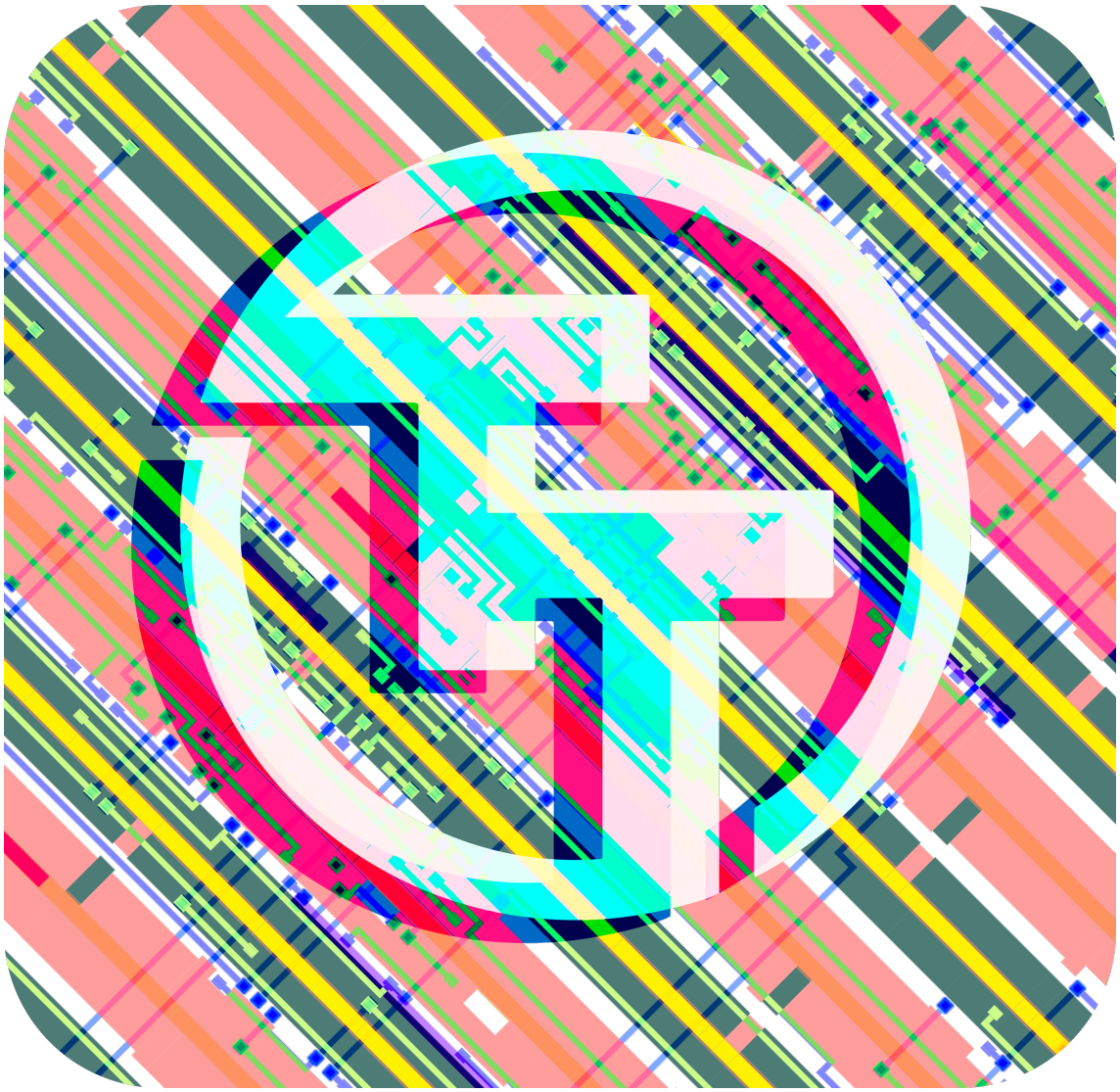


Tiny Tapeout 6 Datasheet

Project Repository

https://github.com/TinyTapeout/tinytapeout-06

February 23, 2025

Contents

Chip map

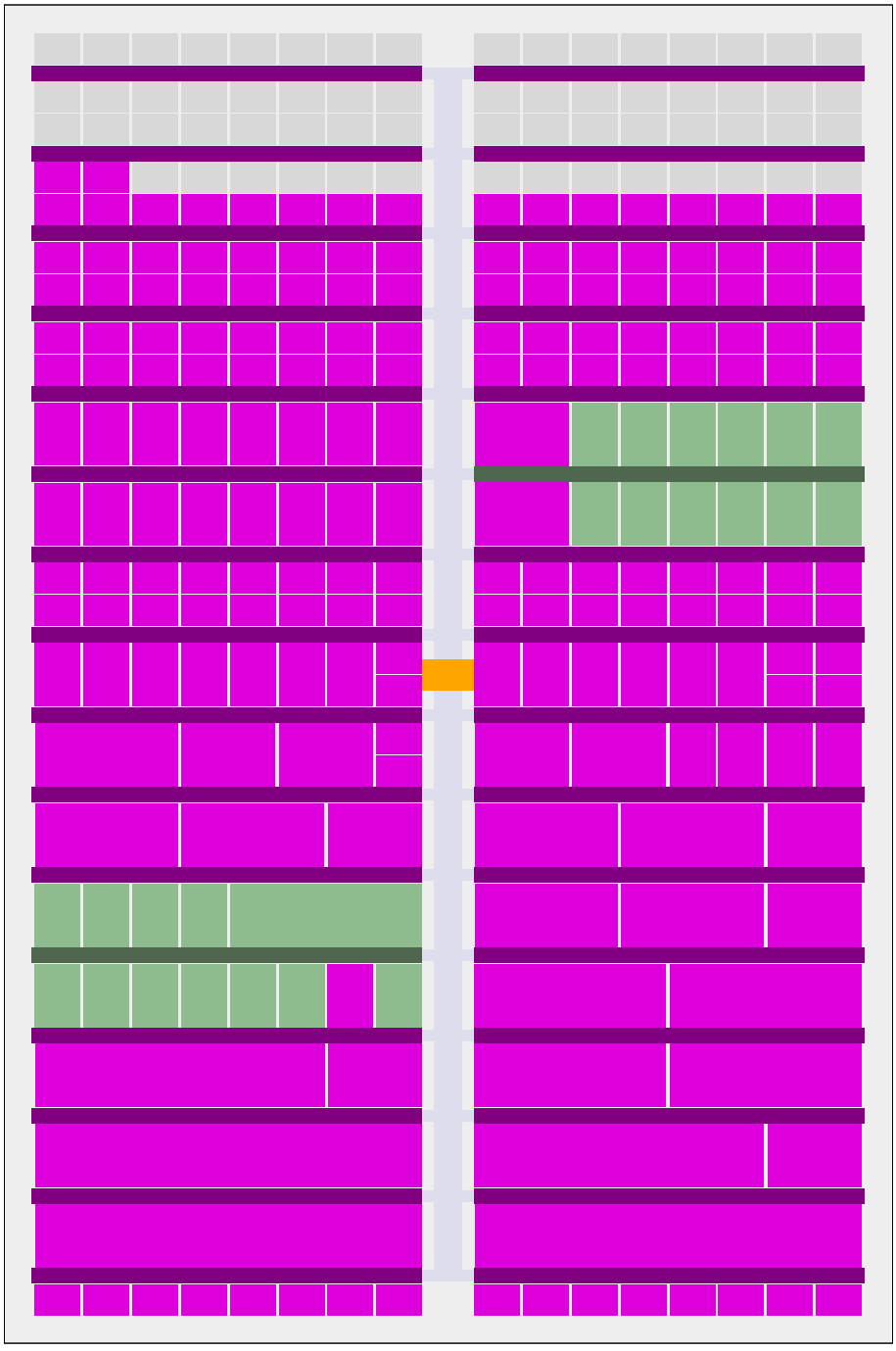


Figure 1: Full chip map

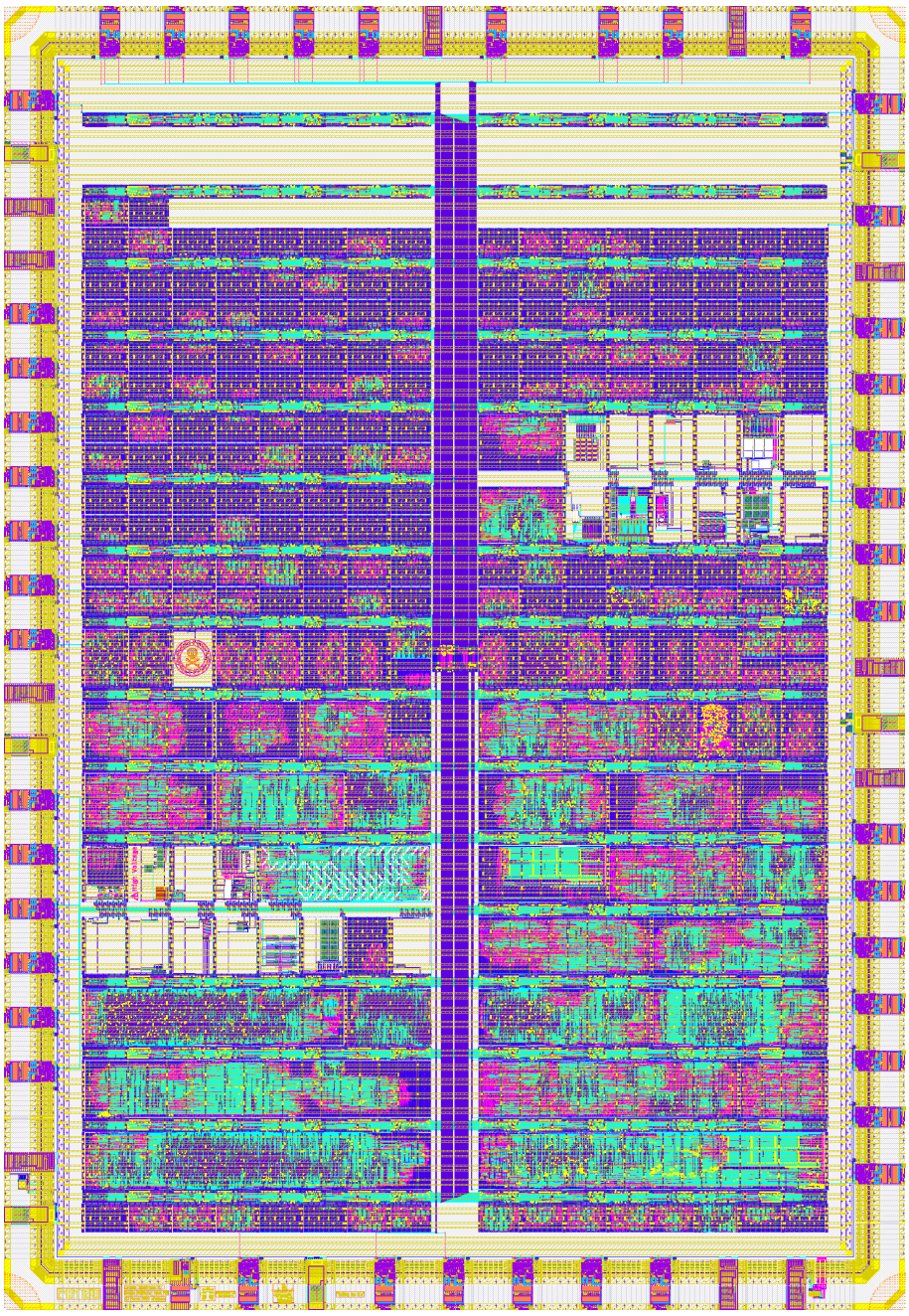


Figure 2: GDS render

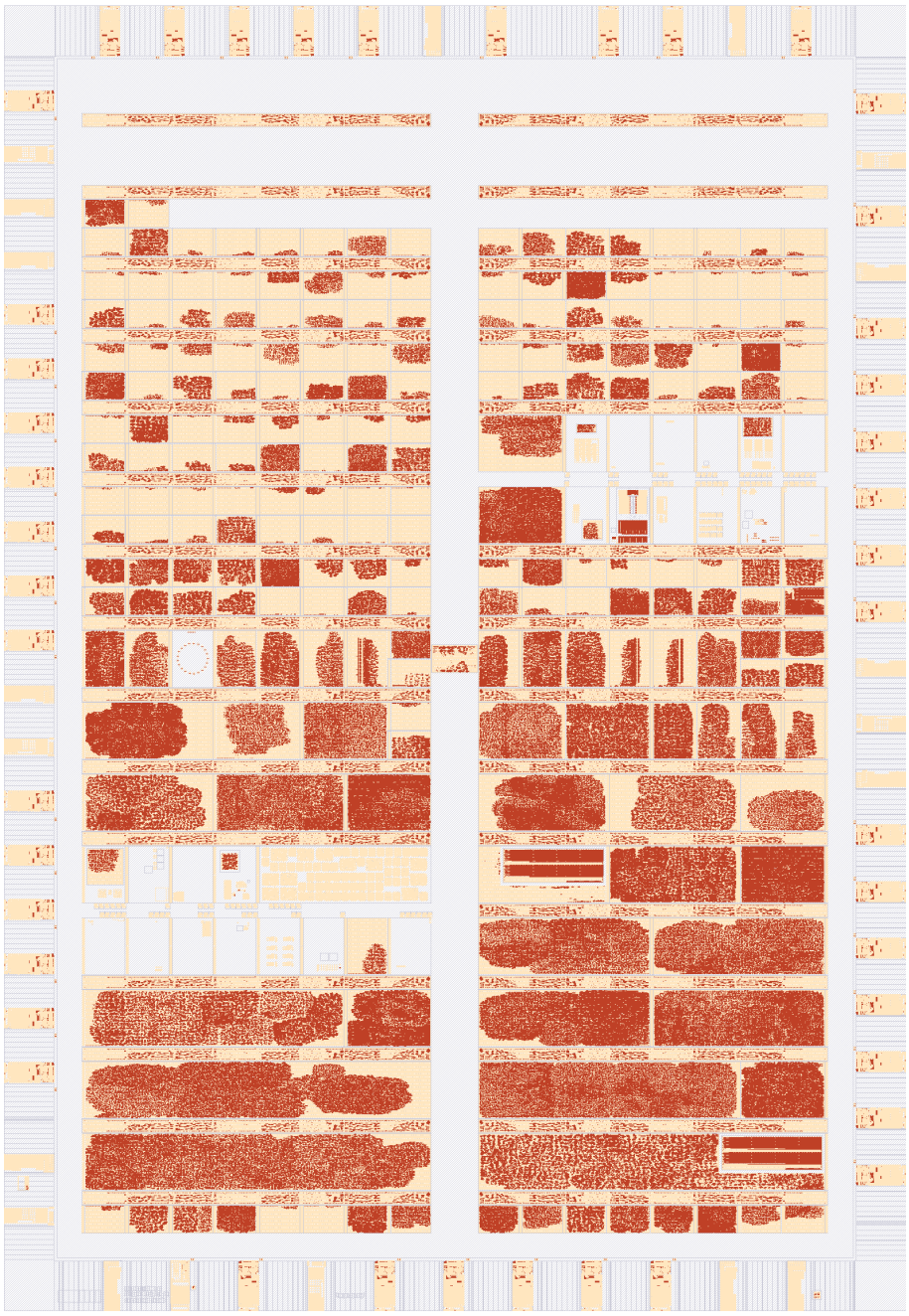


Figure 3: Logic density (local interconnect layer)

Projects

Chip ROM [0]

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

How it works

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

The ROM layout The ROM layout is as follows:

Address	Length	Encoding	Description
0	8	7-segment	Shuttle name (e.g. "tt06"), null-padded
8	8	7-segment	Git commit hash
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	tt06
repo	The name of the repository	TinyTapeout/tinytapeout-06
commit	The commit hash *	a1b2c3d4

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

Here is a complete example of a chip descriptor:

```
shuttle=tt06
repo=TinyTapeout/tinytapeout-06
commit=a1b2c3d4
```

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

How to test

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

Pinout

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

TinyTapeout 06 Factory Test [1]

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

How it works

If `sel` is high, then a counter is output on the output pins and the bidirectional pins (`data_o = counter_o = counter`). If `sel` is low, the bidirectional pins are mirrored to the output pins (`data_o = data_i`).

How to test

Set `sel` high and observe that the counter is output on the output pins (`data_o`) and the bidirectional pins (`counter_o`).

Set `sel` low and observe that the bidirectional pins are mirrored to the output pins (`data_o = data_i`).

Pinout

#	Input	Output	Bidirectional
0	sel	data_o[0]	data_i[0] / counter_o[0]
1		data_o[1]	data_i[1] / counter_o[1]
2		data_o[2]	data_i[2] / counter_o[2]
3		data_o[3]	data_i[3] / counter_o[3]
4		data_o[4]	data_i[4] / counter_o[4]
5		data_o[5]	data_i[5] / counter_o[5]
6		data_o[6]	data_i[6] / counter_o[6]
7		data_o[7]	data_i[7] / counter_o[7]

UCSC HW Systems Collective, TDC [2]

- Author: Tyler Sheaves, Phillip Marlowe, & Dustin Richmond
- Description: A tiny TDC constructed entirely of standard cells. Skywater130 FA-2 delay element
- GitHub repository
- HDL project
- Mux address: 2
- Extra docs
- Clock: 17241379 Hz

How it works

A tiny TDC

How to test

Setup VCS on you local machine, cd to test run: `make SIM=vcs GATES=yes`

External hardware

Just pins

Pinout

#	Input	Output	Bidirectional
0	lanuch clock	hw[0]	
1	capture clock	hw[1]	
2	pg_src	hw[2]	
3	pg_bypass	hw[3]	
4	pg_in	hw[4]	
5	pg_tog	hw[5]	
6	valid_in	hw[6]	
7		valid_out	

Most minimal extension of friend's 'CPU In a Week' in a day [4]

- Author: Gregory Kollmer
- Description: 8-bit Single-Cycle CPU
- GitHub repository
- HDL project
- Mux address: 4
- Extra docs
- Clock: 0 Hz

How it works

This project is the most minimal extension (add CLR & MUL to ISA) of Ramyad Hadidi's 8-bit CPU that details the design and implementation of an 8-bit single-cycle microprocessor. The processor includes a register file and an Arithmetic Logic Unit (ALU). The design was crafted to handle a simple instruction set architecture (ISA) that supports basic ALU operations, load/store operations, and status checks for the ALU carry – all within less than a week. While the current version lacks a program counter and external memory, thus omitting any form of jump operations, it provides a solid foundation for understanding basic computational operations within a custom CPU architecture.

ISCA Overview The ISA is straightforward and is primarily focused on register operations and basic arithmetic/logic functions. Below is the breakdown of the instruction set:

```
// ISA ---------------------------------------------------------
//-- R level
`define MVR 4'b0000            // Move Register
`define LDB 4'b0001            // Load Byte into Regsiter
