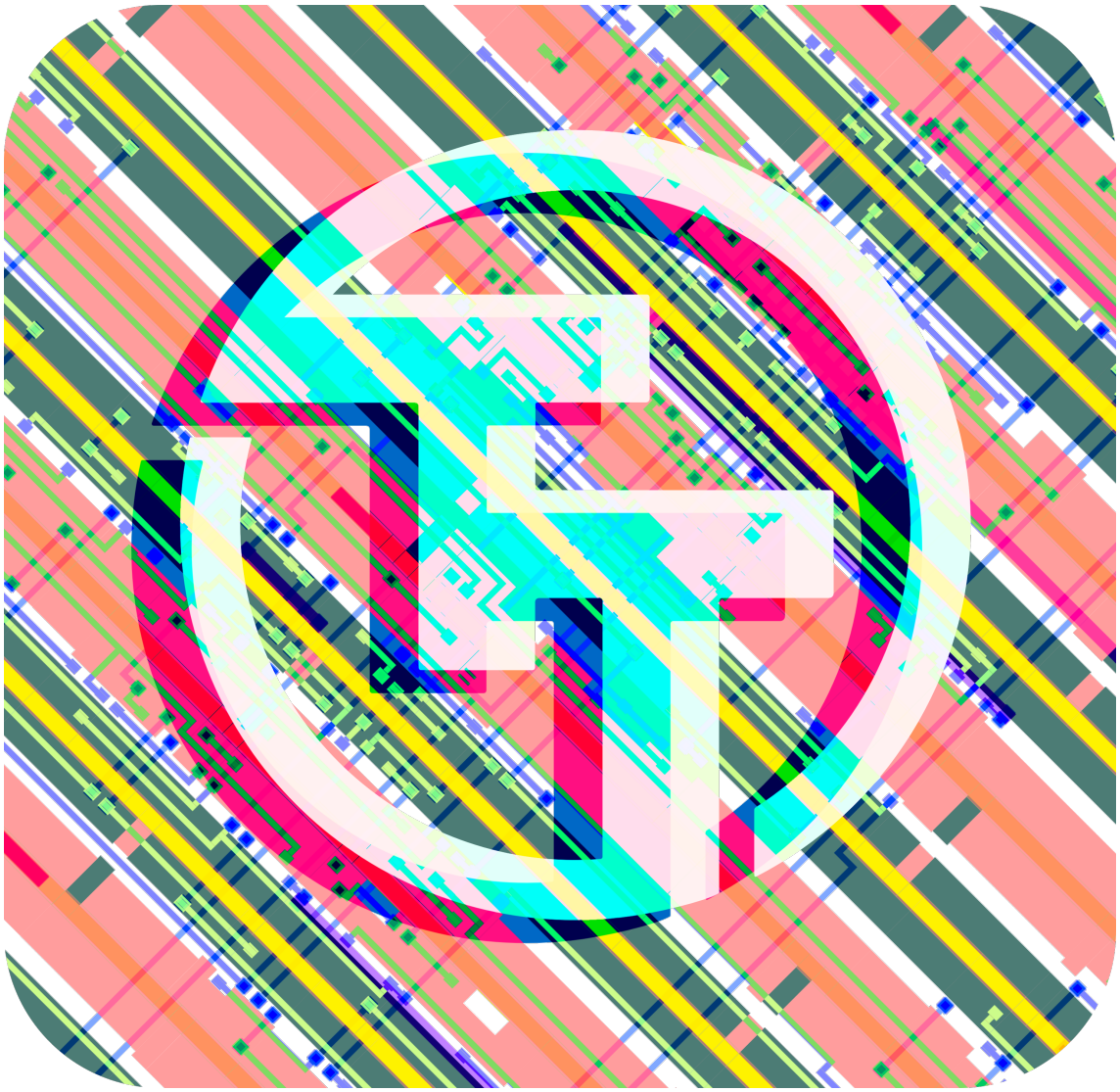


Tiny Tapeout 04 Datasheet

Project Repository

https://github.com/TinyTapeout/tinytapeout-04

December 14, 2023

Contents

Render of whole chip

Full GDS

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
- External hardware:

How it works

ROM memory that contains information about the Tiny Tapeout chip. The ROM is 8-bit wide and 128 bytes long.

The ROM layout The ROM layout is as follows:

Address	Length	Encoding	Description
0	8	7-segment	Shuttle name (e.g. "tt04"), null-padded
32	96	ASCII	Chip descriptor (see below)

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	tt04
repo	The name of the repository	TinyTapeout/tinytapeout-04

Future Tiny Tapeout shuttles may add more keys to the chip descriptor.

Here is a complete example of a chip descriptor:

```
shuttle=tt04
repo=TinyTapeout/tinytapeout-04
```

How to test

Read the ROM contents by setting the address pins and reading the data pins. The first eight bytes of the ROM are 7-segment encoded and contain the shuttle name. 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]	none
1	addr[1]	data[1]	none
2	addr[2]	data[2]	none
3	addr[3]	data[3]	none
4	addr[4]	data[4]	none
5	addr[5]	data[5]	none
6	addr[6]	data[6]	none
7	addr[7]	data[7]	none

TinyTapeout 04 Factory Test [1]

- Author: Sylvain Munaut
- Description: Factory test module
- GitHub repository
- HDL project
- Mux address: 1
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

If input 0 is high, then a counter is output on the outputs and the bidirectional outputs. Otherwise the inputs are mirrored to the outputs.

How to test

Set input 0 high. Check the outputs are toggling.

Pinout

#	Input	Output	Bidirectional
0	sel / data_i[0]	data_o[0] (when sel=0) / counter_o[0] (when sel=1)	counter_o[0]
1	data_i[1]	data_o[1] (when sel=0) / counter_o[1] (when sel=1)	counter_o[1]
2	data_i[2]	data_o[2] (when sel=0) / counter_o[2] (when sel=1)	counter_o[2]
3	data_i[3]	data_o[3] (when sel=0) / counter_o[3] (when sel=1)	counter_o[3]

#	Input	Output	Bidirectional
4	data_i[4]	data_o[4] (when sel=0) / counter_o[4] (when sel=1)	counter_o[4]
5	data_i[5]	data_o[5] (when sel=0) / counter_o[5] (when sel=1)	counter_o[5]
6	data_i[6]	data_o[6] (when sel=0) / counter_o[6] (when sel=1)	counter_o[6]
7	data_i[7]	data_o[7] (when sel=0) / counter_o[7] (when sel=1)	counter_o[7]

VGA clock [2]

- Author: Matt Venn
- Description: Shows the time on a VGA screen
- GitHub repository
- HDL project
- Mux address: 2
- Extra docs
- Clock: 31500000 Hz
- External hardware: R2R dac for the VGA signals

How it works

Races the beam! Font is pre generated and loaded into registers. 6 bit colour keeps register count low.

How to test

Hook up a VGA monitor to the outputs and provide a clock at 31.5 MHz. Connect the 6 bit colour output up with resistors to make a R2R DAC. See the circuit here: https://github.com/mattvenn/6bit-pmod-vga

Pinout

#	Input	Output	Bidirectional
0	clock	hsync	none
1	reset	vsync	none
2	adjust hours	r0	none
3	adjust minutes	r1	none
4	adjust seconds	g0	none
5	none	g1	none
6	none	b0	none
7	none	b1	none

7 segment seconds [3]

- Author: Matt Venn
- Description: Count up to 10, one second at a time.
- GitHub repository
- HDL project
- Mux address: 3
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

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

Puts the bottom 8 bits of the counter on the bidirectional outputs.

With all the inputs set to 0, the internal 24 bit compare is set to 10,000,000. This means the counter will increment by one each second.

If any inputs are non zero, then the input will be used as an bits 11 to 18 of the 24 bit compare register. Example: setting the inputs to 00010000 will program 16384 into the compare register. With a 10MHz clock the counter will increment ~610 times per second.

How to test

After reset, the counter should increase by one every second with a 10MHz input clock. Experiment by changing the inputs to change the counting speed.

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

#	Input	Output	Bidirectional
6	compare bit 17	segment g	second counter bit 6
7	compare bit 18	dot	second counter bit 7

Number Factorizer [4]

- Author: Marno van der Maas
- Description: Takes the input and computes its factors
- GitHub repository
- HDL project
- Mux address: 4
- Extra docs
- Clock: 10000000 Hz
- External hardware: seven-segment display

How it works

This design uses a set of registers to compute the modulo of all factors up to 19 in one or two steps. The modulus of non-trivial numbers is calculated using the following trick:

```
n % k = (
          (2^0 % k) * n[0] +
          (2^1 % k) * n[1] +
          ...              +
          (2^6 % k) * n[6] +
          (2^7 % k) * n[7]
        ) % k
      = big_sum % k
```

The values of `2^x % k` can be computed ahead of time and are hardcoded in the design. Also we don't actually care about the modulus but rather about when the modulus is equal to zero, because that means that `k` is a factor. Since the final result of `big_sum` is guaranteed to be less than or equal to `(k - 1) * 8`, we can exhaustively list all the values for which the modulus is zero by:

```
(n % k == 0) = (
                  big_sum == (k * 0) ||
                  big_sum == (k * 1) ||
                  big_sum == (k * 2) ||
                  ...                ||
                  big_sum == (k * m)
                )
```

Where `m` is the largest integer for which `k * m <= (k - 1) * 8`.

Factors between 0x1 (decimal 1) and 0xF (decimal 15) are shown in a loop on the seven segment display with a one second delay between each factor. This design uses some combinatorial logic to convert from binary to hexadecimal for the seven segment display. The second delay is achieved by a clock divider logic.

The output pins show the prime factors of the input number. If the input is zero, the output is set to the bottom 8 bits of the counter for debugging purposes.

How to test

After reset and input set to 0, the counter on the seven segment display should increase by one every second with a 10 MHz input clock from `0x1` to `0xF`. The outputs are the lower bits of the internal counter that increases every cycle. The dot on the seven segment display should be off.

For inputs other than 0, the seven segment display shows the factors one by one, cycling back at the end. The factor 1 is displayed for all inputs and only factors up to 15 are shown. For example for 6, the factors 1, 2, 3 and 6 will be shown on the display. For 7, the factors 1 and 7 will be shown on the display. For 23, only the factor 1 will be show on the display. It will also use the output pins to indicate the prime factors, where the least significant bit represents 2 and the most significant bit represents 19. For example for 6, only 2 and 3 are set to 1. For 7, only 7 is set to 1. For 23, all the outputs are zero. The dot on the seven segment display is only on when the input number is prime.

Hexadecimals are displayed using the decimal configurations (without modifications). And then the hexadecimal values specified here.

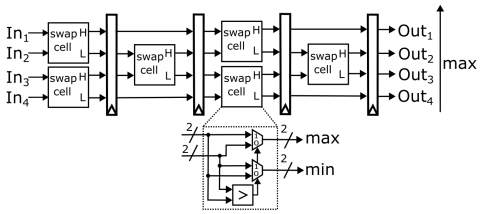
Please reset the design before giving your input. Also, you can have a look at the testbench for more thorough testing.

Pinout

#	Input	Output	Bidirectional
0	Fist bit of number to factor	Segment a (hex)	Is 2 a factor?
1	Second bit of number to factor	Segment b (hex)	Is 3 a factor?
2	Third bit of number to factor	Segment c (hex)	Is 5 a factor?
3	Fourth bit of number to factor	Segment d (hex)	Is 7 a factor?
4	Fifth bit of number to factor	Segment e (hex)	Is 11 a factor?
5	Sixth bit of number to factor	Segment f (hex)	Is 13 a factor?

#	Input	Output	Bidirectional
6	Seventh bit of number to factor	Segment g (hex)	Is 17 a factor?
7	Eigth bit of number to factor	Segment dot (is prime)	Is 19 a factor?

Odd even sorter [5]



- Author: Vasileios Titopoulos
- Description: An odd even sorter of four 2-bit values
- GitHub repository
- HDL project
- Mux address: 5
- Extra docs
- Clock: 25 000 000 Hz
- External hardware:

How it works

The sorter takes the inputs from ui_in[7:0] signals and rearranges them properly to uo_out[7:0] signals after they pass from the four internal pipeline registers.

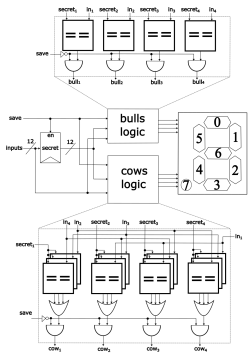
How to test

Connect a clock for the internal registers and use the inputs ui_in[7:0] to assign the values to the design. The sorted values are provided through uo_out[7:0] signals

Pinout

#	Input	Output	Bidirectional
0	I0/In1[0]	O0/Out1[0]/segment a	none
1	I1/In1[1]	O1/Out1[1]/segment b	none
2	I2/In2[0]	O2/Out2[0]/segment c	none
3	I3/In2[1]	O3/Out2[1]/segment d	none
4	I4/In3[0]	O4/Out3[0]/segment e	none
5	I5/In3[1]	O5/Out3[1]/segment f	none
6	I6/In4[0]	O6/Out4[0]/segment g	none
7	I7/In4[1]	O7/Out4[1]/dot	none

The Bulls and Cows game [6]



- Author: Giorgos Dimitrakopoulos
- Description: An implementation of the Bulls and Cows game
- GitHub repository
- HDL project
- Mux address: 6
- Extra docs
- Clock: 25 000 000 Hz
- External hardware:

How it works

The bulls and cows is a game where the users try to discover the exact pattern of secret numbers

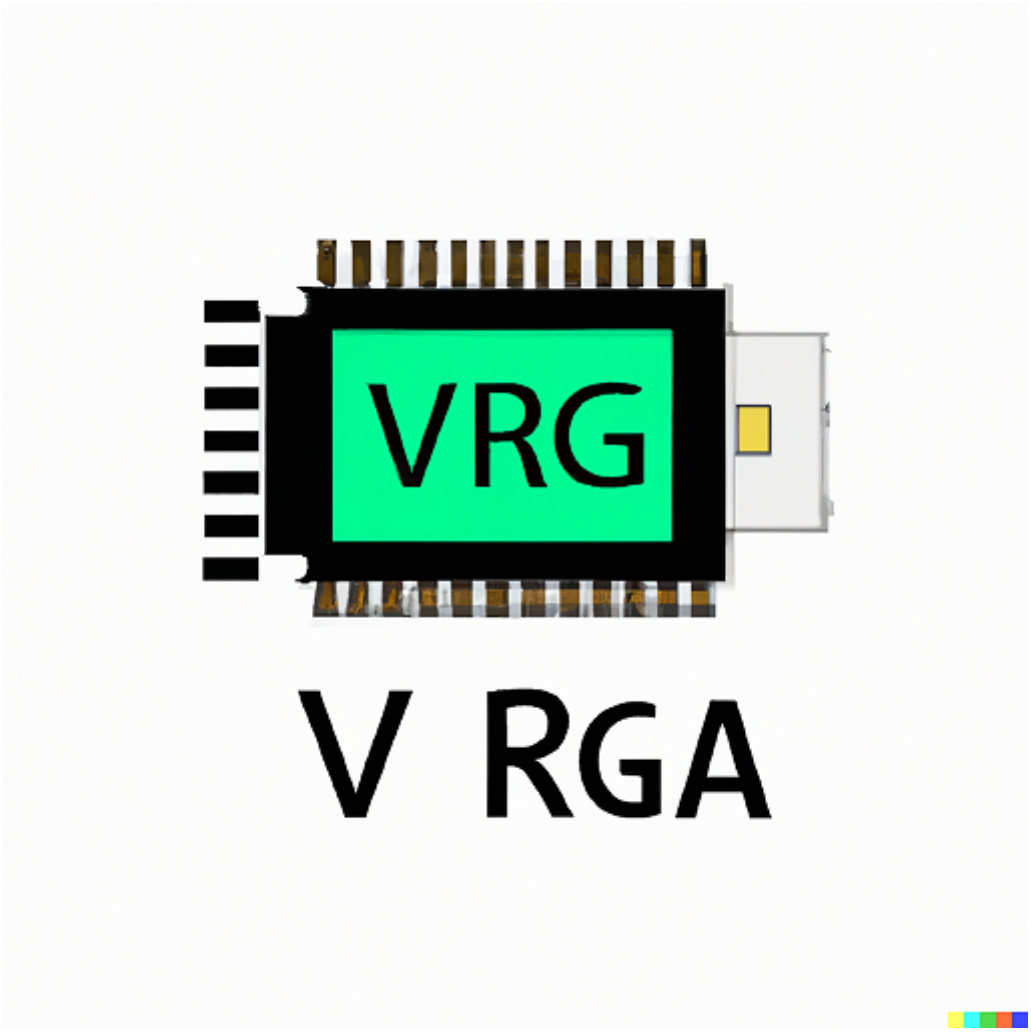
How to test

Firstly, a secret pattern of four 3-bit numbers, which are different with each other, should be set. This is done through the save signal. After the secret number is set, another user tries to find the correct pattern of numbers. In order for the correct pattern to be found, the signals of cows and bulls are pinpointing as to whether the user input had any match with the secret pattern. The cows indication shows as to whether the input matches with any secret number but it is not in the right position. On the other hand, the bulls indication shows as to whether the input matches and is placed correctly. The purpose of the game is for the user to achieve four bulls indications. For the signals of bulls and cows the indications are showed through the seven segment display. In the seven segment display the bulls indication is placed in the top region (0-1-5-6) and the cows indication is placed in the bottom region (2-3-4-7) of the seven segment display.

Pinout

#	Input	Output	Bidirectional
0	I0/Number1[0]	O0/bulls[1]/segment a	I7/Number3[0]
1	I1/Number1[1]	O1/bulls[2]/segment b	I8/Number3[1]
2	I2/Number1[2]	O2/cows[0]/segment c	I9/Number3[2]
3	I3/Number2[0]	O3/cows[1]/segment d	I10/Number4[0]
4	I4/Number2[1]	O4/cows[2]/segment e	I11/Number4[1]
5	I5/Number2[2]	O5/bulls[0]/segment f	I12/Number4[2]
6	I6/Save the secret number	O6/bulls[3]/segment g	none
7	none	O7/cows[3]/dot	none

VGA Output for Arduino [16]



- Author: Devin Atkin
- Description: The final goal of this project is to create an arduino VGA driver. Currently it's nothing
- GitHub repository
- HDL project
- Mux address: 16
- Extra docs
- Clock: 25175000 Hz
- External hardware: You're going to need to hook up a VGA output to the chip alongside the clock, to control it you'll need some form of microcontroller

How it works

The name is overly ambitious but that's why I'll submit it to future submisisons as I add more features. :)

This project uses the CLK input to generate a VGA output, this will default to a Random noise output, the output can be set to a background colour using the SPI interface. 32'b1000_0000_1111_1100_0000_0000_0000_0000 32-bit configuration word configuration[31:30] = 2'b11 - Set output mode 00 = Random Noise, 01 = Solid Configuration Set Colour, 10 = Coloured Text (Color set by config), 11 = Bouncing Ball configuration[29:24] = 6'b111111 - Background Colour for Solid Colour and Colour Text configuration[23] = (Forced to 0 given utilization issues character memory array data input) configuration[22] = (Forced to 0 given it increases utilization too high, this may be implemented if I choose to join TT05 in the future)character memory array write configuration[21] = Write for the character memory. (Has issues due to utilization, may be implemented if I join TT05 in the future) configuration[20:15] = Character memory being written to row memory address

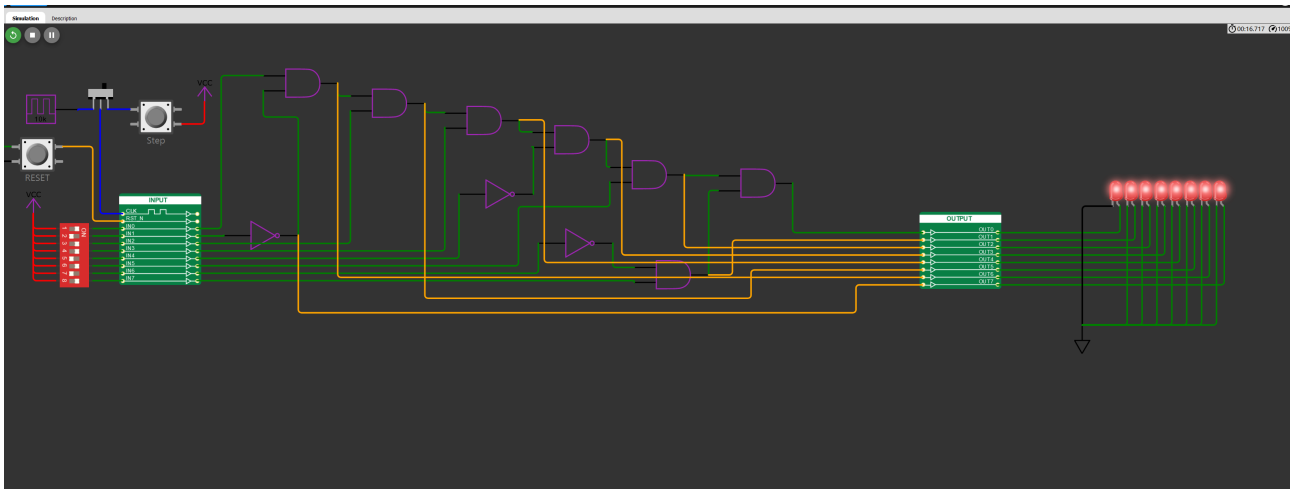
How to test

The design has been simulated and tested with the eric eastwood simulator using the output generated by tb.v. The output will be tested using a Basys 3 board prior to the final sumbission time permitting. The design is currently set up to output a 640x480 60Hz VGA signal. The output is currently set to a random test pattern. The design will have a way to adjust background colour, and draw onto the display using a few basic SPI commands. (Not yet implemented) I'm testing my output using a VGA simulator tool online https://madlittlemods.github.io/vga-simulator/ and will be testing on a Basys 3 board prior to submission.

Pinout

#	Input	Output	Bidirectional
0	SPI MOSI	VGA HSync	SPI MISO
1	SPI CLK	VGA VSync	none
2	SPI CS	VGA Red 0	none
3	none	VGA Red 1	none
4	btn_up	VGA Green 0	none
5	btn_down	VGA Green 1	none
6	none	VGA Blue 0	none
7	none	VGA Blue 1	none

Digital Cipher & Interlock System [17]



- Author: Eric German MKME Lab
- Description: Digital Cipher with 256 combinations & one solution which sets output to high
- GitHub repository
- Wokwi project
- Mux address: 17
- Extra docs
- Clock: 0 Hz
- External hardware: NA

How it works

Can be used as a simple puzzle demo or as a safety chain/interlock on equipment. Being hardware interlocks without microcontroller logic it mimics a standalone safety relay function which is used to verify all subsystems are online before allowing machinery to run. The high or low input can be tied to the sensors and switches in the safety chain. Only when all are in the desired state will the output be OKAY/HIGH. NO and NC switches/sensors can be tied to the appropriate pins. Feedback signals are provided from gate outputs by FB1 through FB7

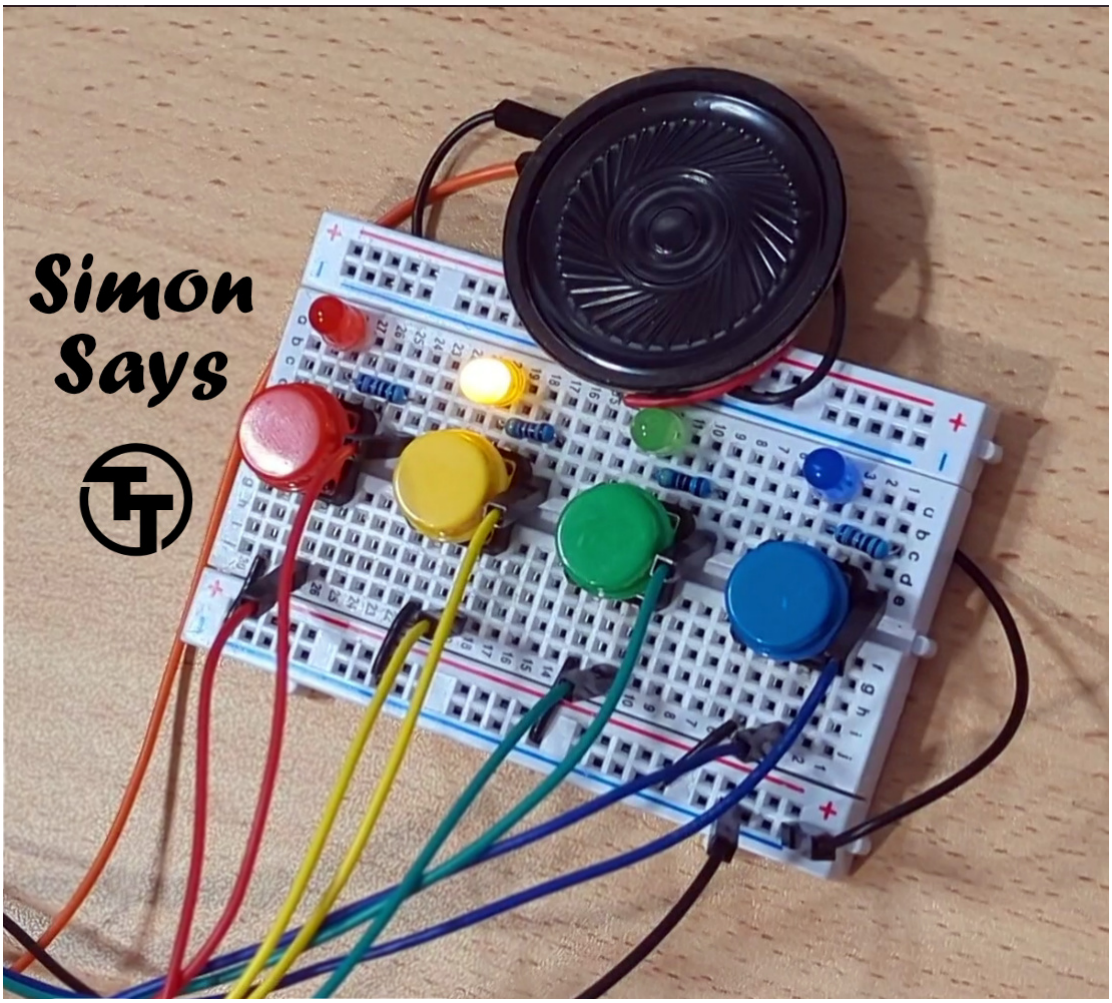
How to test

Provide below inputs on the required pins to activate output

Pinout

#	Input	Output	Bidirectional
0	HIGH	HIGH All Chain Unlocked	none
1	LOW	FB1 Feedback signal	none
2	HIGH	FB2 Feedback signal	none
3	HIGH	FB3 Feedback signal	none
4	LOW	FB4 Feedback signal	none
5	HIGH	FB5 Feedback signal	none
6	LOW	FB6 Feedback signal	none
7	HIGH	FB7 Feedback signal	none

Simon Says game [18]



- Author: Uri Shaked
- Description: A simple memory game
- GitHub repository
- HDL project
- Mux address: 18
- Extra docs
- Clock: 50000 Hz
- External hardware: Four push buttons (with pull-down resistors), four LEDs, and optionally a speaker/buzzer and two digit 7-segment display

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/371755521090136065 (including wiring diagram).

How to test

You need four buttons, four LEDs, resistors, and optionally a speaker/buzzer and a two digit 7-segment display for the score.

Ideally, you want to use 4 different colors for the buttons/LEDs (red, green, blue, yellow).

1. Connect the buttons to pins `btn1`, `btn2`, `btn3`, and `btn4`, and also connect each button to a pull down resistor.
2. Connect the LEDs to pins `led1`, `led2`, `led3`, and `led4`, matching the colors of the buttons (so `led1` and `btn1` have the same color, etc.)
3. Connect the speaker to the `speaker` pin.
4. Connect the seven segment display as follows: `seg_a` through `sev_g` to individual segments, `dig1` to the common pin of the tens digit, `dig2` to the common pin of the ones digit. Set `seginv` according to the type of 7 segment display you have: high for common anode, low for common cathode.
5. Reset the game, and then press any button to start it. Enjoy!

Note: the game requires 50KHz clock input.

Pinout

#	Input	Output	Bidirectional
0	btn1	led1	seg_a
1	btn2	led2	seg_b
2	btn3	led3	seg_c
3	btn4	led4	seg_d
4	seginv	speaker	seg_e
5	none	dig1	seg_f
6	none	dig2	seg_g

#	Input	Output	Bidirectional
7	none	none	none

YKM 7-seg driver [19]

- Author: Yeo Kheng Meng
- Description: Shows the string ykM_1St_CHIP character by character
- GitHub repository
- Wokwi project
- Mux address: 19
- Extra docs
- Clock: 0.5 Hz
- External hardware: 7-segment LCD

How it works

The string is shown by individual characters to the 7-segment LCD. By default with all pins except Clock being Low, the chip will cycle through all the characters depending on clock speed. To display individual characters manually, set HIGH to counter pin and BCD. Then select the bits 0-3 manually.

How to test

See how_it_works.

Pinout

#	Input	Output	Bidirectional
0	clock	7-segment a	none
1	none	7-segment b	none
2	none	7-segment c	none
3	none	7-segment d	none
4	Disable counter. This is active-high.	7-segment e	none
5	Driven by BCD or counter. High for BCD, Low for counter.	7-segment f	none
6	BCD bit 3	7-segment g	none
7	BCD bit 2	none	none

Configurable PID Block [20]

- Author: Maxim Vasic
- Description: It was meant to be a final project, but that was undercut.
- GitHub repository
- HDL project
- Mux address: 20
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

It's a PID block to be configured with I2C. GPIOs 7 through 2 are for error/control, and GPIOs 1 and 0 are SDA and SCL.

How to test

Configure with the I2C frame, see the I2C files for the "addresses". See the I2C test file for an example.

Pinout

#	Input	Output	Bidirectional
0	none	segment a	Error/Control
1	none	segment b	Error/Control
2	none	segment c	Error/Control
3	none	segment d	Error/Control
4	none	segment e	Error/Control
5	none	segment f	Error/Control
6	none	segment g	SDA
7	none	dot	SCL

PWM audio [21]

- Author: Yeo Kheng Meng
- Description: Takes in 8-bit audio over a parallel (port) interface then generates an analog audio signal like a Covox Speech Thing.
- GitHub repository
- HDL project
- Mux address: 21
- Extra docs
- Clock: 10000000 Hz
- External hardware: A 0.1uF capacitor to ground is recommended on the 2 audio output pins

How it works

This is meant as a parallel port sound card like a Covox Speech Thing. Instead of R-2R resistors, the chip will generate the analog audio output using PWM and First-order sigma-delta modulator.

How to test

No particular test required.

Pinout

#	Input	Output	Bidirectional
0	Bit 0 of Parallel port (LSB)	Standard PWM audio output	Direct from input 0
1	Bit 1 of Parallel port	Sigma-delta modulator output	Direct from input 1
2	Bit 2 of Parallel port	From ena pin	Direct from input 2
3	Bit 3 of Parallel port	From clk pin	Direct from input 3
4	Bit 4 of Parallel port	From rst_n pin	Direct from input 4
5	Bit 5 of Parallel port	Static 1	Direct from input 5
6	Bit 6 of Parallel port	Static 0	Direct from input 6
7	Bit 7 of Parallel port (MSB)	Static 1	Direct from input 7

4-bit ALU [22]

- Author: David Bertuch
- Description:
- GitHub repository
- HDL project
- Mux address: 22
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

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

Puts the bottom 8 bits of the counter on the bidirectional outputs.

With all the inputs set to 0, the internal 24 bit compare is set to 10,000,000. This means the counter will increment by one each second.

If any inputs are non zero, then the input will be used as an bits 11 to 18 of the 24 bit compare register. Example: setting the inputs to 00010000 will program 16384 into the compare register. With a 10MHz clock the counter will increment ~610 times per second.

How to test

After reset, the counter should increase by one every second with a 10MHz input clock. Experiment by changing the inputs to change the counting speed.

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

#	Input	Output	Bidirectional
6	compare bit 17	segment g	second counter bit 6
7	compare bit 18	dot	second counter bit 7

RGB Mixer [23]

- Author: Matt Venn
- Description: Use 3 rotary encoder to control 3 PWM generators
- GitHub repository
- HDL project
- Mux address: 23
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

3 PWM generators are fed by 3 debounced encoder peripherals.

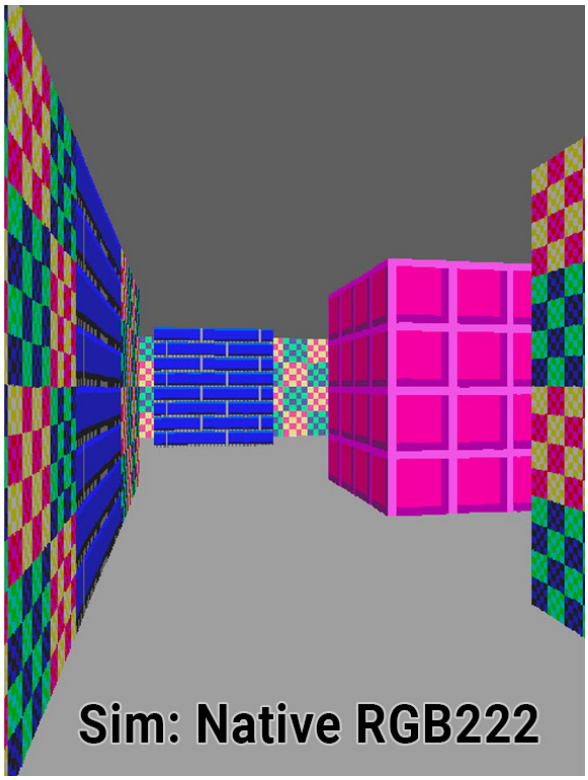
How to test

Connect 3 digital rotary encoders to the first 6 inputs. Changing the encoders will change the PWM outputs on the first 3 outputs.

