it works

It's the digital block from algofoogle's TT08 VGA FUN! project, with a simple sequencer loop to make it work standalone and some dithering to simulate 8 bit output on the 4 bit Digilent PmodVGA.

How to test

Plug it into a VGA monitor, reset the project, then sit back and enjoy.

You can also manually select the mode and bit depth if you override the sequencer by pulling bit 7 of the input high.

External hardware

Digilent PmodVGA

Pinout

#	Input	Output	Bidirectional
0	(mode bit 0)	r0	g0
1	(mode bit 1)	r1	g1
2	(mode bit 2)	r2	g2
3	(depth bit 0)	r3	g3
4	(depth bit 1)	b0	hsync
5	(depth bit 2)	b1	vsync
6	(depth bit 3)	b2	
7	override sequencer	b3	

Example of Bad Synchronizer [552]

- Author: Darryl Miles project from Eric Smith
- Description: Figure 29.3 from Dally & Harting
- GitHub repository
- HDL project
- Mux address: 552
- Extra docs
- Clock: 0 Hz

How it works

Badly

This project is based on (https://github.com/ericsmi/tt07-bad-synchronizer) but for IHP130.

How to test

Align clocks, push them apart, look for bit errors

External hardware

A way to generate a clock

Pinout

#	Input	Output	Bidirectional
0	clk1	stage3[0]	stage2[0]
1		stage31	stage21
2		stage32	stage22
3		stage3[3]	stage2[3]
4		skew	stage1[0]
5			stage11
6			stage12
7			stage1[3]

Pulse Width Counter [553]

- Author: Devin Atkin
- Description: Pulse Width Counter Accessible Over UART
- GitHub repository
- HDL project
- Mux address: 553
- Extra docs
- Clock: 50000000 Hz

How it works

Simple Test Project which counts the width of an input square wave returning the time high, time low, and period.

How to test

This project works by counting the time high, time low, and period of the input signal. The design can be tested by feeding in a PWM signal and reading how the output changes.

External hardware

No external hardware is required for this proejct ot function.

Pinout

#	Input	Output	Bidirectional
0	freq_in	time_hi_lo_per[0]	uart_tx
1	out_sel[0]	time_hi_lo_per1	uart_rx
2	out_sel1	time_hi_lo_per2	uart_tx_ready
3	out_sel2	time_hi_lo_per[3]	uart_tx_valid
4	out_sel[3]	time_hi_lo_per[4]	uart_rx_valid
5		time_hi_lo_per[5]	uart_rx_ready
6		time_hi_lo_per[6]	
7		time_hi_lo_per[7]	

Ring Oscillator (5 inverter) [555]

- Author: Darryl Miles
- Description: Ring Oscillator (5 inverter)
- GitHub repository
- HDL project
- Mux address: 555
- Extra docs
- Clock: 0 Hz

How it works

Set ui_in[0] to 1 to enable oscillator.

How to test

Measure frequency at outputs.

External hardware

Frequency counter.

Pinout

#	Input	Output	Bidirectional
0	ring enable	divide-by-1	divide-by-256
1		divide-by-2	divide-by-512
2		divide-by-4	divide-by-1024
3		divide-by-8	divide-by-2048
4		divide-by-16	divide-by-4096
5		divide-by-32	divide-by-8192
6		divide-by-64	divide-by-16738
7		divide-by-128	divide-by-32768

Frequency counter [577]

- Author: Matt Venn
- Description: measured frequency of a signal on pin 0 and displays on the 7 segment display
- GitHub repository
- HDL project
- Mux address: 577
- Extra docs
- Clock: 10000000 Hz

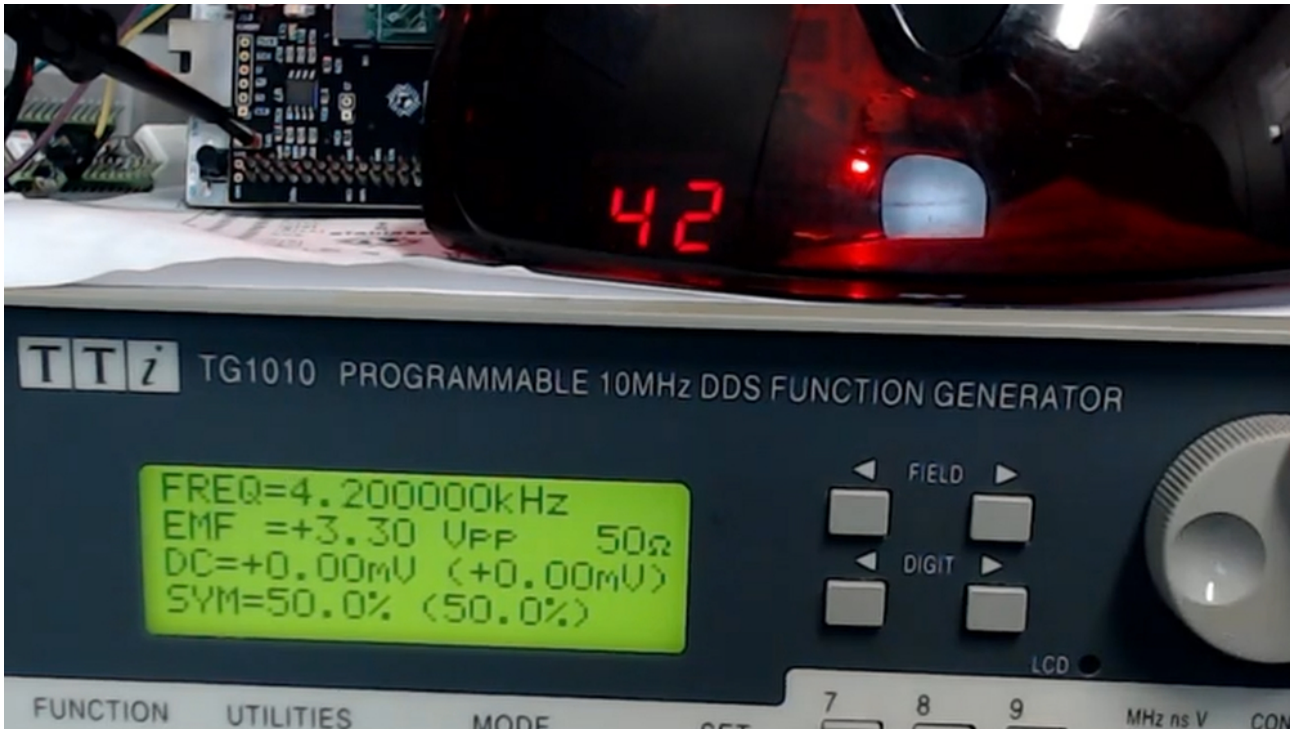


Figure 40: demo

How it works

Debounces the input signal and counts how many transistions occur in a given period. A state machine then divides the count by ten by repeatedly subtracting ten and then displays the tens and units on the seven segment display.