`define STB 4'b0010            // Store Byte from Regsiter
`define RDS 4'b0011            // Read (store) processor status
// 1'b0100 NOP
// 1'b0101 NOP
// 1'b0110 NOP
// 1'b0111 NOP
//-- Arithmatics
`define NOT {1'b1, `ALU_NOT}
`define AND {1'b1, `ALU_AND}
```

```
`define ORA {1'b1, `ALU_ORA}
`define ADD {1'b1, `ALU_ADD}
`define SUB {1'b1, `ALU_SUB}
`define XOR {1'b1, `ALU_XOR}
`define INC {1'b1, `ALU_INC}
`define MUL {1'b1, `ALU_MUL}
// 1'b1111 NOP
```

How to test

The processor has been tested through a suite of 12 testbenches, each designed to validate a specific functionality or operation. These testbenches cover basic ALU operations, data movement between registers, and the load/store functionalities. Although basic operational tests are passing, timing interactions between instructions have not been exhaustively verified, and it is anticipated that a sophisticated compiler would handle these timing considerations effectively, reminiscent of approaches taken in historical computing systems. [ADD TESTS FOR MUL EXTENSION]

External hardware

Currently, the processor does not interface with any external hardware components. It operates entirely within a simulated environment where all inputs and outputs are managed through testbenches. This setup is ideal for educational purposes or for foundational experimentation in CPU design.

Pinout

#	Input	Output	Bidirectional
0	Register 1 (R1) Address bit 0	Data out bit 0 (either register data / Processor stat)	Data in bit 0 / Register 3 (R3) Address bit 0
1	Register 1 (R1) Address bit 1	Data out bit 1 (either register data / 0)	Data in bit 1 / Register 3 (R3) Address bit 1

#	Input	Output	Bidirectional
2	Register 1 (R1) Address bit 2	Data out bit 2 (either register data / 0)	Data in bit 2 / Register 3 (R3) Address bit 2
3	Register 1 (R1) Address bit 3	Data out bit 3 (either register data / 0)	Data in bit 3 / Register 3 (R3) Address bit 3
4	Instruction ISA Opcode bit 0	Data out bit 4 (either register data / 0)	Data in bit 4 / Register 2 (R2) Address bit 0
5	Instruction ISA Opcode bit 1	Data out bit 5 (either register data / 0)	Data in bit 5 / Register 2 (R2) Address bit 1
6	Instruction ISA Opcode bit 2	Data out bit 6 (either register data / 0)	Data in bit 6 / Register 2 (R2) Address bit 2
7	Instruction ISA Opcode bit 3	Data out bit 7 (either register data / 0)	Data in bit 7 / Register 2 (R2) Address bit 3

SPDIF to I2S decoder [6]

- Author: Jørgen Kragh Jakobsen
- Description: Convert audio from SPDIF to I2S format for ClassD amp MA12070p
- GitHub repository
- HDL project
- Mux address: 6
- Extra docs
- Clock: 48000000 Hz

How it works

SPDIF audio is a well known and commonly used industry standard for audio distibution on a single optical or electical interface. It signal both audio data and the clock in the same signal. I2S is a well known and commenly used industry standard for audio distibution on a 3 wire interface. It has clock(BCK), left/right sync(WS) and data signal(D0).

Digial audio amplifiers and DAC's often use I2S as input interface.

I have coded up a spdif converter that oversample the spdif using a 27Mhz system clock. One process regenerates the i2c_bck clock signal by counting number of system clock between spdif input data edges. The spdif singal is a phase mark kind of encoding with a couple of sync words. This decoder only looks for Left and right sync word that indecate when to toggle the i2c_ws signal.

The audio PCM samples will get decode by looking for phase/mark changes and the pcm bits are shifted in to a left and right pcm_sample fifo. From here the i2s_d0 signal is generate by shifting out of the non active fifo in reversed order.

Code base The system consist of 3 major block:

- A register bank
- I2C interface to read/write to the register bank
- The audio interface

The register back is written in Golang and generates HDL systemVerilog to support packed structs and packages. All systemVerilog sources are converted to verilog using sv2v during the FPGA build script used for testing.