Pinout

#	Input	Output	Bidirectional
0	encoder 0 pin a	pwm 0	n/a
1	encoder 0 pin b	pwm 1	n/a
2	encoder 1 pin a	pwm 2	n/a
3	encoder 1 pin b	n/a	n/a
4	encoder 2 pin a	n/a	n/a
5	encoder 2 pin b	n/a	n/a
6	n/a	n/a	n/a
7	n/a	n/a	n/a

raybox-zero [33]



- Author: algofoogle (Anton Maurovic)
- Description: Simple VGA ray caster game demo
- GitHub repository
- HDL project
- Mux address: 33
- Extra docs
- Clock: 25000000 Hz
- External hardware: VGA connector with RGB222 DAC

How it works

NOTE: Expect updates after the TT04 datasheet is made. Check tt04-raybox-zero's README (https://github.com/algofoogle/tt04-raybox-zero) for latest info.

This framebuffer-less VGA display generator is 'racing the beam' to yield a simple realtime "3D"-like render of a game map using ray casting. It's inspired by Wolf3D and based on Lode's Raycasting tutorial (https://lodev.org/cgtutor/raycasting.html). Think of it as a primitive 'GPU' using a grid map of wall blocks, with basic texture mapping and flat-coloured floor/ceiling. No doors or sprites – but maybe in TT05? In TT04's 130nm process we use 4x2 tiles (~0.16mm^2) at ~48% density.

Without a framebuffer, rendering/animation occurs at full speed. Registers store the 'POV' (Point of View) to render. It's expected that a host controller implements game/motion logic and calculates the POV, then sending it to the chip via SPI (`ss_n`/`sclk`/`mosi`). An MCU or low-spec CPU should do. I've been bit-banging SPI with a Raspberry Pi Pico.

At reset the POV registers are set to an angled view of the inbuilt 16x16 grid map.

NOTE: "FPS games" like Wolf3D use a landscape display, i.e. normal desktop monitor orientation. I designed this as a portrait display (rotated 90° clockwise) for silicon area optimisations that come with rendering by scanline instead of by column. If you don't want a sideways monitor, design a game/demo using this different perspective. For example, image Mario's 1st-person view of his 2D platform world...

Features

- 640x480 VGA display at ~60Hz from 25MHz clock (25.175MHz ideal)
- Registered and unregistered digital VGA outputs: RGB222 and H/VSYNC
- Portrait "FPS" orientation
- Hard-coded 16x16 map with 3 textures: light- and dark-side variations
- SPI interface to set POV with debug option to see POV register bits
- 'SPI2' interface to set ceiling colour, floor colour, or floor 'leak'
- Reset loads an interesting POV. Optional 'demo mode' inputs can vary it.
- HBLANK and VBLANK outputs as optional interrupt requests

A warning about turning your screen on its side

As stated, this is designed to drive a display with a *portrait* orientation when used as a "first person shooter" but BEWARE: The backlights failed on *two* old flat panel VGA displays (from circa 2003) not long after I turned them on their sides. Coincidence? Age? A CCFL failure mode? Not sure. I'm using a monitor from 2008 now.

How to test

Attach a VGA connector's HSYNC and VSYNC to the chip's respective outs with (say) inline 100R resistors for protection. Connect at least `red[1]`, `green[1]`, `blue[1]` with inline 270R resistors, or better yet use an R2R DAC on each colour output *pair*. Make sure VGA GND is connected, of course.

Pull up `reg` to select 'registered outputs'. Without this, you will get the unregistered versions, which might be murky or have some timing issues – I included this option for testing purposes. In the actual ASIC version of this, I expect the registered outputs will be much cleaner, but we'll see.

Supply a 25MHz clock (or ideally 25.175MHz), and assert the `reset` signal, and you should get a clockwise-90°-rotated display of textured walls with dark grey ceiling (right-hand side) and light grey floor (left-hand side).

Pull up the `debug` input and you should see little squares show up in the corner of the screen that represent the current state of the POV registers.

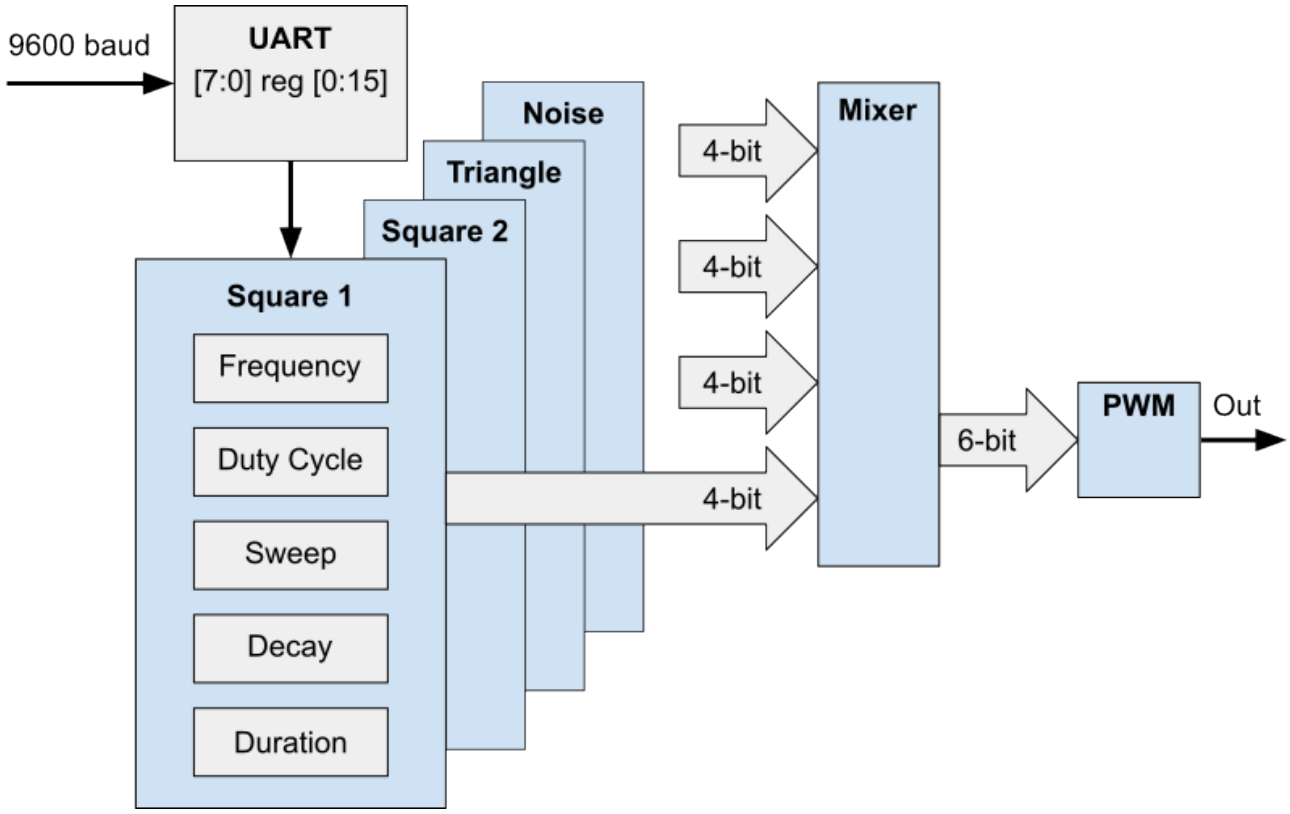
Pull up either/both of `inc_px/py` and the view should drift along slowly. This is 'demo mode'. Don't be alarmed when it goes through walls, or for periods when you see half the screen is just grey and the other half is flickering different colours – this just means you're moving *through* a wall block.

Changing POV: https://github.com/algofoogle/tt04-raybox-zero#write-pov-via-spi

Pinout

#	Input	Output	Bidirectional
0	SPI in: sclk	hsync_n	Out: o_hblank
1	SPI in: mosi	vsync_n	Out: o_vblank
2	SPI in: ss_n	red[0]	SPI2 in: reg_sclk
3	debug	red[1]	SPI2 in: reg_mosi
4	inc_px	green[0]	SPI2 in: reg_ss_n
5	inc_py	green[1]	none
6	reg	blue[0]	none
7	none	blue[1]	none

ChipTune [37]



- Author: Wallace Everest
- Description: Vintage 8-bit sound generator
- GitHub repository
- HDL project
- Mux address: 37
- Extra docs
- Clock: 1789773 Hz
- External hardware: Computer COM port

How it works

ChipTune implements an 8-bit Programmable Sound Generator (PSG). Input is from a serial UART interface. Output is PWM audio.

Overview This project replicates the Audio Processing Unit (APU) of vintage video games.

Statistics

- Tiles: 2x2
- DFF: 417
- Total Cells: 2549
- Utilization: 32%

TinyTapeout 4 Configuration

TT04 devices from the eFabless Multi-Project Wafer (MPW) shuttle are delivered in QFN-64 packages, mounted on a daughterboard for breakout.

Based on data from:

- https://github.com/efabless/caravel_board/blob/main/hardware/breakout/caravel-M.2-card-QFN/caravel-M.2-card-QFN.pdf
- https://github.com/TinyTapeout/tt-multiplexer/blob/main/docs/INFO.md
- https://open-source-silicon.slack.com/archives/C016N88BX44/p1688915892223379

MPRJ_IO Pin Assignments

Signal	Name	Dir	QFN	PCB
mprj_io[0]	jtag	in	31	
mprj_io[1]	sdo	out	32	
mprj_io[2]	sdi	in	33	
mprj_io[3]	csb	in	34	
mprj_io[4]	sck	in	35	
mprj_io[5]	user_clk	out	36	
mprj_io[6]	clk	in	37	
mprj_io[7]	rst_n	in	41	
mprj_io[8]	ui_in[0]	in	42	
mprj_io[9]	ui_in[1]	in	43	
mprj_io[10]	ui_in[2]	in	44	
mprj_io[11]	ui_in[3]	in	45	
mprj_io[12]	ui_in[4]	in	46	
mprj_io[13]	ui_in[5]	in	48	
mprj_io[14]	ui_in[6]	in	50	
mprj_io[15]	ui_in[7]	in	51	
mprj_io[16]	uo_out[0]	out	53	
mprj_io[17]	uo_out[1]	out	54	
mprj_io[18]	uo_out[2]	out	55	
mprj_io[19]	uo_out[3]	out	57	
mprj_io[20]	uo_out[4]	out	58	
mprj_io[21]	uo_out[5]	out	59	

Signal	Name	Dir	QFN	PCB
mprj_io[22]	uo_out[6]	out	60	
mprj_io[23]	uo_out[7]	out	61	
mprj_io[24]	uio[0]	bid	62	
mprj_io[25]	uio[1]	bid	2	
mprj_io[26]	uio[2]	bid	3	
mprj_io[27]	uio[3]	bid	4	
mprj_io[28]	uio[4]	bid	5	
mprj_io[29]	uio[5]	bid	6	
mprj_io[30]	uio[6]	bid	7	
mprj_io[31]	uio[7]	bid	8	
mprj_io[32]	sel_ena	in	11	
mprj_io[33]	spare		12	
mprj_io[34]	sel_inc	in	13	
mprj_io[35]	spare		14	
mprj_io[36]	sel_rst_n	in	15	
mprj_io[37]	spare		16	

APU Operation

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

An explanation of register functions can be found on the NESDEV website. Only the lower 4-bits of the address are decoded.

- https://www.nesdev.org/wiki/APU

UART Operation

- A register address and data are recovered from two consecutive serial bytes.
- A byte with the msb=0 is considered the first byte with 7-bits of data.
- A byte with msb=1 is considered the second byte with the remaining 1-bit of data and a 6-bit address.
- A ready flag is generated after receiving the second byte.

```
Byte 1                              Byte 2
Start D0 D1 D2 D3 D4 D5 D6 0 Stop  Start D7 A0 A1 A2 A3 A4 A5 1 Stop
```

Pin Assignments

Signal	Name	Signal	Name
clk	12 MHz	ena	spare
rst_n	spare	uio_oe[7:0]	spare
ui_in[0]	spare	uo_out[0]	blink
ui_in[1]	spare	uo_out[1]	link
ui_in[2]	rx	uo_out[2]	tx
ui_in[3]	spare	uo_out[3]	pwm
ui_in[4]	spare	uo_out[4]	square1
ui_in[5]	spare	uo_out[5]	square2
ui_in[6]	spare	uo_out[6]	triangle
ui_in[7]	spare	uo_out[7]	noise
uio_in[7:0]	spare	uio_out[7:0]	spare

- CLK is a 1.789733 MHz clock
- BLINK is an LED status indicator with a 1 Hz rate
- LINK is an LED activity indicator of the RX signal
- PWM is the pulse-width modulated audio output
- RX is a UART input (9600,8,N,1)
- TX generates a frame synchonization character (0x80)

How to test

The ChipTune project can be interfaced to a computer COM port (9600,n,8,1). An analog PWM filter and audio driver are needed for the test rig.

The following serial strings will activate example functions:

```
# Square 1
08 82 30 80 00 84 00 86 #Clear
27 83 02 81 7E 85 08 86 #PlaySmallJump
27 83 02 81 7C 84 09 86 #PlayBigJump
13 83 1E 81 3A 84 0A 86 #PlayBump
19 83 1E 81 0A 84 08 86 #PlayFireballThrow
4B 83 1F 81 6F 85 08 86 #PlaySmackEnemy
1C 83 1E 81 7E 85 08 86 #PlaySwimStomp
08 82 3F 81 17 84 01 86 #400Hz
# Square 2
30 88 08 8A 00 8C 00 8E #Clear
18 89 7F 8A 71 8C 08 8E #PlayTimerTick
```

```
0D 89 7F 8A 71 8C 08 8E #PlayCoinGrab
1F 89 14 8B 79 8D 0A 8E #PlayBlast
7F 88 5D 8A 71 8C 08 8E #PlayPowerUpGrab
3F 89 08 8A 17 8C 01 8E #400Hz
# Triangle
00 91 00 92 00 94 00 96 #Clear
00 92 0B 95 00 96 40 91 #400Hz
# Noise
30 98 00 9A 00 9C 00 9E #Clear
00 9A 00 9E 05 9D 3F 98 #300Hz
```

Pinout

#	Input	Output	Bidirectional
0	None	Blink	None
1	None	Link	None
2	RX	TX	None
3	None	PWM	None
4	None	Square1	None
5	None	Square2	None
6	None	Triangle	None
7	None	Noise	None

OpenSource PWM Peripheral [48]

- Author: Medinceanu Paul-Catalin
- Description: The purpose of this project is to develop an Opensource PWM Peripheral with advanced functions and configurations. These capabilities are needed mostly in Power Electronics, where fine tuning of the control signals is crucial. The two standout functions that I have implemented are the deadband and the synchronization between counters.
- GitHub repository
- HDL project
- Mux address: 48
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

The peripheral contains 3 counters, each of them with 2 comparators and 2 deadband generators. The counter is set to go to a value, the comparator is wired to the registers of the counter and decides when to set or clear the squarewave signal. By setting the counter to switch at different values, any duty cycle can be achieved. The project can be configured to generate specific PWM signals by writing the desired configuration at the right address in the register file. The register map can be found in the 'doc' folder of the repo.

How to test

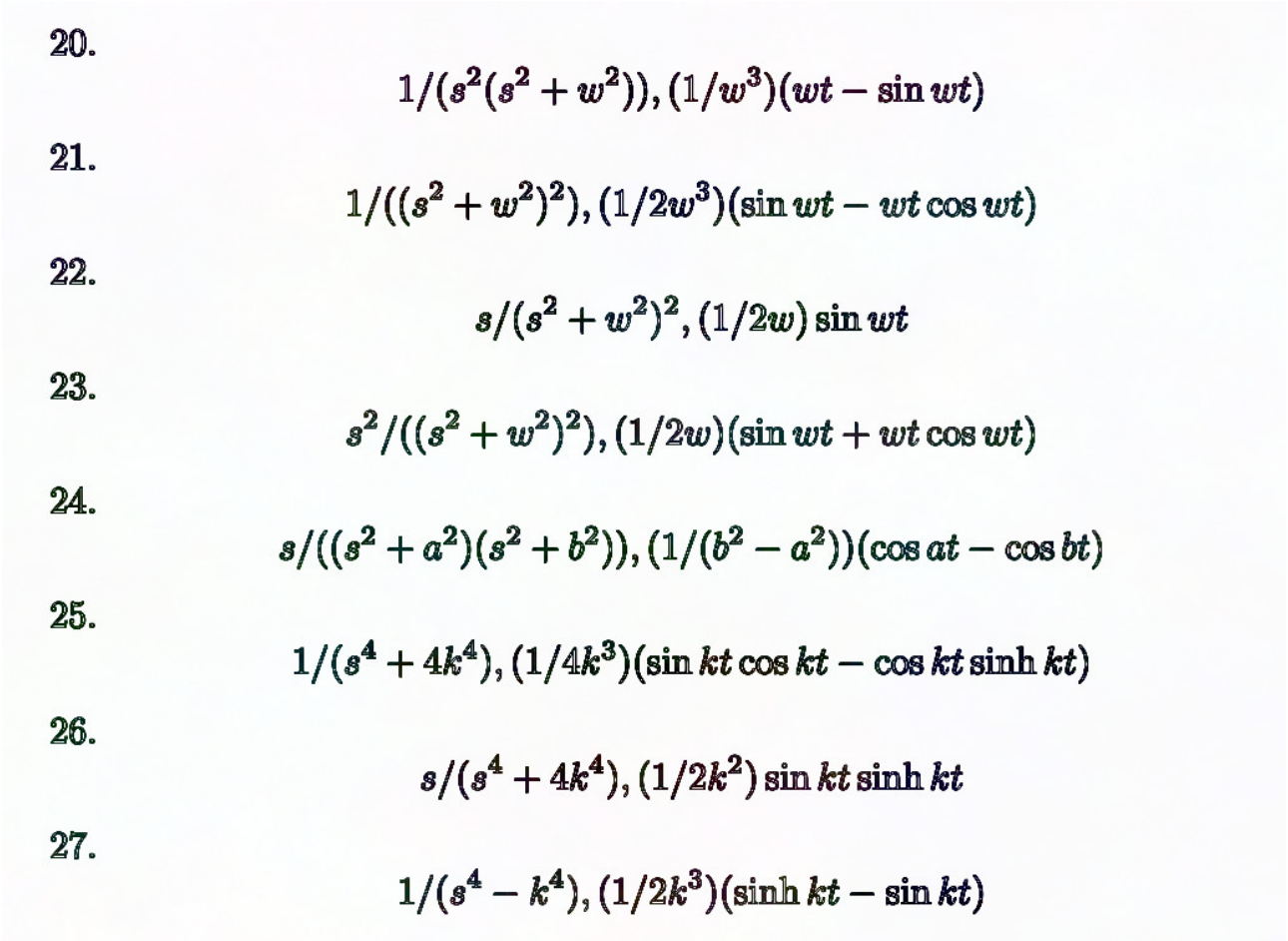
To test the project, it should be tied through the input and bidirectional pins to a microcontroller. A C code(for a specific microcontroller) will be available on the repo to configure the peripheral through the serial terminal. After the setup is done, the PWM signals will be visible at the output pins. An alternative way to configure the peripheral is through the onboard switches. The write enable is held high, the address is set next on the input pins and finally the data is set at the bidirectional pins. The process is repeated for each register of the PWM module in use.

Pinout

#	Input	Output	Bidirectional
0	i_address[5]	unused	io_data[7]

#	Input	Output	Bidirectional
1	i_address[4]	unused	io_data[6]
2	i_address[3]	o_pwm1A	io_data[5]
3	i_address[2]	o_pwm1B	io_data[4]
4	i_address[1]	o_pwm2A	io_data[3]
5	i_address[0]	o_pwm2B	io_data[2]
6	unused	o_pwm3A	io_data[1]
7	i_write_en	o_pwm3B	io_data[0]

Experiment Number Six: Laplace LUT [50]



20.

$$1/(s^2(s^2+w^2)), (1/w^3)(wt-\sin wt)$$

21.

$$1/((s^2+w^2)^2), (1/2w^3)(\sin wt - wt\cos wt)$$

22.

$$s/(s^2+w^2)^2, (1/2w)\sin wt$$

23.

$$s^2/((s^2+w^2)^2), (1/2w)(\sin wt + wt\cos wt)$$

24.

$$s/((s^2+a^2)(s^2+b^2)), (1/(b^2-a^2))(\cos at - \cos bt)$$

25.

$$1/(s^4+4k^4), (1/4k^3)(\sin kt\cos kt - \cos kt\sinh kt)$$

26.

$$s/(s^4+4k^4), (1/2k^2)\sin kt\sinh kt$$

27.

$$1/(s^4-k^4), (1/2k^3)(\sinh kt - \sin kt)$$

- 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: 50
- Extra docs
- Clock: 1 Hz
- External hardware:

How it works

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 43. 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 (0-45) 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.

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
7	Clock divider disable	RHS_BIT_7	LHS_BIT_7

Karplus-Strong String Synthesis [52]

- Author: Chinmay Patil
- Description: Plucked string sound synthesizer
- GitHub repository
- HDL project
- Mux address: 52
- Extra docs
- Clock: 256000 Hz
- External hardware:

How it works

This is simplified implementation of Karplus-Strong (KS) string synthesis based on papers, Digital Synthesis of Plucked-String and Drum Timbres and Extensions of the Karplus-Strong Plucked-String Algorithm.

A register map controls and configures the KS synthesis module. This register map is accessed through a SPI interface. Synthesized sound samples can be accessed through the I2S transmitter interface.

SPI Frame

SPI Mode: CPOL = 0, CPHA = 1

The 16-bit SPI frame is defined as,

$\textRead/\overline\textWrite$	$\textAddress[6:0]$	$\textData[7:0]$

Register Map

The Register Map has 16 Registers of 8-bits each. It is divided into configuration and status registers,

Complete register map is described in the repository at https://github.com/pyamnihc/tt04-um-ks-pyamnihc.

I2S Transmitter

The 8-bit signed sound samples can be read out at `f_sck = 256 kHz` through this interface.

How to test

Connect a clock with frequency `f_clk = 256 kHz` and apply a reset cycle to initialize the design, this sets the audio sample rate at `fs = 16 kHz`. Use the spi register map or the ui_in to futher configure the design. The synthesized samples are sent continuously on the I2S transmitter interface.

Pinout

#	Input	Output	Bidirectional
0	~rst_n_prbs_15, ~rst_n_prbs_7	segment a	sck_i
1	load_prbs_15, load_prbs_7	segment b	sdi_i
2	freeze_prbs_15	segment c	sdo_o
3	freeze_prbs_7	segment d	cs_ni
4	i2s_noise_sel	segment e	i2s_sck_o
5	~rst_n_ks_string	segment f	i2s_ws_o
6	pluck	segment g	i2s_sd_o
7	NOT CONNECTED	dot	prbs_15

USB Device [54]

- Author: Darryl Miles
- Description: USB FullSpeed/LowSpeed device (proof-of-concept)
- GitHub repository
- HDL project
- Mux address: 54
- Extra docs
- Clock: 48000000 Hz
- External hardware: USB Connector, 2 x 68 ohm resistors, 1k5 ohm resistor

How it works

This text will be updated nearer the scheduled TT04 redistribution time (early 2024) along with the project github README.md and gh-pages documentation. Please regenerate your documentation.

This is a hardware implementation of a USB device end hardware interface, should be compliant with USB1.1 FS/LS (not HS).

It is designed to be driven and commanded by a CPU over a native bus (such as WishBone). Due to the limited IO ports with TinyTapeout there is a TT2WB WishBone driver that provides the ability to perform WishBone 32bit data-path transactions inside the module over the narrower TT IO ports. The hardware design is capable of being any kind of USB device, this includes (and it not limited to) CDC, HID, audio, storage as the CPU sets the identity of the device in software to the host.

While I intend to drive the TT IC with an FPGA development board myself, it should be possible for the RP2040 providing a 48MHz clock to this project to achieve some kind of hello world over USB. The controller clock uses the global CLK pin (this is expected to be 48MHz but may well work at a range of other clock rates), the PHY interface uses BIDI port 3 (this must be 48MHz to provide timing in both full-speed and low-speed modes).

You can fire WishBone commands at the TT2WB interface, this maps to most of the ports IN/OUT/BIDI. This is currently abstracted away (via API) for testbench purposes and it is intended this also be the case for the FPGA/RP2040 programming interfaces. At this time the best documentation around this is to look at the TT2WB.py and tt04_to_wishbone.v in the project.

The configuration options were reduced to squeeze something to demonstrate a working endpoint into a 2x2 tile space, this is possible because SpinalHDL is good a generating hardware designs based on complex parametrization that allow features to be turned

on-off easily. The limited 2x2 tile space (ideally it wanted 2x4) has resulted in some limitations:

- the total number of endpoints is reduced from the full 16 down to 4 (numbered 0 to 3)
- the total buffer space available is reduced to just 52 bytes (this may be only enough space for a single active endpoint to operate, a standard serial CDC ideally need 3 endpoints working). The buffer space is provided by DFF registers and has a particular layout for control information and headers, this results in a total of 52 bytes only allowing a single endpoint to operate with a MaxPacketLen=8. If a 2x3 tile were possible 96-108 bytes of buffer would be possible which would allow 3 endpoints to operaten all at MPL=8 or a single endpoint at upto MPL=64 or some combination in between.

It is necessary to create a suitable USB cable to connect to the BIDI port0 and port1, this is expected to be the same cable and pinout scheme as the tt04-usbcbc project that is also present in TT04. This recommends a 68ohm series resistor for each of the Data+ and Data- lines, along with a pull-up resistor 1k5. The single pull-up resistor needs to be positioned appropriately for full-speed (on Data+) and low-speed (on Data-) modes.

How to test

This text will be updated nearer the scheduled TT04 redistribution time (early 2024) along with the project github README.md and gh-pages documentation. Please regenerate your documentation.

The original bus interface documentation can be found at https://spinalhdl.github.io/SpinalDoc-RTD/master/SpinalHDL/Libraries/Com/usb_device.html

The original source can be found at https://github.com/SpinalHDL/SpinalHDL/tree/dev/lib/src/

This hardware was originally designed for FPGA so some modifications were made in the areas of:

- Improving USB specification (a couple of potential bugs/out-of-spec items)
- Use of combinational logic versions of CRC5/CRC16 function blocks.
- Fixing features that seemed 95 percent written and present in the code but obviously not working or tested (support dual full-speed and low-speed in same hardware stack)
- Running on ASIC (clocks/resets)
- Optimizing for ASIC (UsbTimer counter widths, DFF buffer reduction squeeze, endpoint reduction squeeze)
- Encapsulating WishBone bus inside a TinyTapeout User project.

- More items I've already forgotten on the way (but can document from code walk later)

The cocotb tests cover a significant number of the features and provide VCD output demonstrating almost everything possible with this hardware.

The Verilator/coverage showed the 2 main areas I do not exercise host suspend/resume and device resumeIt support. Plus a number of error scenarios and a few non-critial minor features of the hardware.

I hope by early 2024 to have available some FPGA and some RP2040 application code to assist demonstration.

Pinout

#	Input	Output	Bidirectional
0	tt2wb input bit0	tt2wb output bit0	USB D+ (bidi)
1	tt2wb input bit1	tt2wb output bit1	USB D- (bidi)
2	tt2wb input bit2	tt2wb output bit2	Interrupt (output only)
3	tt2wb input bit3	tt2wb output bit3	Phy Clock 48MHz (input only)
4	tt2wb input bit4	tt2wb output bit4	tt2wb control ACK (output only)
5	tt2wb input bit5	tt2wb output bit5	tt2wb control CMD bit0 (input only)
6	tt2wb input bit6	tt2wb output bit6	tt2wb control CMD bit1 (input only)
7	tt2wb input bit7	tt2wb output bit7	tt2wb control CMD bit2 (input only)

Audio-PWM-Synth [64]

- Author: Thorsten Knoll
- Description: Generate Audio with a PWM output.
- GitHub repository
- HDL project
- Mux address: 64
- Extra docs
- Clock: 12MHz Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	none	pwm_audio_low	none
1	none	pwm_audio_high	none
2	none	none	none
3	none	none	none
4	none	none	none
5	none	none	none
6	none	none	none
7	none	none	none

German Traffic Light [71]

- Author: Paul Knoll
- Description: Simulation of a german traffic light
- GitHub repository
- Wokwi project
- Mux address: 71
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

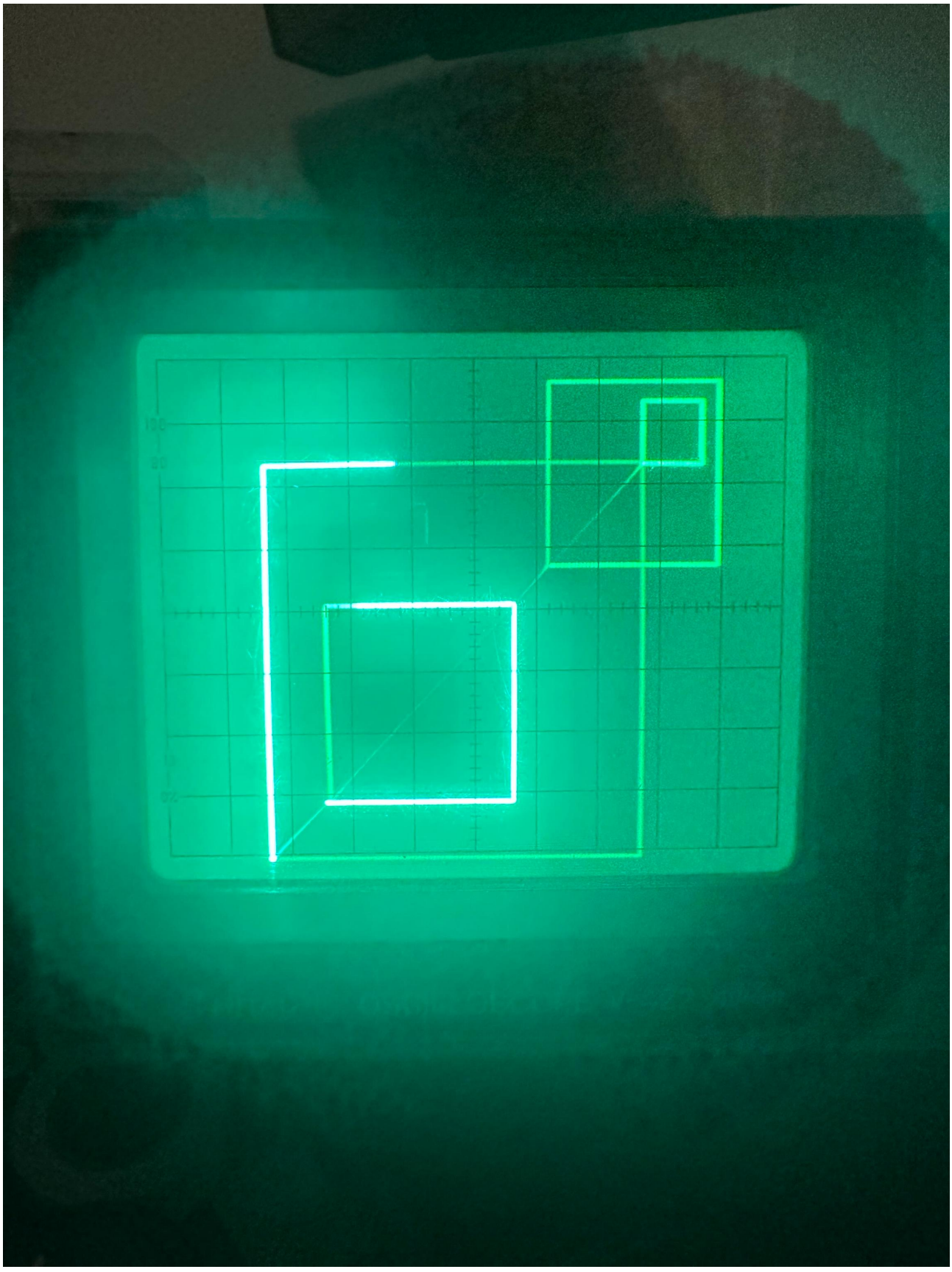
How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	none	red LED	none
1	none	yellow LED	none
2	none	green LED	none
3	none	none	none
4	none	none	none
5	none	none	none
6	none	none	none
7	none	none	none

Dandy VGA [96]



- Author: Blaise Saunders

- Description: Serial vector graphics adapter
- GitHub repository
- HDL project
- Mux address: 96
- Extra docs
- Clock: 10000000 Hz
- External hardware: 2 DACs, vector graphics display, fastish UART input

How it works

This module is a serial vector graphics adapter [or VGA for short ;^)] it has a small set of state machine based primitive instructions that can be programmed over serial and animated etc. It does a nice job of drawing squares and I'm hoping 3D graphics can be achieved with use of the line primitive. There are 16 instruction registers that can be programmed and updated live over serial.