How to test

Apply a signal to the signal input and the frequency will be measured and displayed on the seven segment. The dot is used to select between display tens and units.

To change the count period (to get accurate counts), set it to match the clock frequency: clock_mhz * 100 - 1. So for a 10MHz clock, set to 999. Set the desired period (top 4 bits ui_in and all of uio_in) on the bidirectional inputs and toggle load input.

To debug, enable debug mode and check the bidirectional outputs for state machine, clock count and edge count information.

External hardware

A dual seven segment PMOD and a frequency source.

Pinout

#	Input	Output	Bidirectional
0	signal to measure	segment a	count period bit 00 or debug state bit 0
1	debug mode	segment b	count period bit 01 or debug state bit 1
2	load new period	segment c	count period bit 02 or debug clock bit 0
3	none	segment d	count period bit 03 or debug clock bit 1
4	count period bit 11	segment e	count period bit 04 or debug clock bit 2
5	count period bit 10	segment f	count period bit 05 or debug edge bit 0
6	count period bit 9	segment g	count period bit 06 or debug edge bit 1
7	count period bit 8	digit select	count period bit 07 or debug edge bit 2

SPI Test [579]

- Author: Harald Pretl
- Description: Simple SPI-based register file (for the testing the flow)
- GitHub repository
- HDL project
- Mux address: 579
- Extra docs
- Clock: 20000000 Hz

How it works

A simple serial 16b register (mini SPI) with magic cookie detection is implemented. In addition, a 16bit sigma-delta modulator of 1st or 2nd order is included.

Further, a sine generator (based on a LUT) with programmable frequency can be selected to drive the input of the DAC.

How to test

- Load the shift register in a serial way.
- Check the magic cookie detection.
- Check the digital output (low- and high-byte) of the loaded data word.
- Check the dc output voltage of the delta-sigma modulator by using an external RC lowpass filter.
- Check the sine output of the delta-sigma modulator by using an external RC lowpass filter.

External hardware

Just a way to set digital inputs is needed. A scope for monitoring output signals would be good. A voltmeter can be used to inspect the DAC output voltage. If the sine generator is used for the DAC input, a scope can be used to monitor the sine signal.

Pinout

#	Input	Output	Bidirectional
0	SPI clk (SCLK)	SPI data out (MISO)	register b0
1	SPI data in (MOSI)	cookie detected (loaded 0xCAFE)	register b1
2	SPI load (CS)	XOR of SPI clk and SPI data in	register b2
3	select output byte (0 = low, 1 = high)		register b3
4	sinegen scale factor (LSB)		register b4
5	sinegen scale factor (MSB)		register b5
6	select ds-modulator input (0 = SPI register, 1 = sine generator)	inverted output of delta-sigma modulator	register b6
7	order of delta-sigma modulator (0 = 1st, 1 = 2nd)	output of delta-sigma modulator	register b7

One Sprite Pony [581]

- Author: Leo Moser
- Description: This SVGA design has exactly one trick up its sleeve: it displays a sprite!
- GitHub repository
- HDL project
- Mux address: 581
- Extra docs
- Clock: 40000000 Hz

How it works

A one-trick pony is someone or something that is good at doing only one thing. Accordingly, a one-sprite pony can display only one sprite, and that's exactly what this design does:

This Verilog design produces SVGA 800x600 60Hz output with a background and one sprite. Internally, the resolution is reduced to 100x75, thus one pixel of the sprite is actually 8x8 pixels. The design operates at a 40 MHz pixel clock.

The sprite is 12x12 pixel in size and is initialized at startup with a pixelated version of the Tiny Tapeout logo.

An SPI receiver accepts various commands, e.g. to replace the sprite data, change the colors or set the background.

How to test

Connect a Tiny VGA to the output Pmod connector. By default, you should see the TinyTapeout logo moving around the screen. By sending commands over SPI via the bidirectional Pmod you can change the sprite and the background, set the sprite position and much more. See the longer documentation for all commands.

External hardware

Tiny VGA Pmod

Pinout

#	Input	Output	Bidirectional
0	SPI mode	R1	CS
1		G1	MOSI
2		B1	MISO
3		VS	SCK
4		R0	Vertical Pulse
5		G0	Horizontal Pulse
6		B0	
7		HS	

I2C EEPROM Project Selection [583]

- Author: Uri Shaked
- Description: Prototype for reading the selected Tiny Tapeout design address from an I2C EEPROM
- GitHub repository
- HDL project
- Mux address: 583
- Extra docs
- Clock: 20000000 Hz

How it works

This is a prototype for automatic project selection in Tiny Tapeout. It reads the design id from an I2C EEPROM and selects the corresponding project. The design id is stored in the first two bytes of the EEPROM memory as a big endian 16-bit number. The design id is then used to select the project by setting the `ctrl_sel_inc` signal to the lowest 10 bits of the design id.

The project also prints the 16-bit number read from the EEPROM to the UART port on `uo_out[4]` (baud rate 115200, 8N1) for debugging purposes.