The register map has a seciton for the audio interface - and holds 8 registers for write operation through a dedicated I2C interface to the amplifier. Default it will set amplifer

volume on address 0x40 and amplifer audioformat to std i2s. More commands can be added by software through the system I2C interface.

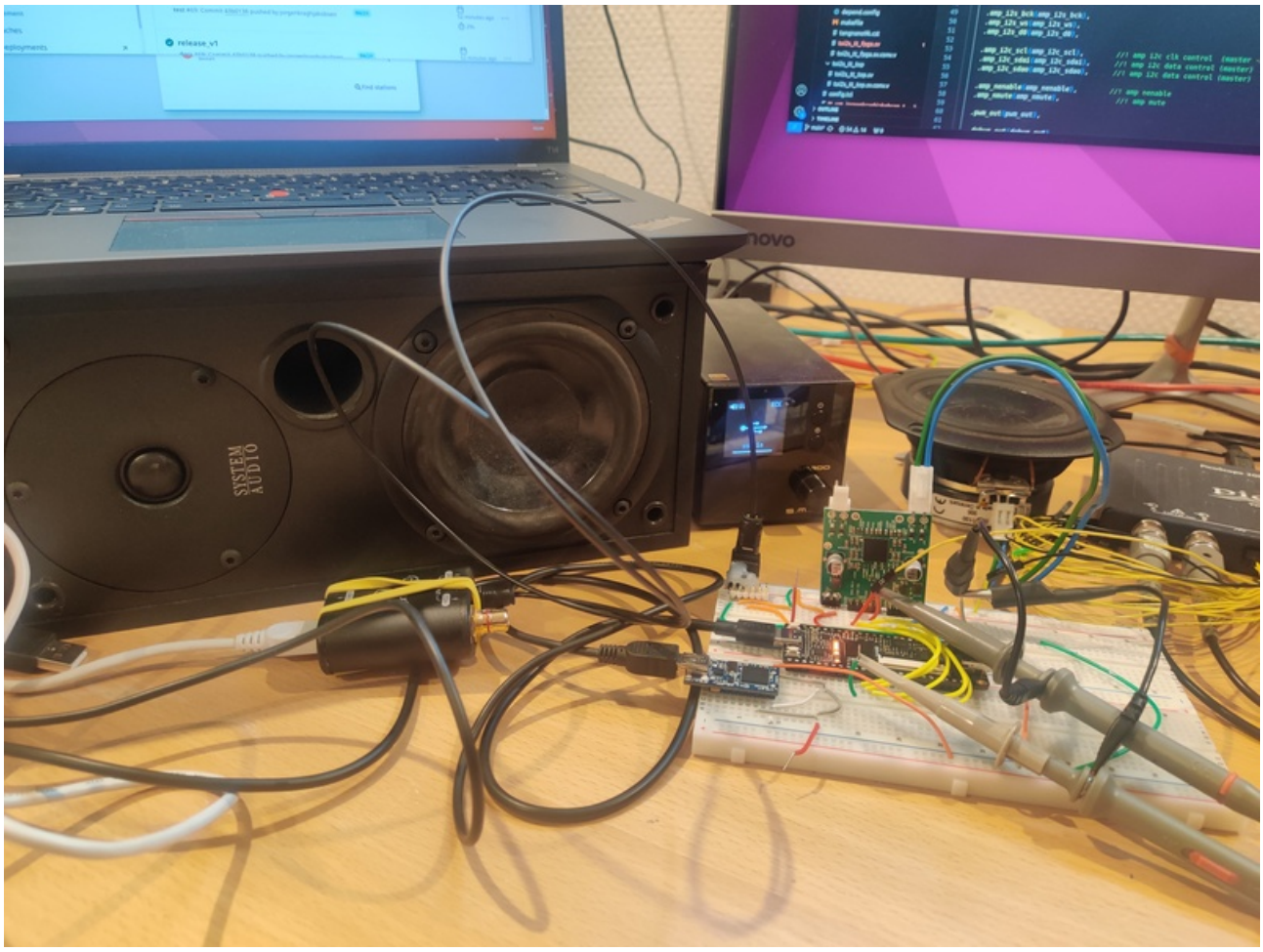


Figure 4: FPGA test implementation

How to test

If no smoke coming out after supply has been applied all good :-)

Apply optical audio from a spdif source - if sounds good - it works :-)

External hardware

Amplifier module MA12070p and ftdi usb to i2c module :-)

Pinout

#	Input	Output	Bidirectional
0	rx_in	amp_i2s_bck	i2c_scl
1	debug_in	amp_i2s_ws	i2c_sda
2		amp_i2s_d0	amp_i2c_scl
3		amp_nenable	amp_i2c_sda
4		amp_nmute	
5			
6			
7		pwm_out	

Chisel Pong [8]

- Author: Tjark Petersen
- Description: A basic Pong game using VGA implemented in Chisel.
- GitHub repository
- HDL project
- Mux address: 8
- Extra docs
- Clock: 50000000 Hz

How it works

This is a Chisel template

How to test

Currently we use cocotb, this shall change to ChiselTest

External hardware

non by default

Pinout

#	Input	Output	Bidirectional
0	left player up	r[1]	state[0]
1	left player down	g[1]	state[1]
2	right player up	b[1]	state[2]
3	right player down	v-sync	v-sync
4	engage left player autopilot	r[0]	h-sync
5	engage right player autopilot	g[0]	left player lost
6	not used	b[0]	right player lost
7	not used	h-sync	game tick

8080 CPU [12]

- Author: Emily Schmidt
- Description: It's an Intel 8080-compatible CPU core that can hopefully run Microsoft BASIC, CP/M, etc.
- GitHub repository
- HDL project
- Mux address: 12
- Extra docs
- Clock: 1000000 Hz

How it works

It's a 8080-compatible CPU. It needs the RP2040 to simulate RAM and I/O.

How to test

TBD

External hardware

RP2040.

Pinout

#	Input	Output	Bidirectional
0	bus_handshake_ack	bus_handshake_req	data_bus[0]
1	debug_req	bus_state[0]	data_bus[1]
2	int_req	bus_state[1]	data_bus[2]
3		bus_io	data_bus[3]
4		cpu_fetch	data_bus[4]
5		cpu_in_debug	data_bus[5]
6		cpu_halted	data_bus[6]
7		cpu_int_ack	data_bus[7]

Tiny Zuse [14]

- Author: Florian Stolz
- Description: Minimal Implementation of a Zuse Z3-style FPU for Addition/Subtraction
- GitHub repository
- HDL project
- Mux address: 14
- Extra docs
- Clock: 10000000 Hz

Quick Overview

This is a partial recreation of the original Zuse Z3 ALU. In Germany the Zuse Z3 is generally regarded as the first computer, however, unlike the ENIAC it is not turing-complete. Even though it may not be the first turing-complete computer, to the best of my knowledge, it and its predecessor the Zuse Z1 contains the first implementation of a floating point unit. It works purely on floating point numbers and only understands a few commands: loading/storing, reading in data and performing addition/subtraction, multiplication/division and lastly computing the square root. It only employs two registers and 64 memory locations.

This is not a faithful recreation, because I did not want to convert the relay-based logic 1:1 to verilog. The memory is missing as well. However, it retains the original floating point format as well as algorithms.

Number Representation

This project uses the Zuse Z3 floating point format, but without using hidden digits. All floats must be normalized, meaning the mantissa must be within 1.0 to 1.99999. The mantissa is 15 bits long, the msb must always be 1 to comply with the previously mentioned normalization (normally, this digit is hidden and used implicitly, but not in this design).

A number is represented via: `+/- x * 2 ^ e`. The sign is represented by a single bit (1 = positive, 0 = negative). X is the mantissa. E is the exponent: A signed 7 bit number!

In order to convert a decimal number, for example, 42.24 to the Z3 format perform the following steps:

1. The number is positive, so the sign bit is 1. s = 1

2. Convert the integer part to binary. 42 = 101010
3. The highest bit is in position 5 (counting from 0). Thus, e = 5
4. Now convert the fractional part to binary:

0.24 * 2 = 0.48 (0)

0.48 * 2 = 0.96 (0)

0.96 * 2 = 1.92 (1)

0.92 * 2 = 1.84 (1)

0.84 * 2 = 1.64 (1)

0.64 * 2 = 1.28 (1)

0.28 * 2 = 0.56 (0)

0.56 * 2 = 1.12 (1)

(this would continue, but we have enough digits to fill our mantissa)

Our number is thus: 101010.00111101

6. Take the number above and remove the dot. Now you got your mantissa
7. Thus in general the Z3 number will be 1 (sign) | 0000101 (exponent) | 10101000111101 (mantissa)
8. You can verify this by computing: `2^5 *( 2^0 + 2^-2 + 2^-4 + 2^-8 + 2^-9 + 2^-10 + 2^-11 + 2^-13) = 42.23828125`

This is not exactly 42.24, which is to be expected, because some decimal numbers are not representible in binary, thus inducing a rounding error.

Here are some example bit strings in Python format, which you can send to the FPU:

`b'\x85\xab\x00' = 42.75`

`b'\x82\xe0\x00' = 7.0`

The number 0 is represented by any value, which has the exponent -64.

Infinity is represented by any value, which has the exponent 63.

How to test

The design was created for a 10 MHz clock and uses a 9600 baud UART connection for communication. It lets you load values into the FPU and perform addition or subtraction.

The commands require a single byte and are defined as follows:

0x82: Set the R1 register

0x83: Set the R2 register

0x84: Set the status register (e.g., overflow, underflow)

0x85: Read the R1 register

0x86: Read the R2 register

0x87: Read the result register

0x88: Perform R1 + R2

0x89: Perform R1 - R2

0x8A: Perform R1 * R2

0x8B: Perform R1 / R2

0x8C: Perform sqrt(R1)

After sending the command to write a register you need to send 3 additional bytes where the first byte contains the sign bit (7) and the exponent (6:0), the following byte defines the mantissa bits 15 to 7. Remember that bit 15 must be 1. The last byte defines the lower mantissa bits. Notice how we transmit 16 bits but only use 15 bits of information. The lowest bit of the last byte is thus ignored. You do not get an ack from the board! Simply read the register back if you are unsure if the transmission worked.

If you send a read command, you will receive 3 bytes in the exact same format as above. First the sign and exponent in the first byte followed by the mantissa bytes.

If you send a command to compute a result, you will receive no answer. You will have to manually read the result register. Do not worry, the FSM should wait until the FPU is done, so no reading of undefined data will happen!

Using the example values from above, here is a complete command sequence:

`b'\x82\x85\xab\x00'` sends the number 42.75

`b'\x83\x82\xe0\x00'` send the number 7.0

`b'\x88'` sends the ADD command

`b'\x87'` reads the result register

The result should be `b'\x85\xc7\x00'`

The status register can signify the following events:

The result was zero bit position 0

The computation overflowed bit position 1

The computation underflowed bit position 2

External hardware

Use the on board RPi 2400 for uart connections. It uses the default uart ports suggested by tiny tapeout.

Pinout

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

7-segment-FUN [32]

- Author: Armin Hartl
- Description: Many different Animations on an 7-Segment-Display
- GitHub repository
- HDL project
- Mux address: 32
- Extra docs
- Clock: 10000000 Hz

How it works

Simplified is this project just a counter, which speed can be changed, combined with animations for a 7-segment display, which also can be switched trough.

How to test

The default setting should display the numbers 0 to 9, which should change every second. The design can be tested by pressing the different input buttons and seeing if the speed respectively the animation changes.