How to test

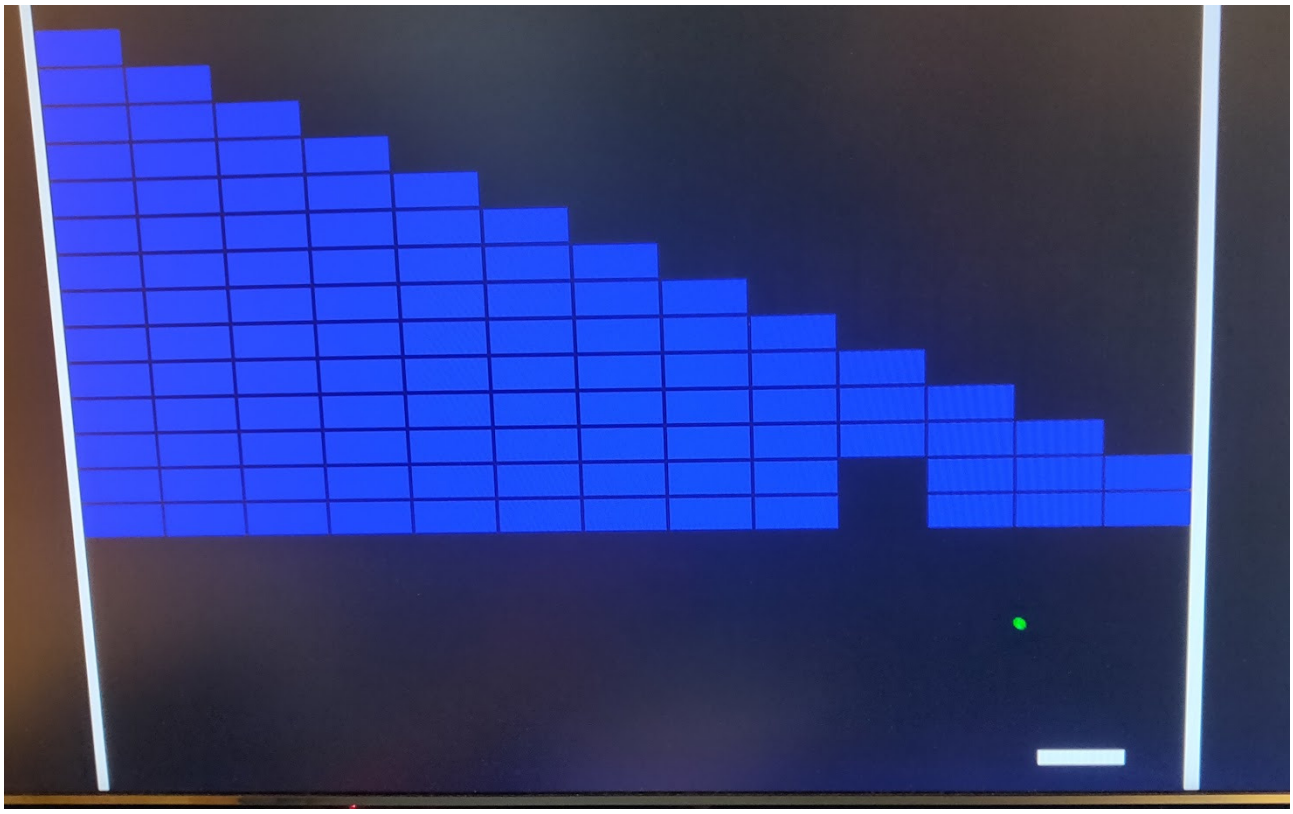
The device outputs 8 bits on the bidirectional IO and 8 bits on the output pins, one is for X and the other is for Y display on a vector graphics display, most oscilloscopes with XY mode should work well. Can convert to analog with a simple R2R resistor ladder or whatever method you like best :). You can find some test code that can be flashed a Teensy or similar and hook it up to pin 1 on the input to send some primitives to be drawn: https://github.com/DavidoRotho/tt04-davidroth-dandy-vga

Pinout

#	Input	Output	Bidirectional
0	UART RX input, 921600 baud input. Tested with Teensy 4.1	binary X axis output [7:0]	binary Y axis output [7:0]
1	Safe/Unsafe mode toggle, unsafe high. Whether or not to wait while	n/a	n/a

#	Input	Output	Bidirectional
2	binary graphics clock divider for compensating for slow DAC drive speed [2:7]	n/a	n/a
3	n/a	n/a	n/a
4	n/a	n/a	n/a
5	n/a	n/a	n/a
6	n/a	n/a	n/a
7	n/a	n/a	n/a

Tiny Breakout [98]



- Author: Robbert-Jan de Jager
- Description: This is a small breakout game implemented in HDL. It uses a VGA connector to output the video signal. The game is controlled by 3 buttons. The left button moves the paddle to the left, the right button moves the paddle to the right and the action button starts the game. The game is over when all blocks are destroyed or when the ball hits the bottom of the screen.
- GitHub repository
- HDL project
- Mux address: 98
- Extra docs
- Clock: 25175000 Hz
- External hardware: 3x 2bit DAC for the red, green and blue video signals. VGA connector. 3 buttons.

How it works

Basic operation The core of the design is the vga_timing module. This module generates all the required timing signals. Some of these signals like hsync and vsync are used to generate the video signal, while others like the horizontal and vertical position

are used to generate the graphics. The horizontal and vertical sync signals are also used for the game logic.

Before outputting the video signal the video mux selects the correct input color to display. It does so based on the highest priority component that wants to output a color.

We have multiple painter modules. These generate from the current game state and the current horizontal and vertical position the correct color to display. Ideally the painters would not contribute to the game logic, but for optimization reasons they do.

While drawing a frame the game logic keeps track of collisions. It does so by checking if multiple painters want to draw at the same position. If so it will latch a collision, which will be processed after drawing the frame.

At the end of each frame the game logic will calculate the next ball position, taking collisions into account. The collision with the paddle is special. To have an entertaining game that does not play the same every time the ball will bounce off the paddle at a different angle depending on where it hits. This is done by splitting the paddle into multiple segments and checking for collisions with these segments. The game logic will look at the segment when a paddle collision was registered. An exception to the end of frame gamestate update is the breaking of blocks. It would require too much memory to keep the updated state for the next frame. Instead we will update the row of blocks that was just finished drawing.

We can display a grid of 13x16 blocks. This requires 208 bits of memory. This is a lot of memory for such a small design and takes up a lot of space. To reduce the number of connections the state has been put into a shift register that outputs one row at a time. This shift register is rotated 13 bits when we reach the end of drawing a row of blocks. Also we can write to the shift register the new block state if a block has been broken. This is done one clock cycle before shifting to the next row. 1 clock cycle after shifting to the next row we load the current row into a buffer which will be used to update the state.

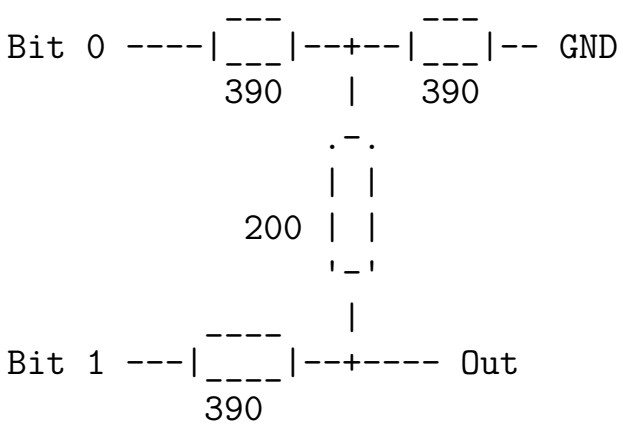
How to test

This is a small breakout game implemented in HDL. It uses a VGA connector to output the video signal. The game is controlled by 3 buttons. The left button moves the paddle to the left, the right button moves the paddle to the right and the action button starts the game. The game is over when all blocks are destroyed or when the ball hits the bottom of the screen.

Required hardware This project requires a VGA monitor and a VGA DAC. An easy way to create the VGA DAC is to use 3 2-bit R2R DACs. The 2-bit R2R DACs can be created using 2 resistors per bit. The resistors should be 200Ohm and 390Ohm. For the 3.3V power supply.

What has not been verified is the current sourcing capability of the ASIC, If it can not at least source 10mA through each pin and 30mA through the power supply pins you should add a buffer before the DAC.

The VGA DAC should be connected as follows:



Every color should have an identical copy of this DAC. The red DAC should be connected to the red VGA pin, the green DAC to the green VGA pin and the blue DAC to the blue VGA pin. The outputs of the DACs should be connected to the VGA connector. The HSync and VSync pins should also be connected to the VGA connector. The following connections need to be made to the VGA connector:

- Red DAC output to VGA connector pin 1
- Green DAC output to VGA connector pin 2
- Blue DAC output to VGA connector pin 3
- HSync to VGA connector pin 13
- VSync to VGA connector pin 14
- GND to VGA connector pin 5, 6, 7, 8

SPI interface For changing the game state externally you can use the SPI interface. The SPI interface returns the current game state when reading and accepts a few commands when writing. The SPI interface uses 16 bit words.

The returned state is as follows:

- bit 0-12: The block state of the current row. Use HBlank and VBlank to determine which row is currently being drawn.

- bit 13: right button state
- bit 14: left button state
- bit 15: action button state
- bit 16: collision state. This bit is set when a collision has been detected.
- bit 17: ball out of bounds. This bit is set when the ball is off screen.
- bit 18: game state: 0 = game idle, 1 = game running
- bit 19-20: remaining lives

When writing the first word is the command word, the following words are the data words for the command. Command words:

- 0x0000: Do nothing. Usefull when you want to read the state.
- 0x0001: Write a row state. This will shift the state to the next row. Be sure to only use this during the VBlank and call this with 15 words to completely draw the screen.
- 0x0002: Send control word. The next word is the control word. The control word is as follows:
  - bit[0]: Send the stop game command.

Board configuration The ASIC requires an input clock of 25.175MHz. The 7-Segment display is not used.

Pinout

#	Input	Output	Bidirectional
0	MOSI	HSync	MISO
1	SCK	VSync	HBlank
2	slave select	Red output bit 0	VBlank
3	none	Red output bit 1	sound output. Connect to a speaker with amplifier.
4	none	Green output bit 0	none
5	Button left	Green output bit 1	none
6	Button right	Blue output bit 0	none
7	Button action	Blue output bit 1	none

VC 16-bit CPU [99]

- Author: Paul Campbell
- Description: VC 16-bit CPU - RISV-C cpu
- GitHub repository
- HDL project
- Mux address: 99
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

See the README.MD for more info

How to test

Included is an assembler and a memory image for executing

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

Risc-V Nano V [100]

- Author: Michael Bell
- Description: RV32E bit serial processor
- GitHub repository
- HDL project
- Mux address: 100
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

A minimal RV32E processor using an SPI RAM for instructions and data.

The SPI RAM and a UART are connected to the bidi IOs. The SPI RAM is clocked at the same speed as the input clock.

The CPU has no instruction or data cache and effectively runs at clock speed / 32. More details can be found in the nanoV repo.

Restrictions/unimplemented parts of RV32E:

- register x3/gp is hardcoded to 0x00001000 (this allows data in the low 6KB of RAM to be accessed cheaply).
- register x4/tp is hardcoded to 0x10000000 (this allows the GPIO and UART to be accessed cheaply)
- The ebreak instruction (and interrupts in general) are not implemented.

The gp and tp registers are not written to by normal programs compiled by gcc, so the regular build of gcc can be used to build programs.

The inputs and outputs are for general purpose use from the CPU, the outputs can be written at address 0x10000000, and inputs read at 0x10000004.

The UART runs at clock speed / 128 (e.g. 93750 baud with a 12MHz clock). Bytes can be written or read at address 0x10000010. The UART provides 2 bits of status at address 0x10000014:

- Bit 1: Received data waiting
- Bit 0: Data transmit in progress

There are no transmit or receive FIFOs, before a program sends a byte it should check bit 0 is low before writing. The peripheral library in nanoV-sdk does this.

How to test

Attach an SPI RAM or use the RP2040 emulated SPI RAM and build prorgams using the nanoV-sdk, found in the nanoV-sdk repo.

The SPI RAM outputs are disabled when Reset is asserted, allowing the RAM to be reprogrammed easily.

Pinout

#	Input	Output	Bidirectional
0	General purpose input 0	segment a / GP output 0	SPI RAM MOSI
1	General purpose input 1	segment b / GP output 1	SPI RAM CSn
2	General purpose input 2	segment c / GP output 2	SPI RAM SCK
3	General purpose input 3	segment d / GP output 3	SPI RAM MISO
4	General purpose input 4	segment e / GP output 4	UART RX
5	General purpose input 5	segment f / GP output 5	UART TX
6	General purpose input 6	segment g / GP output 6	UART RTS
7	General purpose input 7	dot / GP output 7	SPI RAM ~Hold

USB CDC (Serial) [101]

- Author: Uri Shaked
- Description: USB to Serial bridge
- GitHub repository
- HDL project
- Mux address: 101
- Extra docs
- Clock: 48000000 Hz
- External hardware:

How it works

All the magic happens in https://github.com/davidthings/tinyfpga_bx_usbserial.

How to test

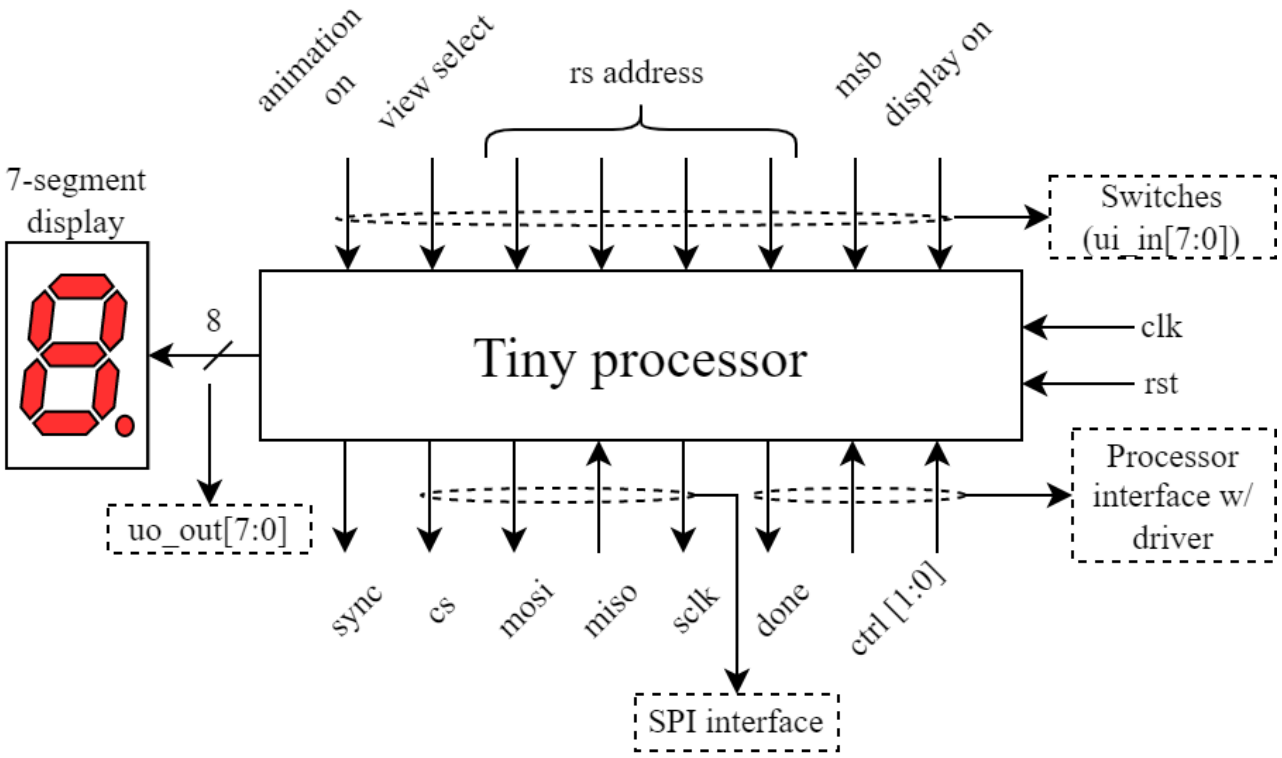
Set the clock frequency to 48 MHz. Connect usb_p and usb_n pins to D+ / D- USB pins through 68 ohm resistors. Pull up D+ with 1.5k resistor.

The device should appear as a serial port on your computer. Data received from USB host will appear on the output pins (rx) when `rx_ready` goes high. You can send data to the USB device by waiting for `tx_ready` to go high, setting the input pins (tx) to the byte you'd like to transmit, and setting `tx_valid` high.

Pinout

#	Input	Output	Bidirectional
0	tx[0]	rx[0]	usb_p
1	tx[1]	rx[1]	usb_n
2	tx[2]	rx[2]	tx_valid
3	tx[3]	rx[3]	tx_ready
4	tx[4]	rx[4]	rx_valid
5	tx[5]	rx[5]	rx_ready
6	tx[6]	rx[6]	dp_pu_o
7	tx[7]	rx[7]	configured

Tiny processor [102]



- Author: Kosmas Alexandridis
- Description: An 8-bit processor
- GitHub repository
- HDL project
- Mux address: 102
- Extra docs
- Clock: 25 000 000 Hz
- External hardware: FPGA, a device that supports SPI (optional)

How it works

The design is an 8-bit processor that supports communication with a single external device through the Serial Periferal Interface or SPI protocol, and has the capability to animate the seven segment display. To use the processor an additional external driver is needed. In this project we use an Digilent Nexys A7 FPGA. The FPGA is programmed w/ the driver.sv module. The driver's internal storages (imem, dmem) are initialized w/ .mem files. The driver then sends this data to the processor and signals it to begin execution. Once execution is finished the user can view the contents of the GPRs or watch an animation on the 7-segment display.

How to test

1. Write and assemble a simple program using the provided assembler (more on that in the README.md) to generate a .mem file. This file will be used to initialize the instructions' memory of the processor. Make a similar .mem initialization file for registers.
2. Open Xilinx' Vivado and create a project containing all the necessary file (e.g. driver.sv, tp.xdc).
3. Replace the desired file paths in the **readmemh** macros in the driver.sv module for instructions and data.
4. Connect the processor to the FPGA.
5. Program the FPGA using Vivado.

- Turn on the switch connected to the **drive** signal of the driver module. This will signal the driver to begin intializing the processor and signal it to start execution.
- Use the first switch to turn the 7-seg display [on] and [off].
- Use the second switch to select which 4-bit values (msbs [on] or lsbs [off]) of a Byte, you wish to see on the 7-seg display.
- The switches[5:2] represent the 4-bit address used to index the 14 registers available for display.
- The sixth switch changes the source between instruction [on] and data [off] memory.
- The last switch enables the animation of the 7-seg display. If it is turned on, the display's source is the animation register (x9). Otherwise it displays the data stored in one of the processor's memories.

Note: Unless the processor has stopped executing (is in its `IDLE` state), the contents of its memories will not be clearly visible on the display.

Pinout

#	Input	Output	Bidirectional
0	Display on/off	segment a	Driver_ctrl[0] (I)
1	Most Significant Bits	segment b	Driver_ctrl[1] (I)
2	RS_addr[0]	segment c	Done executing (O)
3	RS_addr[1]	segment d	Serial clock (O)
4	RS_addr[2]	segment e	MISO (I)
5	RS_addr[3]	segment f	MOSI (O)
6	View select	segment g	Chip select (O)
7	Animation on/off	dot	Sync (O)

fft-4-tt [103]

- Author: Foivos Chaloftis
- Description: A simple FFT Calculator downscaled for deployment with the Tiny Tapeout 04 Physical PCB
- GitHub repository
- HDL project
- Mux address: 103
- Extra docs
- Clock: 1000 Hz
- External hardware: 2x buttons, Way to input 8-bit data, Way to display/read 8-bit data

How it works

This is a simplified Fast Fourier Transform implementation (*based on the radix-2 Cooley–Tukey algorithm*) that can be scaled-up to larger precision and more points. Designed for low complexity circuits requiring large DFT calculations, sacrificing speed. This specific implementation offers 4-point, 8-point, and 16-point versions of the Fourier Transform, while having the precision set to 4 bits.

For the first part, it integrates reverse bit ordering, placing data to their corresponding memory address as they are being input from the user. Afterwards, data, along with the weights, are fed through a single butterfly module (2-point DFT), responsible for all the calculations, controlled by the control unit which delegates the data reading/writing throughout each clock cycle. Once finished, the data output process begins. The FFT is calculated using signed fixed-point arithmetic. The decimal range is between -1 and 0.875. Any results bigger/smaller than the previous, will be capped at the maximum/minimum value possible.

How to test

Connect the proper I/O for inserting data/controlling the circuit, and displaying/reading output. Follow the steps as shown below:

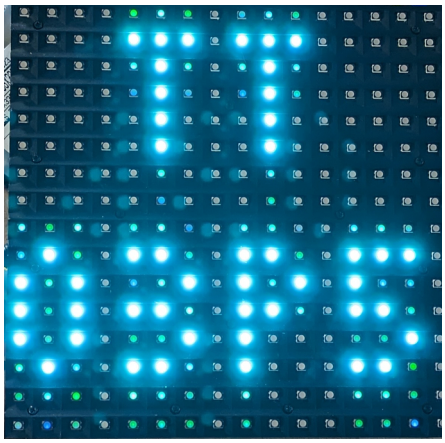
- **Step 1**: Reset the IC by momentarily enabling the rst signal.
- **Step 2**: Cycle through the 3 modes (0: 4-point, 1: 8-point, 2: 16-point FFT) shown on the 7-segment display using the mode_change pin, and select the mode you wish to use by using the enter pin.

- **Step 3**: Insert data in the specified format (Q1.3 real and Q1.3 imaginary), and use enter pin to input each point. After inserting all data, the FFT computation will begin.
- **Step 4**: Use the enter pin again to read the data from the output pins.
- **Step 5**: Once all data points are read, the display will show an F, indicating that the data reading is finished.
- **Step 6**: Use enter pin to repeat process form **Step 2**.

Pinout

#	Input	Output	Bidirectional
0	imaginary_in[0]	segment a/imaginary_out[0]	mode_change
1	imaginary_in[1]	segment b/imaginary_out[1]	enter
2	imaginary_in[2]	segment c/imaginary_out[2]	none
3	imaginary_in[3]	segment d/imaginary_out[3]	none
4	real_in[0]	segment e/real_out[0]	none
5	real_in[1]	segment f/real_out[1]	none
6	real_in[2]	segment g/real_out[2]	none
7	real_in[3]	dot/real_out[3]	none

LED Panel Driver [112]



- Author: Tom Keddie
- Description: Drives a 16x16 P10 LED panel
- GitHub repository
- HDL project
- Mux address: 112
- Extra docs
- Clock: 12000000 Hz
- External hardware: led panel, level converter to 5V logic

How it works

- The circuit updates a P10 16x16 LED display module
- It initially displays the string TT03P5
- It provides a 1.2Mbaud uart input to
  - paint pixels
  - erase pixels
  - clear the display
  - change the displayed colour
- Functionality is limited by resource availability
  - single colour at once
  - no double buffer, updates may have artifacts
- Mode pin to allow for 2 different clocking patterns

How to test

- Connect the display module as per the outputs
- Connect the uart
- Power on and see the TT03P5 text
- If the display is swapped by quadrant change the mode pin
- Use the script(s) in the software directory to control the display

Pinout

#	Input	Output	Bidirectional
0	uart	red0	red1
1	mode	blue0	blue1
2	none	b	green1
3	none	blank	none
4	none	green0	none
5	none	a	none
6	none	clk	none
7	none	latch	none

OSU Counter [113]

- Author: Mehmet Aksoy
- Description: flip flop counter
- GitHub repository
- Wokwi project
- Mux address: 113
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

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

Even digits [114]

- Author: Ibrahim Eskikurt
- Description: Even digits
- GitHub repository
- Wokwi project
- Mux address: 114
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

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

Traffic light [115]

- Author: Guvanch Gulmyradov
- Description: OSU RET training
- GitHub repository
- Wokwi project
- Mux address: 115
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

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

Tutorial4 [116]

- Author: Delwar
- Description: Tutorial4
- GitHub repository
- Wokwi project
- Mux address: 116
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

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

Grain-Flex-FPGA [117]

- Author: Rice Shelley
- Description: FPGA designed in SpinalHDL.
- GitHub repository
- HDL project
- Mux address: 117
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Grain-Flex-FPGA is a 1 CLB FPGA with 8 IO buffers. The single CLB has four 4 input LUTs each with an optional flipflop at the output. The CLB has four inputs and four outputs that can be mapped to any IO buffer. Each LUT input pin can be mapped to any CLB input or any other LUT's output including its own.

How to test

The FPGA is programmed with a simple scan chain. The SpinalHDL project has a few examples of creating a bit stream. (Hopefully VTR support coming soon)

Pinout

#	Input	Output	Bidirectional
0	Scan chain clock	Scan chain data out	FPGA IO Buffer
1	Scan chain active high reset	NA	FPGA IO Buffer
2	Scan chain enable	NA	FPGA IO Buffer
3	Scan chain data in (sampled on rising edge of scan chain clock)	NA	FPGA IO Buffer
4	NA	NA	FPGA IO Buffer
5	NA	NA	FPGA IO Buffer
6	NA	NA	FPGA IO Buffer
7	NA	dot	FPGA IO Buffer

BFCPU [118]

- Author: Michael Yenik
- Description: Hardware BF CPU
- GitHub repository
- HDL project
- Mux address: 118
- Extra docs
- Clock: 50000000 Hz
- External hardware: bus device (see above, probably RP2040 w/ fw)

How it works

A hardware CPU for the brainfuck esolang, with some BFISA extensions!

The program and data memory don't remotely fit onto the given area, so they are handled externally using a custom asynchronous bus protocol. The bus can perform certain types of transactions (read data, write data, read char from I/O, write char to I/O, read next program word, read previous program word). These correspond to reading/writing data memory, reading program memory (the program counter is implicitly kept outside the BFCPU since it only requests next/prev instructions), and doing I/O (BF . and , instructions).

The bus is controlled by the BFCPU, with the BFCPU setting the bus type and ctrl pins, then setting the rdy output pin. When the bus device implementing the data/program/IO sees a rising edge on rdy, it looks at the type/ctrl pins to know what to do. In order to prevent bus conflicts, the BFCPU does not drive the bus unless the bus en pin is set by the bus device.

This allows the bus device to see the rising rdy edge, get ready to read whatever the BFCPU wants to put on the bus, set bus en, read it, and then unset bus en. If the BFCPU is trying to read something, then the bus device can simply drive the bus to the requested value.

Once the bus device has either read what the BFCPU has to say or driven the bus, it sets the ack input to the BFCPU to allow the BFCPU to continue the transaction. The BFCPU will accept the ack by setting rdy low, the device must continue to set ack until rdy goes low. In the case that the bus device is driving the bus to a requested value, it must continue to drive the bus until rdy goes low.

When a bus transaction is initiated, the type of transaction the CPU is trying to perform is put onto the bus type pins

- 000 - read data

- 001 - write data
- 010 - read char
- 011 - write char
- 100 - read next program byte
- 101 - read prev program byte

Since the data being read may be at an arbitrary 15 bit address, and we don't have enough pins to easily make address and data lines, the address and data are multiplexed onto the bus. When reading or writing from data memory, the address will be written onto the bus one byte at a time, then the data to be read/written will be placed onto the bus. In order to coordinate with the device on the bus about which phase of the transaction it's in, we use the two bus ctrl pins

- 00 - lower byte of address (for data read/write)
- 01 - upper byte of address (for data read/write)
- 11 - data phase (for data read/write, IO, and program read)

The BFCPU also supports a simple extension allowing from 2 up to 14 consecutive $+$, -, $<$, and $>$ to be compressed into a-m, n-z, A-M, N-Z, respectively. So the following BF program '++++»»' can be compressed into 'cC'.

How to test

It needs a device paired with it that can read the bus signals and interpret the reads/writes correctly in order to operate. See the description above, as well as src/test.py in the github repo for an example. Hopefully there will also eventually be some RP2040 firmware in te repo to use with it!

Pinout

#	Input	Output	Bidirectional
0	bus en	bus rdy	bus bit 0
1	bus ack	bus ctrl bit 0	bus bit 1
2	unused	bus ctrl bit 1	bus bit 2
3	unused	bus type bit 0	bus bit 3
4	unused	bus type bit 1	bus bit 4
5	unused	bus type bit 2	bus bit 5
6	unused	halted	bus bit 6
7	unused	unused	bus bit 7

AI Decelerator [119]

- Author: machinaut
- Description: Systolic array for matrix multiplication
- GitHub repository
- HDL project
- Mux address: 119
- Extra docs
- Clock: 50000000 Hz
- External hardware:

How it works

Implements a 2x2 outer product, such that it can stream accumulate a product of a 2xN and Nx2 matrix.

It's pipelined to run 4-clock-cycle blocks, and has a single 4-stage floating point multiply-accumulate.

Chip diagram:

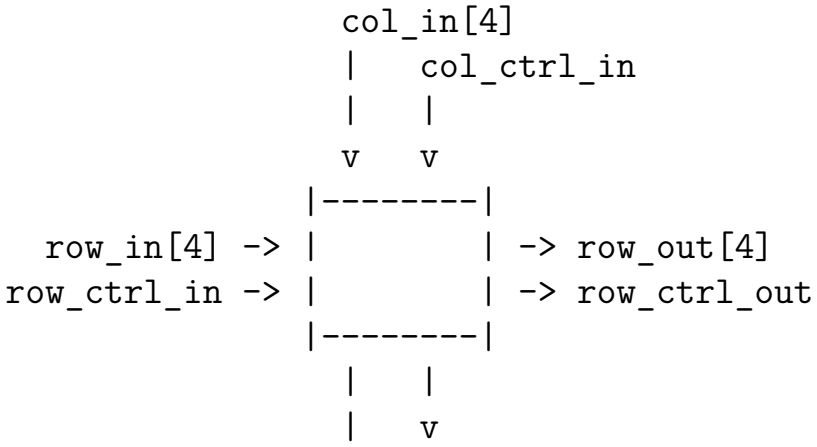
This is designed to be placed in a grid of tiles, with each element having 4+1 wires connecting each side.

Columns move data from north to south (inputs to outputs), and rows move data west to east (inputs to outputs).

The wires between each tile is 4 data wires and 1 control wire, and are all 1-directional.

Shared clock and reset are not shown, but are assumed to be connected to all tiles.

(See inputs/outputs at the bottom for pin assignment)



```
                    v   col_ctrl_out
                    col_out[4]
```

Block timing diagram:

We have 4 clock cycles per block, which advances at the positive edge of each clock.

We'll use "block" to denote the whole block, and "count" to count the cycles within a block.

Note the control inputs are always passed through exactly from inputs to outputs on the next block. However data outputs might be different from data inputs (see modes below).

Key: ci = col_in, ri = row_in, cc = col_ctrl, rc = row_ctrl, co = col_out, ro = row_out

```
       Block | 0       | 1       | 2       | ...
       Count | 0 1 2 3 | 0 1 2 3 | 0 1 2 3 | ...
-------------|---------|---------|---------|-----
      col_in | blk0_ci | blk1_ci | blk2_ci | ...
 col_ctrl_in | blk0_cc | blk1_cc | blk2_cc | ...
      row_in | blk0_ri | blk1_ri | blk2_ri | ...
 row_ctrl_in | blk0_rc | blk1_rc | blk2_rc | ...
-------------|---------|---------|---------|-----
     col_out |         | blk0_co | blk1_co | ...
col_ctrl_out |         | blk0_cc | blk1_cc | ...
     row_out |         | blk0_ro | blk1_ro | ...
row_ctrl_out |         | blk0_rc | blk1_rc | ...
```

Modes:

The shared control bits (from both row and column) decide what to do with the data in the block.

```
col_ctrl | row_ctrl | mode
---------|----------|------------
0000     | 0000     | passthrough
0WX0     | 1YZ0     | multiply-accumulate
1000     | 0100     | read-write accumulator 0
1100     | 0000     | read-write accumulator 1
```

Passthrough Mode will pass through data unchanged (current block data will be sent out as the next block).

Multiply-accumulate Mode will interpret the input data as FP8 vectors, and multiply them and accumulate them (see math below). This mode will also pass through the data unchanged (current block data will be sent out as the next block). W, X, Y, Z specify the FP8 format for the inputs (see below).

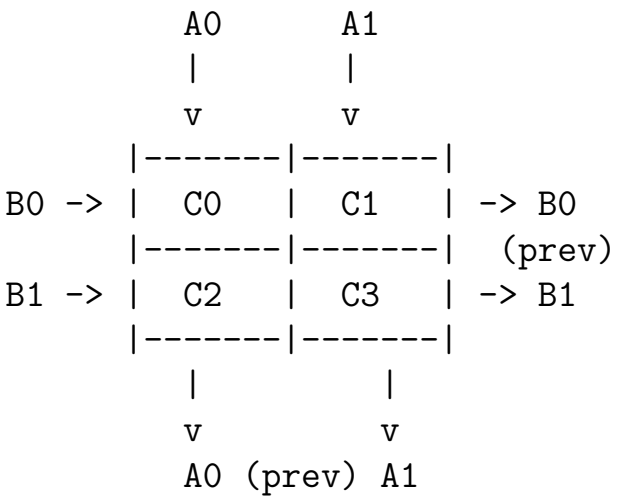
Read-write accumulator modes will shift input with the accumulator data and output data. This is used to simultaneously read-out the current accumulator state, and write-in the next accumulator state.

RW mode	cci	rci	ci	ri	co (next)	ro (next)
0	0100	1000	C0 (new)	C1 (new)	C0 (prev)	C1 (prev)
1	0000	1100	C2 (new)	C3 (new)	C2 (prev)	C3 (prev)

Multiply-Accumulate Math:

We interpret the column data as vector A0, A1, and the bits of the control input specify the FP8 format of A0, A1. Ditto for row data and B0, B1. The format bit is 0 if the value is E5M2 and 1 if the value is E4M4. See this paper for details on the FP8 formats: https://arxiv.org/pdf/2209.05433.pdf

The accumulators (C0, C1, C2, C3) are all FP16.



Systolic Tiling:

Each block controls what should happen in the following block.

Notionally, this could be used in a systolic tile pattern of N * M tiles, moving data along columns and rows. This hasn't been tested. Note that this still works with

reading and writing accumulators since all the values are shifted block by block along the columns and rows.

How to test

I have no idea what clock speeds are safe for this, so probably start out slow and work your way up until there are glitches. (This is like only 4 pipeline stages for a full multiply-accumulate, so it has some nasty propagation chains)

To compute A * B + C, where A is a 2xN matrix, B is a Nx2 matrix, and C is a 2x2 matrix, do the following: (A and B can be mixtures of FP8 formats, and C is FP16)

Use the read-write accumulator mode to write in C over two blocks (skip this if you want to start with C = 0)

```
Block 0:
    col_in: C_0,0 (FP16)
    row_in: C_0,1 (FP16)
    col_ctrl_in: b1000
    row_ctrl_in: b0100
Block 1:
    col_in: C_1,0 (FP16)
    row_in: C_1,1 (FP16)
    col_ctrl_in: b1100
    row_ctrl_in: b0000
```

Then use the multiply-accumulate mode for N blocks

```
Block K:
    col_in: A_0,K A_1,K (FP8, FP8)
    row_in: B_K,0 B_K,1 (FP8, FP8)
    col_ctrl_in: b0WX0 (where W, X are FP8 format bits for A0, A1)
    row_ctrl_in: b1YZ0 (where Y, Z are FP8 format bits for B0, B1)
```

Finally read out the result from the accumulator, just like you wrote it in

```
Block 0:  (Note we care about the outputs here)
    col_ctrl_in: b1000
    row_ctrl_in: b0100
    col_out: C_0,0 (FP16)
```

```
    row_out: C_0,1 (FP16)
Block 1:  (Note we care about the outputs here)
    col_ctrl_in: b1100
    row_ctrl_in: b0000
    col_out: C_1,0 (FP16)
    row_out: C_1,1 (FP16)
```

Example:

I have a 2xK matrix A, with values `A00` through `A1K`, and a Kx2 matrix B, with values `BK0` through `BK1`, and a 2x2 matrix C with values `C00` through `C11`. We want to compute `A * B + C = D` where D is a 2x2 matrix with values `D00` through `D11`.