How to test

You need to connect a 1 kbit or 2 kbit I2C EEPROM (e.g. 24C01 or 24C02), program the selected Tiny Tapeout design address into the first two bytes of the EEPROM memory, and connect the EEPROM to the SCL/SDA pins with a 10k pull-up resistor. After connecting the EEPROM, reset the design, and and then observe the ctrl output pins to see if the design id is correctly read from the EEPROM:

- `ctrl_sel_rst_n` should go low, and then high
- `ctrl_sel_inc` should pulse high according to the lowest 10 bits of the number stored in the EEPROM
- `ctrl_ena` should go high

For more information about those signals, please refer to the Tiny Tapeout Multiplexer documentation.

External hardware

I2C EEPROM (24C01 or 24C02) and two 10k pull-up resistors on SCL and SDA pins.

Pinout

#	Input	Output	Bidirectional
0		ctrl_sel_rst_n	SCL
1		ctrl_sel_inc	SDA
2		ctrl_ena	
3			
4		uart_tx	
5			
6			
7			

Color Bars [585]

- Author: Rebecca G. Bettencourt
- Description: VGA demo resembling NTSC color bars
- GitHub repository
- HDL project
- Mux address: 585
- Extra docs
- Clock: 0 Hz

How it works

Displays a test pattern on the screen resembling NTSC color bars. Optionally, you can add a station ID, make the ID scroll, and make the color bars scroll.

The colors displayed are NOT accurate to actual NTSC color bars. This cannot be used to adjust NTSC video equipment; it's just for fun.

Figure 41: Color bars with station ID

How to test

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

- `show_id` (`ui_in[0]`) to add a station ID,
- `custom_id` (`ui_in[1]`) to use a custom ID (address on `uio_out`, data on `ui_in[7:4]`),
- `scroll_id` (`ui_in[2]`) to make the ID scroll,
- `scroll_bars` (`ui_in[3]`) to make the color bars scroll.

External hardware

TinyVGA PMOD

Pinout

#	Input	Output	Bidirectional
0	show_id	R1	A0 (custom id)
1	custom_id	G1	A1 (custom id)
2	scroll_id	B1	A2 (custom id)
3	scroll_bars	VSync	A3 (custom id)
4	D3 (custom id)	R0	A4 (custom id)
5	D2 (custom id)	G0	A5 (custom id)
6	D1 (custom id)	B0	A6 (custom id)
7	D0 (custom id)	HSync	A7 (custom id)

SPELL [586]

- Author: Uri Shaked
- Description: SPELL is a minimal, cryptic, stack-based programming language crafted for The Skull CTF
- GitHub repository
- HDL project
- Mux address: 586
- 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

Pinout

#	Input	Output	Bidirectional
0	run	sleep/porta[0]	portb[0]
1	step	stop/porta1	portb1
2	load	wait_delay/porta2	portb2
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_sel1	porta[6]	portb[6]
7		porta[7]	portb[7]

Crispy VGA [587]

- Author: James Meech
- Description: The scrolling VGA example from the vga playground but as you set more inputs high it gets successively more crispy
- GitHub repository
- HDL project
- Mux address: 587
- Extra docs
- Clock: 0 Hz

How it works

This project "Crispy VGA" takes as input the output of a standard tiny tapeout VGA project. Crispy VGA then adds a programmable amount of random noise to the VGA signal and passes it through to the output. The uio_in[0] input sets the noise on the hsync signal. The uio_in1 input sets the noise on the B signal. The uio_in2 input sets the noise on the G signal. The uio_in[3] input sets the noise on the R signal. The uio_in[4] input sets the noise on the vsync. The uio_in[5] signal sets the noise level applied to the R, G, and B wires to high or low. The uio_in[0:5] inputs set the succesively increasing noise levels on the audio signal.

How to test

Plug an existing tiny tapeout VGA project into the input of this design. Plug the output of this design into a standard VGA input monitor. Power up both tiny tapeout boards and select the appropriate control bits for the level of noise that you want to see on the output VGA signal.

External hardware

You will need a VGA input monitor and a computer that can output a VGA signal or a second tiny tapeout ASIC with a working VGA design that follows the standard pinout. You will also need two tiny tapeout VGA adapters and two VGA cables.

Pinout

#	Input	Output	Bidirectional
0	R1 vga input	R1 vga input	Crispy input bit 0 that toggles the noise on the hsync signal on or off. Also adds one bit of noise to audio.
1	G1 vga input	G1 vga input	Crispy input bit 1 toggles the noise on the B signal on or off. Also adds one bit of noise to audio.
2	B1 vga input	B1 vga input	Crispy input bit 2 toggles the noise on the G signal on or off. Also adds one bit of noise to audio.
3	vsync vga input	vsync vga input	Crispy input bit 3 toggles the noise on the R signal on or off. Also adds one bit of noise to audio.
4	R[0] vga input	R[0] vga input	Crispy input bit 4 that toggles the noise on the vsync signal on or off. Also adds one bit of noise to audio.
5	G[0] vga input	G[0] vga input	Crispy input bit 5 that sets the noise level applied to the R, G, and B wires to high or low. Also adds one bit of noise to audio.
6	B[0] vga input	B[0] vga input	Audio input bit
7	hsync vga input	hsync vga input	Audio output bit

Snow [608]

- Author: sylefeb
- Description: Demo on TinyTapeout? Let's do something!
- GitHub repository
- HDL project
- Mux address: 608
- Extra docs
- Clock: 25000000 Hz

This demo is written in Silice, my HDL. Here is the actual source. Silice now fully support TinyTapeout as a build target.

How it works

But does it work?

How to test

Plug the VGA+audio PMODs to the board and run. Maybe it works?

External hardware

See https://tinytapeout.com/competitions/demoscene/

Pinout

#	Input	Output	Bidirectional
0		R1	
1		G1	
2		B1	
3		VS	
4		R0	
5		G0	
6		B0	
7		HS	Audio

TTL Pulse Generator [609]

- Author: Adonai Cruz
- Description: Simple TTL Pulse Generator
- GitHub repository
- HDL project
- Mux address: 609
- Extra docs
- Clock: 0 Hz

How it works

This ASIC is a simple TTL pulse generator with 16 pre-programmed pulse sequences.

How to test

To test the ASIC select the desired pulse sequence using 4-bits on pins ui[0:3]. The TTL pulse output is on pin uo[0].