External hardware

You might need a breadboard and buttons for the controls, as well as a 7-segment display if not available.

Pinout

#	Input	Output	Bidirectional
0	btn1_incAni	segment a1	
1	btn2_decAni	segment b2	
2	btn3_incSpeed	segment c3	
3	btn4_decSpeed	segment d4	
4		segment e5	
5		segment f6	
6		segment g7	
7			

Asynchronous Down Counter [33]

- Author: Alen Music
- Description: Counter
- GitHub repository
- Wokwi project
- Mux address: 33
- Extra docs
- Clock: 0 Hz

How it works

The Project is a Asynchronous 3 Bit Down Counter.In the asynchronous counter, an external clock pulse is provided for only the first Flip-Flop, thereafter the output of the 1st Flip-Flop acts as a clock pulse for the second Flip-Flop and so on. In the case of synchronous Flip-Flops, all the Flip-Flops are triggered simultaneously by an external clock pulse.

How to test

Pressing the button in succession will make the counter count.

External hardware

7SEG-Display

Pinout

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

Synthesized Time-to-Digital Converter (TDC) [35]

- Author: Harald Pretl
- Description: Synthesized TDC based on an interleaved delay line
- GitHub repository
- HDL project
- Mux address: 35
- Extra docs
- Clock: 50000000 Hz

How it works

This is a simple synthesized time-to-digital converter (TDC), consisting of a delay line and parallel capture FF. Depending on `__TDC_INTERLEAVED__` either a simple or an interleaved delay line is implemented.

In the TT 1x1 block size a 192-stage interleaved delay can be fitted.

How to test

Apply two signals to `ui_in[0]` and `clk`.

After capturing (rising edge of `clk`) the result (i.e., the time delay between rising edge of `ui_in[0]` and `clk`) can be muxed-out to `uo_out[7:0]` using `ui_in[7:3]` as byte-wise selector. `ui_in[7:3]=0000` gives result byte 0, `ui_in[7:3]=0001` gives result byte 1, etc.

External hardware

Two signal generators generating logical signals with a programmable delay.

Pinout

#	Input	Output	Bidirectional
0	Start signal of TDC (stop signal is clk)	Result LSB	
1		Result	
2		Result	
3		Result	
4	output select	Result	

#	Input	Output	Bidirectional
5	output select	Result	
6	output select	Result	
7	output select	Result MSB	

luckyCube [37]

- Author: Lejla Rahmanovic-Abdic
- Description: Lucky Number 6
- GitHub repository
- Wokwi project
- Mux address: 37
- Extra docs
- Clock: 0 Hz

How it works

In this project, goal was to create an engaging digital dice experience. The primary goal is to display the word "roll" on the LED display, accompanied by the illumination of the number "6" – a universally recognized lucky number in dice games.

How to test

Play with switch

External hardware

-

Pinout

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

playwithnumbers [39]

- Author: Rukija Hafizovic
- Description: Even/Odd Numbers
- GitHub repository
- Wokwi project
- Mux address: 39
- Extra docs
- Clock: 0 Hz

How it works

The function of this project is to enable the user to determine accurate even or odd numbers. No matter which number (1-8) the user turns on, the LED bulbs will not light evenly. Using dip switch 8 user can choose which number will be used and according to logicgates(AND, OR, XOR, NAND, NOT) LED and 7seg dipslay will work.

How to test

To get even/odd numbers you have to switch on 2,4,6 and 8 (2,4,6,8 ->„1“ and 1,3,5,7 ->„0“). All blue led bulbs should light and 8 have to be displayed.

External hardware

List external hardware used in your project: seven segment display

Pinout

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

2 Player Game [41]

- Author: Gabriel Silva, Tristan Peterson, Conner F
- Description: Count down timer game
- GitHub repository
- HDL project
- Mux address: 41
- Extra docs
- Clock: 20000000 Hz

How it works

Counting timer game! 4 buttons, start, reset, p1, p2. The goal time is seen on the screen after reset, once start is selected the timer will begin at 0.0 and go to 9.9. After 3.0 seconds the display is hidden, guess the answer!

How to test

If you select the Player button and Start button you can see the time the player selected. THis only works after both players have answered

External hardware

You need a PMOD 7 segment disp 1286-1065-ND, 4 button PMOD 1286-1145-ND

Pinout

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

Oscillating Bones [42]

- Author: Uri Shaked
- Description: A stylish ring oscillator built from SkullFET transistors
- GitHub repository
- HDL project
- Mux address: 42
- Extra docs
- Clock: 0 Hz

How it works

A simple yet stylish ring oscillator that uses a chain of 21 SkullFET inverters to generate a square wave output. Based on simulation, the oscillator should have a frequency of around 90 MHz.

How to test

Connect an oscilloscope to the `osc_out` (ou_out pin 0) pin and enjoy the show. You can also observe the divided frequency outputs on `osc_div_2`, `osc_div_4`, and `osc_div_8`.

Simulation results

The following graph shows the output of the oscillator and the divided outputs. It was generated by running `make -C sim` and patiently waiting for the simulation to finish:

The outputs are shifted by 2 volts to make them easier to see in the graph. "uo_out[0]" is the main output of the oscillator, and "uo_out[1]", "uo_out[2]", and "uo_out[3]" are the divided outputs.

Note that the simulation results do not include all the parasitics, only the main ones. The actual frequency of the oscillator will probably be lower than the simulated one.

#	Input	Output	Bidirectional

Pinout

#	Input	Output	Bidirectional
0		osc_out	
1		osc_div_2	
2		osc_div_4	
3		osc_div_8	
4			
5			
6			
7			

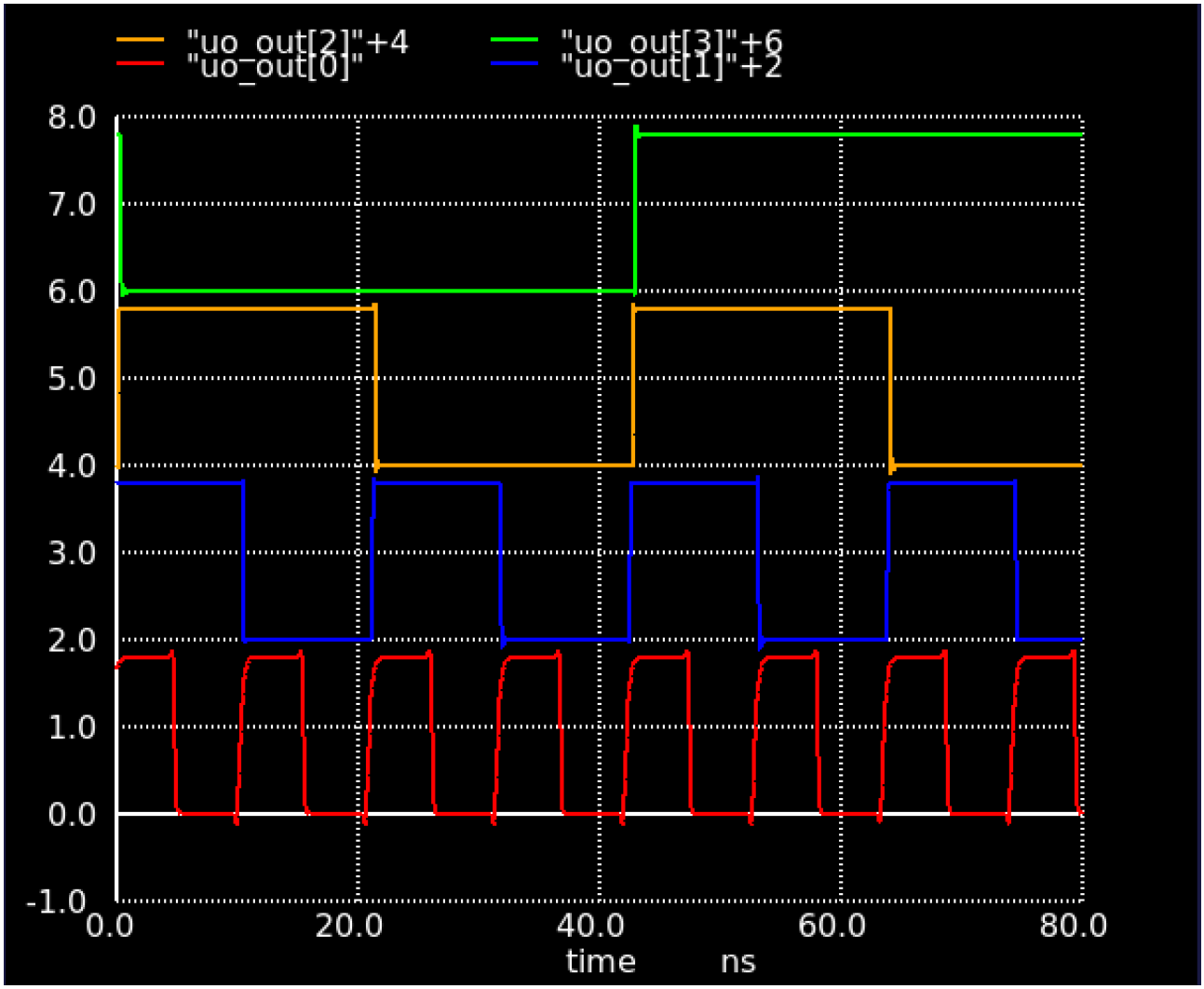


Figure 5: Simulation results

drops [43]

- Author: Philipp Ploeckinger
- Description: Arcade Style game, which lets you collect vertical droplets on an 8x8 pixel display
- GitHub repository
- HDL project
- Mux address: 43
- Extra docs
- Clock: 0 Hz

How it works

This project uses two mechanical buttons and an 8x8 display to plan arcade style game called drops. The goal is to move a bar horicontal in order to catch the vertical falling drops. The player starts with a fixed numer of lifes. Each time the drop is missed, the lifes are deducted by one. When all lifes are used, the game is over an can be restarted with the reset button.

How to test

After plugging everything in as specified in the info.yaml file, the display should light up. If this is not the case, change row and colum pins

There are two things that need to be tested and eventually corrected:

- Drop moving upwards: change the column pins (7 to 0, 0 to 7 etc)
- Bar mowing in wrong direction: either change left and right button or siwcht row pins (7 to 0, 0 to 7 etc)

External hardware

In addition to the Tiny Tapeout board there are two buttons, and an 8x8 display necessary. Base on your desired connection of the buttons you might need an additional power source.

Pinout

#	Input	Output	Bidirectional
0	push button - right	display column 0	display row 0
1	push button - left	display column 1	display row 1
2		display column 2	display row 2
3		display column 3	display row 3
4		display column 4	display row 4
5		display column 5	display row 5
6		display column 6	display row 6
7		display column 7	display row 7

Flappy Bird [45]

- Author: Robin Hohensinn
- Description: Flappy Bird
- GitHub repository
- HDL project
- Mux address: 45
- Extra docs
- Clock: 0 Hz

How it works

The design of the chip allows playing a simplified version of Flappy Bird on an 8x8 LED matrix. For peripheral hardware, only two buttons for controlling the bird's position and an 8x8 LED matrix are required. After successful software testing using Waveform, the design was synthesized in a Github repository. Following successful Waveform testing, the circuit was verified for functionality using an FPGA chip.

The 8-bit outputs act as the "High" signals for the LED matrix, while another set of 8-bit outputs serve as the "LOW" signals, forming a grid pattern conceptually. This setup enables individual LEDs to be lit up through precise control of one row and one colomn. Ensuring correct installation of the LED matrix and using appropriately sized resistors for protection is essential.