The basic steps are

- Write in the initial value of C
- Stream in the values of A and B, and multiply-accumulate
- Read out the accumulated result, and call it D

For simplicity, we'll assume all the A and B values are E5M2 format, but remember they can be configured per-value.

C and D are both in FP16 format.

Remember that each block is four clock cycles, and each clock cycle 1/4 of the inputs and outputs are transmitted.

Block	col_in	row_in	cci	rci	col_out	row_out
0	C00	C01	1000	0100		
1	C10	C11	1100	0000		
2	A00A10	B00B01	0000	1000		
3	A01A11	B10B11	0000	1000		
...	...	...	...	...		
K + 2	A0KA1K	BK0BK1	0000	1000		
K + 3			1000	0100	D00	D01
K + 4			1100	0000	D10	D11

So we can compute this in just K + 4 blocks, where K is the inner size of the A and B matrix product.

(Also we can save 2 blocks if C is zero, since that's the reset value of the accumulators)

Pinout

#	Input	Output	Bidirectional
0	Row Data In 0	Row Data Out 0	Row Control Out
1	Row Data In 1	Row Data Out 1	Col Control Out
2	Row Data In 2	Row Data Out 2	Row Control In
3	Row Data In 3	Row Data Out 3	Col Control In
4	Column Data In 0	Column Data Out 0	Unused
5	Column Data In 1	Column Data Out 1	Unused
6	Column Data In 2	Column Data Out 2	Unused
7	Column Data In 3	Column Data Out 3	Unused

Tiny (3-bit) LFSR [160]

- Author: Thomas Klassert
- Description: Tiny (3-bit) LFSR
- GitHub repository
- Wokwi project
- Mux address: 160
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Pseudo-random number generator using a 3-bit linear-feedback shift register

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	clock	random bit 0	none
1	none	random bit 1	none
2	none	random bit 2	none
3	none	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

Pulsed Plasma Thruster (PPT) Controller [161]

- Author: Jurica Kundrata
- Description: Controller module which generates triggering pulses for a Pulse Plasma Thruster, configurable via I2C interface.
- GitHub repository
- HDL project
- Mux address: 161
- Extra docs
- Clock: 32768 Hz
- External hardware:

How it works

The controller generates a given number of pulses at a given frequency and pulse width. The controller is to be used to control a Pulsed Plasma Thruster, i.e. its HV driver. The parameters of the controller can be given using the I2C slave interface. The I2C address is defined by 0b101 and the lower nibble of the input port. The I2C register map is:

ADDR | XX | Description

0x00 | RW | Clock divider value for the controller

0x01 | RW | Lower byte of pulse period

0x02 | RW | Higher byte of pulse period (6 LSBs used)

0x03 | RW | Lower byte of pulse width

0x04 | RW | Higher byte of pulse width (6 LSBs used)

0x05 | RW | Total number of pulses

0x06 | – | N/A

0x07 | RW | Run command - initiate the pulse train

0x08 | RO | Number of pulses fired

0x09 | – | N/A

0x0A | RO | Done signal - active when the pulse train is finished

The controller can be run without configuration via I2C using the fifth bit of the input port (run_override signal).

Verified by iverilog simulations and FPGA prototyped on Xilinx Basys 3 board.

How to test

Controller is designed to be used with a 32.768 kHz clock. It can be used without configuration via I2C interface by pulling HIGH the run_override signal (ui_in[4]). After reset the controller will produce a pulse train with 0.25 Hz frequency, 1/32s width and 16 pulses. The pulses are shown on the 7-seg display as alternating between dash '-' and zero '0'. Also it will produce a 1 Hz divided clock at div_clock port, i.e. dot port of the output port (7-seg display).

Pinout

#	Input	Output	Bidirectional
0	I2C address bit 0	ON if pulse train HIGH	I2C SCL bus line
1	I2C address bit 1	ON if pulse train HIGH	I2C SDA bus line
2	I2C address bit 2	ON if pulse train HIGH	none
3	I2C address bit 3	ON if pulse train HIGH	none
4	run_override	ON if pulse train HIGH	none
5	none	ON if pulse train HIGH	none
6	none	OFF if pulse train HIGH	none
7	none	div_clk	none

SAP-1 CPU [162]

- Author: Jayraj Desai
- Description: Implementaion of Simple As Possible (SAP-1) CPU based on the book Digital Computer Electronics by Albert Paul Malvino and Jerald A. Brown
- GitHub repository
- HDL project
- Mux address: 162
- Extra docs
- Clock: 0 Hz
- External hardware: Clock generator and a switch to provide reset signal

How it works

This project is an implementation of a CPU called SAP-1 as referred in the book 'Digital Computer Electronics' by Albert Paul Malvino and Jerald A. Brown. Book's PDF is available online, a simple internet search will point you to the PDF of the book. Difference in my implementation versus the one mentioned in the book is that they have used shared bus architecture, which was possible due to use of TTL tristate buffers but in CMOS implementation I am not aware of a simple way to infer a tristate buffer in verilog so made some changes to adapt their architecture. Another important thing to notice is that even though this project has full implementation of LDA, ADD, SUB and OUT instructions, I have not implemented interface to an external memory, instead I have hard coded a 16 x 8 bit memory in the memory_16x8_rom.v which kind of simulates memory. Hence this CPU can run only one code.

This Project needs two input only clk and rst_n. I plan to provide input clock by generating it outside using a 555 timer and using a simple switch to provide rst_n signal.

How to test

As mentioned in how it works this project needs two inputs clk and rst_n. Once these signals are applied after few clock pulses you should see output of a fixed code at the output pins which can be viewed using set of 8 LEDs connected serially through resistors.
Expected output is binary 01 (This is output of first instruction that loads accumulator with value 01 which is stored at address 0x9 in memory) , 03 (This is output of second instruction which is add accumulator with value 02 which is stored at 0xA address in memory), 06 (This is output of third instruction which is add accumulator with value 03 which is stored at 0xB address in memory) and 02 (This is output of third

instruction which is subtract accumulator with value 04 which is stored at 0xC address in memory).

Pinout

#	Input	Output	Bidirectional
0	clk = input clock	uo_out[0]	none
1	rst_n = active low reset	uo_out[1]	none
2	none	uo_out[2]	none
3	none	uo_out[3]	none
4	none	uo_out[4]	none
5	none	uo_out[5]	none
6	none	uo_out[6]	none
7	none	uo_out[7]	none

Multi-channel pulse counter with serial output, v01a [163]

- Author: Adrian Novosel, Dinko Oletic
- Description: Counts number of digital pulses occuring within a time interval across four input channels, and periodically outputs the values out using serial output. Wokwi implementation.
- GitHub repository
- Wokwi project
- Mux address: 163
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Design consists of four 12-bit counters with overflow detection, a mm:ss real-time clock (RTC), a parallel-input-serial output (PISO) readout register, controlled by a readout state-machine. The counters store number of intermittently-occuring short digital input pulses, accumulated within the RTC's time-measurement interval 00:00 - 59:59, at each of the four input channels. Periodically, after every RTC overflow (1 h with assumed 1 Hz RTC input clock signal), the state-machine performs sequential serial readout of the RTC time and all channels, and resets all channel counters. Additionally, readout and individual channel reset is initiated by overflow at any of individual input channel counter. This an early work-in progress implementation of digital portion of a low-power sensor interface for readout of a multichannel acoustic emission detector, based on MEMS-array of piezoelectric microresonators for passive ultrasonic band-pass filtering: https://ieeexplore.ieee.org/document/9139151. Design is generally applicable for low-power wake-up sensor interfaces, acoustic event detection, non-destructive testing, particle-counters, or as a generic pulse-counting digital building block.

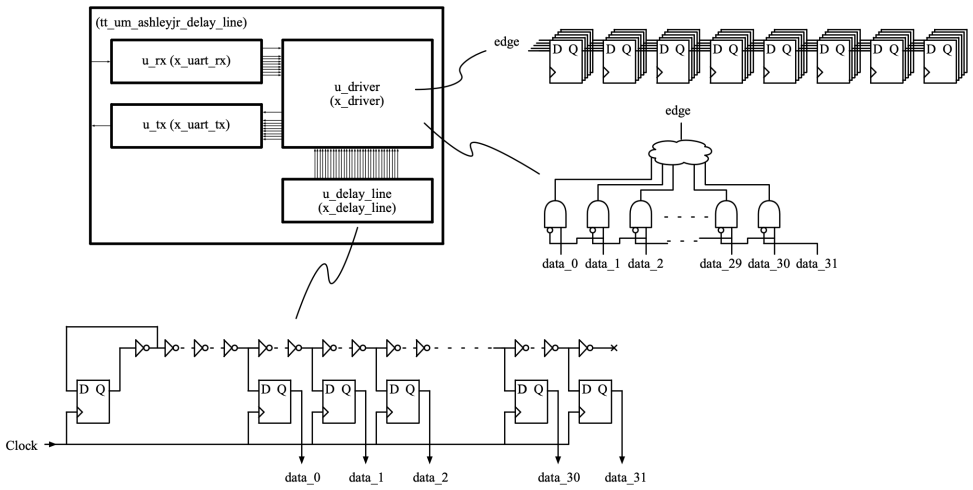
How to test

Input signals are short rising-edge digital pulses, connected to input pins "ch1", "ch2", "ch3", "ch4". Output data becomes ready for serial readout at the output pin "serial_out" when overflow is signalled via the output "ready" pin ovf_global. Output bits are serially clocked-out using the input pin "clk". Specifically, RTC overflow is signalled via output pin "ovf_RTC_out", and overflow at an individual channel via the pin "ovf_ch_out". The rest of output pins are used for debugging of the state-machine's internal states.

Pinout

#	Input	Output	Bidirectional
0	reset	serial_out	none
1	ch1	ovf_global	none
2	ch2	a0_out	none
3	ch3	a1_out	none
4	ch4	a2_out	none
5	clk	SL_out	none
6	RTC	ovf_RTC_out	none
7	none	ovf_ch_out	none

Delay Line [164]



- Author: Ashley J. Robinson
- Description: A simple delay line with instrumentation
- GitHub repository
- HDL project
- Mux address: 164
- Extra docs
- Clock: 50000000 Hz
- External hardware: FTDI Cable

How it works

- A delay line output changes based on time delay of different variables such as process, voltage and temperature.
- There are may different delay line architectures.
  - https://springerplus.springeropen.com/articles/10.1186/s40064-016-2090-z.
- This implementation is a simple tapped delay line.
- The continually changing data races through a chain of inverters.
- The chain is sampled at different stages to become a digital signal.
- An edge detection circuit is used find the rising which is then converted in a binary value.
- A bank of flops is used to sample 8 sequential rising edge values.

How to test

- https://github.com/ashleyjr/tt04-delay-line/blob/main/src/test/silicon_test.py
  - This python script uses pyserial to run a set of tests on the design
  - python3 silicon_test.py –help
- UART
  - The UART is the only interface to the design
    - 9600 baud
    - Least significant bit first
    - 1 Start bit
    - 8 Data bits
    - No parity bit
    - 1 Stop bit
    - Taken from https://github.com/ashleyjr/rtl-uart
  - The bottom 4 bits [3:0] of the UART frame make up the command
  - 4'h0: Shift In
  - Shift the top 4 bits [7:4] of the frame in to memory
    - The memory is shifted 4 places to the left
    - The data is placed in to the bottom 4 bits [3:0]
    - This command is to test the silicon and debug software
  - 4'h1: Shift Out
    - Shift the top 8 bits [39:32] of memory out to UART Tx
    - The memory is shifted 8 places to the left
  - 4'h2: Full Sample
    - Take a full 32-bit sample from the delay line and place in memory
    - The sample is placed in to the bottom 32 bits [31:0]
    - The shift out command may be used to read the sample
  - 4'h3: Scope

    - Take 8 samples from the delay line at a 25MHz sample rate
    - These sample use the edge detection logic to find the position of the rising edge
    - These samples are 5 bits wide
    - The samples are shifted in to the memory
        - Sample 0: [39:35]
        - Sample 1: [34:30]
        - Sample 2: [29:25]
        - Sample 3: [24:20]
        - Sample 4: [19:15]
        - Sample 5: [14:10]
        - Sample 6: [9:5]
        - Sample 7: [4:0]
    - The shift out command may be used to read the sample
- 4'h4 to 4'hF inclusive
    - Ignored

Pinout

#	Input	Output	Bidirectional
0	UART Rx	UART Tx	Tied Low
1	none	Tied Low	Tied Low
2	none	Tied Low	Tied Low
3	none	Tied Low	Tied Low
4	none	Tied Low	Tied Low
5	none	Tied Low	Tied Low
6	none	Tied Low	Tied Low
7	none	Tied Low	Tied Low

Simple Piano [165]

- Author: Sarthak Raheja and Bittu N
- Description: An eight octave twelve key piano with two inbuilt songs. The design can be customized and incorporated as per user requirement in multiple use cases.
- GitHub repository
- HDL project
- Mux address: 165
- Extra docs
- Clock: 1000000 Hz
- External hardware: 12 momentary switches, 4 toggle switches, an led bar and a speaker

How it works

Description: Twelve Independent Tonal Frequencies and Two Pre-defined Songs

Introduction: In this ASIC, we generate twelve independent tonal frequencies and two predefined songs based on RTTTL. This design features sixteen input switches for selecting one of two modes, one of eight octaves, any one of twelve notes. In the demo mode one out of two keys can be used to select the song.

Tone Generation: Clock generation: The design makes use of a 1MHz clock to generate tonal frequencies/notes. A tone gen module generates a specific frequency, which is configured based on user input. Song demo mode: The design includes a note sequencer that steps through one of two predefined songs. The songs are stored as a list of note-duration pairs. The sequence is generated from an RTTTL description of the songs.

Output Channels: The ASIC generates one single ended square wave output for the notes and the remaining outputs are used to drive an LED bar visualizer.

Clock, Enable and Reset: Clock Input: The ASIC requires an external clock signal of 1MHz to synchronize its operations. This clock signal ensures that all generated tones and sentences are coherent and correctly timed. This allows us a frequency resolution of less than 0.5%. Enable: The design only produces output when the enable pin is held high. Reset Mechanism: The chip features a reset input, allowing you to reset its internal state and restart the frequency generation process if needed.

How to test

The design requires a 1MHz clock. The design needs to be reset before using and enable must be set to 1 for output. To test the piano function, set the mode toggle to 0 and press any one of the twelve keys just as you would on a piano. To test the demo function, set the mode toggle to 1 and press either the C or C# keys for a short song. The audio output is single ended and must be fed to an amplifier.

Pinout

#	Input	Output	Bidirectional
0	Note E	LED bar [6]	Mode piano = 0 & demo = 1
1	Note F	LED bar [5]	Octave [3]
2	Note F#	LED bar [4]	Octave [2]
3	Note G	LED bar [3]	Octave [1]
4	Note G#	LED bar [2]	Note C
5	Note A	LED bar [1]	Note C#
6	Note A#	LED bar [0]	Note D
7	Note B	Audio out	Note D#

Ripple-Carry Adder [166]

- Author: Yannick Reiß
- Description: Add two bytes.
- GitHub repository
- Wokwi project
- Mux address: 166
- Extra docs
- Clock: 0 Hz
- External hardware: 16 input devices, 8 output devices

How it works

Combination of one half adder and 7 full adder, directly connected.

How to test

Connect switches to set the input bytes. Use LEDs or some other kind of output device to view the sum of the two input bytes.

Pinout

#	Input	Output	Bidirectional
0	MSB Byte 1	MSB Result	MSB Byte 2
1	6	6	6
2	5	5	5
3	4	4	4
4	3	3	3
5	2	2	2
6	1	1	1
7	LSB Byte 1	LSB Result	LSB Byte 2

Led Multiplexer Display [167]

- Author: Baciu Florin-George | BFG-e
- Description: Stores 4 characters and displays them on a 16x4 led matrix
- GitHub repository
- HDL project
- Mux address: 167
- Extra docs
- Clock: 100 Hz
- External hardware: 16x4 led matrix

How it works

If the load is high, the design will load the specified hex char(input data pins) at the specified location in memory (input char position), therwise the design will go trough the display columns and represent the chars using the internal character map. The design should be reseted before use

How to test

Use the input data to add chars to the internal memory.

Pinout

#	Input	Output	Bidirectional
0	input data 0	active column number 0	not used
1	input data 1	active column number 1	not used
2	input data 2	active column number 2	not used
3	input data 3	active column number 3	not used
4	input char position 0	line output 0	not used
5	input char position 1	line output 1	not used
6	load	line output 2	not used
7	not used	line output 3	not used

LED Matrix Driver [176]

- Author: Michael Bella
- Description: Serial data input 8x8 common anode led matrix driver.
- GitHub repository
- HDL project
- Mux address: 176
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

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

Puts the bottom 8 bits of the counter on the bidirectional outputs.

With all the inputs set to 0, the internal 24 bit compare is set to 10,000,000. This means the counter will increment by one each second.

If any inputs are non zero, then the input will be used as an bits 11 to 18 of the 24 bit compare register. Example: setting the inputs to 00010000 will program 16384 into the compare register. With a 10MHz clock the counter will increment ~610 times per second.

How to test

After reset, the counter should increase by one every second with a 10MHz input clock. Experiment by changing the inputs to change the counting speed.

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

#	Input	Output	Bidirectional
6	compare bit 17	segment g	second counter bit 6
7	compare bit 18	dot	second counter bit 7

8-bit FIFO with depth 16. [177]

- Author: Steve Jenson
- Description: 8-bit FIFO with depth 16 and FWFT read operation
- GitHub repository
- HDL project
- Mux address: 177
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

A FIFO queue is a first-in, first-out digital device that allows two parties to communicate with a channel of limited size by following specific rules: one party writes, the other reads. The first thing written will be the first thing read. Reading an empty queue is disallowed and writing to a full queue is disallowed. Empty and full status can be checked via the proper status pin before use. In psuedo-code, two parties can communicate with this FIFO as follows:

Party A

```
while !full:
  write_entry(item)
```

Party B

```
while !empty:
  item = read_entry()
```

The queue works in First-Word Fall-Through mode meaning that the top item is always available on the read bus even if you haven't set `read_request` high. If you want to see the next item in the queue on your next read, be sure to set `read_request` high.

`almost_full` and `almost_empty` signals exist so you can batch reads and writes. Instead of checking for `full` or `empty` on each read or write you can instead check `almost_full` or `almost_empty` and batch read or writes based on how many slots are available. For this design taped out in TinyTapeout 4, almost_full means 12 of 16 slots have been used and almost_empty means that 12 of 16 slots are free.

How to test

Write an item, read an item. Check the status bits. See the unit tests for more ideas!

Pinout

#	Input	Output	Bidirectional
0	item[0]	item[0]	empty (read-only)
1	item[1]	item[1]	full (read-only)
2	item[2]	item[2]	underflow (read-only)
3	item[3]	item[3]	overflow (read-only)
4	item[4]	item[4]	almost_empty (read-only)
5	item[5]	item[5]	almost_full (read-only)
6	item[6]	item[6]	write_enable (set this high to write a value)
7	item[7]	item[7]	read_enable (set this high to read the latest entry from the FIFO)

Pong [178]

- Author: Robbert-Jan de Jager
- Description: This is a small pong game implemented in HDL. It uses a VGA connector to output the video signal. The game is controlled with 3 buttons per player. The left button moves the paddle to the left, the right button moves the paddle to the right and the action button starts the game. The game is over when one player is out of lives.
- GitHub repository
- HDL project
- Mux address: 178
- Extra docs
- Clock: 25175000 Hz
- External hardware: 3x 2bit DAC for the red, green and blue video signals. VGA connector. 3 buttons.

How it works

Basic operation The core of the design is the vga_timing module. This module generates all the required timing signals. Some of these signals like hsync and vsync are used to generate the video signal, while others like the horizontal and vertical position are used to generate the graphics. The horizontal and vertical sync signals are also used for the game logic.

Before outputting the video signal the video mux selects the correct input color to display. It does so based on the highest priority component that wants to output a color.

We have multiple painter modules. These generate from the current game state and the current horizontal and vertical position the correct color to display. Ideally the painters would not contribute to the game logic, but for optimization reasons they do.

While drawing a frame the game logic keeps track of collisions. It does so by checking if multiple painters want to draw at the same position. If so it will latch a collision, which will be processed after drawing the frame.

At the end of each frame the game logic will calculate the next ball position, taking collisions into account. The collision with the paddle is special. To have an entertaining game that does not play the same every time the ball will bounce off the paddle at a different angle depending on where it hits. This is done by splitting the paddle into multiple segments and checking for collisions with these segments. The game logic will look at the segment when a paddle collision was registered. An exception to the end of frame gamestate update is the breaking of blocks. It would require too much memory to keep the updated state for the next frame. Instead we will update the row of blocks that was just finished drawing.

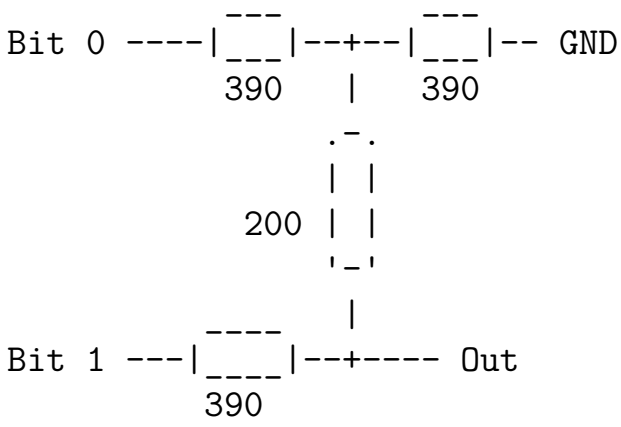
How to test

This is a small breakout game implemented in HDL. It uses a VGA connector to output the video signal. The game is controlled with 3 buttons per player. The left button moves the paddle to the left, the right button moves the paddle to the right and the action button starts the game. The game is over when all blocks are destroyed or when the ball hits the bottom of the screen.

Required hardware This project requires a VGA monitor and a VGA DAC. An easy way to create the VGA DAC is to use 3 2-bit R2R DACs. The 2-bit R2R DACs can be created using 2 resistors per bit. The resistors should be 200Ohm and 390Ohm. For the 3.3V power supply.

What has not been verified is the current sourcing capability of the ASIC, If it can not at least source 10mA through each pin and 30mA through the power supply pins you should add a buffer before the DAC.

The VGA DAC should be connected as follows:



Every color should have an identical copy of this DAC. The red DAC should be connected to the red VGA pin, the green DAC to the green VGA pin and the blue DAC to the blue VGA pin. The outputs of the DACs should be connected to the VGA connector. The HSync and VSync pins should also be connected to the VGA connector. The following connections need to be made to the VGA connector:

- Red DAC output to VGA connector pin 1
- Green DAC output to VGA connector pin 2
- Blue DAC output to VGA connector pin 3
- HSync to VGA connector pin 13
- VSync to VGA connector pin 14
- GND to VGA connector pin 5, 6, 7, 8

Board configuration The ASIC requires an input clock of 25.175MHz. The 7-Segment display is not used.

Pinout

#	Input	Output	Bidirectional
0	none	HSync	none
1	none	VSync	HBlank
2	Button P2 left	Red output bit 0	VBlank
3	Button P2 right	Red output bit 1	sound output. Connect to a speaker with amplifier.
4	Button P2 action	Green output bit 0	none
5	Button P1 left	Green output bit 1	none
6	Button P1 right	Blue output bit 0	none
7	Button P1 action	Blue output bit 1	none

8 panel display"" [179]

- Author: Jimmy Hartford
- Description: 4 different patterns displaying 8 panels on the 7-seg display
- GitHub repository
- Wokwi project
- Mux address: 179
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

1- figure 8 2- CHS 3- CUSHIng 4- roboticS

How to test

Explain how to test your project

Pinout

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

Traffic Light [180]

- Author: Courtney
- Description: Lights
- GitHub repository
- Wokwi project
- Mux address: 180
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

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

Model Railway turntable polarity controller [181]

- Author: Joop aan den Toorn
- Description: A controller that automatically switches the polarity of DC-type turntables
- GitHub repository
- Wokwi project
- Mux address: 181
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

A controller that automatically switches the polarity of DC-type turntables, to prevent short-circuits when rotating the turntable along DC-powered tracks. Every track that connects the turntable to the main tracks must include a short, isolated 'sensing track' between the normal tracks and the turntable. When the turntable rotates and makes contact with any of the tracks, it powers the sensing element. If the polarity does not match that of the main tracks, the turntable polarity controller will invert it.

This controller is designed to work with a Fleischmann 6152 turntable, using the connecting elements 6153 as sensing elements. The sensing elements must be isolated from the main tracks.

The polarity controller assumes that every sensing element has two tracks. Both are connected with a high-impedance resistor to the main track they respectively connect to. e.g.

```
---<MAIN L>---[ISOLATOR]---<SENS L>---[CONTACT]---<TURNTABLE L>---
    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
---<MAIN R>---[ISOLATOR]---<SENS R>---[CONTACT]---<TURNTABLE R>---


                          ___
MAIN TRACK R/L    -------|    \mbox{\textbackslash}
                    |    |     \mbox{\textbackslash}
                  High R   | XOR  |-----Polarity switch signal
                    |    |    /
SENSING TRACK R/L--------|___/
```

- Every track and corresponding sensing element form two inputs for an external XOR port. when polarity is not equal, the XOR output will go from 0 -> 1,

which is fed into the polarity controller [IN0-IN7, D1-D4]. Any of the XOR ports can command a change in polarity.
- The tracks are suitably connected through resistors and (zener) diodes to the XOR ports, to convert to digital signals and prevent an overvoltage on its inputs
- If the polarity controller receives a high input for prolonged periods of time, it assumes an error state where the turntable is not powered. To this end, connect [OUT6], the change-polarity control signal through a suitable RC delay to [D0] to trigger an error.
- The turntable polarity controller controls a full-bridge driver on [OUT2] and [OUT3] that connects the turntable to the main tracks.
- In the original design, the polarity controller uses digital control signals to control the full-bridge driver. The setup uses N-channel mosfets, where the gate voltage supply is generated by boosting the track voltage.
- XOR input circuitry, XOR ports, full bridge driver, mosfets and gate-voltage supply are all external to the polarity controller, but must be implemented if this design is to be used.
- The clock signal can be generated by the controller. Connect IO port [D6] and [D7] using an RC network to tune the frequency.
- The controller assumes a reset signal is always activated before the controller is used. An RC network to VCC can keep the reset signal active while powering up. Make sure to connect a switch that can ground the reset input to be able to activate the controller after an error.

POLARITY CONTROLLER
The inputs that can trigger a polarity change are connected through OR ports, hence every input can trigger a polarity change.

The trigger is used to drive a binary counter that counts down from 3..0 on the clock signal, which in turn generates signals for the full bridge driver, which is controlled by OUT2 and OUT3. To avoid shoot-through while switching in the FETs that connect tracks to either polarity, the outputs are only enabled on counts 1 and 3, using counts 0 and 2 to turn all FETS off. The counter generates a trigger when at value 0 and 2, so as to always end up in count 1 or 3. For miscellaneous purposes, the counter signal is provided on [OUT0] and [OUT1].
The counter trigger signal is provided on output OUT6, which can be connected through an RC delay on D0 to trigger an error when the trigger is active for an unusually long period of time. This may happen in case of short circuits at the tracks or other electrical issues. All driver outputs are disabled while an error is active.

The reset signal removes errors and disables the counter and bridge driver outputs.

An inverter network can be used to generate a clock by the IC. To this end, connect D6 and D7 through an RC network. The delay will tune the clock frequency.

How to test

Always reset the controller before using it. After a reset, provide a HIGH polarity switch trigger signal on IN0..IN7 or D1..D4 and provide a clock signal on CLK. The counter output must count down between 3..0 on the clock frequency while the trigger signal is HIGH. Also, OUT4 and OUT6 must be HIGH while the trigger is provided.

Remove the trigger signal. OUT4 must immediately output a LOW.
The counter must now stop at either 1 or 3 but never at 0 or 2. OUT6 will remain HIGH while counting, but must be LOW when the counter has stopped.

If a particular track polarity is A and its inverse is B, the following conditions must be met:

```
COUNT = 3, polarity A is active. OUT2 = HIGH, OUT3 = LOW
COUNT = 2, tracks are disabled.  OUT2 = LOW,  OUT3 = LOW
COUNT = 1, polarity B is active. OUT2 = LOW,  OUT3 = HIGH
COUNT = 0, tracks are disabled.  OUT2 = LOW,  OUT3 = LOW
```

Outputs OUT0..OUT3 must be LOW when a LOW->HIGH pulse is provided on D0, triggering an error which is indicated by a HIGH on OUT7.
Apply a reset signal to enable the outputs again, and ensure OUT7 is LOW.

Connect D6 and D7 through an RC network. Verify a self-oscillation is observed at output D7

Pinout

#	Input	Output	Bidirectional
0	IN0: trigger signal to change polarity	OUT0: counter signal	D0: error input. Use to activate an error and disable outputs.
1	IN1: trigger signal to change polarity	OUT1: counter signal	D1: trigger signal to change polarity
2	IN2: trigger signal to change polarity	OUT2: full bridge control signal for polarity A	D2: trigger signal to change polarity
3	IN3: trigger signal to change polarity	OUT3: full bridge control signal for polarity B	D3: trigger signal to change polarity

#	Input	Output	Bidirectional
4	IN4: trigger signal to change polarity	OUT4: indicates a polarity switch trigger is active on one of the inputs	D4: trigger signal to change polarity
5	IN5: trigger signal to change polarity	OUT5: VCC	D5: not used
6	IN6: trigger signal to change polarity	OUT6: counter enabled signal	D6: oscillator input
7	IN7: trigger signal to change polarity	OUT7: active error signal	D7: oscillator output

Customizable UART string tx [182]

- Author: Tiny Tapeout 02 (J. Rosenthal)
- Description: Customizable UART Transmitter Supports sending multiple ASCII characters over UART. Each column of flip flops stores a single ASCII character. To modify a character, change bits 6-0 by modifying whether the respective multiplex is set to VCC or GND.

To add characters, copy and paste a column. Connect the output of the new column (Q port of the upper-most D-flip flop) to the input of the stage to the left (bottom-left most multiplexer port). Remember to connect the multiplexer select signal and the clock to the new column as well.

Lastly, delete the the output of the first column (Q port of the upper-most D-flip flop) and create a new connection to the to the input of the new stage you've added (bottom-left most multiplexer port).

- GitHub repository
- Wokwi project
- Mux address: 182
- Extra docs
- Clock: 1 Hz
- External hardware:

How it works

TODO:FIXME

How to test

To begin transmission

1. Set the Arduino serial baud rate `Serial.begin(&lt;baud rate&gt;);` in the *.ino file to 300
2. Set the Wokwi clock frequency `&quot;attrs&quot;: { &quot;frequency&quot;: &quot;300&quot; }` in the diagram.json to 300 as well
3. Set the slide switch to the clock
4. Set SW7 to OFF ("Load")
5. Set SW8 to ON ("Output Enable")
6. Set SW7 to ON ("TX")

If there's no output from the Wokwi Arduino serial monitor, try toggling SW7 OFF and ON again.

Congratulations! You should now see your customized letters being output!

Note that garbage characters may be printed upon initialization.

Pinout

#	Input	Output	Bidirectional
0	N/A	segment a	none
1	N/A	segment b	none
2	N/A	segment c	none
3	N/A	segment d	none
4	N/A	segment e	none
5	N/A	segment f	none
6	load/tx	segment g	none
7	output enable	N/A	none

7-Seg 'Tiny Tapeout' Display [183]

- Author: Tiny Tapeout 02 (J. Rosenthal)
- Description: This circuit will output a string of characters ('tiny tapeout') to the 7-segment display.
- GitHub repository
- Wokwi project
- Mux address: 183
- Extra docs
- Clock: 1 Hz
- External hardware:

How it works

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

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	N/A	segment a	none
1	N/A	segment b	none
2	N/A	segment c	none
3	Clock Disable (Test Mode)	segment d	none
4	Test Logic A	segment e	none
5	Test Logic B	segment f	none
6	Test Logic C	segment g	none

#	Input	Output	Bidirectional
7	Test Logic D	N/A	none

UART character tx [192]

- Author: Tiny Tapeout 02 (J. Rosenthal)
- Description: This circuit will output a string of characters ('tiny tapeout') to the uart.
- GitHub repository
- Wokwi project
- Mux address: 192
- Extra docs
- Clock: 1 Hz
- External hardware:

How it works

Supports ASCII characters from 0x40 (@) to 0x5F (_), including capital letters from the latin alphabet.

How to test

Example Sending 'A'

1. Set the Arduino serial baud rate `Serial.begin(&lt;baud rate&gt;);` in the *.ino file to 300
2. Set the Wokwi clock frequency `&quot;attrs&quot;: { &quot;frequency&quot;: &quot;300&quot; }` in the diagram.json to 300 as well
3. Set SW7 to OFF ("Load")
4. Set SW2 to ON and SW3-6 to OFF
5. Set SW7 to ON ("TX")
6. Set SW8 to ON ("Output Enable") If there's no output from the Wokwi Arduino serial monitor, try toggling SW7 OFF and ON again. Congratulations! You should now see the letter being output in the Wokwi Arduino Serial monitor at the bottom of the simulation. Note that garbage characters may be printed upon initialization.

Pinout

#	Input	Output	Bidirectional
0	n/a	segment a	none
1	bit 0	segment b	none
2	bit 1	segment c	none
3	bit 2	segment d	none
4	bit 3	segment e	none
5	bit 4	segment f	none
6	load/tx	segment g	none
7	output enable	N/A	none

Padlock [193]

- Author: Tiny Tapeout 02 (J. Rosenthal)
- Description: Set a code for your precious safe
- GitHub repository
- Wokwi project
- Mux address: 193
- Extra docs
- Clock: Hz
- External hardware:

How it works

Set a code for your precious safe! **Controls**

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

How to test

Press the green button in the top left of the pane to begin the simulation. Set your desired code using Switches 3 to 5. Once you've done so, toggle Switch 8 to ON then back OFF–the safe is now set! Turn ON Switch 2, and press the push button. The red LED labeled "Locked" should turn on and the seven segment display should show "L" (for locked). Next turn OFF Switch 2 to begin entering codes.