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	Sequence Select bit 0	TTL output	
1	Sequence Select bit 1		
2	Sequence Select bit 2		
3	Sequence Select bit 3		
4			
5			
6			
7			

8-bit ALU based on 2x 74181 [610]

- Author: Caio Alonso da Costa
- Description: 8-bit ALU implemented with 2x 4-bit slice arithmetic logic unit 74181 with SPI interface
- GitHub repository
- HDL project
- Mux address: 610
- 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    |
```

- 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    |
```

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.

External hardware

Not required.

Pinout

#	Input	Output	Bidirectional
0	cpol	decod_reg[0]	
1	cpha	decod_reg1	
2		decod_reg2	
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]	

Iterative MAC [611]

- Author: Raju Machupalli
- Description: Iterative multiply and accumulation unit for ML accelerators
- GitHub repository
- HDL project
- Mux address: 611
- 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_in1	uo_out1	uio_in1
2	ui_in2	uo_out2	uio_in2
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]

VGA Tiny Logo (1 tile) [612]

- Author: Renaldas Zioma
- Description: Large 480x480 pixels Tiny Tapeout logo with bling and dithered colors crammed into 1 tile!
- GitHub repository
- HDL project
- Mux address: 612
- Extra docs
- Clock: 25175000 Hz

How it works

Compressed VGA Logo

How to test

Connect to VGA monitor

External hardware

TinyVGA PMOD, VGA monitor

Pinout

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

TTIHP TinyVGA FUN! [613]

- Author: algofoogle (Anton Maurovic) + htfab
- Description: The digital part of TT08 VGA FUN! with a simple sequencer loop
- GitHub repository
- HDL project
- Mux address: 613
- Extra docs
- Clock: 0 Hz

How it works

It's the digital block from algofoogle's TT08 VGA FUN! project, with a simple sequencer loop to make it work standalone and some dithering to simulate 8 bit output on the 2 bit TinyVGA Pmod.

How to test

Plug it into a VGA monitor, reset the project, then sit back and enjoy.

You can also manually select the mode and bit depth if you override the sequencer by pulling bit 7 of the input high.

External hardware

TinyVGA Pmod

Pinout

#	Input	Output	Bidirectional
0	(mode bit 0)	r1	
1	(mode bit 1)	g1	
2	(mode bit 2)	b1	
3	(depth bit 0)	vsync	
4	(depth bit 1)	r0	
5	(depth bit 2)	g0	
6	(depth bit 3)	b0	
7	override sequencer	hsync	

SkyKing Demo [614]

- Author: Nicklaus Thompson
- Description: Types some text over an image of a plane flying into the sunset
- GitHub repository
- HDL project
- Mux address: 614
- Extra docs
- Clock: 25200000 Hz

How it works

The demo conststs of a static image of a passenger jet flying off into the sunset with a text overlay at the bottom that fills in character-by-character. The text begins typing immediately after reset, so it is likely that the entire text animation will complete before the VGA monitor recognizes the signal. It is best to view this demo in the VGA playground due to the timing issue. The project also includes demos to test some oscilloscope XY display PMODs I'm working on. The demos for these PMODs are both circles and they can be accessed by setting ui_in[0] high and using ui_in1 to select the demo.

How to test

The demo runs automatically if all inputs are low. If ui_in[1:0] = 2'b01, an unrelated demo for a 1-PMOD XY display driver will play. If ui_in[1:0] = 2'b11, a demo for a 2-PMOD XY display driver will play.

External hardware

The demo requires the Tiny VGA PMOD on UO. The XY demos require either a 1-PMOD driver on UO, or a 2-PMOD driver on UO and UIO. The demo does not include audio.

Pinout

#	Input	Output	Bidirectional
0	0: VGA, 1: XY	ui[1:0] = 0 -> HS, 1 -> Trig, 3 -> Y0	ui1 = 0 -> 1'b0, 1 -> X0
1	0: XY 1, 1: XY 2	ui[1:0] = 0 -> R0, 1 -> Y5, 3 -> Y2	ui1 = 0 -> 1'b0, 1 -> X2
2		ui[1:0] = 0 -> G0, 1 -> X7, 3 -> Y4	ui1 = 0 -> 1'b0, 1 -> X4
3		ui[1:0] = 0 -> B0, 1 -> X5, 3 -> Y6	ui1 = 0 -> 1'b0, 1 -> X6
4		ui[1:0] = 0 -> VS, 1 -> Y6, 3 -> Y1	ui1 = 0 -> 1'b0, 1 -> X1
5		ui[1:0] = 0 -> R1, 1 -> Y4, 3 -> Y3	ui1 = 0 -> 1'b0, 1 -> X3
6		ui[1:0] = 0 -> G1, 1 -> X6, 3 -> Y5	ui1 = 0 -> 1'b0, 1 -> X5
7		ui[1:0] = 0 -> B1, 1 -> X4, 3 -> Trig	ui1 = 0 -> 1'b0, 1 -> X7

One Bit PUF [615]

- Author: Yimin Gao and Ceylan Morgul
- Description: It is a PUF based on a difference of two registers
- GitHub repository
- HDL project
- Mux address: 615
- Extra docs
- Clock: 0 Hz