How to test

To test this version use waveform tests or an oscilloscop.

External hardware

two buttons and a 8x8 Led Matrix https://de.aliexpress.com/item/32857281704.html?gatewayAd

Pinout

#	Input	Output	Bidirectional
0	up-Button	row of display-Matrix	col of display-Matrix
1	down-Button	row of display-Matrix	col of display-Matrix
2	not used	row of display-Matrix	col of display-Matrix
3	not used	row of display-Matrix	col of display-Matrix

#	Input	Output	Bidirectional
4	not used	row of display-Matrix	col of display-Matrix
5	not used	row of display-Matrix	col of display-Matrix
6	not used	row of display-Matrix	col of display-Matrix
7	not used	row of display-Matrix	col of display-Matrix

4-Bit CPU mit FSM [47]

- Author: Jacqueline Gislai
- Description: Mini CPU, that can do simple calculations and logic operations as well as storing and loading values and execute shifting operations
- GitHub repository
- HDL project
- Mux address: 47
- Extra docs
- Clock: 0 Hz

How it works

Based on the input information, a few operations can be used to process the input values. If there should be operations done with two or more values, store- and loading-operations have to be executed in between before going to the next calculation step.

How to test

For example giving the CPU a value to store and next to load into the accumulator by giving the correct operation codes and the storage address. Next giving the CPU the next value and the operation that should be processes onto those to values, for example a AND function. The result will be given to the output converted to a 8-Bit signal.

Pinout

#	Input	Output	Bidirectional
0	storage address Bit 0	output data Bit 0	input write-access in storage
1	storage address Bit 1	output data Bit 1	
2	storage address Bit 2	output data Bit 2	
3	storage address Bit 3	output data Bit 3	
4	value of input data Bit 0	output data Bit 4	input Operation Code Bit 0
5	value of input data Bit 1	output data Bit 5	input Operation Code Bit 1
6	value of input data Bit 2	output data Bit 6	input Operation Code Bit 2
7	value of input data Bit 3	output data Bit 7	input Operation Code Bit 3

FP4 x 8-bit matrix multiplier [64]

- Author: ReJ aka Renaldas Zioma
- Description: 4-bit floating point (E3M0) x 8-bit matrix multiplier block
- GitHub repository
- HDL project
- Mux address: 64
- Extra docs
- Clock: 48000000 Hz

How it works

Matrix multiplication is implemented using a systolic array architecture.

How to test

Every cycle feed packed weight data to Input pins and input data to Bidirectional pins. Strobe Enable pin to start receiving results of the matrix multiplication on the Output pins.

External hardware

MCU is necessary to feed weights and input data into the accelerator and fetch the results.

Pinout

#	Input	Output	Bidirectional
0	2nd FP4 weight LSB	result LSB	(in) activations LSB
1	2nd FP4 weight	result	(in) activations
2	2nd FP4 weight	result	(in) activations
3	2nd FP4 weight MSB	result	(in) activations
4	1st FP4 weight LSB	result	(in) activations
5	1st FP4 weight	result	(in) activations
6	1st FP4 weight	result	(in) activations
7	1st FP4 weight MSB	result MSB	(in) activations MSB

CORA-16 [66]

- Author: Andrew Dona-Couch
- Description: Simple 16-bit CPU
- GitHub repository
- HDL project
- Mux address: 66
- Extra docs
- Clock: 0 Hz

Couch's One-Register Accumulator machine, 16-bit width.

How it works

One register should be enough for anybody. Well, there's also the program counter, status flags, stack pointer, data pointer, but who's counting?

External SPI memory is used for a simple instruction fetch/execute cycle. High-bandwidth I/O is provided through a full byte-width input and output bus. The machine allows single-stepping through execution to aid debugging.

Pin	Function
`step`	Set high for a clock cycle to step, hold high to run.
`busy`	When high, the machine is currently working on an instruction.
`halt`	When high, the machine has halted execution.
`trap`	When `halt` is low and `trap` is high, the machine has trapped. Step once to attempt recovery (success depends significantly on context).
Note: when both `halt` and `trap` are high, the machine has experienced an irrecoverable fault, please reset.	
`in[7:0]`	General-purpose byte input. Use as data source `IN` for any one-argument instruction.
`out[7:0]`	General-purpose byte output. Set with the `OUT` instruction.

How to test

1. Load the program to run into the external SPI RAM.
2. Reset the CPU.
3. Raise `step` high for a clock for each instruction to step.
4. Hold `step` high to run free (you are advised to handle `trap`).
5. Observe `busy`, `halt` and `trap` for the module status.

External hardware

The module expects an SPI RAM attached to the relevant SPI pins. The onboard Raspberry Pi emulation should work just fine.

Instruction set

Status byte	7	6	5	4	3	2	1	0
x	x	**E**lse	x	x	**C**arry	**N**eg	**Z**ero	

Impact on the status flags is documented as:

- `-`: No effect
- `0`: The flag is cleared to zero
- `1`: The flag is set to one
- `#`: The flag is affected by the operation

One-byte instructions

Name	Bit Pattern	Description	Status
Nop	`0000 0000`	No operation	`---- ----`
Halt	`0000 0001`	Halt machine	`---- ----`
Trap	`0000 0010`	Trap execution	`---- ----`
Drop	`0000 0011`	Drop a word from the stack	`---- ----`
Push	`0000 0100`	Push a word to the stack	`---- ----`
Pop	`0000 0101`	Pop a word from the stack to the accumulator	`---- ----`
Return	`0000 0110`	Return to the address on top of the stack	`---- ----`

Name	Bit Pattern	Description	Status
Not	`0000 0111`	One's complement of the accumulator	`---- -1##`
Out Lo	`0000 1000`	Output the low byte of the accumulator	`---- ----`
Out Hi	`0000 1001`	Output the high byte of the accumulator	`---- ----`
Set DP	`0000 1010`	Set the data pointer value to the accumulator value	`---- ----`
Test	`0000 1011`	Set the status flags based on the accumulator value	`---- --##`
Branch Indirect	`0000 1100`	Add the accumulator to the program counter	`---- ----`
Call Indirect	`0000 1101`	Call the subroutine address in the accumulator	`---- ----`
Status	`0001 0000`	Load the status flags into the accumulator	`---- ----`
Load Indirect	`0100 01mm`	Load a word from the address in the accumulator, using addressing mode `m` (bug: modes not supported)	`---- ----`

Two-byte instructions

Name	Bit Pattern	Description	Status
Load	`1000 0sss vvvv vvvv`	Load a value into the accumulator	`---- ----`
Store	`1001 0sss vvvv vvvv`	Store a value to memory	`---- ----`
Add	`1000 1sss vvvv vvvv`	Add a value to the accumulator	`---- -###`
Sub	`1001 1sss vvvv vvvv`	Subtract a value from the accumulator	`---- -###`
And	`1010 0sss vvvv vvvv`	Bitwise and a value with the accumulator	`---- --##`
Or	`1010 1sss vvvv vvvv`	Bitwise or a value with the accumulator	`---- --##`

Name	Bit Pattern	Description	Status
Xor	`1011 0sss vvvv vvvv`	Bitwise exclusive or a value with the accumulator	`---- --##`
Shift	`1011 1sss vvvv vvvv`	Shift the accumulator (see note below on direction)	`---- -###`
Branch	`1100 0pp pppp pppp`	Add the offset `p` to the program counter	`---- ----`
Call	`1101 0pp pppp pppp`	Call the subroutine at address `p`	`---- ----`
If	`1111 000 0000 cccc`	Skip the following instruction if the condition doesn't hold	`---- ----`

Many of these instructions specify a source type `s` and value `v`. These are the options:

Source Type	Bit Pattern	Interpretation
Const Lo	`000`	Take the value `v` as the low byte of a constant
Const Hi	`001`	Take the value `v` as the high byte of a constant
Input Lo	`010`	Input the low byte, ignore the value `v`
Input Hi	`011`	Input the high byte, ignore the value `v`
Data Direct	`100`	Read a value from the address `v` (relative to the data pointer)
Data Indirect	`101`	Read a pointer from the address `v` (relative to the data pointer), and load a value from that address
Stack Direct	`110`	Read a value from the address `v` (relative to the stack pointer)
Stack Indirect	`111`	Read a pointer from the address `v` (relative to the stack pointer), and load a value from that address

Note: the SHIFT instruction stashes the shift direction within this source field.

Source Type	Shift Bit	Source Limitation
Constant	Lo/Hi	Only 8-bit constants supported
Input	Lo/Hi	Only 8-bit inputs supported
Memory	Addr[0]	Only aligned addresses supported (TODO: maybe require that everywhere??)

The following table lists the condition codes for the IF instruction.

Condition	Bit Pattern	Description
Zero	0000	Skip the next instruction if the Z bit is cleared
Not Zero	0001	Skip the next instruction if the Z bit is set
Else	0010	Skip the next instruction if the E bit is cleared
Not Else	0011	Skip the next instruction if the E bit is set
Neg	0100	Skip the next instruction if the N bit is cleared
Not Neg	0101	Skip the next instruction if the N bit is set
Carry	0110	Skip the next instruction if the C bit is cleared
Not Carry	0111	Skip the next instruction if the C bit is set

Three-byte instructions

Name	Bit Pattern	Description	Status
Call Word	0011 1110 wwww wwww wwww wwww	Call the subroutine at address w	---- ----
Load Immediate Word	0011 1111 wwww wwww wwww wwww	Set the accumulator to w	---- ----

Pinout

#	Input	Output	Bidirectional
0	Data In 0	Data Out 0	SPI MOSI
1	Data In 1	Data Out 1	SPI CS
2	Data In 2	Data Out 2	SPI CLK
3	Data In 3	Data Out 3	SPI MISO

#	Input	Output	Bidirectional
4	Data In 4	Data Out 4	Step
5	Data In 5	Data Out 5	Busy
6	Data In 6	Data Out 6	Halt
7	Data In 7	Data Out 7	Trap

kstep [68]

- Author: Kevin OConnor
- Description: Generate step/dir pulses for stepper motor drivers
- GitHub repository
- HDL project
- Mux address: 68
- Extra docs
- Clock: 50000000 Hz

How it works

This project can produce timed pulses suitable for controlling stepper motor drivers. It is similar to a PWM controller, but has additional control over the number of pulses generated and an ability to gradually change the timing between each pulse.

Commands are sent via SPI. Each SPI message should have 40 bits and be in the following format: `&lt;1-bit rw&gt;&lt;7-bit address&gt;&lt;32-bit data&gt;`

The `rw` bit should be 1 to indicate a write.

The following commands are available:

```
W 0x10 <pin polarity>: This controls default state of all uo_out pins.
W 0x11 <any>: Clear shutdown state.
W 0x12 <step_duration>: Set the duration of step pulses (in clock ticks).
W 0x20 <count/add>: Set count (upper 16 bits), add (lower 16 bits), and s
W 0x21 <interval>: Set interval between pulses (ticks). Submit with addr
W 0x22 <direction>: Set stepper direction (pin uo_out[1]) during step pul
W 0x30 <any>: Reset last step time to zero.
W 0x70 <clock>: Set the current clock counter.
R 0x70: Read the current clock counter.
```

There are also two control pins separate from the SPI interface: `signal_irq` and `signal_shutdown`. The `signal_irq` signal is raised high to indicate that there is space to submit a new schedule entry (via writes to address 0x21 and 0x20). If the `signal_shutdown` reads a high value that the device will return all uo_out pins to their configured polarity (that is it will stop pulsing the step pin). To clear the shutdown state, return the `signal_shutdown` to a low state and issue a write to address 0x11.