Pinout

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

8bits Counter by AI [194]

- Author: Noritsuna Imamura
- Description: This verilog code is generated by LLaMa2 on PC.
- GitHub repository
- HDL project
- Mux address: 194
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

This is a simple 8bits counter. This was generated automatically by an Edge AI. Its EdgeAI environment is that of https://github.com/noritsuna/Edge_Circuit_Designer .

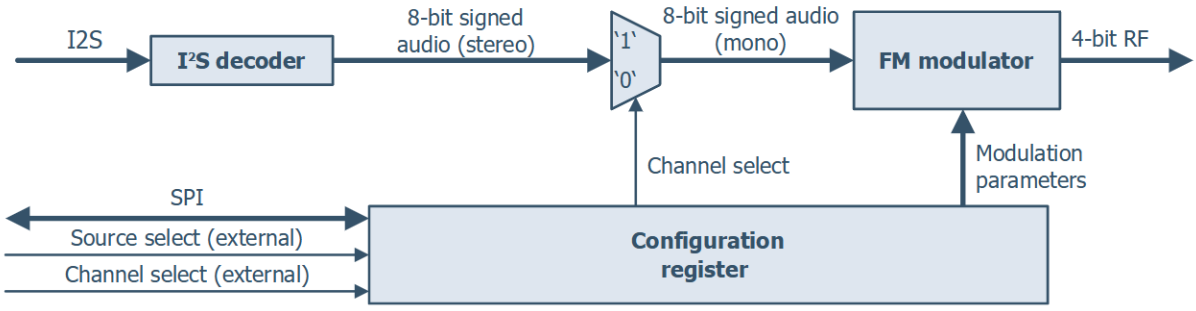
How to test

Check reset, ena, counter function.

Pinout

#	Input	Output	Bidirectional
0	none	counter bit 0	none
1	none	counter bit 1	none
2	none	counter bit 2	none
3	none	counter bit 3	none
4	none	counter bit 4	none
5	none	counter bit 5	none
6	none	counter bit 6	none
7	none	counter bit 7	none

FM Transmitter [195]



- Author: Jan Kral (jan.kral@vut.cz), Ondrej Kolar (ondrej.kolar@vut.cz)
- Description: FM transmitter with I2S input
- GitHub repository
- HDL project
- Mux address: 195
- Extra docs
- Clock: 50000000 Hz
- External hardware: 4-bit R-2R DAC, I2S source, SPI (optional)

How it works

Our design takes an audio signal and modulates it to a higher carrier frequency, using *FM modulation*. The modulator in our design is based on a *numerically controlled oscillator* (NCO) with several modifications.

The *frequency control word*, which increments the *phase accumulator*, is being added with the audio signal. This results in the phase increments proportional to the current audio sample level. The variation directly determines the actual shift of the output signal frequency. For the conversion of phase to a harmonic signal (sine wave) NCOs usually use look-up tables or *CORDIC algorithm*. However, both of these methods are resource-heavy, therefore the design adopts a very rough, piecewise linearized approximation of the sine function. The main upside of this approach is the lightweight implementation, which utilizes only simple bit-shifting and addition operations.

Since the output digital-to-analog converter suggested below is not followed by a *reconstruction filter*, the output signal will not be present only on a single frequency but also on several higher ones, sometimes called *mirrors* (as they appear on frequencies mirrored by the sampling frequency and its multiples). Thanks to this, it is possible to get the signal in the range of FM broadcast band, even with the sampling frequency lower than the carrier frequency.

How to test

Disclaimer! Our design is not intended for real *on air* use. Any signals generated by our design are far from ideal and require proper filtering. Improper use will most probably violate your local regulations. Use only at your own risk!

For testing the design you need to provide an audio source using the I2S bus interface. You can use for example Raspberry Pi. For the output, you need to build a DAC. A simple *R–2R resistor ladder network*) should be enough for testing. The schematic is provided in our GitHub repository.

Pinout

#	Input	Output	Bidirectional
0	i2s_clk	dac[0] (LSB)	none
1	i2s_din	dac[1]	none
2	i2s_ws	dac[2]	none
3	i2s_ws_align_pin	dac[3] (MSB)	none
4	audio_chan_sel_pin	none	spi_clk (in)
5	multiply_sel_pin	none	spi_csn (in)
6	dith_disable_pin	none	spi_mosi (in)
7	none	none	spi_miso (inout)

Test 4x4 memory [196]

- Author: Marchand Nicolas
- Description: A 4x4 memory adapted from :https://www.researchgate.net/figure/Structure-of-SRAM-Cell-The-design-of-SRAM-usually-involves-edge-triggered-flip-flops-The_fig3_324963843
- GitHub repository
- Wokwi project
- Mux address: 196
- Extra docs
- Clock: 1 Hz
- External hardware: the switches and the 7-segment can be enough - either

How it works

it uses 16 flip flop logic to create the memory of 4 lines of 4bits

2 switches controle the lines, 4 switches sets the bits of a line, 2 switches setup the read and chip select (CS to be 1 to work), 4 outputs show the inverted value of the stored bit for the first line of 4 bits 4 outputs show the actual 4 bits of the selected line (updated by switching CS or RD)

truth table operation | CS | RD | Line1 | Line2 | In3 | In2 | In1 | In0 | Out3 | Out2 | Out1 | Out0 |

No operation | 0 | X | X | X | X | X | 0 | 0 | X | X | X | X |

Write operation | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | X | X | X | X | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | X | X | X | X |
1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | X | X | X | X |
1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | X | X | X | X |

Read operation | 1 | 1 | 0 | 0 | X | X | X | X | 1 | 0 | 0 | 0 | | 1 | 1 | 0 | 1 | X | X | X | X | 0 | 1 | 0 | 0 | | 1 | 1 | 1 | 0 | X | X | X | X | 0 | 0 | 1 | 0 | | 1 | 1 | 1 | 1 | X | X | X | X | 0 | 0 | 0 | 1 |

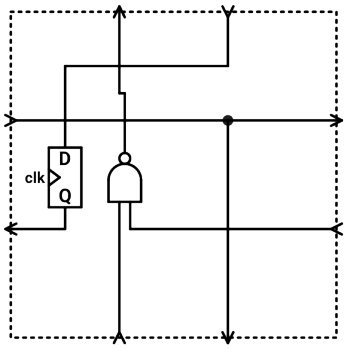
How to test

Setting the input switch to on should store the data and turn the corresponding LED of the 7-segment ON or off regarding to the stored value.

Pinout

#	Input	Output	Bidirectional
0	in 0 - updates the value of bit0 of the selected line with in4 and in5	out 0 - segment a - value of bit0 of the selected line with in4 and in5	none
1	in 1 - updates the value of bit1 of the selected line with in4 and in5	out 1 - segment b - value of bit1 of the selected line with in4 and in5	none
2	in 2 - updates the value of bit2 of the selected line with in4 and in5	out 2 - segment c - value of bit2 of the selected line with in4 and in5	none
3	in 3 - updates the value of bit3 of the selected line with in4 and in5	out 3 - segment d - value of bit3 of the selected line with in4 and in5	none
4	in 4 - selects the line with in5	out 4 - segment e - control of bit0 of the first line (value Q-)	none
5	in 5 - selects the line with in4	out 5 - segment f - control of bit1 of the first line (value Q-)	none
6	clk/step push-button to select write or read operation or can be automated on the clock	out 6 - segment g - control of bit2 of the first line (value Q-)	none
7	in 7 - chip select, allways ON (1) to wrok	out 7 - dot - control of bit3 of the first line (value Q-)	none

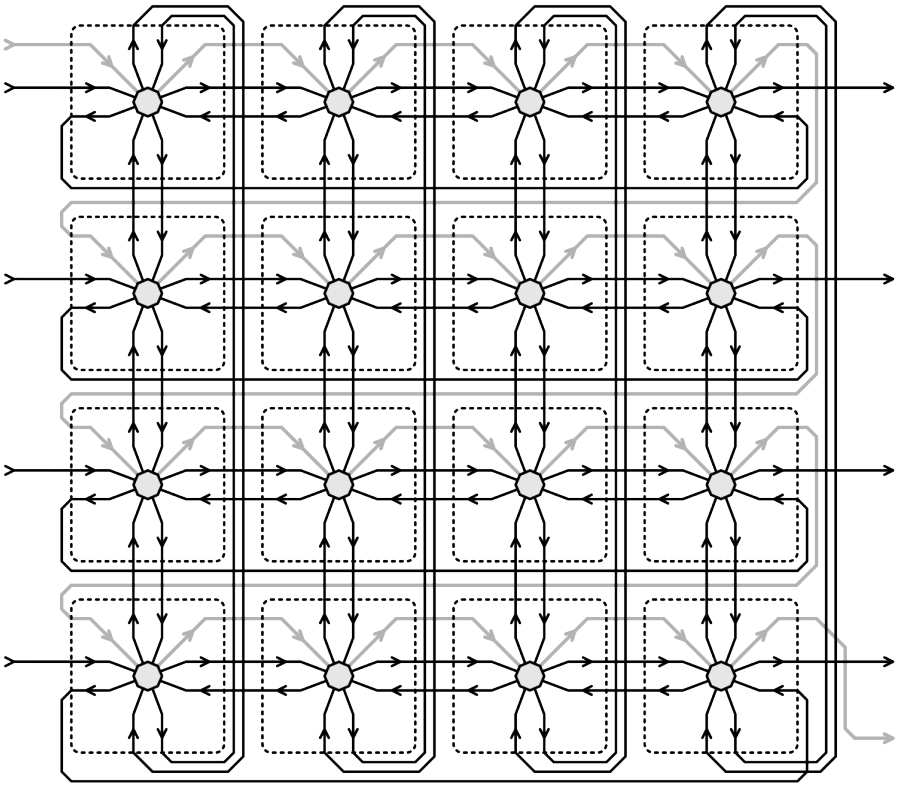
ROTFPGA v2 [197]



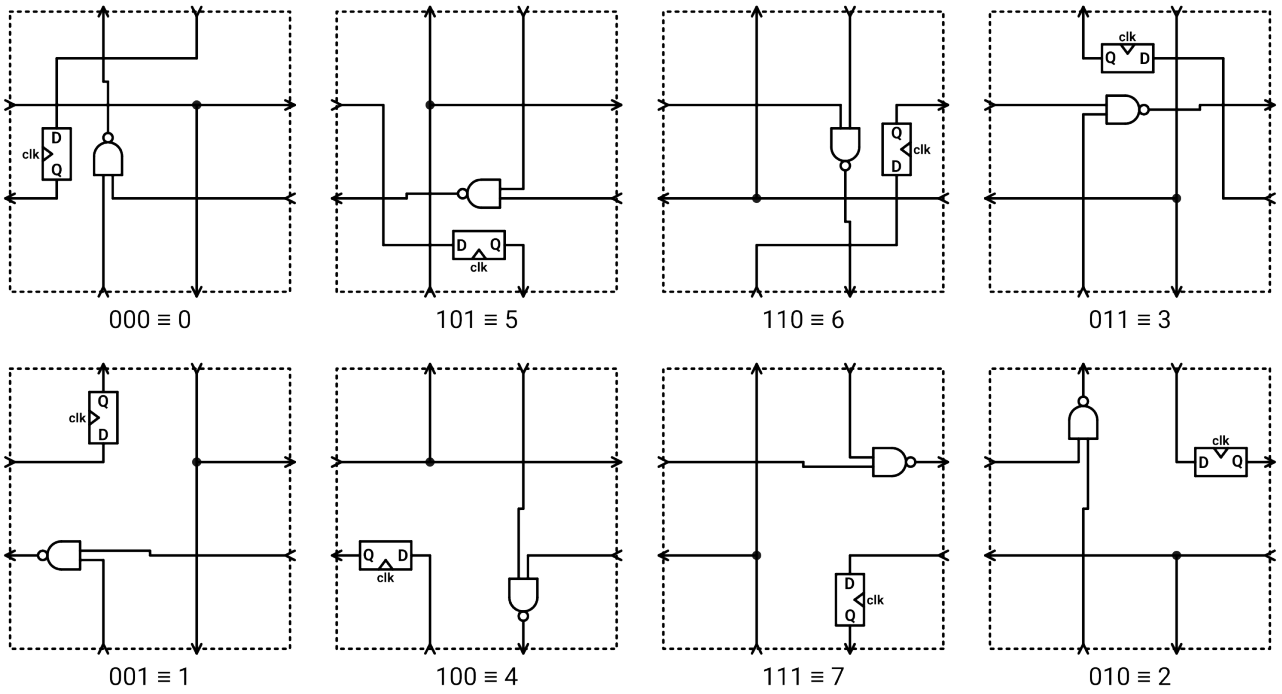
(a) single tile

00		11		00		11	
	10		01		10		01
11		00		11		00	
	01		10		01		10
00		11		00		11	
	10		01		10		01
11		00		11		00	
	01		10		01		10

(b) loop breaker classes



(c) grid model



(d) rotations and reflections

- Author: htfab
- Description: A reconfigurable logic circuit made of identical rotatable tiles

- GitHub repository
- HDL project
- Mux address: 197
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

ROTFPGA v2 is a reconfigurable logic circuit built from identical copies of the tile in Figure *(a)* containing a NAND gate, a D flip-flop and a buffer, with each tile individually rotated or reflected as described by the FPGA configuration. It is a 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 as shown in Figure *(d)*. (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.

Figure *(c)* shows a 4×4 model of the tile grid. 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 (black arrows). When *scan enable* is 1, it sets the flip-flop to the value received through the scan chain instead (grey arrows). 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 according to Figure *(b)*. 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.

How to test

Follow the test suite in `src/test.py`.

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

Arithmetic logic unit of four operations between two 8-bit numbers [198]

- Author: Alejandro Araya, María Bogantes, Isaías González
- Description: Calculates addition, multiplication, logical xor and shift left operations between two numbers.
- GitHub repository
- HDL project
- Mux address: 198
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

This circuit is designed to solve four arithmetic logic operations between two 8-bit numbers. The numbers are entered from a 4x4 matrix keyboard. The data entered from the keyboard is manipulated with decoders, encoders and registers, to finally reach an ALU. In the ALU one of the operations of addition, multiplication, xor or shift left will be calculated.

The circuit generates a two-bit counter that goes to a decoder, the decoder is responsible for activating the keyboard columns high. Pressing the keyboard columns will cause them to switch from high to low, resulting in the matrix_in input. The data that enter to matrix_in goes to an encoder. The encoder, according to the input, will have as output a hexadecimal value, which will be saved if en_reg is active. When en_reg is active, the data is saved at the address provided by switches 2 to 3. This address is the location where the data will be saved in the register bank.

The operands that enter the ALU are obtained from the register bank, the addresses of these operands are indicated with switches 4 to 5 and 6 to 7. To indicate the ALU operation, switches 0 to 1 are used, depending on the value entered, one of the following operations will be performed:

- 00 - A + B
- 01 - A * B
- 10 - A xor B
- 11 - A « 1

Finally, the result of the operation is obtained in the 8-bit alu_r output.

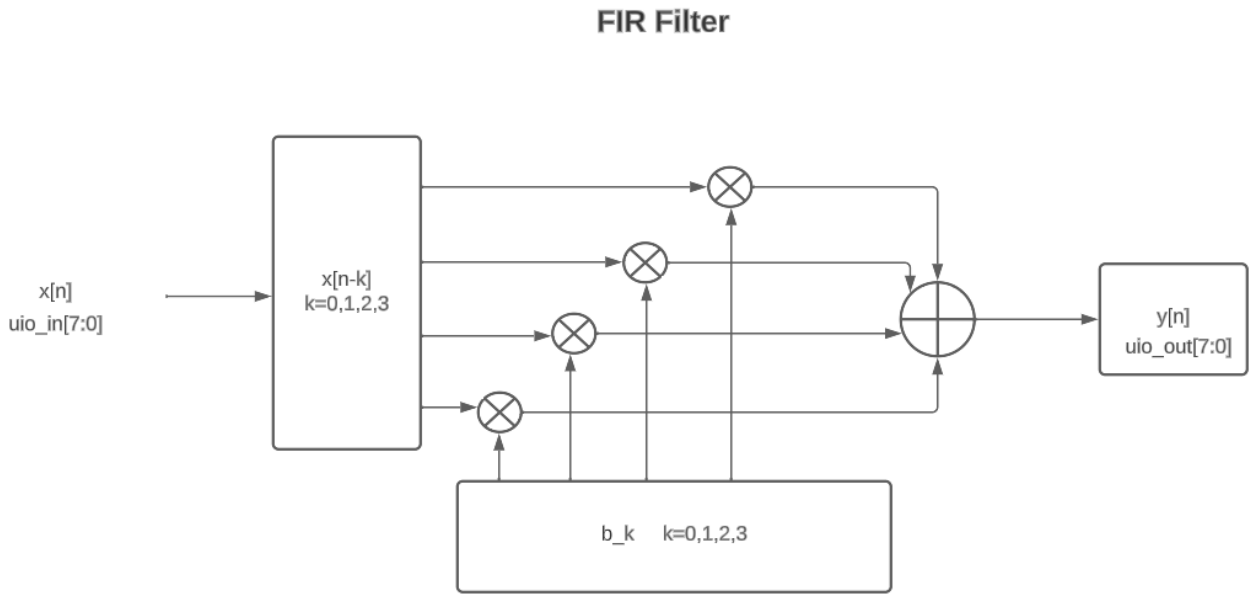
How to test

After reset, the counter should increase by one every second with a 10MHz input clock.

Pinout

#	Input	Output	Bidirectional
0	op [1:0] (selects operation)	alu_r [7:0] (operation result)	matrix_in [0:3] (keyboard data)
1	add_s [3:2]	n/a	en_reg (if active, saves kayboard data in the register bank)
2	add_op1 [5:4] (defines first operand direction)	n/a	2bc [1:0] (two bit counter)
3	add_op2 [7:6] (defines second operand direction)	n/a	z (zero flag)
4	n/a	n/a	n/a
5	n/a	n/a	n/a
6	n/a	n/a	n/a
7	n/a	n/a	n/a

FIR Filter [199]



- Author: Daniel González
- Description: FIR Filter with 4 coefficients
- GitHub repository
- HDL project
- Mux address: 199
- Extra docs
- Clock: 10000000 Hz
- External hardware: Requires a microcontroller to send the x[n] values for the input signal

How it works

Uses for coefficients which can be defined by the user with the swtichs, or by default 1s are used. The last 4 values of the input signal are multiplied by the coefficients and sum to generate the output value.

How to test

You have to send the input signal x[n] with the uio_in, and the output uio_out will be the output signal y[n]

Pinout

#	Input	Output	Bidirectional
0	{'ui_in': 'Assign custom coefficients, ui_in[7:3] with sel [2:1] and enable [0]'}	{'uio_out': 'Output y[n]'}	None
1	{'uo_out': 'does nothing'}	n/a	n/a
2	{'ui_in': 'Input x[n]'}	n/a	n/a
3	{'uio_oe': 'does nothing'}	n/a	n/a
4	{'ena': 'enables the shift register for taking the inputs'}	n/a	n/a
5	{'clk': 'clock'}	n/a	n/a
6	{'rst_n': 'when 0 past values of x[n] are set to 0 and coefficients to 1s'}	n/a	n/a
7	n/a	n/a	n/a

Tamagotchi [208]

- Author: Fabian Alvarez
- Description: Simple Console Tamagotchi
- GitHub repository
- HDL project
- Mux address: 208
- Extra docs
- Clock: 27000000 Hz
- External hardware:

How it works

Connect rx and tx to a serial terminal. The game will start automatically, if the tamagotchi dies, press the reset button to start again. feed all the stats to keep the tamagotchi alive (food, sleep, happiness, hygiene, social), when the tamagotchi are sleeping, you cannot interact with it. Controls:

- E: feed
- S: sleep
- P: play
- B: clean
- T: talk
- W: wake up

How to test

Use a 27MHz clock. Connect rx and tx to a serial terminal. The game will start automatically, if the tamagotchi dies, press the reset button to start again.

Pinout

#	Input	Output	Bidirectional
0	rx	tx	none
1	none	none	none
2	none	none	none
3	none	none	none
4	none	none	none
5	none	none	none

#	Input	Output	Bidirectional
6	none	none	none
7	none	none	none

LFMPDM (Lightning Fast Matrix Programmable Design Module) [209]

- Author: Emilio Baungarten
- Description: 8xLUTs 4-input
- GitHub repository
- HDL project
- Mux address: 209
- Extra docs
- Clock: Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	i_addr_load_data[0]	o_Data	i_LUT [0]
1	i_addr_load_data[1]	none	i_LUT [1]
2	i_addr_load_data[2]	none	i_LUT [2]
3	i_addr_load_data[3]	none	i_LUT [3]
4	i_Data	none	none
5	i_config_enable	none	none
6	none	none	none
7	none	none	none

7 SEGMENTS CLOCK [210]

- Author: Juan Carlos Garcia Lopez
- Description: 7 SEGMENTS CLOCK
- GitHub repository
- HDL project
- Mux address: 210
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	disp_type	ampm	none
1	fmt	segments_	none
2	prog	disp_select_	none
3	adjust	segment_select_	none
4	n/a	n/a	none
5	n/a	n/a	none
6	n/a	n/a	none
7	n/a	n/a	none

Multi Pattern LED Sequencer [211]

- Author: Francisco Javier Rodriguez Navarrete
- Description: Project for the MPLS LED lights
- GitHub repository
- HDL project
- Mux address: 211
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

The 'tt_um_MultiPatternLEDSequencer_RSYO3000.v' is just a wrapper for the tinytapeout I/O. Inside this, theres a top wrapper for the project called 'MultiPatternLEDSequencer.v' that connects to the tinytapeout wrapper and instances the following modules: 'PLL_10MHztoNHz.v' and 'MPLS.v'

The 'PLL_10MHztoNHz.v' contains a verilog that can change the 10MHz frequency input to 1Hz, 2Hz 5Hz and 50Hz, to see the LEDs with various frequencies via the ('clk_selector') signal.

The 'MPLS.v' is the main module, it uses a combination of counters, feedback loops, and pattern selection logic to generate different LED patterns. The pattern selection signal ('pattern_sel') determines which LED pattern to display.

- The 'demo_counter' is used to cycle through all the available patterns when ('pattern_sel') is 31.
- The 'pattern_counter' and 'oh_counter' are used to generate specific timing sequences for the LED patterns.
- The 'lfsr_reg' implements a Linear Feedback Shift Register (LFSR) to generate pseudo-random sequences.
- A 'case' selects from the selected ('pattern_sel') to display the selected pattern out of the 30 available patterns through the 8 outputs. The 30 patterns are the following ones:
  - **Pattern 0: All LEDs OFF** Turn off all the LEDs.
  - **Pattern 1: All LEDs ON** Turn on all the LEDs at once.
  - **Pattern 2: Blinking LEDs** Make the LEDs alternate between on and off, creating a blinking effect.

- **Pattern 3: Running lights** The LEDs move in a sequence, like lights running down a line.
- **Pattern 4: Alternating LEDs** Alternate the LEDs in an on-off pattern.
- **Pattern 5: Negative running lights** Similar to pattern 3, but with the LEDs off where they were on, and vice versa.
- **Pattern 6: KR effect** The LEDs flicker and shift, producing a mysterious "Knight Rider" effect.
- **Pattern 7: Bouncing lights** Lights bounce back and forth.
- **Pattern 8: LED wave effect** Create a wave-like pattern that travels along the LEDs.
- **Pattern 9: Alternating LED groups** Divide the LEDs into groups of 2 that alternate turning on and off.
- **Pattern 10: Heartbeat** Make the LEDs pulse in a heartbeat-like rhythm.
- **Pattern 11: p-Random LFSR LEDs** Use a random number generator to make the LEDs light up in a pseudo-random pattern.
- **Pattern 12: XOR All** XOR all counters, creating a unique show.
- **Pattern 13: Binary counter** Display a binary counting sequence on the LEDs.
- **Pattern 14: Clockwise LED rotation** Rotate the LEDs in a clockwise direction.
- **Pattern 15: XOR Pattern** XOR between one hot counter and binary counter.
- **Pattern 16: Bouncing lights** Similar to pattern 7, but with a slightly different bounce effect.
- **Pattern 17: Diagonal Bounce** Make the LEDs bounce diagonally across the LEDs.
- **Pattern 18: Circular Bounce** Create a circular bounce effect, like lights moving in a loop.
- **Pattern 19: Random Bounce** The LEDs bounce pseudo-randomly.
- **Pattern 20: Negative Diagonal Bounce** Like pattern 17, but with the LEDs off where they were on, and vice versa.
- **Pattern 21: Accelerating Bounce** The bouncing effect speeds up over time.

- **Pattern 22: Gravity Effect** LEDs appear to "fall" downward, creating a gravity-like effect.
- **Pattern 23: Spring Effect** Like pattern 8, but with a spring-like bounce in the wave effect.
- **Pattern 24: Reflecting Bounce** Create a bouncing pattern that reflects off the edges.
- **Pattern 25: Double Bounce** Similar to pattern 17, but with the middle LEDs on.
- **Pattern 26: Wave Bounce** A wave-like pattern that bounces back and forth.
- **Pattern 27: Breathing Effect** Make the LEDs "breathe" by gradually brightening and dimming.
- **Pattern 28: Alternating Binary and One-Hot** Switch between binary counting and one-hot encoding.
- **Pattern 29: Alternating LFSR and One-Hot** Alternate between the LFSR sequence and one-hot encoding.
- **Pattern 30: Alternating LFSR and Binary** Switch between the LFSR sequence and binary counting.
- **Pattern 31: DEMO** This mode cycles through all the available patterns automatically, showcasing the variety of the patterns.

How to test

To test the `MPLS` module, follow these steps:

- To create a simulation displaying all the patterns, run the target "make mpls_sim" in the src folder, this will run the testbench via icarus verilog for all patterns with a fixed time of simulation enough to see the patterns.
- There is another target that simulates the PLL and 10MHz to 1Hz, 2Hz, 5Hz and 50Hz with the MPLS module by using "make tt_sim" in the src folder, running this one is not recommended since its meant for long testing and the VCD file can go up to 16GB if you let it run everything.

Pinout

#	Input	Output	Bidirectional
0	N/C	LED[7]	none
1	pattern_sel[4]	LED[6]	none
2	pattern_sel[3]	LED[5]	none
3	pattern_sel[2]	LED[4]	none
4	pattern_sel[1]	LED[3]	none
5	pattern_sel[0]	LED[2]	none
6	clk_selector[1]	LED[1]	none
7	clk_selector[0]	LED[0]	none

Generador de PWM [212]

- Author: Rodrigo Garcia
- Description: This is PWM generator
- GitHub repository
- HDL project
- Mux address: 212
- Extra docs
- Clock: 10000000 Hz
- External hardware: Dos push buttons y un osciloscopio

How it works

General Description

```
The project is a simple Pulse Width Modulation (PWM) generator with a var
using two buttons: one to increase and another to decrease the duty cycle
MHz clock as its time base.
```

Button Signals

```
increase_duty and decrease_duty are signals directly connected to ui_in[0
```

Debounce Logic

```
The code includes debounce logic for the buttons. It uses a counter (coun
(slow_clk_enable). This slow clock is used to sample the button states an
```

PWM Logic

```
The code utilizes a 4-bit counter (counter_PWM) to generate a PWM signal.
DUTY_CYCLE variable, which can be increased or decreased using the deboun
```

DFF_PWM Module

```
This is a simple D flip-flop used for debounce. It samples the D input wh
```

How to test

Button Connections

Connect two buttons to the system. These buttons are used to control the

- Connect input pin 0 to one of the buttons. This button is used to increase the duty cycle of the PWM signal. Make sure the button has a pull-down resistor connected to ensure a defined logical level when not pressed.
- Connect input pin 1 to the other button. This button is used to decrease the duty cycle of the PWM signal. Just like the first button, ensure that this one also has a pull-down resistor connected.

Oscilloscope Connection

Connect an oscilloscope to output pin 0 of the dedicated outputs of the s
measure and display the PWM signal correctly.

Pinout

#	Input	Output	Bidirectional
0	increase_duty	out_pwm	none
1	decrease_duty	none	none
2	none	none	none
3	none	none	none
4	none	none	none
5	none	none	none
6	none	none	none
7	none	none	none

Multi stage path for delay measurements. [213]

- Author: Daniel Mundo, Noel Prado, Victor Vanegas
- Description: Verilog coding for cascaded not gates connected as a ring oscillator. After running the flow it is observed that the synthesizer does not support combinatorial feedback and that it collapsed several cascaded not gates into buffers. The original purpose for the ring oscillator will not be achieved but it is observed that synthesized circuit is still useful for measuring some gate delays that can be compared to theoretical calculations for educational purposes.
- GitHub repository
- HDL project
- Mux address: 213
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

The ui_in signals first two bits are used to control the transmission of the input signal thru the gates all the way to the several external outputs that are taps to different gate stages as to measure different stage delays for educational purposes.

How to test

One can put a square wave generator in the inputs and use a scope to measure the delay of the gates. The delay can be compared with theoretical calculations.

Pinout

#	Input	Output	Bidirectional
0	EN (ui_in[0])	Tap 1 (uo_out[0])	none
1	EN_2 (ui_in[1])	Tap 2 (uo_out[1])	none
2	none	Tap 3 (uo_out[2])	none
3	none	none	none
4	none	none	none
5	none	none	none
6	none	none	none
7	none	none	none

ASCII Text Printer Circuit [214]

- Author: Noel Prado, Daniel Mundo, Angel Orellana and Julio Lopez.
- Description: A circuit that is able to print two different texts. It utilizes 8 output pins, each character is printed as the ASCII character described in 8 bits.
- GitHub repository
- HDL project
- Mux address: 214
- Extra docs
- Clock: 0 Hz
- External hardware: Any microcontroller, we have tested it using TIVA C and a FPGA.

How it works

This circuit is designed to output ASCII-encoded text sequences. The circuit can display two different texts.

Select Input: The select pins (`ui_in[1:0]`), a 2-bit binary input, determine which text sequence will be displayed:

- `2'b00` or `2'b11`: Outputs a sequence of characters that correspond to the beginning of a traditional song from Guatemala.
- `2'b01` or `2'b10`: Outputs a sequence of characters with the names of the people that participated in this project.

The text is displayed character-by-character, with each character's ASCII representation determined by the current value of an internal counter. The counter increments with each clock cycle until the specified limit for the chosen text sequence is reached, at which point it resets, allowing the sequence to be displayed repetitively.

How to test

To test this project, one needs to use an external microcontroller, where one can read digital input pins synchronously. After reading the characters via the input pins, you can send the pins to a computer via UART communication and display the texts on the computer terminal.

Pinout

#	Input	Output	Bidirectional
0	select bit 0	Bit 0	none
1	select bit 1	Bit 1	none
2	none	Bit 2	none
3	none	Bit 3	none
4	none	Bit 4	none
5	none	Bit 5	none
6	none	Bit 6	none
7	none	Bit 7	none

Clock synchronizer [215]

- Author: Mateo Guerrero Gonzalo Hernandez Cesar Azambuya Francisco Veirano
- Description: Testing different ways of clock synchronizers to avoid metastability problems.
- GitHub repository
- HDL project
- Mux address: 215
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

This project is composed by different clock synchronizers subblocks. The function of these subblocks is to mitigate the problem of metastability caused when you have two clock domains and you want to transfer a signal from one domain to another. Then, with a selector input, you can select the output from the different subblocks in order to test and measure their performance to avoid this problem.

So, the project has: A first register at the input to hold the data. This 8 bits-length data is registered by CLK1.

This data is registered by another second register but with another clock, in this case CLK2. The data of the output of this register has a high probability to be metastable.

The output of the first register is the input of the three different blocks to be tested.

1- The first one, 2 flip-flop synchronizer, is composed by two register in cascade (A connected to B), triggered by CLK2. There is probability that while sampling the input A-d by flip flop A in CLK2 clock domain, output A-q may go into metastable state. But during the one clock cycle period of CLK2 clock, output A-q may settle to some stable value. Output of flop B can go to metastable if A does not settle to stable value during one clock cycle, but probability for B to be metastable for a complete destination clock cycle is very close to zero.

2- The second, Recirculation mux synchronization, in order to synchronize data, a control pulse is generated in source clock domain (CLK1) when data is available at source flop. Control Pulse is then synchronized using 2 flip flop synchronizer or pulse synchronizer (Toggle or Handshake) depending on clock ratio between source (CLK1) and destination (CLK2) domain. Synchronized control pulse is used to sample the data on the bus in destination domain. Data should be stable until it is sampled in destination clock domain (CLK2).

3- The third, Toggle block, is the same as Recirculation mux synchronization block, but in this case we generate the control pulse using a Toggle synchronizer. The toggle synchronizer is used to synchronize a pulse generating in source clock domain (CLK1) to destination clock domain (CLK2). A pulse cannot be synchronized directly using 2 FF synchronizer. While synchronizing from fast clock domain (CLk2) to slow clock domain (CLk1) using 2 FF synchronizer, the pulse can be skipped which can cause the loss of pulse detection and hence subsequent circuit which depends upon it, may not function properly.

The project also has enable control block. This block controls that the synchronizer subblocks run only once each, generating a pulse in their respective enable inputs.

How to test

First, you need to define the frequency you want to use with CLK1 and CLK2. After reset, you need to put an input data on data_in bus (uio_in). If you want to use de enable control block, you need to put enable_block input (ui_in[6]) at low level all the time, and rise a pulse of one period of CLK1 duration in trigger input (ui_in[7]). Otherwise, you can put enable_block in high level all the time. In this case, all blocks are going to be updating their values constantly. You will see at the output the signal and examine if the signal has a metastable state or not, switching between the diferent output using the sel input (ui_in[3:1]). Also, you can use the input stb and pulse_in to measure the performance of the blocks associated to these inputs. These signals are used as triggers to register data in CLK2 domain.