How it works

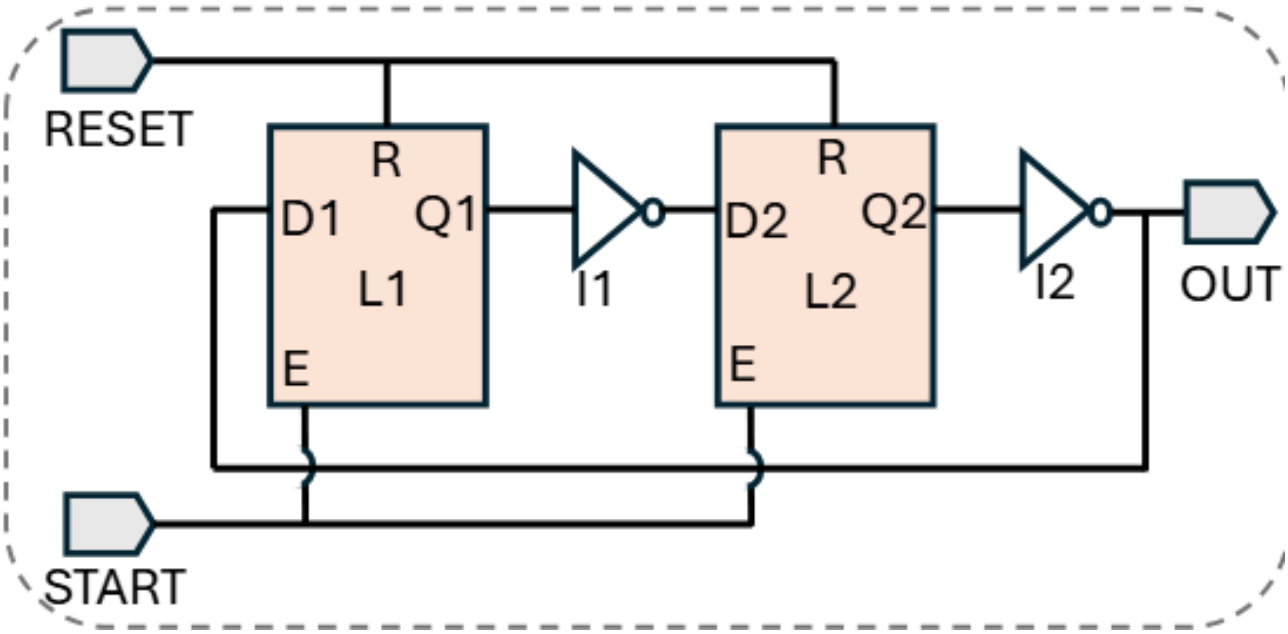


Figure 42: image

How to test

The output is 0 in the reset condition.

External hardware

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

Pinout

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

Cell mux [616]

- Author: htfab
- Description: All the IHP standard cells
- GitHub repository
- HDL project
- Mux address: 616
- Extra docs
- Clock: 0 Hz

How it works

To be filled later

How to test

To be filled later

External hardware

None

Pinout

#	Input	Output	Bidirectional
0	page bit 0	output bit 8*page	input bit 4
1	page bit 1	output bit 8*page+1	input bit 5
2	page bit 2	output bit 8*page+2	
3	page bit 3	output bit 8*page+3	
4	input bit 0	output bit 8*page+4	
5	input bit 1	output bit 8*page+5	
6	input bit 2	output bit 8*page+6	
7	input bit 3	output bit 8*page+7	

One Bit PUF [617]

- Author: Yimin Gao and Ceylan Morgul
- Description: It is a PUF based on a difference of two registers
- GitHub repository
- HDL project
- Mux address: 617
- Extra docs
- Clock: 0 Hz

How it works

This is a PUF design that includese 2**ADDR_BITS x OUT_BITS one_bit_pufs The addr is the address to read OUT_bits of the PUF bits For instance if ADDR_BITS = 2, OUT_BITS = 2 The design will include 8 one_bit_pufs, addr = 2'b10 will read 2 puf bits (OUT[5:4])

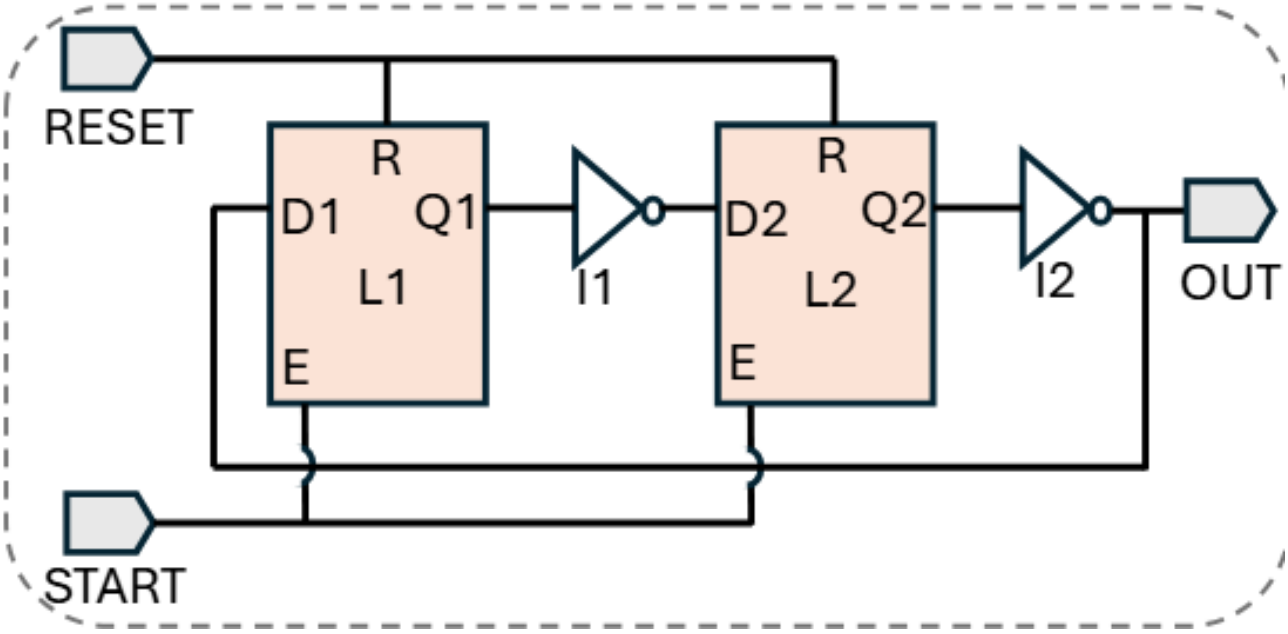


Figure 43: image

How to test

The output is 0 in the reset condition.