How to test

Configure an SPI device. Ensure that the signal_shutdown line is held low. Issue an SPI set pin polarity command. Issue a set pulse duration command. Issue a set clock command. Issue a set interval command. Isuse a set count/add command. Optionally issue additional interval,count,add commands. Observe the step pulses on the uo_out[0] (step) pin.

Pinout

#	Input	Output	Bidirectional
0		step	spi_cs
1		dir	spi_mosi
2		other2	spi_miso
3		other3	spi_sclk
4		other4	signal_irq
5		other5	signal_shutdown
6		other6	
7		other7	

UCSC HW Systems Collective, TDC - BUF2x1 [70]

- Author: Phillip Marlowe, Tyler Sheaves, & Dustin Richmond
- Description: A tiny TDC constructed entirely of standard cells. Skywater130 AND-2 delay element
- GitHub repository
- HDL project
- Mux address: 70
- Extra docs
- Clock: 17241379 Hz

How it works

A tiny TDC

How to test

Setup VCS on you local machine, cd to test run: `make SIM=vcs GATES=yes`

External hardware

Just pins

Pinout

#	Input	Output	Bidirectional
0	lanuch clock	hw[0]	
1	capture clock	hw[1]	
2	pg_src	hw[2]	
3	pg_bypass	hw[3]	
4	pg_in	hw[4]	
5	pg_tog	hw[5]	
6	valid_in	hw[6]	
7		valid_out	

UCSC HW Systems Collective, TDC - MUX2x1 [72]

- Author: Phillip Marlowe, Tyler Sheaves, & Dustin Richmond
- Description: A tiny TDC constructed entirely of standard cells. Skywater130 MUX2x1 delay element
- GitHub repository
- HDL project
- Mux address: 72
- Extra docs
- Clock: 40000000 Hz

How it works

A tiny TDC

How to test

Setup VCS on you local machine, cd to test run: `make SIM=vcs GATES=yes`

External hardware

Just pins

Pinout

#	Input	Output	Bidirectional
0	lanuch clock	hw[0]	
1	capture clock	hw[1]	
2	pg_src	hw[2]	
3	pg_bypass	hw[3]	
4	pg_in	hw[4]	
5	pg_tog	hw[5]	
6	valid_in	hw[6]	
7		valid_out	

Gray scale and Sobel filter [74]

- Author: Diana Natali Maldonado Ramirez
- Description: This project performs grayscale conversion and Sobel filtering with the aim of detecting edges in an image.
- GitHub repository
- HDL project
- Mux address: 74
- Extra docs
- Clock: 10000000 Hz

How it works

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

Below is a block diagram of the implementation:

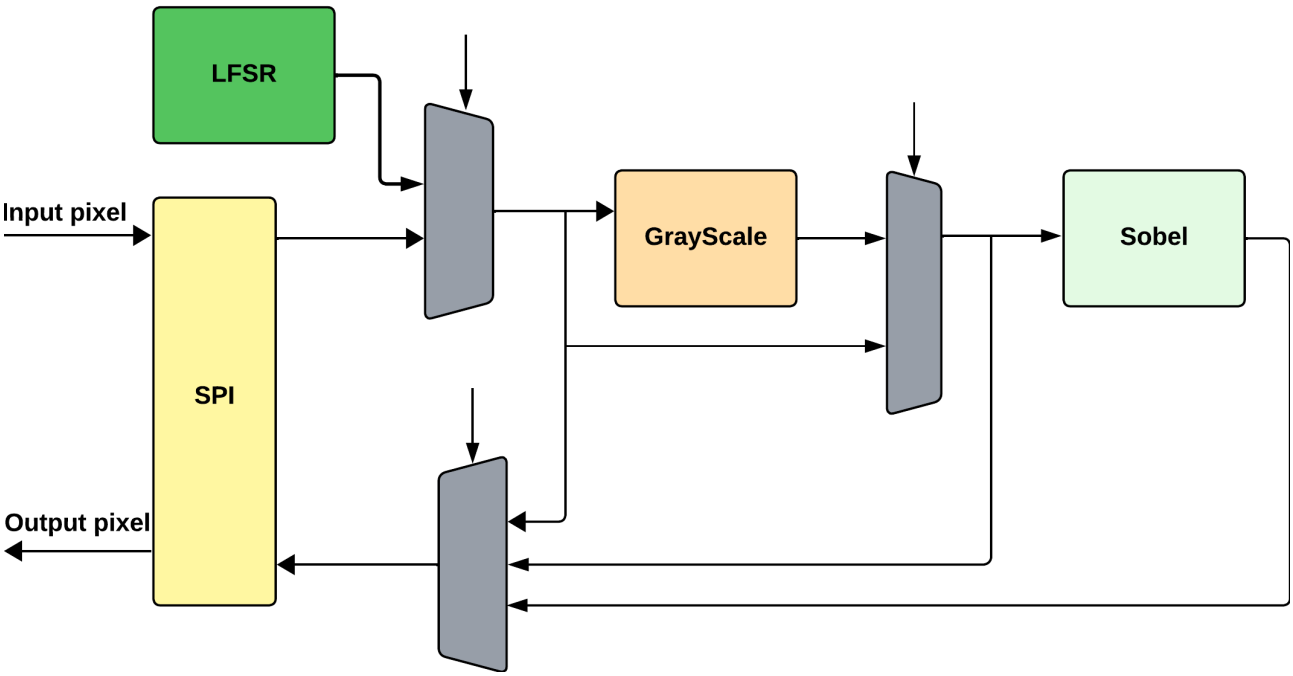


Figure 6: arc

How to test

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

- `ui_in[0]` → SPI Clock

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

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

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

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

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

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

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

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

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	spi_sck_i	spi_sdo_o	LFSR_enable_i
1	spi_cs_i	lfsr_done	seed_stop_i
2	spi_sdi_i	ena	lfsr_en_i

#	Input	Output	Bidirectional
3	select_process_i[0]	output_px[0]	
4	select_process_i[1]	output_px[1]	
5	start_sobel_i	output_px[2]	
6		output_px[3]	
7		output_px[4]	

RGB Mixer demo [76]

- Author: Matt Venn
- Description: Zero to ASIC demo project
- GitHub repository
- HDL project
- Mux address: 76
- Extra docs
- Clock: 10000000 Hz

How it works

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

How to test

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

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

External hardware

Use 3 digital encoders attached to the first 6 inputs.

Pinout

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

Projekt KEIS Hadner Thomas [78]

- Author: Thomas Hadner
- Description: Demodulator for RC Receiver with different Outputs
- GitHub repository
- HDL project
- Mux address: 78
- Extra docs
- Clock: 1000000 Hz

How it works

Decodes PWM-Signal from RC Receiver with counter and threshold values to decide wether to set the output to HIGH or LOW.

How to test

The program can be tested by applying a PWM-Signal to the input with a longer pulse time than 1.9ms, then the output will go to HIGH. If then you apply a PWM-Signal with a pulse time lower than 1.1 the output will go to LOW.

Additionally the 7-Segment-Display will always show how many outputs are currently active (HIGH).

Pinout

#	Input	Output	Bidirectional
0	input PWM of channel 0	segment a	output of channel 0
1	input PWM of channel 1	segment b	output of channel 1
2	input PWM of channel 2	segment c	output of channel 2
3	input PWM of channel 3	segment d	output of channel 3
4	input PWM of channel 4	segment e	output of channel 4
5	input PWM of channel 5	segment f	output of channel 5
6	input PWM of channel 6	segment g	output of channel 6
7	input PWM of channel 7	UART Transmit Wire	output of channel 7

spi_pwm [97]

- Author: djuara
- Description: This is a PWM generator and 8-bit width IO, spi controlled (2 different interfaces, just for testing).
- GitHub repository
- HDL project
- Mux address: 97
- Extra docs
- Clock: 0 Hz

How it works

This design is an SPI controlled PWM generator and 8-pin IO controller. IOs can be configure as output or input. Through registers we can configure number of ticks the PWM signal is ON and the cycle. Ticks are related to the system clk provided externally.

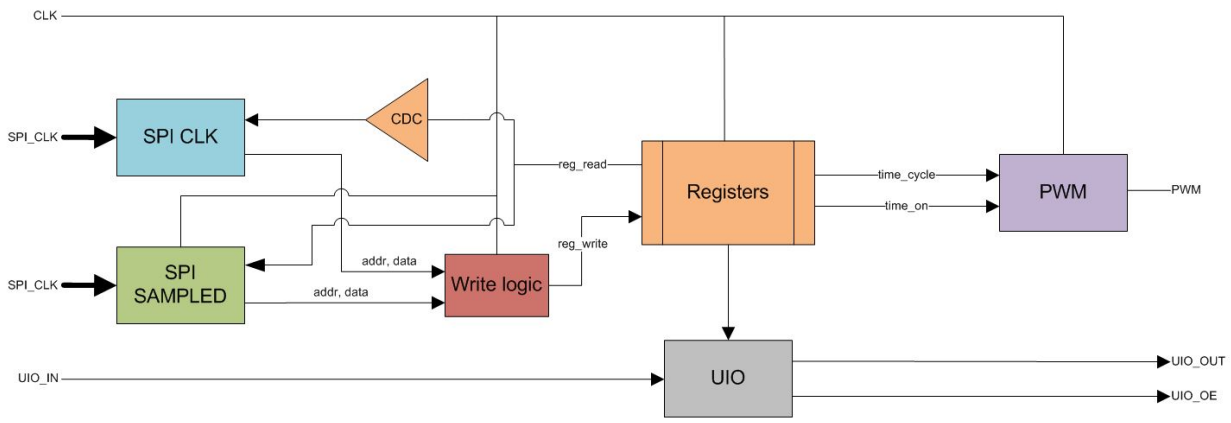


Figure 7: alt text

The design contain 8 registers that can be accessed by the two SPI interfaces. With these registers user can control PWM generator, allowing control of time on and cycle time. Also there are 8 IOs that can be set as inputs or outputs.

If two SPI writes occurs at the same time, SPI_CLK prevails over SPI_SAMPLED.

Configuration example

PWM Configuration of PWM is based on system clk. Registers to be configured are TICKS_ON and TICKS_CYCLE, which is basically the number of ticks of system clk the pwm signal is on and the period.

So assuming a system clk of 50 MHz, if we want to obtain a PWM signal with period 1 ms and duty cycle of 33%:

We need to calculate the number of clk ticks that are in 1 ms:

cycle_ticks = T / T_clk = 1 ms / (1 / 50 MHz) = 50 MHz * 1 ms = 50000 ticks

And now calculate the number of clk ticks the signal is on:

on_ticks = cycle_ticks * duty_cycle = 50000 * 0.33 = 16500 ticks

So configuring the registers with these values, and activating PWM (through external signal or register)

IOs In order to use the IOs, we just need to configure the IO_DIR register in order to set the pin as input or output.

Then, if it is an input, just read the IO_VALUE register, and if it is an output, just write the desired value to the IO_VALUE register.