Pinout

#	Input	Output	Bidirectional
0	clk, clock of the first clock domain.	data_out connected to uo_out. This is the data output of the different block depending of the value of sel.	data_in connected to uio_in, set as input. This is the data input to be synchronized.
1	clk_2 connected to ui_in[0]. This is the second clock domain.	When the input sel = 0 you will see the output of the first FF (triggered by CLK1).	n/a

#	Input	Output	Bidirectional
2	sel connected to ui_in[3:1]. This the selector tu select the different output between the different blocks.	When the input sel = 1 you will see the output of the second FF (triggered by CLK2).	n/a
3	stb connected ui_in[4]. This is the pulse needed to synchronize with the second block.	When the input sel = 2 you will see the first block using to synchronize (triggered by CLK2).	n/a
4	pulse_in connected to ui_in[5]. This is the pulse needed to synchronize with the third block.	When the input sel = 3 you will see the second block using to synchronize (triggered by CLK1 and CLK2).	n/a
5	rst_n. Reset of the system.	When the input sel = 4 you will see the third block using to synchronize (triggered by CLK1 and CLK2).	n/a
6	ena. Will go high when the design is enabled.	When the input sel = 5 you will see the output {stb_in, ctrl, stb_out, A, B1, B2, B3, pulse_out}, the intermediate signals of the different blocks.	n/a
7	enable_block connected to ui_in[6]. This signal is used to enable all subblocks.	When the input sel = 6 you will see the output {0, 0, ena_A, ena_1, ena_2, ena_3, done} the enables of the different blocks.	n/a

Simple PWM Generator [224]

- Author: Daniel Barrios
- Description: Generates a PWM signal with a duty cycle that can be varied with inputs pins
- GitHub repository
- HDL project
- Mux address: 224
- Extra docs
- Clock: 5000000 Hz
- External hardware:

How it works

The PWM Generator takes a clock and generates a PWM by comparating the selected bus against a counter. Another input is added to determine the maximum resolution for the counter (meaning high resolution requires more bits for counting, which results in an overall lower frequency). By dinamically changing the counter max it is easy to generate the new signal. Also a DFF is added to the output of the comparator in order to synchronize the signal and reduce the possible glitches that can arise by changing values mid-run.

How to test

To test, just connect the duty bus to the desired value at the output, while also setting the Maximum Bits in the bidirectional pins to the desired quantity (max - 111). After pressing restart the PWM should work as desired.

Pinout

#	Input	Output	Bidirectional
0	Duty[0]	PWM output	Bit selector [0]
1	Duty[1]	none	Bit selector [1]
2	Duty[2]	none	Bit selector [2]
3	Duty[3]	none	none
4	Duty[4]	none	none
5	Duty[5]	none	none
6	Duty[6]	none	none

#	Input	Output	Bidirectional
7	Duty[7]	none	none

CLK Frequency Divider [225]

- Author: Ramon Sarmiento
- Description: Generates several frequency clock signals from a user-selected M module
- GitHub repository
- HDL project
- Mux address: 225
- Extra docs
- Clock: Hz
- External hardware:

How it works

The frequency divider consists of a counter module M chosen by the user as an input to the project. When the counter reaches the value M-1, a signal is enabled which will function as a clock of another 7-bit counter and each output of this counter can be used as a clock signal, each signal is divided by 2 in frequency. The pulse of the M module counter was enabled as output but it is not recommended to use it as clock signal, but it can be used in other applications.

How to test

To test the design you only need to choose the M module with the switches on the dedicated inputs.

Pinout

#	Input	Output	Bidirectional
0	Modulo[0]	segment a	CLK 1/2
1	Modulo[1]	segment b	CLK 1/4
2	Modulo[2]	segment c	CLK 1/8
3	Modulo[3]	segment d	CLK 1/16
4	Modulo[4]	segment e	CLK 1/32
5	Modulo[5]	segment f	CLK 1/64
6	Modulo[6]	segment g	CLK 1/128
7	Modulo[7]	dot	Modulo M signal

UIS Traffic Light [226]

- Author: Jorge Eduardo Angarita Pérez
- Description: Traffic light control for LatinPractice Bootcamp at UIS
- GitHub repository
- HDL project
- Mux address: 226
- Extra docs
- Clock: 32768 Hz
- External hardware: Three Different Color LEDs (Optional)

How it works

This is a Finite State Machine (FSM) that utilizes an instantiated module of the "CLK Frequency Divider" by Ramón Sarmiento to perform internal counting for a traffic light control system.

How to test

Set the first input to "1" and await the activation of the Red light. It will remain active for 30 seconds, provided the correct frequency is employed. Afterward, it will transition to the Green state within 3 seconds, remaining in this state for an additional 20 seconds. Finally, it will transition back to the Red state over the course of 3 seconds.

Pinout

#	Input	Output	Bidirectional
0	Start	Green Light	none
1	none	Yellow Light	none
2	none	Red Light	none
3	none	none	none
4	none	none	none
5	none	none	none
6	none	none	none
7	none	none	none

4 bit adder [227]

- Author: Nestor Matajira
- Description: Add two 4 bit numbers
- GitHub repository
- HDL project
- Mux address: 227
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Consists of 4 adders, 1 half adder and 3 full adders

How to test

Between bit 0 to 3 set the first number, and from pin 4 to 7 set the second number.

Pinout

#	Input	Output	Bidirectional
0	A[0]	S[0]	none
1	A[1]	S[1]	none
2	A[2]	S[2]	none
3	A[3]	S[3]	none
4	B[0]	S[4]	none
5	B[1]	none	none
6	B[2]	none	none
7	B[3]	none	none

8-bit ALU [228]

- Author: Nicolas Orcasitas Garcia
- Description: 8-bit ALU with 8 operations
- GitHub repository
- HDL project
- Mux address: 228
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

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

Collatz Conjecture [229]

- Author: Sergio Sebastian Oliveros Sepulveda
- Description: A circuit that computes the Collatz orbit
- GitHub repository
- HDL project
- Mux address: 229
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

The circuit takes an 8-bit input value and produces two outputs: the number of iterations required to reach 1 and a status bit indicating whether the calculation is in progress or complete. Once the process is finished, the circuit keeps the value of the iterations indefinitely so that it can be checked.

How to test

To test the circuit it is necessary to have as input the clock signal, a number of maximum 8 bits and the rst_n signal to start the iterations. It is taken into account that rst_n is at 0 when it is active, so once it takes this value, the circuit begins to perform the calculations until it reaches 1, then it keeps the values.

Pinout

#	Input	Output	Bidirectional
0	clk	Process indicator (busy bit)	Number of iterations to reach 1 (8 bits number)
1	ena	n/a	n/a
2	rst_n	n/a	n/a
3	Number to test	n/a	n/a
4	n/a	n/a	n/a
5	n/a	n/a	n/a
6	n/a	n/a	n/a
7	n/a	n/a	n/a

8 bit 4 data sorting network [230]

- Author: Emmanuel Díaz Marín
- Description: The circuit orders the 4 input numbers according to their value, with the highest number at output 0
- GitHub repository
- Wokwi project
- Mux address: 230
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Sorting networks can be visualized as combinatorial circuits where a set of denoted compare-swap (CS) circuits can be connected in accordance to a specific network topology. This way the CS circuit is formed by a full adder configured as a subtractor and a pair of multiplexers, the carry of the subtractor is used for the selection of the multiplexer.

How to test

1. Reset signal.
2. Enter the 4 8-bit inputs (8 clock positive flanks).
3. Enable the control/load signal for 8 clock positive flanks.
4. See the results.

Pinout

#	Input	Output	Bidirectional
0	Number 1	Highest number	none
1	Number 2	Second highest number	none
2	Number 3	Third highest number	none
3	Number 4	Fourth highest number	none
4	none	Not used	none
5	none	Not used	none
6	none	Not used	none
7	Control	Not used	none

BCD to 7 segments [231]

- Author: Josue Marcelo Castillo Acosta, Kaylee Michelle Diaz Rodriguez, Juliana Hernandez Hernandez
- Description: un decodificador de binario a decimal
- GitHub repository
- Wokwi project
- Mux address: 231
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

emplea una funcion logica para que mediante un numero binario salga el numero en el display

How to test

ingresar en las entradas un numero binario

Pinout

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

4 bit full adder [240]

- Author: Hugo Jesús Navarro Hernández, David Mora Mendez, Nadia Fernanda Barradas Solis, Juan Giovani Landa Cervantes
- Description: A full adder of binary numbers with logic gates
- GitHub repository
- Wokwi project
- Mux address: 240
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

"Takes 2 numbers and adds them using binary code with logic gates"

How to test

"Selecting the different switches and switching them on and off to turn on the leds, each switch has an assigned value, such as 1, 2, 4, 8, 16"

Pinout

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

Circuito Religioso [241]

- Author: Eunice Husai Garcia Javier, Axel Daniel Luna Carmona, Aneesa Miranda Peredo García, Daniel Alberto Gil Martinez
- Description: Un circuito BCD display 7 segmentos que despliega caracteres para formar una palabra
- GitHub repository
- Wokwi project
- Mux address: 241
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Son compuertas logicas que crean una funcion para desplegar caracteres en un display de 7 segmentos. Eb este caso los caracteres forman dos palabras por lo que utilizamos una entrada de seleccion para elegir cual de ambas se muestran.

How to test

En las entradas 0 a 3 escribir un numero binario con la ultima entrada (3) como el bit menos significante. Al escribirlos en orden de 0 a 9 se desplegará la palabra, la cual se selecciona con la entrada 7.

Pinout

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

Demultiplexor NAND [242]

- Author: Mauricio Caballero Hernández - Alejandro Duran Morales - Marvin Yahir Salamanca Ramirez - Kevin Ortiz Sarate
- Description: Demultiplexor de 3 entradas independientes y 3 entradas de dirección que arrojan valore logicos de 0 y 1 en 8 salidas diferentes, constituido por compuertas NAND y NOT, imitando el comportamiento de un LS138
- GitHub repository
- Wokwi project
- Mux address: 242
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Introduciendo un total de 6 señales en el circuito se puede arrojar una señal negada (un valor de 0 logico) en una de las 8 salidas disponibles. Las primeras 3 entradas dentro del circuito son clasificadas como entradas de dirección y se encargan de configurar el Demultiplexor, los otros 3 puertos de entrada admiten valores de entrada independientes que terminan por infliur en las entradas de las compuertas logicas NAND y eso enconjunto permite que se arrojen valores logicos, predominando los estados altosen 7 de 8 salidas, mientras que la salida restante arroja un valor logico de 0 (lo cual admite un total de 8 combinaciones posibles con resultados diferentes). Todo el cuerpo del Demultiplexor esta conformado por compuertas NAND de 4 entradas y su demanda de compuertas NOT es minima en comparacion.

How to test

Para probar el circuito es necesario utilizar un dip switch de 6 entradas donde las primeras 3 posiciones conformaran las entradas de dirección (E0-E2) mientras que las posiciones 4 a 6 seran las entradas independientes (A0-A2). En su estado natural, (sin señales de entrada más que la E0), se arrojara el estado bajo a O0, Para arrojar un valor de 0 a la salida O1 es necesario mantener activa la entrada de direccion E0 y la entrada independiente A0. Para cambiar a la salida O2 se mantiene la entrada de dirección E0 y la entrada A1. Para cambiar a la salida O3 se mantiene la entrada de dirección E0 y la entrada A0 + A1. Para cambiar a la salida O4 se mantiene la entrada de dirección E0 y la entrada A2. Para cambiar a la salida O5 se mantiene la entrada de dirección E0 y la entrada A0 + A2. Para cambiar a la salida O6 se mantiene la entrada de dirección E0 y la entrada A1 + A2. Para cambiar a la salida O7 se mantiene la

entrada de dirección E0 y la entrada A0 + A1 + A2. Para que todas las salidas arrojen un valor logico de 1 se necesita que se activen las entradas E1 + E2.

Pinout

#	Input	Output	Bidirectional
0	E0 (Entrada de dirección)	segment a	none
1	E1 (Entrada de dirección)	segment b	none
2	E2 (Entrada de dirección)	segment c	none
3	A0 (Entrada independiente)	segment d	none
4	A1 (Entrada independiente)	segment e	none
5	A2 (Entrada independiente)	segment f	none
6	n/a	segment g	none
7	n/a	dot	none

Sumador/Sustractor de 3 bit con acarreo y prestamo [243]

- Author: ONIX-M50
- Description: Este es un pequeño proyecto para la iniciativa VLSI, el cual consta de un circuito que realiza tanto la suma como la sustracción de dos numeros de 3 bits, los cuales pueden venir acompañados de un bit de acarreo o prestamo respectivamente.
- GitHub repository
- Wokwi project
- Mux address: 243
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Este circuito recibe 2 numeros de 3 bits cada uno y opcionalmente un bit de acarreo y/o un bit de prestamo. El circuito realiza tanto la adición como la sustracción de dichos números y el resultado de ambas operaciones es entregado a la salida. Se utilizan 4 pines de salida para el resultado de la suma y 4 pines para el resultado de la resta.

How to test

Las entradas IN0 a IN2 corresponden al primer número de 3 bits, mientras que las entradas IN4 a IN6 corresponden al segundo número de 3 bits, por otro lado, las entradas IN3 e IN7 corresponden al los posibles bits de acarreo y préstamo respectivamente. Las salidas OUT0 a OUT3 corrreponden al resultado de la suma de los npumeros binarios de entrada, siendo OUT3 el acarreo. Mientras que las salidas OUT4 a OUT7 corresponden al resultado de la resta de los números binarios de entrada, siendo OUT4 el prestamo.

Pinout

#	Input	Output	Bidirectional
0	A0	X0	none
1	A1	X1	none
2	A2	X2	none
3	carry	carry	none
4	B0	Y0	none

#	Input	Output	Bidirectional
5	B1	Y1	none
6	B2	Y2	none
7	borrow	borrow	none

Hardware Lock [244]

- Author: Lautiux
- Description: A simple hardware pin made with a shift register formed by joining various flip-flops.
- GitHub repository
- Wokwi project
- Mux address: 244
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Entering the right pin the output turns on.

How to test

Enter the following pin serially one bit at a time through the input 0 (10100100010). For a 1 turn input 0 ON and send a clock pulse, for a 0 turn input 0 OFF and send a clock pulse. Do that 11 times and with the correct pin the output shoud turn on, if not it shoud stay off.

Pinout

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

Custom falling and rising edge detection [245]

- Author: Kelvin Kung
- Description: Build a custom edge detection circuit using flip-flop
- GitHub repository
- Wokwi project
- Mux address: 245
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

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

4-bit-alu [246]

- Author: Angelo Machorro
- Description: 4 bit alu
- GitHub repository
- HDL project
- Mux address: 246
- Extra docs
- Clock: 0 Hz
- External hardware: buttons or dip switches

How it works

Its a 4-bit alu, can make addition, substraction, multiplication, division and bitwise-and.

How to test

A and B input are the data for ALU, S is the operation selects, in the next table we presents the operations

operation	S	Result
addition	0	A + B
substraction	1	A - B
multiplication	2	A * B
division	3	A/B
bitwise and	4	A & B

Pinout

#	Input	Output	Bidirectional
0	A3	R7	S0
1	A2	R6	S1
2	A1	R5	S2
3	A0	R4	none
4	B3	R3	none
5	B2	R2	none
6	B1	R1	none

#	Input	Output	Bidirectional
7	B0	R0	none

Angardo's pong [247]

- Author: Angel Orellana
- Description: Is a pong game
- GitHub repository
- HDL project
- Mux address: 247
- Extra docs
- Clock: 10000000 Hz
- External hardware: 6 pushbutton, 8x8 neopixel matrix

How it works

This is a pong game, it uses a neopixel led matrix 8x8 as display. To control the game, each player has two push buttons to move up and down the paddle, an extra push button is used to start the game. The game ends when the ball touch the left or the right side of the matrix. To set the initial conditions of the game press the reset button.

How to test

To use the project you must to connect the 6 push butttons (two for player 1, two for player 2, one for start game, and 1 for reset) with an pull up resistor, also you need conect the data in pin of neopixel matrix to driver_neopixel output.

Pinout

#	Input	Output	Bidirectional
0	start	driver_neopixel	none
1	p1_up	none	none
2	p1_down	none	none
3	p2_up	none	none
4	p2_down	none	none
5	none	none	none
6	none	none	none
7	none	none	none

(11,7) hamming code encoder and decoder with UART [256]

- Author: LEOGLM
- Description: (11,7) hamming code encoder and decoder using UART Protocol
- GitHub repository
- HDL project
- Mux address: 256
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

The project is divided into two parts: a (11,7) Hamming code encoder connected to a UART transmitter, and a decoder connected to a UART receiver. Both the encoder and decoder share the same set of input/output ports, which can be switched by inserting an impulse at ui_in[0]. The encoder adds four parity bits to a sequence of parallel data, improving its error detection and correction capabilities. The UART transmitter then rearranges and sends the coded data in series. On the receiving end, the UART receiver receives the message in series and converts it back to parallel form for further processing. Finally, the decoder decodes the message, corrects any potential errors, and outputs the original message, ensuring reliable and accurate data transmission.

How to test

To test the encoder, a sequence of parallel data can be inserted and the resulting coded data in series can be checked for accuracy. For the decoder, a sequence of coded data in series can be inputted, with a maximum of one bit error, to verify whether the decoder can output the correct data.

Pinout

#	Input	Output	Bidirectional
0	encoder and decoder switching	state	encoder_input/decoder_output
1	encoder_enable	encoder_output	encoder_input/decoder_output
2	decoder_input	decoder_output	encoder_input/decoder_output
3	encoder_input	decoder_output	encoder_input/decoder_output
4	encoder_input	decoder_output	encoder_input/decoder_output

#	Input	Output	Bidirectional
5	encoder_input	decoder_output	encoder_input
6	encoder_input	decoder_output	none
7	encoder_input	decoder_output	none

Multi-channel pulse counter with serial output, v01b [257]

- Author: Adrian Novosel, Dinko Oletic
- Description: Counts number of digital pulses occuring within a time interval across four input channels, and periodically outputs the values out using serial output. Verilog HDL implementation.
- GitHub repository
- HDL project
- Mux address: 257
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Design consists of four 12-bit counters with overflow detection, a mm:ss real-time clock (RTC), a parallel-input-serial output (PISO) readout register, controlled by a readout state-machine. The counters store number of intermittently-occuring short digital input pulses, accumulated within the RTC's time-measurement interval 00:00 - 59:59, at each of the four input channels. Periodically, after every RTC overflow (1 h with assumed 1 Hz RTC input clock signal), the state-machine performs sequential serial readout of the RTC time and all channels, and resets all channel counters. Additionally, readout and individual channel reset is initiated by overflow at any of individual input channel counter. This an early work-in progress implementation of digital portion of a low-power sensor interface for readout of a multichannel acoustic emission detector, based on MEMS-array of piezoelectric microresonators for passive ultrasonic band-pass filtering: https://ieeexplore.ieee.org/document/9139151. Design is generally applicable for low-power wake-up sensor interfaces, acoustic event detection, non-destructive testing, particle-counters, or as a generic pulse-counting digital building block.

How to test

Input signals are short rising-edge digital pulses, connected to input pins "ch1", "ch2", "ch3", "ch4". Output data becomes ready for serial readout at the output pin "serial_out" when overflow is signalled via the output "ready" pin ovf_global. Output bits are serially clocked-out using the input pin "clk". Specifically, RTC overflow is signalled via output pin "ovf_RTC_out", and overflow at an individual channel via the pin "ovf_ch_out". The rest of output pins are used for debugging of the state-machine's internal states.

Pinout

#	Input	Output	Bidirectional
0	ch1	serial_out	none
1	ch2	ovf_global	none
2	ch3	a0_out	none
3	ch4	a1_out	none
4	RTC	a2_out	none
5	clk	SL_out	none
6	reset	ovf_RTC_out	none
7	none	ovf_ch_out	none

State machine of an impulse counter [258]

- Author: Adrian Novosel
- Description: This design is not meant to be a standalone circuit. It is a state machine of my bachelor's thesis project which was also submitted to Tiny Tapeout. This submittal will be used for debugging and will give a better insight into the working principle of its source design. (https://github.com/DinkoOletic/tt04-wokwi_unizgfer_multich_pulse_counter_v01a)
- GitHub repository
- Wokwi project
- Mux address: 258
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

By clocking the state machine adress pins a2, a1 and a0 change their values every 12 clocks. Shift/load pin is set to 1 and then back to zero every 12 clocks. Other outputs are used for debugging.

How to test

After an impulse on reset pin and a subsequent impulse on the ovf pin, you can start clocking the circuit. Adress pins marked with "a" should follow the sequence: "100"->"000"->"001"->"010"->"011"->"000". Shift/load pin wil be set to 1 after every adress change and then set back to 0 on the next clock. The sequence will repeat itself.

Pinout

#	Input	Output	Bidirectional
0	reset	counter flop1	a2
1	ovf	counter flop2	a1
2	clk	counter flop3	a0
3	bi oe	counter flop4	shift/load
4	none	zero	sm flop1
5	none	one	sm flop 2
6	none	global reset	sm flop 3

#	Input	Output	Bidirectional
7	none	ovf	sm flop 4

Logic Circuit 1 [259]

- Author: Patryk Warnke MY REMOTE IOT
- Description: Logic Circuit Timing Test. Serial 1
- GitHub repository
- Wokwi project
- Mux address: 259
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

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

Variable Duty-Cycle TRNG [260]

- Author: Thomas Pluck
- Description: Generates a random bit with a given probability with a special ring oscillator
- GitHub repository
- HDL project
- Mux address: 260
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

This design is a close adaptation of the concept presented by:

Minh-Hai Nguyen; Cong-Kha Pham A wide frequency range and adjustable duty cycle CMOS ring voltage controlled oscillator https://ieeexplore.ieee.org/abstract/document/5670690

This design has seven 7-stage ring oscillators that have a final stage NAND which is takes as input the inverter chain and the `ena` wire - that is, the oscillator should only run when `ena` is high.

The oscillators are all tied at their NAND-outputs to a single node which passes the signal through an "inverter bias" system that are biased using Harald Pretl's vDAC system which uses the first 4 input pins to select a voltage:

https://github.com/iic-jku/tt03-tempsensor/tree/main/src

Note: This circuit is only active when `ena` is enabled.

The inverter-bias system pass through two D-flip flops connected in series with their `set` signals tied high and `reset` being connected to the `rst_n` control wire and their clocks being controlled by `CLK`. Finally, these are passed through a `ena`-enabled `AND` which gates the final output of the TRNG.

How to test

Simulate the analog, hope for the best.

Pinout

#	Input	Output	Bidirectional
0	signal	sample	none
1	control 1	none	none
2	control 2	none	none
3	control 3	none	none
4	none	none	none
5	none	none	none
6	none	none	none
7	none	none	none

Pseudo Random Number Generator [261]

- Author: International Hellenic University - Department of Information and Electronic Engineering
- Description: This project implements a pseudo-random number generator using Verilog. It generates pseudo-random numbers and outputs them to drive two 7-segment displays.
- GitHub repository
- HDL project
- Mux address: 261
- Extra docs
- Clock: 50000000 Hz
- External hardware: 7 segment display

How it works

The pseudo-random number generator is based on LFSR. It takes three inputs:

- `clk`: Clock input.
- `en`: Enable signal.
- `rst_n`: Active-low reset signal.

The generator produces 8-bit pseudo-random numbers, which can be used to control two 7-segment displays. As source of pseudo-randomnes used 2 LFSRs. One that works with 8 bit and one with 16 bit. The circuit dont have input data, just 1 clock and 1 enable pin. The output will be a 7 segment display, so 14 output pins. The 16 bit LFSR produce the input of a 16-to-8 multiplexer and the 8 bit LFSR produce the selection bits of 16-to-8 multiplexer. The 16-to-8 multiplexer is implemented by 8 2-to-1 multiplexers. The final stage is convert the data readable by 2 7 segment displays.

LFSR16 —— bit 0 ——> | 16-to-8 mux | —— bit 1 ——> | |———-> output 0 . | |———-> output 1 . | | . . |inputs | . —— bit 15 ——> | | . | | | |———-> output 7 LFSR8 —— bit 0 ——> |select | —— bit 1 ——> | | . | | . | | . | | —— bit 7 ——> | |

How to test

To test the pseudo-random number generator, you can follow these steps:

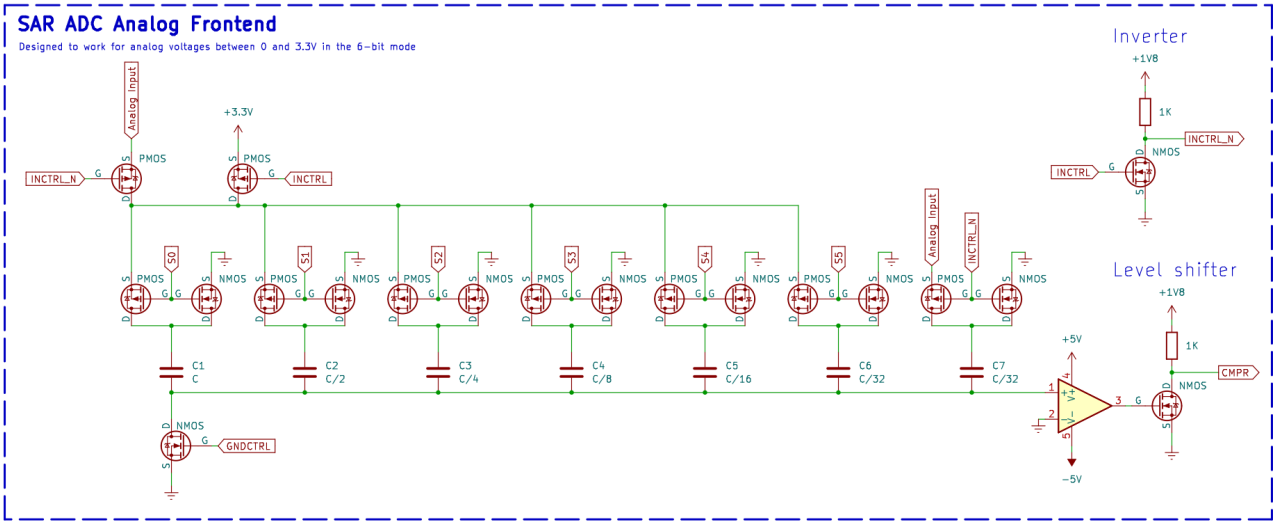
- Connect the `clk`, `en`, and `rst_n` signals appropriately.
- Connect 2 7 segments display as output.

- Apply clock pulses at 50MHz and high logic control signals at `en` to generate and display pseudo-random numbers.

Pinout

#	Input	Output	Bidirectional
0	clk	segment0 a	segment1 b
1	en	segment0 b	segment1 c
2	rst_n	segment0 c	segment1 d
3	n/a	segment0 d	segment1 e
4	n/a	segment0 e	segment1 f
5	n/a	segment0 f	segment1 g
6	n/a	segment0 g	n/a
7	n/a	segment1 a	n/a

SAR ADC Backend [262]



- Author: Hugo Frisk
- Description: A digital backend of a successive approximation digital to analog converter (SAR ADC) featuring two interfaces: I2C or an 11-bit parallel bus.
- GitHub repository
- Wokwi project
- Mux address: 262
- Extra docs
- Clock: 100000 Hz
- External hardware: Op-amp, some precise capacitors, resistors, negative supply, and, p-channel and n-channel mosfets

How it works

This is the digital backend of a charge redistrbution successive approximation analog to digital converter (SAR ADC). A SAR ADC converts an analog voltage to a digital value by successively recreating better and better approximations of the input analog signal. The analog frontend consists of a bank of capacitors where every capacitor has half of the previous capacitors capacitance. Each capacitor can be connected to either the positive or the negative supply, controlled by the digital backend. This forms a variable capacitive divider, or in simpler terms, a digital to analog converter with very high output impedence. The voltage created by the divider is sent to a comparator and compared with a reference voltage. In this implementation, the hold circuit that samples the analog input is combined with the capacitor bank.

See this document from Texas Instruments that the design is based on: https://www.ti.com.cn/cn/lit/an/slyt176/slyt176.pdf

How to test

The precision of the ADC can be set to either 11-bits or 6-bits (for faster measurements). When PRECSEL is low, the ADC is in 11-bit mode. When PRECSEL is high, the ADC is in 6-bit mode. S6 through S10 should be left floating when in 6-bit mode.

In the picture of this design there is an example circuit of how the analog frontend could be built. Note that it is not tested so use your own judgement and don't blow up your chip.

There are two ways to interface with the ADC:

1. Through a 6- or 11-bit paralell bus: To make a measurement, pulse START/BUSY high. While the measurement is taking place the pin will remain high as to signal that it is busy. When the pin goes low, the measurement is done. Store the measurement by reading S0 through S10. S0 is MSB and S10 is LSB.
2. Through I2C: Connect SCL and SDA to a microcontroller with pullups. The I2C address of the ADC can be configured to either 0x34 or 0x35. When ADRSEL is high, the address is 0x34 and when it is low it is 0x35. To make a measurement, send a write command to configured address with the data 0x01. Read the measurement by requesting 2 bytes from configured address. The ADC will NACK the request if it is still busy with the measurement.

Make sure RST_N is pulsed low after a power cycle.

The clock speed that this design works at is yet to be determined and is left as an exercise to the engineer.

Pinout

#	Input	Output	Bidirectional
0	CMPR	S0	START/BUSY
1	PRECSEL	S1	SCL
2	ADRSEL	S2	SDA
3	none	S3	S8
4	none	S4	S9
5	none	S5	S10
6	none	S6	INCTRL
7	none	S7	GNDCTRL

FCFM 7-segment display [263]

- Author: Diego Sanz
- Description: Displays UCHILE-FCFM- into the 7-segment display
- GitHub repository
- Wokwi project
- Mux address: 263
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

It uses a Ripple Counter made from DSR flip-flops to count each character from UCHILE-FCFM- and displays it into the 7-segment display, the logic behind was optimized using the Quine McCluskey algorithm.

How to test

You can test the desing by stepping the clock to see UCHILE-FCFM- in the 7-segment display. Also you can disable the counter with the 3th input and input a counter value by hand using the 4th to 7th inputs.

Pinout

#	Input	Output	Bidirectional
0	clock	segment a	none
1	reset	segment b	none
2	none	segment c	none
3	counter_disable	segment d	none
4	A (most significant bit of the counter)	segment e	none
5	B	segment f	none
6	C	segment g	none
7	D (least significant bit of the counter)	dot	none

another ring oscillator based temperature sensor [272]

- Author: Rodrigo Munoz (UCH)
- Description: 4 differents Ring oscillator whose frecuency depends on temperature. It project is based on https://github.com/JorgeMarinN/tt03_ac3e-usm_ro-based_tempsens
- GitHub repository
- HDL project
- Mux address: 272
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

After ena=1 and the reset cycle, a counter counts the number of cycles of the selected ring oscillator in one period of the system clock. the ring oscillator is selected by osc_sel input, each oscillator have different frequency.

This count is added cumulatively, the number of counts added is given by the entry sum_sel: (sum_sel+1)*4 = number of counts added.

When the data 0x00 (START CODE) is received by the uart, the sum total of three bytes long is sent back in LSB first.

additionally, on each clock cycle the output of the oscillator cycle counter is divided by 2 and sent to the output uo_out[7:1] and uio_out[7:0].

How to test

After reset and enable are set, the ring oscillator should start and then when a START code (0x00) is received by UART, the cumulative sum value of 3 bytes is sent back. The oscillator counter divided by 2 is present on uo_out[7:1] and uio_out[7:0].

Pinout

#	Input	Output	Bidirectional
0	ui_in[0] = clk_external	uo_out[0] = tx (UART tx)	['uio_out[7:0] = oscillator counter bits 15 to 8']

#	Input	Output	Bidirectional
1	ui_in[1] = clk_sel (select the system clock input)	uo_out[7:1] = oscillator counter bits 7 to 1	n/a
2	ui_in[4:2] = sum_sel (number of oscillator counts added (sum_sel+1)*4)	n/a	n/a
3	ui_in[5] = rx (UART RX)	n/a	n/a
4	ui_in[7:6] = osc_sel (select one of 4 ring oscillators)	n/a	n/a
5	n/a	n/a	n/a
6	n/a	n/a	n/a
7	n/a	n/a	n/a

RO-based temperature sensor with hysteresis [273]

- Author: Francisco Aguirre, Francisca Donoso, based on design by Daniel Arevalos and Jorge Marín
- Description: Ring oscillator whose frequency depends on temperature, with a hysteresis module for temperature detection.
- GitHub repository
- HDL project
- Mux address: 273
- Extra docs
- Clock: 1000000 Hz
- External hardware:

How it works

This temperature sensor uses a ring oscillator connected to a counter to determine the number of cycles within a clock period. The numbers of cycles are averaged across 4 samples, using a simple 2-bit right shift, and then sent through UART as well as the standard I/O pins. As a ring oscillator's frequency is related to the temperature, we can then use this output to determine the temperature vs frequency characteristic.

How to test

After reset and enable are set, the ring oscillator should start. You can then use the UART rx channel to send the START code, which will kick-start the data-sending module. This module will start outputting the average of the last 4 samples of the ring oscillator to the UART tx channel. These samples are taken every clock cycle, which you can adjust by providing an external clock through the external clock pin, and then using the selector pin to switch to it. We suggest using a clock around 1MHz, as otherwise the counter may either be too fast or too slow. Additionally, you can send a REG code to adjust the registers used by the hysteresis module, allowing you to set the upper threshold, and then the lower threshold, through transmission from the tx channel.

Pinout

#	Input	Output	Bidirectional
0	clk_internal	tx	counter[0]
1	clk_sel	temp_warn	counter[1]

#	Input	Output	Bidirectional
2	enable_inv_osc	n/a	counter[2]
3	enable_nand_osc	n/a	counter[3]
4	rx	n/a	counter[4]
5	osc_sel	n/a	counter[5]
6	n/a	n/a	counter[6]
7	n/a	n/a	counter[7]

Microrobotics FSM [274]

- Author: Lucas Irribarra, Felipe Rifo
- Description: Simple FSM for Micro-srobots
- GitHub repository
- HDL project
- Mux address: 274
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

A simple FSM controls the direction of a micro-robot based on sensors placed around the robot. The speed is driven by a PWM signal with an 8-bit resolution that can be set through input pins when the RESET pin is active.

The design also supports the use of an H bridge to allow the motors bidirectional rotation.

How to test

After reset, the PWM value should be setted by the input pins and the signal is on one of the output pins for testing. Experiment by changing the inputs to change the states of the FSM and the motor directions.