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	addr[0]	puf_out[0]	
1	addr1	puf_out1	
2	addr2	puf_out2	
3	addr[3]	puf_out[3]	
4		puf_out[4]	
5		puf_out[5]	
6		puf_out[6]	
7	start_signal	puf_out[7]	

Power gating test (1x1) [618]

- Author: htfab
- Description: Placeholder for a power gated test design (preliminary work on supporting power gated designs on later IHP shuttles)
- GitHub repository
- HDL project
- Mux address: 618
- Extra docs
- Clock: 0 Hz

How it works

To be filled later

How to test

To be filled later

External hardware

None

Pinout

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

INTERCAL ALU [619]

- Author: Rebecca G. Bettencourt
- Description: An ALU for the five operators of the INTERCAL programming language.
- GitHub repository
- HDL project
- Mux address: 619
- Extra docs
- Clock: 0 Hz

How it works

As an educational project, it is inevitable that Tiny Tapeout would attract various pedagogical examples of common logic circuits, such as ALUs. While ALUs for common operations such as addition, subtraction, and binary bitwise logic are surprisingly common, it is much rarer to encounter one that can calculate the five operations of the INTERCAL programming language. Due to either the cost-prohibitive nature of Warmenhovian logic gates or general lack of interest, such a feat has never been performed until now. With chip production finally within reach of the average person, all it takes is one person who has more dollars than sense to design the fabled INTERCAL ALU (Arrhythmic Logic Unit).

The pin assignments for this design are roughly as follows. The /OE (output enable) and /WE (write enable) signals are active low, so should be set HIGH by default.

#	Dedicated Input	Dedicated Output	Bidirectional I/O
0	A0 (address)	D0 (output only)	D0 (input and output only)
1	A1 (address)	D1 (output only)	D1 (input and output only)
2	S0 (selector)	D2 (output only)	D2 (input and output only)
3	S1 (selector)	D3 (output only)	D3 (input and output only)
4	S2 (selector)	D4 (output only)	D4 (input and output only)
5	S3 (selector)	D5 (output only)	D5 (input and output only)
6	/OE (output enable)	D6 (output only)	D6 (input and output only)
7	/WE (write enable)	D7 (output only)	D7 (input and output only)

This ALU has two 32-bit registers, B and A (in no particular order). (These may also be thought of as four 16-bit registers, AL, AH, BL, and BH.) To write a byte to a register, set A0 and A1 to the byte address, set S0 LOW for the A register or HIGH for the B register, set S1 through S3 LOW, set the bidirectional I/O pins to the byte

value, set /WE LOW, then set /WE HIGH again. (Do not set S1 through S3 HIGH when writing, or else something unpredictable will happen, most likely nothing.)

To read a register or result, set A0 and A1 to the byte address, set S0 through S3 to the desired operation, set /OE LOW, read the byte value from the bidirectional I/O pins, then set /OE HIGH. Results can also be read from the dedicated outputs; the dedicated outputs are not affected by the /OE signal, as they do not need to care about your feelings.

The operations supported are listed below. An attempt was made to make it understandable.

						Address				
						A	3	2	1	0
Selector						**A1**	1	1	0	0
S	**S3**	**S2**	**S1**	**S0**	**Operation**	**A0**	1	0	1	0
0	0	0	0	0	A		AH		AL	
1	0	0	0	1	B		BH		BL	
2	0	0	1	0	AND16		& AH		& AL	
3	0	0	1	1	AND32		& A			
4	0	1	0	0	OR16		V AH		V AL	
5	0	1	0	1	OR32		V A			
6	0	1	1	0	XOR16		? AH		? AL	
7	0	1	1	1	XOR32		? A			
8	1	0	0	0	MINGLE16L		AL $ BL			
9	1	0	0	1	MINGLE16H		AH $ BH			
10	1	0	1	0	SELECT16		AH~BH		AL~BL	
11	1	0	1	1	SELECT32		A ~ B			

Operations 0 and 1 simply return the current value of the A or B register, respectively. This corresponds with the values of S0 through S3 used in write mode. This is not unintentional. This might also explain why S1 through S3 must be LOW in write mode.

Operations 2 through 7 correspond to INTERCAL's unary AND, unary OR, and unary XOR operators, represented by ampersand (&), book (V), and what (?), respectively. From the INTERCAL manual:

These operators perform their respective logical operations on all pairs of adjacent bits, the result from the first and last bits going into the first bit of the result. The effect is that of rotating the operand one place to the right and ANDing, ORing, or XORing with its initial value. Thus, #&77 (binary = 1001101) is binary 0000000000000100 = 4, #V77 is binary 1000000001101111 = 32879, and #?77 is binary 1000000001101011 = 32875.

Operations 2, 4, and 6 work on the 16-bit halves of the A register independently, while operations 3, 5, and 7 work on the 32-bit whole of the A register.

Operations 8 and 9 correspond to INTERCAL's *interleave* (also called *mingle*) operator, represented by big money ($). From the INTERCAL manual:

The interleave operator takes two 16-bit values and produces a 32-bit result by alternating the bits of the operands. Thus, #65535$#0 has the 32-bit binary form 101010....10 or 2863311530 decimal, while #0$#65535 = 0101....01 binary = 1431655765 decimal, and #255$#255 is equivalent to #65535.

Operation 8 returns the interleave of the lower halves of A and B, while operation 9 returns the interleave of the upper halves of A and B. (Should the chip fabrication process allow for it, operation 8½ will, of course, return the interleave of the middle halves of A and B.)

Operations 10 and 11 correspond to INTERCAL's *select* operator, represented by sqiggle (~). From the INTERCAL manual:

The select operator takes from the first operand whichever bits correspond to 1's in the second operand, and packs these bits to the right in the result. Both operands are automatically padded on the left with zeros. [...] For example, #179~#201 (binary value 10110011~11001001) selects from the first argument the 8th, 7th, 4th, and 1st from last bits, namely, 1001, which = 9. But #201~#179 selects from binary 11001001 the 8th, 6th, 5th, 2nd, and 1st from last bits, giving 10001 = 17. #179~#179 has the value 31, while #201~#201 has the value 15.

To help understand the select operator, the INTERCAL manual also provides a helpful circuitous diagram.

Use of operations 12 and above is not recommended, unless undefined behavior is required.