Ports

Port	in/out	Description
ui_in[7]	in	Input and'ed with ena and reported in bit 7 of reg 0x01
ui_in[6]	in	Control start of PWM externally
ui_in[5]	in	CS signal of SPI_SAMPLED
ui_in[4]	in	MOSI signal of SPI_SAMPLED
ui_in[3]	in	SCLK signal of SPI_SAMPLED
ui_in[2]	in	CS signal of SPI_CLK
ui_in[1]	in	MOSI signal of SPI_CLK
ui_in[0]	in	SCLK signal of SPI_CLK
uo_out[7:3]	out	Always 0
uo_out[2]	out	PWM output
uo_out[1]	out	MISO signal of SPI_SAMPLED
uio_in[7:0]	in	Input signals of IOs
uio_out[7:0]	out	Output signals of IOs
uio_oe[7:0]	out	OE signals of IOs
ena	in	Design selected signal
clk	in	System clk

Port	in/out	Description
rst_n	in	Active low reset

Registers

Reg	Addr	Addr	Description	Default
ID	0x00	R	Identification register	0x96
PWM_CTRL	0x01	R/W	Control register	0x00
TICKS_ON_LSB	0x02	R/W	Ticks PWM signal is on LSB	0x14
TICKS_ON_MSB	0x03	R/W	Ticks PWM signal is on MSB	0x82
TICKS_CYCLES_LSB	0x04	R/W	PWM period in ticks LSB	0x50
TICKS_CYCLES_MSB	0x05	R/W	PWM period in ticks MSB	0xC3
IO_DIR	0x06	R/W	Set the dir of each IO pin	0x00
IO_VALUE	0x07	R/W	Set the IO_output value	0x00

Only 3 bits of address are taken into account for addressing.

When PWM is active, registers cannot be written.

ID This register is read only, it's value is 0x96.

PWM_CTRL This register controls the PWM. Bit 0 control if it's on (Bit 0 set) or off (Bit 0 clear). This register also contain the AND value of inputs ui_in[7] & ena in bit 7.

TICKS_ON LSB and MSB This two registers contains the number of ticks of the system clk that the PWM signal is high. It's a 16 bit wide value, separate in LSB and MSB.

TICKS_CYCLES LSB and MSB This two registers contains the period of the PWM signal in number of ticks of the system clk. It's a 16 bit wide value, separate in LSB and MSB.

IO_DIR In this register each bits configure the direction of each io pin. Value 0 indicates input and value 1 indicate output

IO_VALUE This register contain the value of the io pin. When read it reports the values of uio_in, when writes it sets the values of uio_out (depending on values set in IO_DIR).

SPI Interfaces Registers are accesed through one of the two SPI interfaces. Both interfaces share the access to the registers, so just one interface can be accessed at the same time.

SPI CLK This interface is clocked with the sclk clock of the SPI.

To write a register, 16 bits must be written.

- Bit 15 (MSB, first sent) is the R/W bit, for writes, must be 0
- Bits 14 to 11 are ignored
- Bit 10 to 8 is address
- Bit 7 to 0 is data to be written

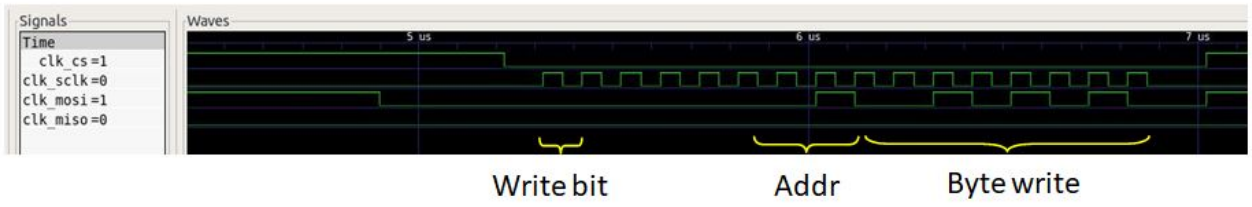


Figure 8: alt text

To read a register, 24 bits must be sent

- Bit 23 (MSB, first sent) is the R/W bit, for reads, must be 1
- Bits 22 to 19 are ignored
- Bit 18 to 16 is address
- Bit 15 to 8 is dummy bits
- Bit 7 to 0 is data read in MISO line

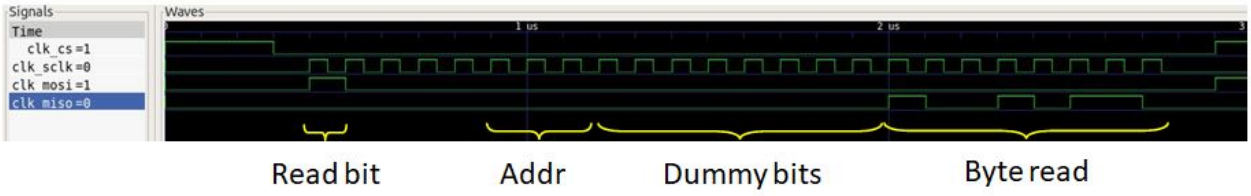


Figure 9: alt text

SPI SAMPLED This interface is sampled with the system clk. Theoretical maximum frequency is 25e10^6

To write a register, 16 bits must be written.

- Bit 15 (MSB, first sent) is the R/W bit, for writes, must be 0
- Bits 14 to 11 are ignored
- Bit 10 to 8 is address
- Bit 7 to 0 is data to be written

Figure 10: alt text

To read a register, 16 bits must be sent

- Bit 15 (MSB, first sent) is the R/W bit, for reads, must be 1
- Bits 14 to 11 are ignored
- Bit 10 to 8 is address
- Bit 7 to 0 is data read in MISO line

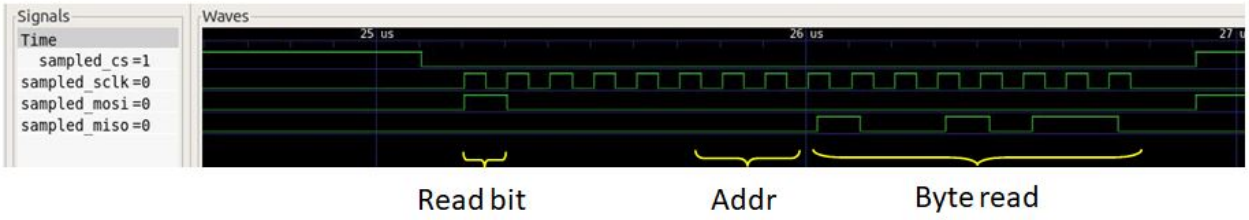


Figure 11: alt text

How to test

In order to test reads, you can read the ID register (0x00) and the byte received should be 0x96.

In order to test writes, you can write a register different than ID register, and then read it back an check you read the value previously written.

External hardware

Some devices to peform SPI transactions

Pinout

#	Input	Output	Bidirectional
0	clk_sclk	clk_miso	IO0
1	clk_mosi	sampled_miso	IO1
2	clk_cs	pwm	IO2
3	sampled_sclk		IO3
4	sampled_mosi		IO4
5	sampled_cs		IO5
6			IO6
7			IO7

PiMAC [99]

- Author: Steffen Reith
- Description: A simple pipelined multiply and accumulate unit to compute a*b+c
- GitHub repository
- HDL project
- Mux address: 99
- Extra docs
- Clock: 0 Hz

How it works

This circuit is a simple pipelined multiply and accumulate unit to compute a*b+c using SpinalHDL as a generator.

It uses the classic textbook method of multiplication with base 2. So if the numbers a and b are multiplied, the sum of the version of argument a shifted to the left by i bits must be summed up if and only if the ith bit of b is 1.

These bit products, i.e. (a « i) * b(i), are determined in the individual stages of the pipeline and the result is calculated step by step.

The full code can be found at https://github.com/SteffenReith/PiMAC

How to test

Simply feed a, b, and c as 4 bit unsigned integer into the unit. The latency is 3 clocks, hence the (hopefully correct) answer can be found at the result output after 3 cycles.

External hardware

No external hardware it needed.

Pinout

#	Input	Output	Bidirectional
0	a[0]	result[0]	c[0]
1	a[1]	result[1]	c[1]
2	a[2]	result[2]	c[2]
3	a[3]	result[3]	c[3]

#	Input	Output	Bidirectional
4	b[0]	result[4]	
5	b[1]	result[5]	
6	b[2]	result[6]	
7	b[3]	result[7]	

PILIPINAS [101]

- Author: Alexander Co Abad and Dino Dominic Ligutan
- Description: 7-seg Display for PILIPINASLASALLE
- GitHub repository
- Wokwi project
- Mux address: 101
- Extra docs
- Clock: 1 Hz

How it works

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

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

Setting the input pin 7 to HIGH allows for manual override of the BCD value. In this mode, input pins 0-3 controls the BCD value. The text displayed for each BCD value are tabulated below: | **in0** | **in1** | **in2** | **in3** | **Character** | |:——-:|:——-:|:——-:|:——-:|:————-:| | LOW | LOW | LOW | LOW | P | | LOW | LOW | LOW | HIGH | I | | LOW | LOW | HIGH | LOW | L | | LOW | LOW | HIGH | HIGH | I | | LOW | HIGH | LOW | LOW | P | | LOW | HIGH | LOW | HIGH | I | | LOW | HIGH | HIGH | LOW | N | | LOW | HIGH | HIGH | HIGH | A | | HIGH | LOW | LOW | LOW | S | | HIGH | LOW | LOW | HIGH | L | | HIGH | LOW | HIGH | LOW | A | | HIGH | LOW | HIGH | HIGH | S | | HIGH | HIGH | LOW | LOW | A | | HIGH | HIGH | LOW | HIGH | L | | HIGH | HIGH | HIGH | LOW | L | | HIGH | HIGH | HIGH | HIGH | E |

How to test

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

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

External hardware

7-segment display

Pinout

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

GOA - grogu on ASIC [103]

- Author: Simone Corbetta
- Description: Single neuron w/ fixed-point arithmetic
- GitHub repository
- HDL project
- Mux address: 103
- Extra docs
- Clock: 25 Hz

GOA - grogu on ASIC

`GOA` stands for `grogu on ASIC`. It is a reduced version of the CORTEZ chip targeting the Tiny Tapeout 6 run. The `grogu` part comes from the register file design utilities grogu.

The `GOA` design is made of a single neuron with 2 (two) inputs. The register file contains a number of registers for control and observation. The neuron core works on 8 (eight) bits fixed-point arithmetic with 5 (five) reserved for the fraction.

Neuron internals

The next figure shows the simplified block diagram of the Neuron.

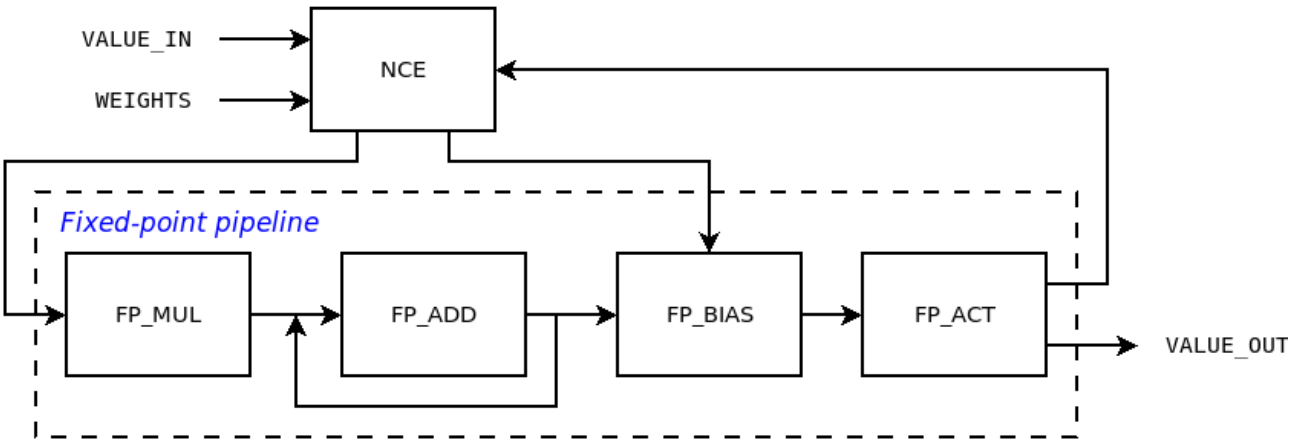


Figure 12: Neuron architecture

The arithmetic pipeline is composed of a number of fixed-point units: multiplier, adder for accumulator and bias, activation function. These primitives are shared, so that a centralized control engine (`NCE`) dispatches one value at a time to the proper block. `WEIGHTS` matrix is externally stored in a local register file, an instance of `grogu`.