Pinout

#	Input	Output	Bidirectional
0	none	motor B left	pwm resolution bit 0
1	none	motor B Right	pwm resolution bit 1
2	none	motor A Left	pwm resolution bit 2
3	none	motor A Right	pwm resolution bit 3
4	none	0	pwm resolution bit 4
5	Front sensor	0	pwm resolution bit 5
6	Left sensor	0	pwm resolution bit 6
7	Rigt sensor	pwm signal	pwm resolution bit 7

MINI ALU [275]

- Author: Vicente Martinez, Cristobal Sanchez, Mauricio Pinto, Antar Derpich
- Description: This project is a Mini Alu with 4 bits, 2 bidirectionals used as inputs that indicates which operation is the Alu doing between XOR, OR, AND & addition
- GitHub repository
- Wokwi project
- Mux address: 275
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

A mini calculator with two bidirectionals that will change between which operations are being executed, then with the switches of the 4 bits A and the 4 Bits B they will define the values that will pass through the mini calculator resulting in 4 bits and in the case of addition a Carry. The bidirectionals bits are defined by: (00:XOR) (01:OR) (10:AND) (11:+)

How to test

First of all, Use the bidirectionals to define all the operations, then try all the possible responses of each operation and see if they fit what it should be.

Pinout

#	Input	Output	Bidirectional
0	A0	O0	S0
1	A1	O1	S1
2	A2	O2	none
3	A3	O3	none
4	B0	Carry	none
5	B1	none	none
6	B2	none	none
7	B3	none	none

PWM Quisquilloso [276]

- Author: Rebeca Orellana, Abigail Alarcon
- Description: Regulates the power or velocity at which a device functions. This PWM was designed to work with an extern clock of 12.5 KHz so the exit has a frequency of 50 Hz and it can control a servomotor.
- GitHub repository
- Wokwi project
- Mux address: 276
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Receives an 8 bits entrance, which the first is conected from a clock made with logicgates. The next bit is linked to the continuation of the clock and so on, with OR entrance that allow comparing the entry numbers until forming just one number written in binary that indicates the time. This has led lights connected in the output to check the clock account.

How to test

Enter a value at the entries between 0 and 255 to determinate the working cycle.

Pinout

#	Input	Output	Bidirectional
0	IN0	OUTC0	PWM
1	IN1	OUTC1	none
2	IN2	OUTC2	none
3	IN3	OUTC3	none
4	IN4	OUTC4	none
5	IN5	OUTC5	none
6	IN6	OUTC6	none
7	IN7	OUTC7	none

CPU 8 bit [277]

- Author: Daniel Arevalos, Patricio Carrasco, Mario Romero, Benjamin Villegas
- Description: Simple CPU of 8 bit
- GitHub repository
- HDL project
- Mux address: 277
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

a simple 8 bit cpu based on 8bitworkshop.

How to test

brief explanation

Pinout

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

A Risc-V Instruction memory i2c programmer [278]

- Author: Pablo Alonso
- Description: This proyect implements an i2c port capable of programing memory of a RISC-V processor
- GitHub repository
- HDL project
- Mux address: 278
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

The i2c port is capable of handle 4 registers:

- register 0: Set the instruction memory to read/write
- register 1: It loads the value of the instruction memory setted in register 0
- register 2: the desired value to be loaded to instruction memory setted in register 0
- register 3: Not used, yet read and write to it is possible.

How to test

Connect any controller with an i2c master port and next is how to read, or write the memory:

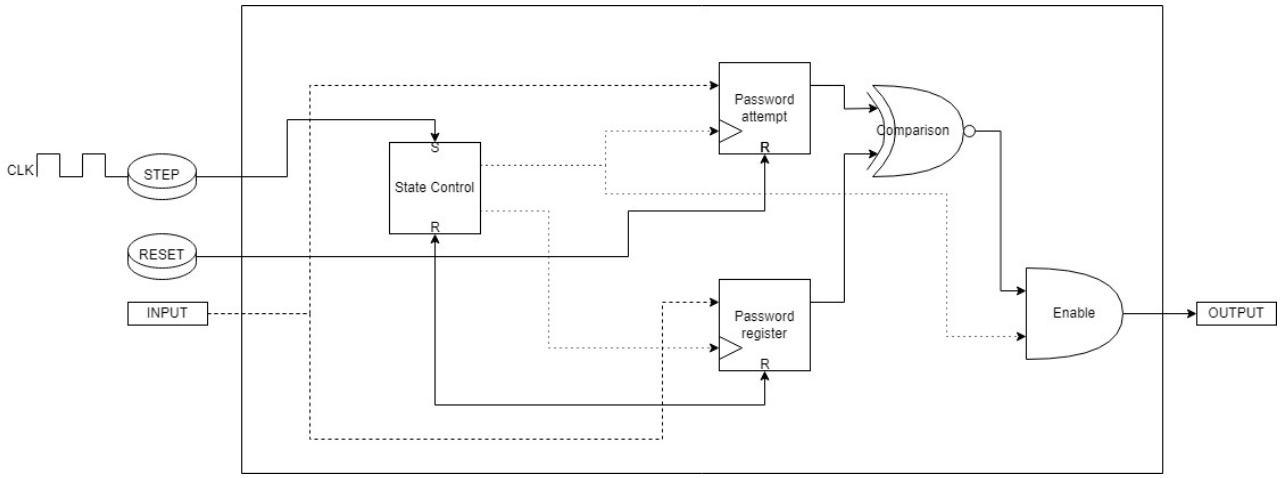
- To Read: MASTER: Start + device_addr [0x55] (7-bits) + master ACK + address_to_read + ACK | | STOP SLAVE: | 8bit_data_from_memory + ACK |

Pinout

#	Input	Output	Bidirectional
0	ext_sda_in	ext_sda_out	none
1	ext_scl_in	ext_scl_out	none
2	ext_i2c_rst	none	none
3	i2c_cs	none	none
4	pc_src	none	none

#	Input	Output	Bidirectional
5	none	none	none
6	none	none	none
7	sda_oe	none	none

IFSC 6-bit Locker [279]



- Author: Gabriel Mota, Luis Davi Kenig Paganella and Vinícius Westphal de Paula
- Description: A lock that receives a 6-bit entry combination.
- GitHub repository
- Wokwi project
- Mux address: 279
- Extra docs
- Clock: 0 Hz
- External hardware: requires two step buttons and some LED or display

How it works

The circuit has 6 input pins (representing a 6-bit password), 1 for the Step button (Clock of the circuit), and 1 for the Reset button. Each of the 6 inputs is connected to 2 registers, called password register and attempt register. Control between the state of registering a password and trying to enter a password is done by a latch connected with its output to one AND gate and its inverted output to another AND gate. When powering up the circuit, a reset must be performed first to ensure its correct operation. Then, it is possible to register the default password using the switches. The first clock pulse through the button registers the default password in 6 D-type flip-flops (DSR), which store this information in the circuit indefinitely until the Reset button is pressed. When the Step button is pressed again, the next clock signals update the other 6 DSR flip-flops in the attempt register. The comparison between each pair of flip-flops is done by an XOR gate with an inverter at the end, as it is necessary to compare when both flip-flops have the same logic state. To prevent the lock from being released when no password is registered (since in the initial state both sets of flip-flops would be cleared), an AND gate is inserted into a set of OR gates connected to the output of each "password register" flip-flop. In other words, the comparison between flip-flops will only be enabled when a password has been registered.

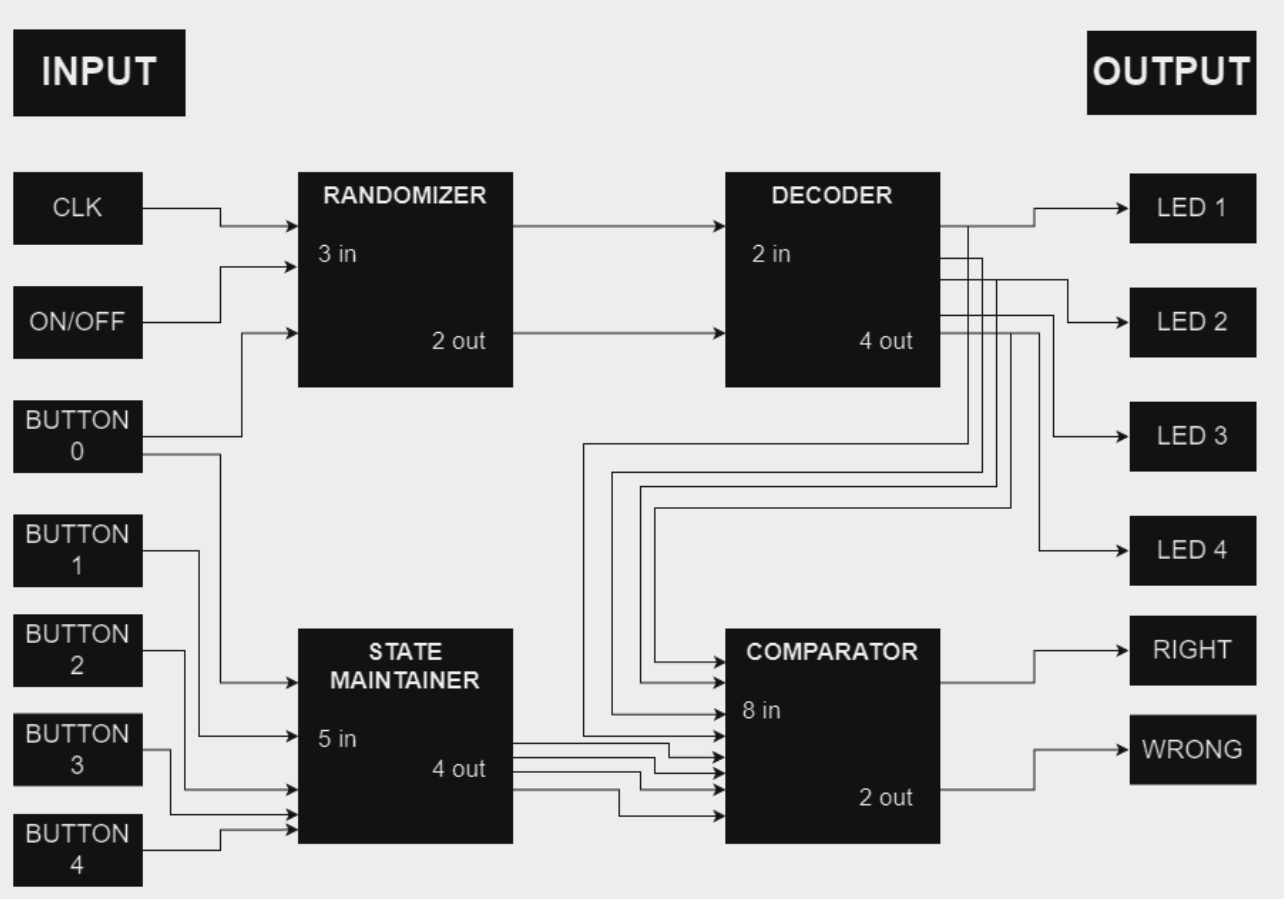
How to test

For the circuit to operate, it is necessary to use a Step button that connects to pin IN0 to VCC when pressed, and another Reset button in the same configuration, connected to pin IN1. The remaining inputs from IN2 to IN7 are connected to 6 switches linked to VCC. When powering up the circuit, the circuit is reset with the Reset button, allowing a password to be registered. The user must adjust the switches as desired to form a combination and then press the Step button to register the password. After that, the next times the Step button is pressed, comparisons are made between the original combination and the tested combination, and a logical signal is sent to the output pins that can be used as desired.

Pinout

#	Input	Output	Bidirectional
0	Step Button	Lock Exit State	none
1	Reset Button	Lock Exit State	none
2	Switch Entry 1	Lock Exit State	none
3	Switch Entry 2	Lock Exit State	none
4	Switch Entry 3	Lock Exit State	none
5	Switch Entry 4	Lock Exit State	none
6	Switch Entry 5	Lock Exit State	none
7	Switch Entry 6	Lock Exit State	none

Randomizer and status checker [288]



- Author: Thomas Linden, Karla Gabrielly Viana Nascimento, Maria Eduarda Amelco, Arthur Hasse
- Description: it randomizes a number between 0 and 3, a corresponding led will light up. After that will compare with a button pressed by the user and say if it was true or false
- GitHub repository
- Wokwi project
- Mux address: 288
- Extra docs
- Clock: 64 Hz
- External hardware:

How it works

General Description: The circuit is a random number generator, that will turn on a random LED between LED 1 and LED 4. Then, the player should press the button with the number corresponding to the LED that is on, and the circuit will say if it is

the RIGHT or WRONG button. The button 0 will make the circuit restart, generating another random number.

Blocks Description: When ON, the circuit will run a 4 bit counter, that will oscilate between 0 and 3 (binary). When the 0 button receives an input, the randomizer will select the number out of the counter, and send it to decoder, that will turn on one of the LEDs.

Now, the user have to press a button between 1 and 4, that correspondes to the LED that is on. This signal is sent to the comparator, that will cross the information with the LEDs that is on, and will sign if was pushed the right or the wrong button with the Right or Wrong LEDs.

As the goal is to use "no hold" buttons, the signal will be sent to the state maintainer, that will keep it in high level, so the right/wrong LEDs keep on. When the button 0 is pressed again, the state maintainer will reset it output.

How to test

In the inputs, the user will need to connect 5 momentary switch labeled from "Button 0" to "Button 4." The clock signal considered in CLK is 64 Hz, but any value above 20 Hz will work as expected.

On the ON/OFF input, a switch should be connected, or it can be driven directly to a high logic level.

On the outputs, 6 LEDs should be connected.

To start, the first step is to turn the switch to ON/OFF, and then press BUTTON 0.

One of the four LEDs will light up.The user should press the button corresponding to the LED that is on. If the correct button is pressed, the RIGHT LED will light up; if another button is pressed, the WRONG LED will light up.

Pinout

#	Input	Output	Bidirectional
0	clock	LED 1	none
1	on/off	LED 2	none
2	button 0	LED 3	none
3	button 1	LED 4	none
4	button 2	LED True	none

#	Input	Output	Bidirectional
5	button 3	LED False	none
6	button 4	none	none
7	seletor	none	none

Simulador de cruzamento de semáforo [289]

- Author: Gabriel Marcio Vieira, Renan Rosa Ferreira, Francisco Eduardo Gonçalves, Dayane Cassuriaga
- Description: Simulator of a traffic light at a two-way intersection with a pedestrian crosswalk.
- GitHub repository
- Wokwi project
- Mux address: 289
- Extra docs
- Clock: 1 Hz
- External hardware:

How it works

This circuit will use the chips internal clock and the PCBs low-level reset. The project in question belongs to the domain of traffic control systems and involves the implementation of a simulation of a traffic light using logical gate circuits. The simulation scenario takes place at an intersection composed of two main avenues, offering drivers the option to proceed straight or make a turn in their respective directions. In this context, four sets of traffic lights are present, but the control logic is applied to only two of them simultaneously. The system also incorporates a state dedicated to pedestrians, activated when all vehicle traffic lights display the red color, thus allowing safe pedestrian crossing. The overall operation of the system is based on logical combinations that determine each of the possible states

How to test

Testing involves observing each traffic light state without punctuation: hardware setup, initial states, input signals, output monitoring, pedestrian activation, and the cycle of states.

Pinout

#	Input	Output	Bidirectional
0	none	S1 Vermelho	none
1	none	S1 Amarelo	none
2	none	S1 Verde	none
3	none	S2 Vermelho	none

#	Input	Output	Bidirectional
4	none	S2 Amarelo	none
5	none	S2 Verde	none
6	none	Pedestres (PD) Vermelho	none
7	none	Pedestres (PD) Verde	none

Full_adder_carry_juang_garzons [290]

- Author: Juan Guillermo Garzón Sánchez
- Description: Es un sumador de 4 bits con carril el cual muestra el resultado mediante una pantalla de 7 segmentos, teniendo la limitación de que solo podrá mostrar resultados menores a 4 bits por lo tanto hasta 15 en decimal, ya que después de este tamaño estará en overflow , los dos números de entrada se introducen en los interruptores, el primer número va de los interruptores del 4 al 1 y el segundo número del 8 al 5
- GitHub repository
- Wokwi project
- Mux address: 290
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

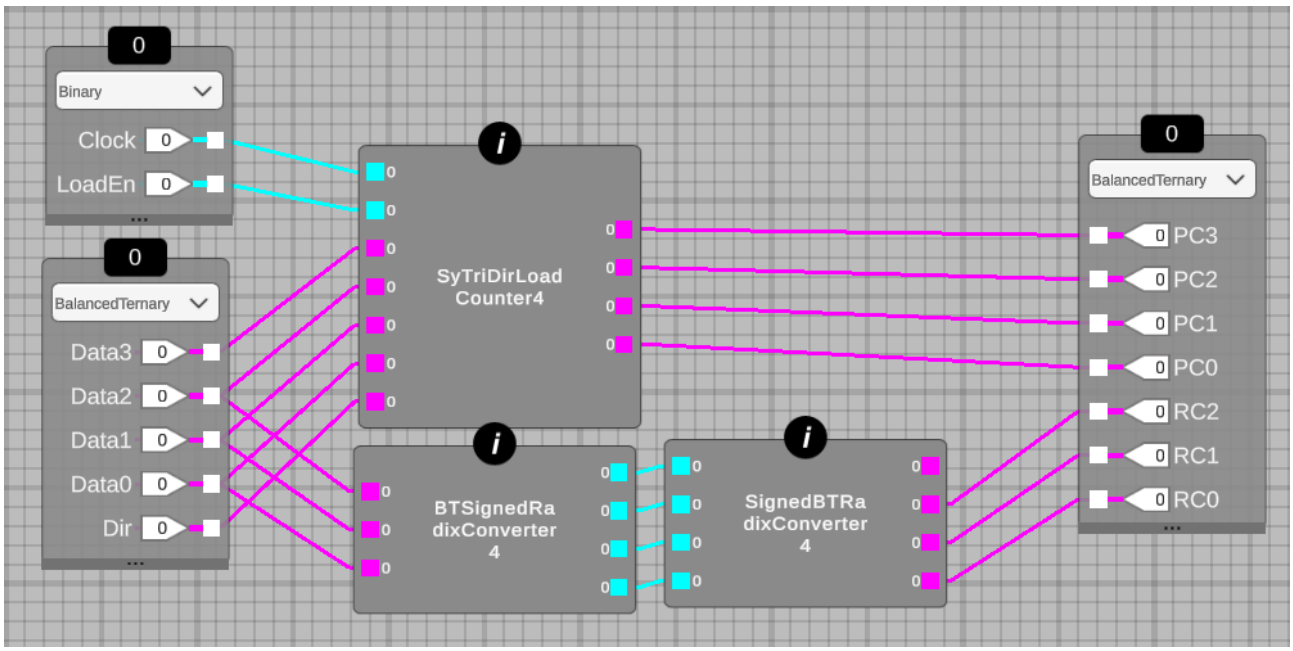
How to test

Explain how to test your project

Pinout

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

4-trit balanced ternary program counter and convertor [291]



- Author: Steven bos
- Description: A 4-trit synchronous balanced ternary (BT) program counter allowing tri-directional counting (up, down, hold) and jump/load the program counter. The other part is a 3-trit asyncronous BT radix converter.
- GitHub repository
- HDL project
- Mux address: 291
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

This design tests various aspects of the MRCS verilog generator in combination with the new pin capabilities of TT. It tests both a sync and async design and uses all the available pins including the bidirectional io. The program counter is the ternary version of the binary one submitted for TT2. It is scalable, loadable (needed for both initialization and jump) and can count up,down and hold. 4-trit BT counters have a range of -40 to 40. The second design has two radix converters: BT to signed binary and immediately followed by a signed binary to BT converter. The output is thus a copy of the input if within range. The design is 4-bit signed binary as intermediate format and has a range of -8 to 7. BT input should thus not be higher or the output

is wrong. The BT encoding is 2'b10 is logical 1, 2'b11 is logical 0, 2'b01 is logical -1 and 2'b00 is illegal.

How to test

The repository has a FPGA folder where a verilog testbench is included with test input. The screenshot shown in the readme is expected as output (a pyramid counting up and down). The design is also tested on the Basys3 FPGA. The FPGA version is only minor different than the verilog file for the ASIC as the data direction (bi-dir) pins are not needed for the FPGA. Inputs {data2,data1,data0} are used for the radix convertor chain and inputs {data3,data2,data1,data0} for the program counter.

Pinout

#	Input	Output	Bidirectional
0	clock	uo_out[7] PC data3 MST	{'uio_in[7]': 'data3 MST'}
1	reset	uo_out[6] PC data3	{'uio_in[6]': 'data3'}
2	ui_in[7] data2 MST (most significant trit)	uo_out[5] PC data2 MST	{'uio_out[5]': 'RC data2 MST'}
3	ui_in[6] data2	uo_out[4] PC data2	{'uio_out[4]': 'RC data2 MST'}
4	ui_in[5] data1 MST	uo_out[3] PC data1 MST	{'uio_out[3]': 'RC data1 MST'}
5	ui_in[4] data1	uo_out[2] PC data1	{'uio_out[2]': 'RC data1 MST'}
6	ui_in[3] data0 MST	uo_out[1] PC data0 MST	{'uio_out[1]': 'RC data0 MST'}
7	ui_in[2] data0	uo_out[0] PC data0	{'uio_out[0]': 'RC data0 MST'}

uDATAPATH_Collatz [292]

- Author: CMUA F.Segura-Quijano, J.S.Moya
- Description: uDATAPATH_Collatz
- GitHub repository
- HDL project
- Mux address: 292
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Our uDATAPATH_Collatz is a component that computes the Collatz Array Sequence for a given input number. The input and output are 8-bit vectors (data bus). The component is synchronous and has a reset signal to read the input data and perform the series calculation. The component implementation is based on a 4-register Datapath and ALU such that the implementation emulates basic assembler instructions programmed in a simple state machine.

How to test

To test the component, you must put a data on the input bus BB_SYSTEM_data_InBUS[7:0], activate the Reset signal. In the output bus BB_SYSTEM_data_OutBUS[7:0] you can see the calculated string. * The series will be calculated close to the clock rate. Visually you will not be able to see the intermediate data unless you use a probe reading per digital channel from an oscilloscope.

Pinout

#	Input	Output	Bidirectional
0	clk	BB_SYSTEM_data_OutBUS[0]	none
1	rst_n	BB_SYSTEM_data_OutBUS[1]	none
2	BB_SYSTEM_data_InBUS[0]	BB_SYSTEM_data_OutBUS[2]	none
3	BB_SYSTEM_data_InBUS[1]	BB_SYSTEM_data_OutBUS[3]	none
4	BB_SYSTEM_data_InBUS[2]	BB_SYSTEM_data_OutBUS[4]	none
5	BB_SYSTEM_data_InBUS[3]	BB_SYSTEM_data_OutBUS[5]	none
6	BB_SYSTEM_data_InBUS[4]	BB_SYSTEM_data_OutBUS[6]	none

#	Input	Output	Bidirectional
7	BB_SYSTEM_data_InBUS[5]	BB_SYSTEM_data_OutBUS[7]	none

Adder [293]

- Author: Juan David Prieto Garzon
- Description: 4-bit Adder
- GitHub repository
- Wokwi project
- Mux address: 293
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Our adder is a component that adds two 4-bit inputs. It was designed with FULL-ADDERS cells. The output is a 5-bit vector taking into account the last carry of the sum.

How to test

To test the adder we only have to put the values of the operands in the 8 input bits. It is a combinational system.

Pinout

#	Input	Output	Bidirectional
0	X0	S0	none
1	X1	S1	none
2	X2	S2	none
3	X3	S3	none
4	Y0	S4	none
5	Y1	n/a	none
6	Y2	n/a	none
7	Y3	n/a	none

Binary to 7 segment [294]

- Author: Juan S Moya & Fredy Segura
- Description: A simple binary to 7-segment decoder
- GitHub repository
- Wokwi project
- Mux address: 294
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

The binary to 7-segment LED decoder has four 1-bit inputs IN3, IN2, IN1, IN0, and seven 1-bit outputs OUT0, OUT1, OUT2, OUT3, OUT4, OUT5, OUT6, for controlling the seven segment of the LED display

How to test

Increase the inputs from 0000 to 1001 to display the numbers from 0 to 9, respectively.

Pinout

#	Input	Output	Bidirectional
0	IN3	OUT0 (segment a)	none
1	IN2	OUT1 (segment b)	none
2	IN1	OUT2 (segment c)	none
3	IN0	OUT3 (segment d)	none
4	none	OUT4 (segment e)	none
5	none	OUT5 (segment f)	none
6	none	OUT6 (segment g)	none
7	none	none	none

Neuron [295]

- Author: David Leonardo Caro Estepa
- Description: Artificial Neuron with 2 inputs of 2 bits
- GitHub repository
- Wokwi project
- Mux address: 295
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Our Neuron is a component that emulates the behavior of an artificial neuron. It is a simple demonstration with 2 inputs of 2 bits that emulate the inputs of the neuron and 2 inputs of 2 bits that emulate the corresponding weights. These weights would simulate values that have already been calculated, that is, the inference state. As an activation function, a small test with a linear comparator was proposed. The output of the neuron is one or zero depending on the value of the inputs and the weights.

How to test

To test the neuron, values must be placed on the 2 2-bit inputs and on the two weight values. The system generates an output at 1 or 0 that represents activation or not of the neuron.

Pinout

#	Input	Output	Bidirectional
0	X0	Y	none
1	X1	n/a	none
2	Y0	n/a	none
3	Y1	n/a	none
4	WX0	n/a	none
5	WX1	n/a	none
6	WY0	n/a	none
7	XY1	n/a	none

Later [304]

- Author: Alejandro Silva
- Description: Later
- GitHub repository
- HDL project
- Mux address: 304
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Later

How to test

Later

Pinout

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

serializer [305]

- Author: Sergio Alejandro Rosales Nuñez
- Description: shift register
- GitHub repository
- HDL project
- Mux address: 305
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	clk	Q	none
1	rst	eos	none
2	D[0]	none	none
3	D[1]	none	none
4	D[2]	none	none
5	D[3]	none	none
6	D[4]	none	none
7	D[5]	none	none

4-bits 1-channel PWM and ALU 4 bits [306]

- Author: Alonso
- Description: This is a 4-bits and 1-channel PWM module
- GitHub repository
- HDL project
- Mux address: 306
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Set ui_in[7:0] as input

How to test

Read u0_out to get the output

Pinout

#	Input	Output	Bidirectional
0	ui_in[7]	uo_out[4]	none
1	ui_in[6]	uo_out[3]	none
2	ui_in[5]	uo_out[2]	none
3	ui_in[4]	uo_out[1]	none
4	ui_in[3]	uo_out[0]	none
5	ui_in[2]	n/a	none
6	ui_in[1]	n/a	none
7	ui_in[0]	n/a	none

up-down counter with parallel load and BCD output [307]

- Author: Diego Hernán Gaytán Rivas
- Description: This device is an up-down counter for numbers ranging from zero to fifteen, with options for enable, clear count, and parallel loading. The count updates at a rate of one second per increment or decrement.
- GitHub repository
- HDL project
- Mux address: 307
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

The design operates by receiving a 10 MHz clock signal and using a frequency divider composed of a counter and a comparator to generate a 1 Hz output signal. This 1 Hz signal is used to feed an up-down counter circuit, which includes a synchronous enable to halt the count, a port to clear the count by setting the output to zero, and a port to indicate the desire to load a value directly from the circuit's inputs. The value is loaded in the next clock cycle, and the count continues from that point. Finally, the count value is decoded into BCD code for display on a seven-segment display.

How to test

After resetting and setting the "en" port high, the device will begin a hexadecimal count with values ranging from 0 to F. Now, if the "up" port is set high, the count will be in ascending order, whereas if it's set low, it will be in descending order. Additionally, there is an option to clear the count by raising the "syn_clear" port, which would reset the counter to zero. In the case where you want to load a value between 0 and F into the counter to start the count, you should set the "load" port high and then lower it to continue the count. The value to load will be taken directly from the first four input ports.

Pinout

#	Input	Output	Bidirectional
0	data in bit 0	segment a	none
1	data in bit 1	segment b	none

#	Input	Output	Bidirectional
2	data in bit 2	segment c	none
3	data in bit 3	segment d	none
4	enable	segment e	none
5	syn_clear	segment f	none
6	up	segment g	none
7	load	dot	none

Later [308]

- Author: Ciro Bermudez
- Description: Later
- GitHub repository
- HDL project
- Mux address: 308
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Later

How to test

Later

Pinout

#	Input	Output	Bidirectional
0	none	none	none
1	none	none	none
2	none	none	none
3	none	none	none
4	none	none	none
5	none	none	none
6	none	none	none
7	none	none	none

Contador con carga [309]

- Author: Cristian Torres
- Description: Es un contador del 0 al 16 con carga
- GitHub repository
- HDL project
- Mux address: 309
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	clk	sal	none
1	rst	segment b	none
2	ini	segment c	none
3	ent	segment d	none
4	none	segment e	none
5	none	segment f	none
6	none	segment g	none
7	none	dot	none

onehot_decoder [310]

- Author: Martin Gonzalez
- Description: this module is a 3bits onehot decoder
- GitHub repository
- HDL project
- Mux address: 310
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	in[0]	out[0]	none
1	in[1]	out[1]	none
2	in[2]	out[2]	none
3	none	out[3]	none
4	none	out[4]	none
5	none	out[5]	none
6	none	out[6]	none
7	none	out[7]	none

CDMA Transmitter/Receiver [311]

- Author: Santiago Robledo Acosta
- Description: This is a CDMA Transmitter/Reciever to academically study the Spread-Spectrum effect while sending singals and to observe pseudonoise
- GitHub repository
- HDL project
- Mux address: 311
- Extra docs
- Clock: 10000000 Hz
- External hardware: OPAM, Testboard, LED

How it works

This is a very simple circuit, it consists in two LSFR register lineary connecated in an specific way in order to generate two m-sequences, in order to generate this pair of m-sequences, we input an initial vaule called seed, this is because we cannot generate a PN signal if the LSFR registers have an initial value of 0s. For this design we used two LSFR with 5 D Flip-Flops. With this we can generate a PN signal with a length of(2^5)-1. With this PN signal and modulus 2 adding the signal we want to tranmit, we generate a CDMA signal, which we are going to study simulating a channel with an OPAM in order to add noise to the CDMA and feed it back to the designed circuit to see the bit-error rate of this device. With this we hope to study and put to test.

- CDMA
- Gold Sequences.
- The effect of the noise in the CDMA.
- Reception process with a simulated channel.
- Apply the knowledge aquired within the Latinpractice Bootcamp initiative and apply the knowledge to design and print in a silicon waffle the proposed device.

How to test

As we used a hardware description language (Verilog), we created an specific testbench for the cdma.v, this testbench simply initializes the input signals to 0 in order to generate the adequate signal such as the clock, a test signal to send that lasts 31 clock cycles (Spread-spectrum), a seed value to load both LSFR The stimulus simple will assign a value to set_i and deactivate it to load the seed, after that, the LSFR have a linear feedback and will contantly generate the Gold signal each clock cycle. With the test signal_i we the system will generate the CDMA signal and assign it to CDMA_o, the Gold signal is assigned to Gold_o, this is the tranmition process.

For the reception process, we simply assign the value of CDMA_o to receptor_i and the output will be observerd at receptor_o, we can observe that we recovered signal_i as it shows the same time diagram as receptor_o.

As for the LED_o it works as a simple indicator that the inputed seed is valid for transmission.

The template will include the verilog file with its testbench.

Pinout

#	Input	Output	Bidirectional
0	signal_i	cdma_o	none
1	seed_i[0]	gold_o	none
2	seed_i[1]	receptor_o	none
3	seed_i[2]	led_o	none
4	seed_i[3]	none	none
5	seed_i[4]	none	none
6	receptor_i	none	none
7	load_i	none	none

clock divider [320]

- Author: Uriel jaramillo
- Description: divide the clock
- GitHub repository
- HDL project
- Mux address: 320
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	none	cout1	none
1	none	cout2	none
2	none	cout3	none
3	none	cout4	none
4	none	cout5	none
5	none	cout6	none
6	none	cout7	none
7	none	cout8	none

reciprocal [321]

- Author: raul pacheco rodriguez
- Description: module reciprocal
- GitHub repository
- HDL project
- Mux address: 321
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	in	out	none
1	none	none	none
2	none	n/a	none
3	none	n/a	none
4	none	n/a	none
5	none	n/a	none
6	none	n/a	none
7	none	n/a	none

Later [322]

- Author: Fabian
- Description: Later
- GitHub repository
- HDL project
- Mux address: 322
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

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

Time Multiplexed Nand-gate [323]

- Author: Frans Skarman
- Description: The furthest you can go in the time/space tradeoff
- GitHub repository
- HDL project
- Mux address: 323
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

TODO

How to test

TODO

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

Octal classifier [324]

- Author: Eduardo Zurek, Margarita Narducci, Diana Rueda
- Description: Classifies images of numbers from 0 to 7 using the 16 most significant pixels
- GitHub repository
- HDL project
- Mux address: 324
- Extra docs
- Clock: 10000000 Hz
- External hardware: N.A

How it works

The system classifies images of numbers from 0 to 7 using the 16 most significant pixels The system´s input or features are the 16 most significant pixels of an 8x8 image with a number from 0 to 7. The output is shown in the 7 segments display.

How to test

In order to emulate an image in the input, the features must be set to one or zero according to the number of the image Example of the image of a zero: feature_10=0 feature_13=1 feature_18=0 feature_19=1 feature_20=0 feature_21=0 feature_26=0 feature_27=1 feature_28=0 feature_34=0 feature_36=0 feature_42=0 feature_43=1 feature_45=0 feature_60=0 feature_61=1

Pinout

#	Input	Output	Bidirectional
0	feature_10 = ui_in[0];	segment a	feature_28 = uio_in[0];
1	feature_13 = ui_in[1];	segment b	feature_34 = uio_in[1];
2	feature_18 = ui_in[2];	segment c	feature_36 = uio_in[2];
3	feature_19 = ui_in[3];	segment d	feature_42 = uio_in[3];
4	feature_20 = ui_in[4];	segment e	feature_43 = uio_in[4];
5	feature_21 = ui_in[5];	segment f	feature_45 = uio_in[5];
6	feature_26 = ui_in[6];	segment g	feature_60 = uio_in[6];
7	feature_27 = ui_in[7];	dot	feature_61 = uio_in[7];

MULDIV unit (4-bit signed/unsigned) [325]

- Author: Darryl Miles
- Description: Combinational Multiply and Divide Unit (signed and unsigned)
- GitHub repository
- HDL project
- Mux address: 325
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Combinational multiply / divider unit (no clock in use)

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 4-bit) * B (multiplicand 4-bit) Expect result P (product 8-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 4-bit) / Dsor (divisor 4-bit) Expect result Q (quotient 4-bit) with R (remainder 4-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.