How to test

The following example calculations found in the INTERCAL manual should be particularly illuminating.

S	A	B	F
MINGLE16L (8)	0	256	65536
MINGLE16L (8)	65535	0	2863311530
MINGLE16L (8)	0	65535	1431655765
MINGLE16L (8)	255	255	65535
SELECT16 (10)	51	21	5 *
SELECT16 (10)	179	201	9
SELECT16 (10)	201	179	17
SELECT16 (10)	179	179	31
SELECT16 (10)	201	201	15
AND16 (2)	77		4
OR16 (4)	77		32879
XOR16 (6)	77		32875

These test cases are included in the (unfortunately Python and not INTERCAL) `test.py` file. As these are likely more INTERCAL operations than any sensible person will ever perform, they should be sufficient for testing purposes. However, for curiosity's sake, an extensive set of additional test cases have also been included.

- Not found in the INTERCAL manual.

External hardware

The ALU may be used without external hardware, although seeing the output values may present a challenge. Instead, it is recommended to use a microcontroller of some sort to drive the inputs and read the outputs, as microcontrollers are designed to do. The implementation of the rest of the INTERCAL language is left as an exercise for the reader.

Further reading

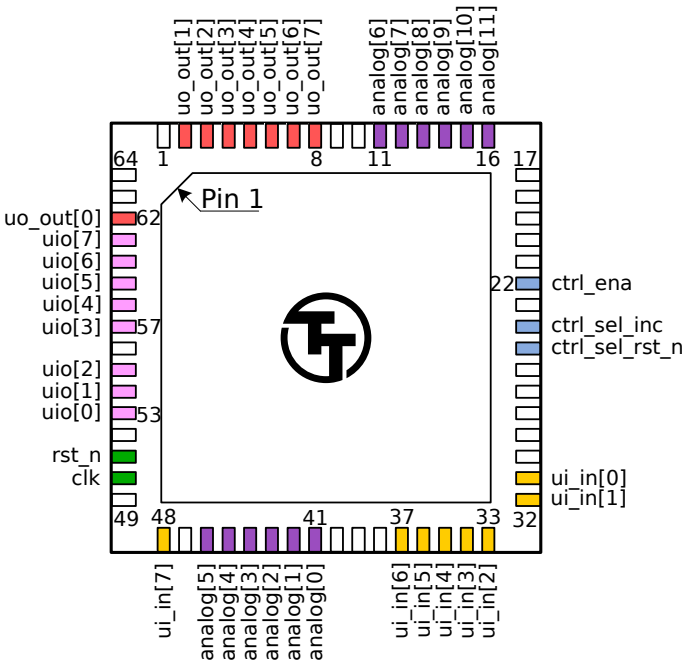
The INTERCAL Programming Language Revised Reference Manual by Donald R. Woods and James M. Lyon with revisions by Louis Howell and Eric S. Raymond (can recommend highly enough)

Pinout

#	Input	Output	Bidirectional
0	A0 (address)	D0	D0
1	A1 (address)	D1	D1
2	S0 (selector)	D2	D2
3	S1 (selector)	D3	D3
4	S2 (selector)	D4	D4
5	S3 (selector)	D5	D5
6	/OE (output enable)	D6	D6
7	/WE (write enable)	D7	D7

Pinout

The chip is packaged in a 64-pin QFN package. The pinout is shown below.



Bottom View

Figure 44: 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:

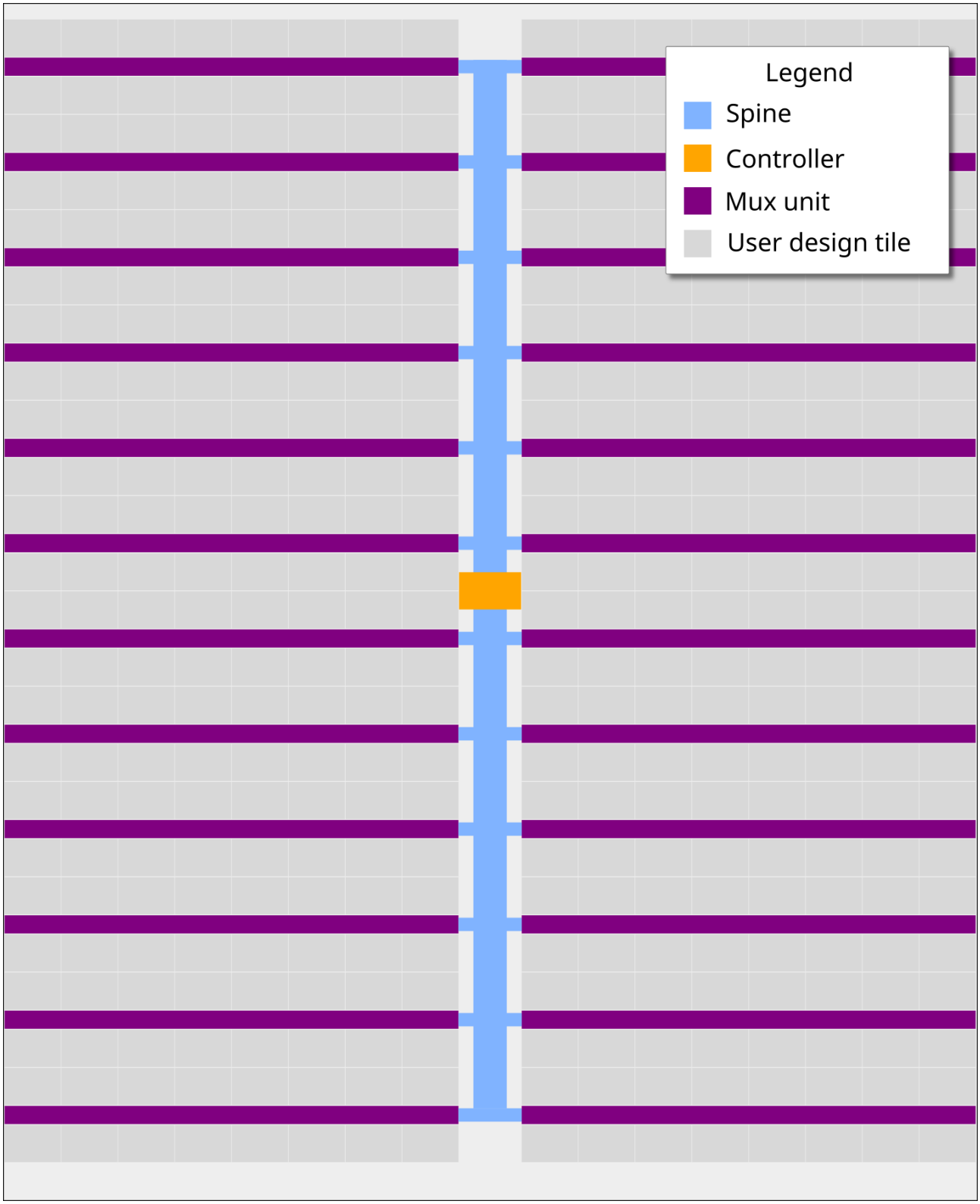


Figure 45: Mux Diagram

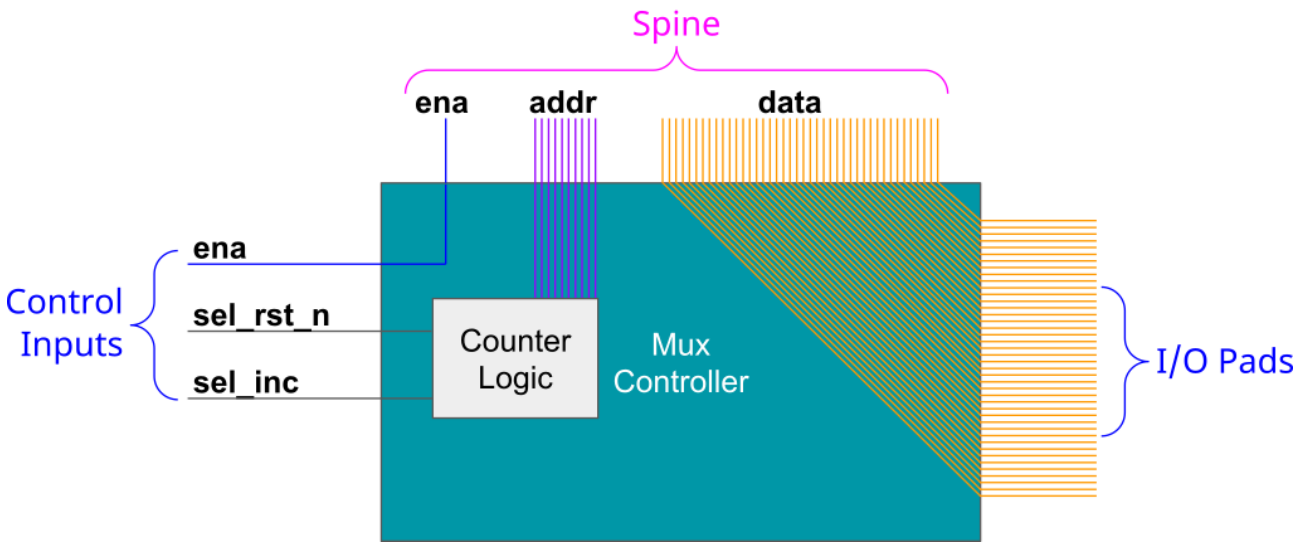


Figure 46: Mux Controller Diagram

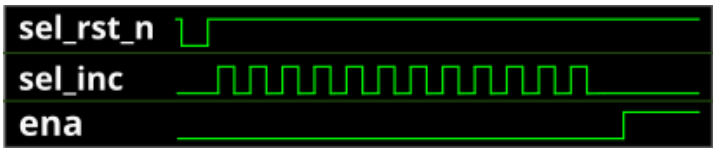


Figure 47: Mux signals for activating the design at address 12

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/3643478076
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

QFN64 pin	Function	Signal
1	Mux Control	ctrl_ena
2	Mux Control	ctrl_sel_inc
3	Mux Control	ctrl_sel_rst_n
4	Reserved	(none)
5	Reserved	(none)

QFN64 pin	Function	Signal
6	Reserved	(none)
7	Reserved	(none)
8	Reserved	(none)
9	Output	uo_out[0]
10	Output	uo_out1
11	Output	uo_out2
12	Output	uo_out[3]
13	Output	uo_out[4]
14	Output	uo_out[5]
15	Output	uo_out[6]
16	Output	uo_out[7]
17	Power	VDD IO
18	Ground	GND IO
19	Analog	analog[0]
20	Analog	analog1
21	Analog	analog2
22	Analog	analog[3]
23	Power	VAA Analog
24	Ground	GND Analog
25	Analog	analog[4]
26	Analog	analog[5]
27	Analog	analog[6]
28	Analog	analog[7]
29	Ground	GND Core
30	Power	VDD Core
31	Ground	GND IO
32	Power	VDD IO
33	Bidirectional	uio[0]
34	Bidirectional	uio1
35	Bidirectional	uio2
36	Bidirectional	uio[3]
37	Bidirectional	uio[4]
38	Bidirectional	uio[5]
39	Bidirectional	uio[6]
40	Bidirectional	uio[7]
41	Input	ui_in[0]
42	Input	ui_in1
43	Input	ui_in2
44	Input	ui_in[3]
45	Input	ui_in[4]

QFN64 pin	Function	Signal
46	Input	ui_in[5]
47	Input	ui_in[6]
48	Input	ui_in[7]
49	Input	rst_n †
50	Input	clk †
51	Ground	GND IO
52	Power	VDD IO
53	Analog	analog[8]
54	Analog	analog[9]
55	Analog	analog[10]
56	Analog	analog[11]
57	Ground	GND Analog
58	Power	VDD Analog
59	Analog	analog[12]
60	Analog	analog[13]
61	Analog	analog[14]
62	Analog	analog[15]
63	Ground	GND Core
64	Power	VDD Core

† 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

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