The project was sketched out and tested with logisim-evolution https://github.com/logisim-evolution/logisim-evolution then exported direct to verilog (as if it was for a FPGA development board using the built in process).

How to test

Setup the input state expect immediate output (after gate propagation delays).

Pinout

#	Input	Output	Bidirectional
0	MUL A[0], DIV Dend[0]	MUL P[0], DIV Q[0]	(unused)
1	MUL A[1], DIV Dend[1]	MUL P[1], DIV Q[1]	(unused)
2	MUL A[2], DIV Dend[2]	MUL P[2], DIV Q[2]	(unused)
3	MUL A[3], DIV Dend[3]	MUL P[3], DIV Q[3]	(unused)
4	MUL B[0], DIV Dsor[0]	MUL P[4], DIV R[0]	DIV error overflow (output only)
5	MUL B[1], DIV Dsor[1]	MUL P[5], DIV R[1]	DIV error divide-by-zero (output only)
6	MUL B[2], DIV Dsor[2]	MUL P[6], DIV R[2]	OPSIGNED mode (input only)
7	MUL B[3], DIV Dsor[3]	MUL P[7], DIV R[3]	MULDIV mode (input only)

RS Write Decodifier [326]

- Author: Francisco Javier Rodriguez Navarrete
- Description: Project for the 12 bit reservation station decodifier
- GitHub repository
- HDL project
- Mux address: 326
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

It works via selecting the corresponding RS needed

How to test

Manupulate the input switches to see the decodifier outputs

Pinout

#	Input	Output	Bidirectional
0	in_rs_write[7]	out_rs_write[7]	in_rs_write[11]
1	in_rs_write[6]	out_rs_write[6]	in_rs_write[10]
2	in_rs_write[5]	out_rs_write[5]	in_rs_write[9]
3	in_rs_write[4]	out_rs_write[4]	in_rs_write[8]
4	in_rs_write[3]	out_rs_write[3]	out_rs_write[11]
5	in_rs_write[2]	out_rs_write[2]	out_rs_write[10]
6	in_rs_write[1]	out_rs_write[1]	out_rs_write[9]
7	in_rs_write[0]	out_rs_write[0]	out_rs_write[8]

Password FSM [327]

- Author: Francisco Javier Rodriguez Navarrete
- Description: Project for the Password FSM
- GitHub repository
- HDL project
- Mux address: 327
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

This is a FSM that has a hard coded password that the user has to guess.

How to test

The password is 3044238

Pinout

#	Input	Output	Bidirectional
0	iv_data[3]	o_acknowledge[7]	none
1	iv_data[2]	o_acknowledge[6]	none
2	iv_data[1]	o_acknowledge[5]	none
3	iv_data[0]	o_acknowledge[4]	none
4	N/C	o_acknowledge[3]	none
5	N/C	o_acknowledge[2]	none
6	i_send_data	o_acknowledge[1]	none
7	i_CE	o_acknowledge[0]	none

Priority e [336]

- Author: Juan Carlos Garcia Lopez
- Description: 7 SEGMENTS CLOCK
- GitHub repository
- HDL project
- Mux address: 336
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	data_in	data_out	none
1	n/a	Valid	none
2	n/a	n/a	none
3	n/a	n/a	none
4	n/a	n/a	none
5	n/a	n/a	none
6	n/a	n/a	none
7	n/a	n/a	none

frecuencimeter [337]

- Author: Juan Carlos Garcia Lopez and Emilio Isaac Baungarten Leon
- Description: frecuencimeter
- GitHub repository
- HDL project
- Mux address: 337
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	data_in	segments_	none
1	n/a	disp_select_	none
2	n/a	segment_select_	none
3	n/a	n/a	none
4	n/a	n/a	none
5	n/a	n/a	none
6	n/a	n/a	none
7	n/a	n/a	none

lfsr random number generator [338]

- Author: Arun A V
- Description: 4-bit Linear Feedback Shift Register with configurable feedback polynomials based on the `mod` input, and it resets to the initial state when `reset` is asserted.
- GitHub repository
- HDL project
- Mux address: 338
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

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

Puts the bottom 8 bits of the counter on the bidirectional outputs.

With all the inputs set to 0, the internal 24 bit compare is set to 10,000,000. This means the counter will increment by one each second.

If any inputs are non zero, then the input will be used as an bits 11 to 18 of the 24 bit compare register. Example: setting the inputs to 00010000 will program 16384 into the compare register. With a 10MHz clock the counter will increment ~610 times per second.

How to test

After reset, the counter should increase by one every second with a 10MHz input clock. Experiment by changing the inputs to change the counting speed.

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

#	Input	Output	Bidirectional
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

i2c_6 bits [339]

- Author: Sergio Alejandro Rosales Nuñez
- Description: i2c address 0x04
- GitHub repository
- HDL project
- Mux address: 339
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

Pinout

#	Input	Output	Bidirectional
0	clk	data_from_master[0]	sda_in
1	rst	data_from_master[1]	sda_out
2	scl	data_from_master[2]	data_to_master[0]
3	none	data_from_master[3]	data_to_master[1]
4	none	data_from_master[4]	data_to_master[2]
5	none	data_from_master[5]	data_to_master[3]
6	none	ctrl	data_to_master[4]
7	none	none	data_to_master[5]

Fastest Finger [340]

- Author: Chris Burton
- Description: Shows which button was pressed first
- GitHub repository
- Wokwi project
- Mux address: 340
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

If when a button is pressed the output for all other buttons is low, the output is set high.

How to test

Connect buttons up to IN0-IN7 and whichever one is pressed first will set the corresponding OUT0-OUT7 high.

Pinout

#	Input	Output	Bidirectional
0	Button 0	LED 0 / segment a	none
1	Button 1	LED 1 / segment b	none
2	Button 2	LED 2 / segment c	none
3	Button 3	LED 3 / segment d	none
4	Button 4	LED 4 / segment e	none
5	Button 5	LED 5 / segment f	none
6	Button 6	LED 6 / segment g	none
7	Button 7	LED 7 / dot	none

Fastest Finger (Clocked) [341]

- Author: Chris Burton
- Description: Shows which button was pressed first
- GitHub repository
- Wokwi project
- Mux address: 341
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

The 3-bit counter increments from the clock input, this then goes through a 3-8 decoder. If the output of the decoder is high and it's corresponding INx is high, OUTx is set high and the clock signal to the adder is interrupted.

How to test

Connect buttons up to IN0-IN7 and whichever one is pressed first will set the corresponding OUT0-OUT7 high.

Pinout

#	Input	Output	Bidirectional
0	Button 0	LED 0 / segment a	none
1	Button 1	LED 1 / segment b	none
2	Button 2	LED 2 / segment c	none
3	Button 3	LED 3 / segment d	none
4	Button 4	LED 4 / segment e	none
5	Button 5	LED 5 / segment f	none
6	Button 6	LED 6 / segment g	none
7	Button 7	LED 7 / dot	none

Oscillators II [342]

- Author: Mikhail Svarichevsky
- Description: Free-running oscillators to verify simulation vs reality + TRNG
- GitHub repository
- HDL project
- Mux address: 342
- Extra docs
- Clock: 1000000 Hz
- External hardware:

How it works

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

How to test

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

Pinout

#	Input	Output	Bidirectional
0	unused	generated clock	none
1	unused	clock divided by 2^1	none
2	shift register clk	clock divided by 2^2	none
3	shift register data	clock divided by 2^3	none
4	clock source id_0	clock divided by 2^4	none
5	clock source id_1	clock divided by 2^9	none
6	clock source id_2	TRNG output	none
7	unused	Bit 11 of shift register	none

Simple ALU [343]

- Author: Rebot449
- Description: A simple ALU with only 6 instructions
- GitHub repository
- HDL project
- Mux address: 343
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Based off the Overture Architecture found in the game "Turing Complete"

The ALU has only 6 instructions which are decided by the 3 least significant bits. xxxxx000: Logical OR xxxxx001: Logical NAND xxxxx010: Logical NOR xxxxx011: Logical AND xxxxx100: Addition of Data_0 + Data_1 xxxxx101: Subraction (Data_1 - Data_0)

How to test

The ALU should perform the logical and arithmetic operations as stated by the coressponding instructions above.

Pinout

#	Input	Output	Bidirectional
0	i_instruction //ALU instruction, 8 bit wide input	o_result //ALU data output, 8 bit wide	none
1	i_data_0 // Data input 0, 8 bit wide input	n/a	none
2	i_data_1 // Data input 1, 8 bit wide input	n/a	none
3	n/a	n/a	none
4	n/a	n/a	none

#	Input	Output	Bidirectional
5	n/a	n/a	none
6	n/a	n/a	none
7	n/a	n/a	none

TinyTapeout 04 Loopback Test Module [352]

- Author: Sylvain Munaut
- Description: Loopback test module
- GitHub repository
- HDL project
- Mux address: 352
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Input 0 goes to output 0 through 6. Output 7 is input4 & input5 & input6 & input7

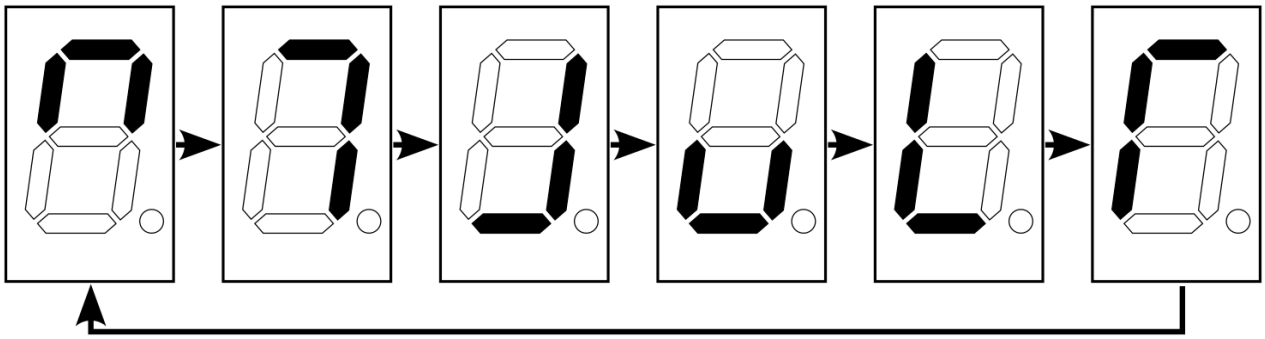
How to test

Toggle input 0, measure the time for output 0 to change.

Pinout

#	Input	Output	Bidirectional
0	in0	mirrors in0	none
1	none	mirrors in0	none
2	none	mirrors in0	none
3	none	mirrors in0	none
4	in4	mirrors in0	none
5	in5	mirrors in0	none
6	in6	mirrors in0	none
7	in7	the value of in4 & in5 & in6 & in7	none

Adjustable Frequency LED Chaser [353]



- Author: Daniel Teal
- Description: Animates a seven-segment display given any clock
- GitHub repository
- Wokwi project
- Mux address: 353
- Extra docs
- Clock: any Hz
- External hardware:

How it works

This is a simple LED chaser animation for a seven-segment display, but a configurable bank of clock dividers allows the animation to run at human speeds (1-10Hz) for all input clock frequencies. The intent is to support a wide range of testing situations: as long as there's any clock and display whatsoever, you get a pretty demo.

It can also be used as a frequency counter in a pinch.

How to test

The chip clock input should be connected to any clock source of nonzero frequency. Examples include a manual tactile switch, a kHz clock source, or the maximum MHz source supported by the chip. The reset input is not used. Standard inputs 0–4 can be set high or low and are ideally adjustable, e.g., via a DIP switch array. These inputs may safely be toggled high and low while the chip is running. Inputs 5–7 are not used. Standard outputs 0–5 should be connected in order to the perimeter elements of a seven-segment display or equivalent. Outputs 6–7 are not connected internally (and so may be connected to a display anyway with no effect). The bidirectional IO is not used.

By default, a new frame of the 6-frame LED chaser pattern is displayed on outputs 0 through 5 every rising clock edge. Inputs 0 through 4 toggle clock dividers of 2x (2^1), 4x (2^2), 16x (2^4), 256x (2^8), and 65536x (2^{16}) respectively. These dividers stack multiplicatively for a maximum clock division of $2^31 = 2.15e9$, which easily slows the maximum expected 50MHz clock to sub-Hertz frequencies.

Combine a subset of the clock dividers for fine adjustment. For example, one might find the original clock signal makes the animation far too fast to see, applying the 65536x divider makes it too slow, and the combination of 256x and 16x dividers (for a total 4096x) works well. Clock division factors are intentionally chosen to be able to express any power of two.

Finally, note the first six frames after chip startup may be partially incorrect due to noise. Later frames should be correct.

Pinout

#	Input	Output	Bidirectional
0	enable 2x (2^1) divider	segment a	n/c
1	enable 4x (2^2) divider	segment b	n/c
2	enable 16x (2^4) divider	segment c	n/c
3	enable 256x (2^8) divider	segment d	n/c
4	enable 65536x (2^{16}) divider	segment e	n/c
5	n/c	segment f	n/c
6	n/c	n/c	n/c
7	n/c	n/c	n/c

Simple QSPI DAC [354]

- Author: Piotr Kuligowski
- Description: Simple implementation of 8-bit R-2R DAC using QSPI interface + bi-directional port test breakout.
- GitHub repository
- Wokwi project
- Mux address: 354
- Extra docs
- Clock: 0 Hz
- External hardware: To test the DAC you need to remove 7seg display and connect external R-2R network to implement DAC.

How it works

After reset, it is a very simple QSPI-based, 8-bit DAC (digital to analog converter). You feed it with 4-bit halves, which are then read on each rising edge of CLK. It also breakouts UIO's pins to test it externally (delays, electrical characteristics, etc).

How to test

After power up set nCS high, set RST_N low, then set RST_N high. If you want to set a new DAC value, first set nCS low, then set 4-bits, beginning from the lowest half. Now trigger a rising edge of CLK, set highest 4-bits and trigger another rising edge of CLK. Now you can continue setting next bytes (divided into 4-bit halves) or set nCS high and start from the beginning, by setting nCS low. You can observe effects on the 7seg display or remove it and implement R-2R network to have the acutal DAC. There is also a break out of bi-directional ports to test them externally.

Pinout

#	Input	Output	Bidirectional
0	DATA0	OUT0	INTERMEDIATE0
1	DATA1	OUT1	INTERMEDIATE1
2	DATA2	OUT2	INTERMEDIATE2
3	DATA3	OUT3	INTERMEDIATE3
4	nCS	OUT4	TESTED_UIO
5	none	OUT5	TESTED_UIO_IN
6	none	OUT6	TESTED_UIO_OUT

#	Input	Output	Bidirectional
7	none	OUT7	TESTED_UIO_OE

AQALU [355]

- Author: Artin Ghanaatpisheh-Sanani and Quardin Lyttle
- Description: 2 bit ALU with 4 Bit Opcode
- GitHub repository
- HDL project
- Mux address: 355
- Extra docs
- Clock: 10000000 Hz
- External hardware: an external circuit with LEDs or a BCD converter to represent the numerical outputs

How it works

4 bit Op Code ALU. OpCodes are as follows-

- 0000 AND
- 0001 OR
- 0010 NOT
- 0011 XOR
- 0100 NAND
- 0101 NOR
- 0110 XNOR
- 0111 Addition
- 1000 Subtraction
- 1001 Multiplication
- 1010 Compare
- 1011 Shift L Logically
- 1100 Shift R Logically
- 1101 Shift L Arithmetically
- 1110 Shift R Arithmetically
- 1111 Running Sum

Running Sum takes the current 4bit number at the input and continously adds it to the output every second.

Compare is 2'b10 when A is greater than B, 2'b01 when B is greater than A. 2'b11 when equal.

NOT treats A and B as a combined 4bit input.

Subtraction anticipates am unsigned number input (treats them as positive numbers essentially) however will give signed output depending on the operation. It acts as A-B.

How to test

Testing can be done by connecting the outputs to LEDs and treating them as an 8 bit output. Seeing how they correspond to the selected OpCodes and inputs.

Pinout

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

Simple TMR [356]

- Author: Piotr Kuligowski
- Description: Simple TMR (triple modular redundancy) voters with error injection option.
- GitHub repository
- Wokwi project
- Mux address: 356
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

It is a simple implementation of one D-flip-flop, three D-flip-flops with TMRed outputs and a single TMRed D-flip-flop that allows to inject errors.

How to test

Set inputs, trigger clock and observe outputs

Pinout

#	Input	Output	Bidirectional
0	FF0_D_SINGLE	Q0_SINGLE	none
1	FF1_D_TMRed_WITH_INJECTION	FF1_OUT0	none
2	FF2_D_TMRed_OR_INJECT_IN0_OF_D1	FF1_OUT1	none
3	FF3_D_TMRed_OR_INJECT_IN1_OF_D1	FF1_OUT2	none
4	FF4_D_TMRed_OR_INJECT_IN2_OF_D1	FF1_TMRed_OUT	none
5	FF1_FF2_FF3_FF4_EN	FF2_TMRed_OUT	none
6	FF1_INJECT	FF3_TMRed_OUT	none
7	CLK	FF4_TMRed_OUT	none

Poor Person's Boundary Scan [357]

- Author: Verneri Hirvonen
- Description: JTAG test logic with a 8-bit TDR for 'external' boundary scan
- GitHub repository
- HDL project
- Mux address: 357
- Extra docs
- Clock: Hz
- External hardware: JTAG debug adapter

How it works

The design contains JTAG test logic with four test data registers:

1. `boundary_scan`
2. `blink_in`
3. `blink_out`
4. `blink_dir`

boundary_scan is connected between fixed input and output registers and can be used to either read the input pins or drive the output pins.

blink_in, blink_out, and blink_dir are connected to pins 6 and 7 of the bidirectional IO and can be used to read/write the bidirectional pins.

The default clk and rst_n are left unconnected because the JTAG interface provides its own clock and reset and those are routed through user IO.

How to test

The cocotb simulation contains helper routines for driving the JTAG state machine and can be used as a example of how to shift values into different JTAG internal registers.

In general the design is controlled with the following procedure:

1. `Reset core either through the TRSTn input or through a reset sequence`
2. `Shift in IR value to select appropriate TDR`
3. `Shift bits to TDR to read/write its value.`

Blinky pins are controlled with the following JTAG TDRs. Each TDR is two bits wide.

```
- blink_in  IR=0x02 = blink pin inputs
- blink_out IR=0x03 = blink pin outputs
- blink_dir IR=0x04 = blink pin directions, high = output, low = input
```

For example, to assign blink[0] high and blinky[1] low, execute the following sequence:

```
1. Shift 0x04 to IR to select blink_dir TDR
2. Shift 0b11 to the selected TDR to set both pins as outputs
3. Shift 0x02 to IR to select blink_in TDR
4. Shift 0b01 to set blink[0] to high and blink[1] to low
```

Pinout

#	Input	Output	Bidirectional
0	boundary_scan input 0	boundary_scan output 0	JTAG TCK (hardcoded input)
1	boundary_scan input 1	boundary_scan output 1	JTAG TMS (hardcoded input)
2	boundary_scan input 2	boundary_scan output 2	JTAG TDI (hardcoded input)
3	boundary_scan input 3	boundary_scan output 3	JTAG TRSTn (hardcoded input)
4	boundary_scan input 4	boundary_scan output 4	JTAG TDO (hardcoded output)
5	boundary_scan input 5	boundary_scan output 5	JTAG state machine in state Test-Logic Reset (hardcoded output)
6	boundary_scan input 6	boundary_scan output 6	Blink pin 0 (bidirectional)
7	boundary_scan input 7	boundary_scan output 7	Blink pin 1 (bidirectional)

Probador de lógica básico [358]

- Author: Felipe R. Serrano Domínguez
- Description: It allows to validate the operation of basic logic devices; gates and flip-flops individually
- GitHub repository
- Wokwi project
- Mux address: 358
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Through common inputs and different outputs for every gate; AND, NAND, OR, NOR, NOT and XOR can be observe the logic states. It also allows you to test the different types of basic flip flops D, RS and JK

How to test

To test logic gates all inputs are common (A,B). The outputs for each gate are addressed to different ports.
To test the flip flops must be selected using the selection pins (6,7). pin 6 enables the JK or the rest. pin 7 states enable D or SR so only one flip flop can be tested at a time. It is expected to be able to validate its basic operation by means of truth tables.

Pinout

#	Input	Output	Bidirectional
0	CLK (FLIP FLOPS)	OUT 0 (GATE AND)	none
1	RST (FLIP FLOPS)	OUT 1 (GATE NAND)	none
2	IN0 (INPUT A GATE)	OUT 2 (GATE OR)	none
3	IN1 (INPUT B GATE)	OUT 3 (GATE NOR)	none
4	IN2 (INPUT J FLIP FLOP)	OUT 4 (GATE NOT)	none
5	IN3 (INPUT K FLIP FLOP)	OUT 5 (GATE XOR)	none
6	IN4 (SET FLIP FLOP)	OUT 6 (FLIP FLOP Q)	none
7	IN5 (D FLIP FLOP)	OUT 7 (FLIP FLOP Q')	none

LIF Neuron, Telluride 2023 [359]

- Author: Paola Vitolo, Andrew Wabnitz, ReJ aka Renaldas Zioma
- Description: Standalone test for a Binarized Leaky Integrate and Fire neuron that is part of the larger experimental design from Telluride Neuromorphic Workshop 2023
- GitHub repository
- HDL project
- Mux address: 359
- Extra docs
- Clock: 10000000 Hz
- External hardware:

How it works

Binarized Leaky Integrate and Fire (LIF) neuron supports binary [0/1] inputs and [-1/1] binarized weights. Inputs are multiplied by weights and accumulated on the internal membrane. Membrane is exponentially decaying with every clock cycle. Once membrane value (potential) reaches threshold, neuron spikes and membrane value is decreased.

```
membrane += inputs * weights
membrane *= decay_factor
membrane -= threshold if membrane > threshold
spike = 1 if membrane > threshold
```

How to test

While reset is held high, input values are assigned to weights After reset cycle neuron is active. The only output of the neuron is the binary spike!

Pinout

#	Input	Output	Bidirectional
0	input1 / weight1	spike	none
1	input2 / weight2	none	none
2	input3 / weight3	none	none
3	input4 / weight4	none	none
4	input5 / weight5	none	none

#	Input	Output	Bidirectional
5	input6 / weight6	none	none
6	input7 / weight7	none	none
7	input8 / weight8	none	none

rusty_adder [368]

- Author: Kevin Webb
- Description: A (tiny) 8 bit adder built using RustHDL
- GitHub repository
- HDL project
- Mux address: 368
- Extra docs
- Clock: 10000 Hz
- External hardware:

How it works

Adds two 8 bit inputs (ui_in and uio_in) together to and outputs via uo_out. Implementation was written in rust using Rust-HDL and exported as verilog

How to test

set ui_in and uio_in and toggle reset line

Pinout

#	Input	Output	Bidirectional
0	input a bit 0	a + b bit 0	input b bit 0
1	input a bit 1	a + b bit 1	input b bit 1
2	input a bit 2	a + b bit 2	input b bit 2
3	input a bit 3	a + b bit 3	input b bit 3
4	input a bit 4	a + b bit 4	input b bit 4
5	input a bit 5	a + b bit 5	input b bit 5
6	input a bit 6	a + b bit 6	input b bit 6
7	input a bit 7	a + b bit 7	input b bit 7

Pinout

The chip is packaged in a 64-pin QFN package. The pinout is shown below.

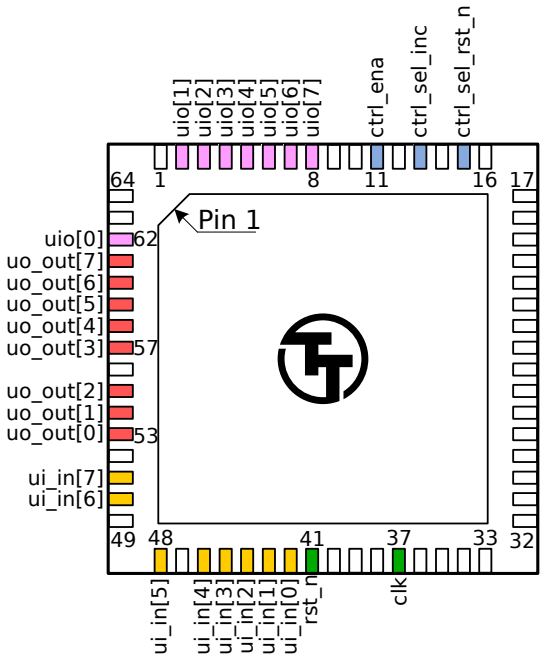


Figure 1: Pinout

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 384 user designs (24 mux units, each with up to 16 designs)
- Designs can have different sizes. The basic unit is a called a tile, and designs can occupy 1, 2, 4, 8, or 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-as (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:

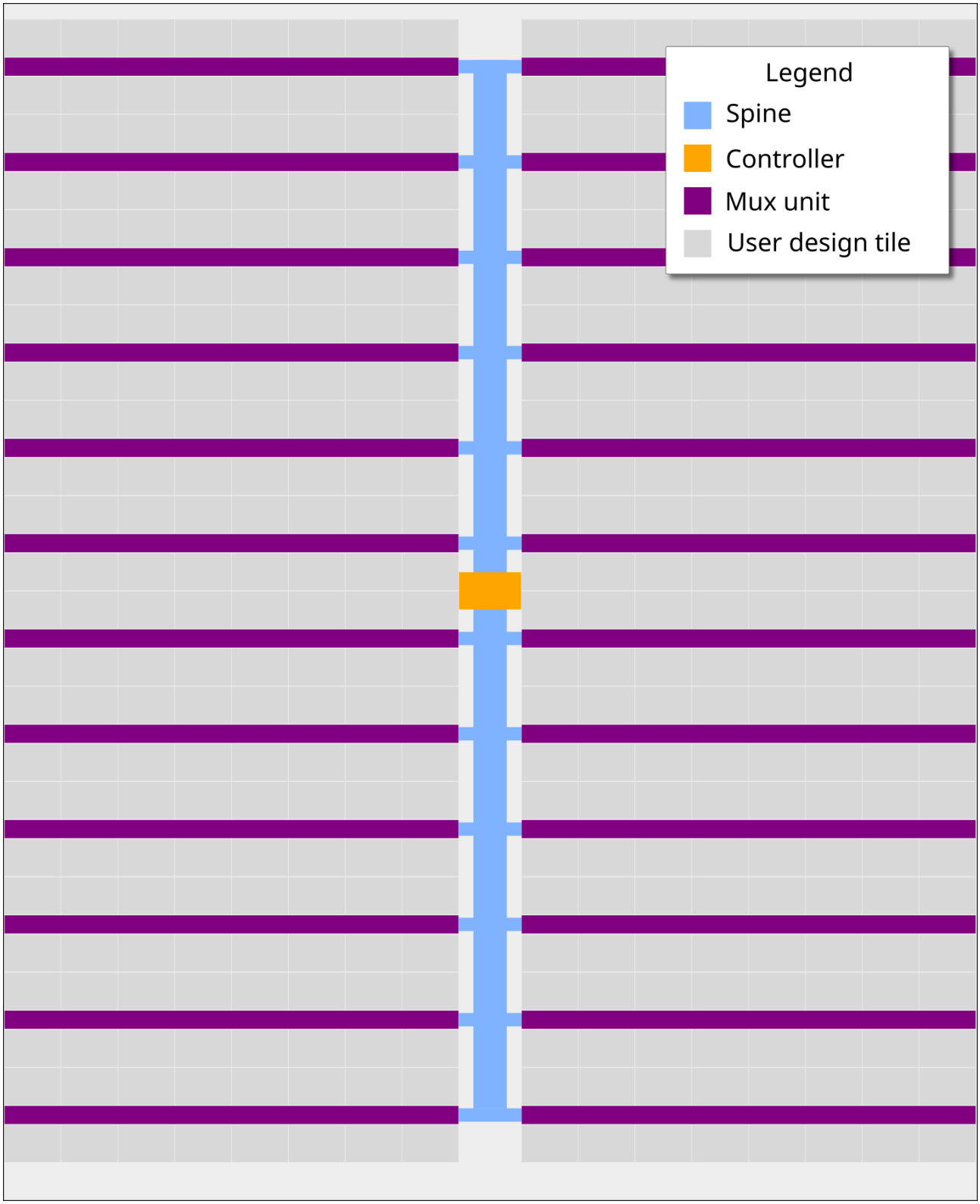


Figure 2: Mux Diagram

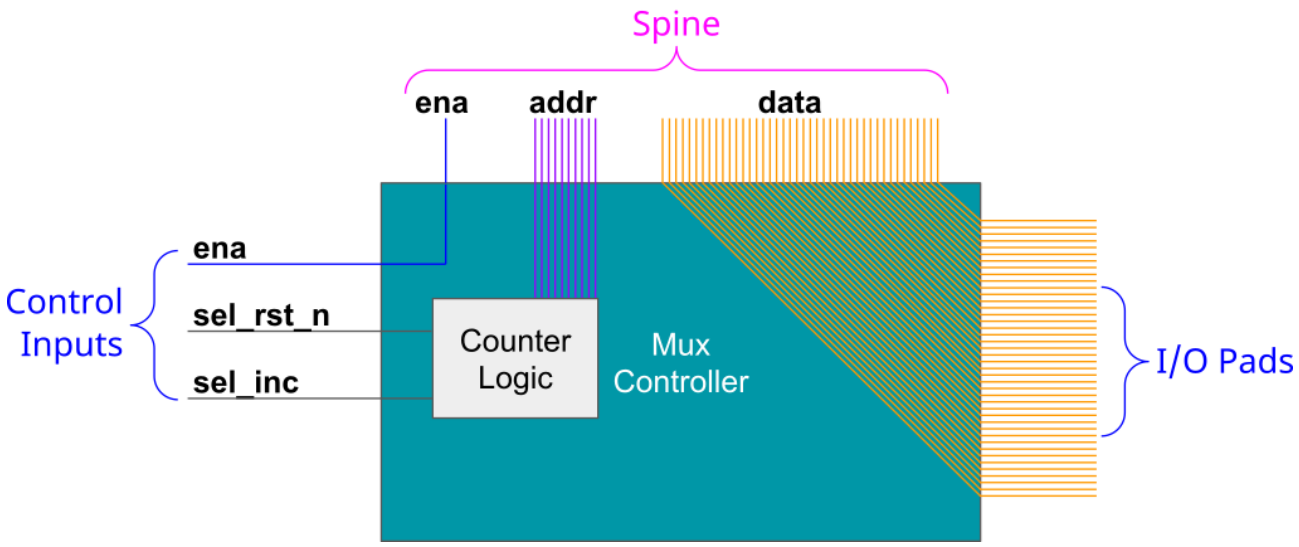


Figure 3: Mux Controller Diagram

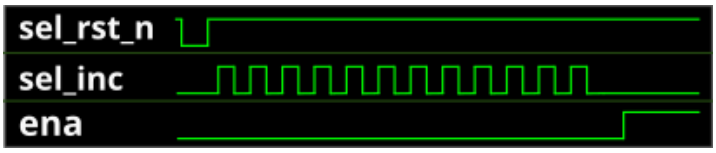


Figure 4: Mux signals for activating the design at address 12

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

mprj_io pin	Function	Signal
0		(none)
1	Housekeeping SPI *	SDO
2	Housekeeping SPI	SDI
3	Housekeeping SPI	CSB
4	Housekeeping SPI	SCK

mprj_io pin	Function	Signal
5	Clock output	user_clock2 †
6	Input	clk
7	Input	n_rst
8	Input	ui_in[0] ‡
9	Input	ui_in[1]
10	Input	ui_in[2]
11	Input	ui_in[3]
12	Input	ui_in[4]
13	Input	ui_in[5]
14	Input	ui_in[6]
15	Input	ui_in[7]
16	Output	uo_out[0]
17	Output	uo_out[1]
18	Output	uo_out[2]
19	Output	uo_out[3]
20	Output	uo_out[4]
21	Output	uo_out[5]
22	Output	uo_out[6]
23	Output	uo_out[7]
24	Bidirectional	uio[0]
25	Bidirectional	uio[1]
26	Bidirectional	uio[2]
27	Bidirectional	uio[3]
28	Bidirectional	uio[4]
29	Bidirectional	uio[5]
30	Bidirectional	uio[6]
31	Bidirectional	uio[7]
32	Mux Control	ctrl_ena
33		(none)
34	Mux Control	ctrl_sel_inc
35		(none)
36	Mux Control	ctrl_sel_rst_n
37		(none)

- The Housekeeping SPI is an SPI interfaces provided by the Caravel harness. You can use it to change the configuration of the GPIO pins and control the clock for the internal Caravel RISC-V core. We do not plan to use it in the Tiny Tapeout Demo board.
  † The `user_clock2` signal outputs the internal clock signal of caravel. You could use it to provide a clock to your design by connecting it to the `clk` input

(mprj_io pin 6). We do not plan to use it in the Tiny Tapeout Demo board.
‡ Internally, there's no difference between `clk`, `n_rst`, 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 bit.

Chip Errata

This section lists the known issues with the chip and suggests workarounds where possible.

Undefined pin states

The state of the bidirectional pins and output pins is not defined when no design is selected. This means that the bidirectional pins may be configured as outputs, with either high or low output values, or as inputs. Take care to avoid shorting the bidirectional pins to other outputs or to VDD or GND when no design is selected. As a workaround, you can connect these pins to external devices or other pins through a resistor.

Sponsored by

Team

Tiny Tapeout would not be possible without a lot of people helping. We would especially like to thank:

- Uri Shaked for wokwi development and lots more
- Sylvain Munaut for help with scan chain improvements
- Mike Thompson for verification expertise
- Jix for formal verification support
- Proppy for help with GitHub actions
- Maximo Balestrini for all the amazing renders and the interactive GDS viewer
- James Rosenthal for coming up with digital design examples
- All the people who took part in TinyTapeout 01 and volunteered time to improve docs and test the flow
- The team at YosysHQ and all the other open source EDA tool makers
- Efabless for running the shuttles and providing OpenLane and sponsorship
- Tim Ansell and Google for supporting the open source silicon movement
- Zero to ASIC course community for all your support
- Jeremy Birch for help with STA
- Aisler for sponsoring PCB development