The NCE expects exactly NUM_INPUTS input values. For each of them, the following process is executed through the pipeline:

1. Multiply current value by its respective weight;
2. Accumulate the value.

Once all values have been received, bias is added and the non-linear activation function is used to determine the output solution.

Fixed-point The entire pipeline works with fixed-point arithmetic. This reduces the complexity of the design. For the Tiny Tapeout run, the fixed-point configuration is: 8 (eight) bits word with 5 (five) bits reserved to the fractional part. All fixed-point operations wrap.

Non-linear activation function tanh() is the chosen non-linear activation function. Thanks to its mathematical properties, it is interesting designing a fully digital filter that implements a piecewise approximation.

Linear interpolation between successive points is carefully chosen to minimize the error. The points where the tanh() function is split are chosen by looking at up to the 4-th derivative. Since the tanh() function is odd symmetric, the digital implementation focuses on half of the problem in the 1st quadrant. The other half of the problem on the 3rd quadrant is derived. The output is shown.

Scalable Configuration Interface The SCI interface has been designed for the CORTEZ chip to address dense register files with a fairly large amount of registers. The SCI is designed to reduce wires and congestion. It consists of an half-duplex communication mechanism with request/ack pairs, useful for low-speed peripheral register access. Is is also latency insensitive. The SCI is inspired by the SPI protocol, with tri-state bus and active-low chip select.

For the single neuron case, the SCI bus is not tri-stated, since there is one single peripheral. This simplfies the implementation.

In general, the SCI interface for one Master and N Slaves is composed of 4 (four) signals (mapping to the tt_um_scorbetta_goa pins is reported in the Pinout section).

SIGNAL	WIDTH	DIRECTION	ROLE
SCI_CSN	N	Master-to-Slave	Active-low peripheral select
SCI_REQ	1	Master-to-Slave	Request
SCI_RESP	1	Slave-to-Master	Response

SIGNAL	WIDTH	DIRECTION	ROLE
SCI_ACK	1	Slave-to-Master	Ack

For detailed information on the SCI protocol please refer to this page.

Examples of Write and Read accesses are shown.

Network emulation A twisted use of the single-neuron design can emulate an entire network made of a number of layers, each with a number of neuron. This is doable thanks to the way the neuron is designed. Basically, the 2-inputs neuron is repeadetely fed with iterative data, coming from either the external world (i.e., input values) or intermediate results (i.e., from the inner core). Mathematically, the MAC operation is distributed in time.

Pinout

PIN	DIRECTION	ROLE
ui_in[0]	input	FPGA clock
ui_in[1]	input	Active-low FPGA reset
ui_in[2]	input	Loopback data
ui_in[3]	input	Unused
ui_in[4]	input	Unused
ui_in[5]	input	Unused
ui_in[6]	input	Debug select [0]
ui_in[7]	input	Debug select [1]
uo_out[0]	output	Shared debug output dbug_out[0]
uo_out[1]	output	Shared debug output dbug_out[1]
uo_out[2]	output	Shared debug output dbug_out[2]
uo_out[3]	output	Shared debug output dbug_out[3]
uo_out[4]	output	Shared debug output dbug_out[4]
uo_out[5]	output	Shared debug output dbug_out[5]
uo_out[6]	output	Shared debug output dbug_out[6]
uo_out[7]	output	Shared debug output dbug_out[7]
uio_in[0]	input	SCI_CSN
uio_in[1]	input	SCI_REQ
uio_out[2]	output	SCI_RESP
uio_out[3]	output	SCI_ACK
uio_out[4]	input	Unused, configured as input
uio_out[5]	input	Unused, configured as input

PIN	DIRECTION	ROLE
uio_out[6]	input	Unused, configured as input
uio_out[7]	input	Unused, configured as input

Debug signals are mapped to output pins uo_out. In total, 32 (thirty-two) signals are exposed to the debug interface. Inputs ui_in[7:6] are used to control which ones, according to the following table.

Configuration

The configuration of the neuron is implemented by means of local registers that hold the values for the weights and the bias. In addition, control registers are used to trigger the neuron operations. All resigsters are 8 (eight) bits wide

REGISTER	OFFSET	TYPE	CONTENTS
WEIGHT_0	0x0	R/W	Weight of input #0
WEIGHT_1	0x1	R/W	Weight of input #1
BIAS	0x2	R/W	Bias
VALUE_IN	0x3	R/W	Input value
CTRL	0x4	R/W	Control register
STATUS	0x5	R	Status register
RESULT	0x6	R	Neuron solution
MULT_RESULT	0x7	R	Intermediate multiplie result
ADD_RESULT	0x8	R	Intermediate adder result w/o bias
BIAS_ADD_RESULT	0x9	R	Intermediate adder result w/ bias

External hardware

The main clock clk is generated by the on-board RP2040 chip. It is used solely for debug purposes. It is mirrored to uo_out[1].

The core clock is instead drawn from ui_in[0]. This is generated by an FPGA residing on an external board. ui_in[0] and clk are mesochronous, and they never interact.

The use of an external clock is required, since the SCI interface (also driven by the FPGA) needs proper synchronization. The FPGA also drives the active-low core reset through ui_in[1]. All control and status information is sent to and retrieved from the ASIC through the SCI interface.

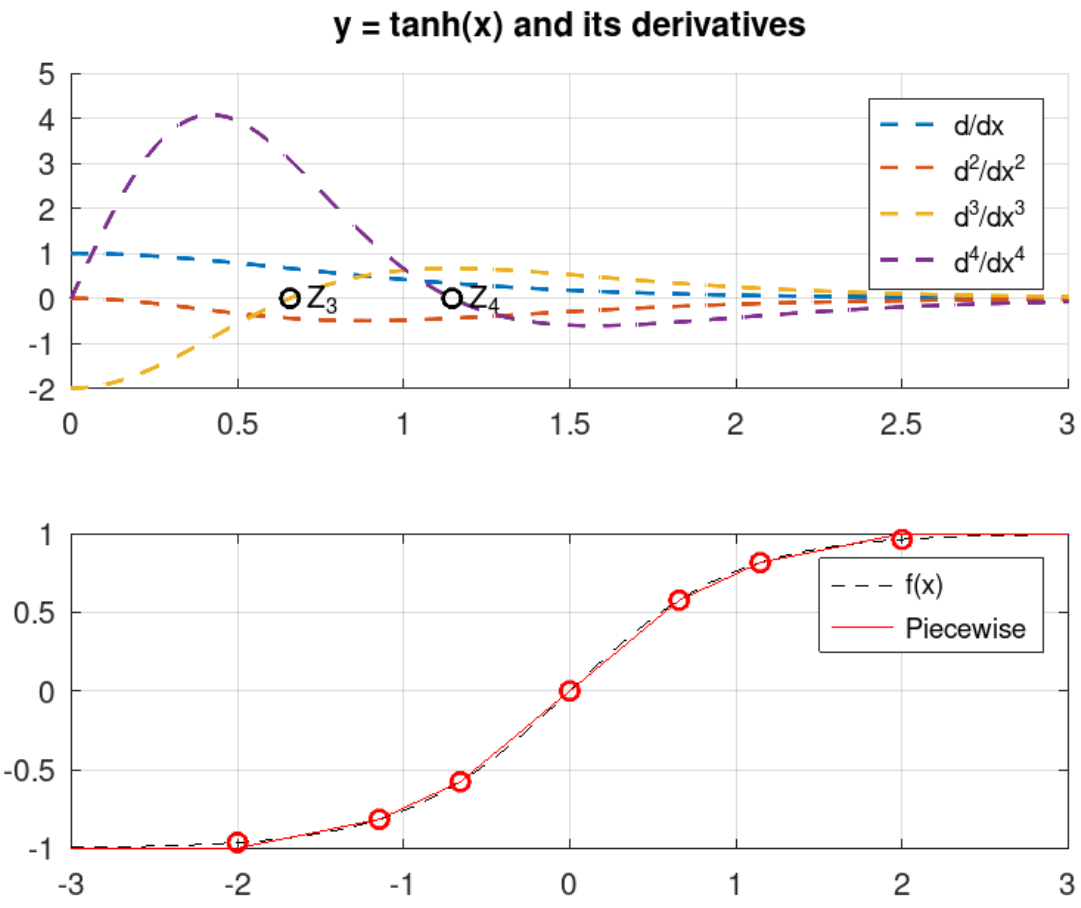


Figure 13: Derivatives

Example: Write 8'hf0 at address 4'ha

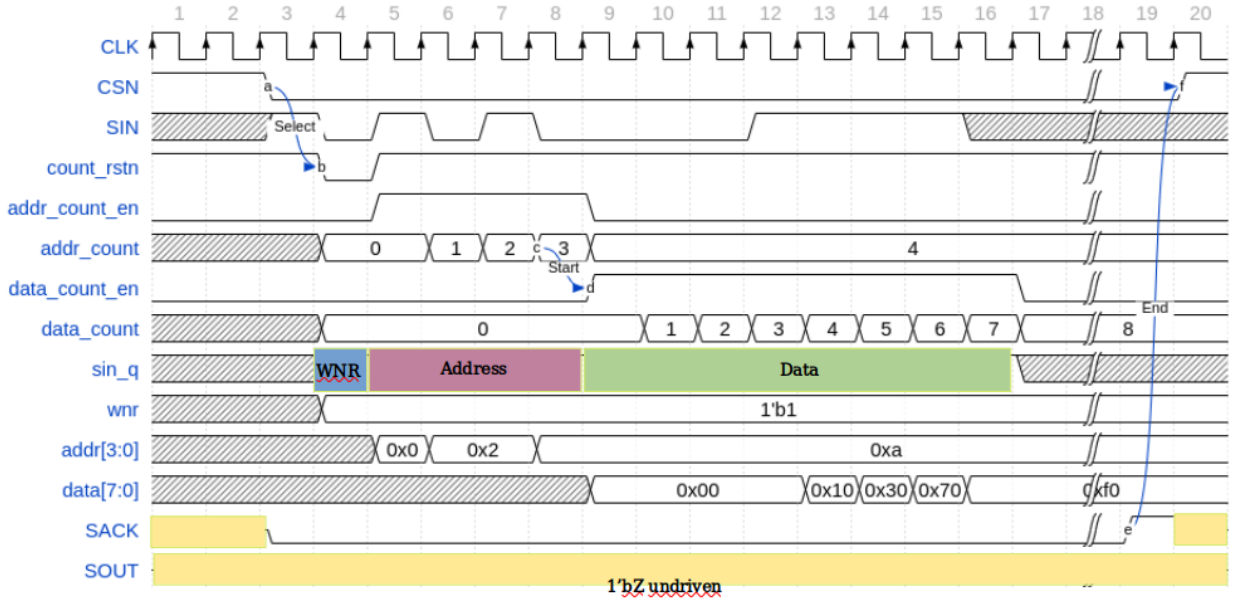


Figure 14: Write access

Example: Read from address 4'ha

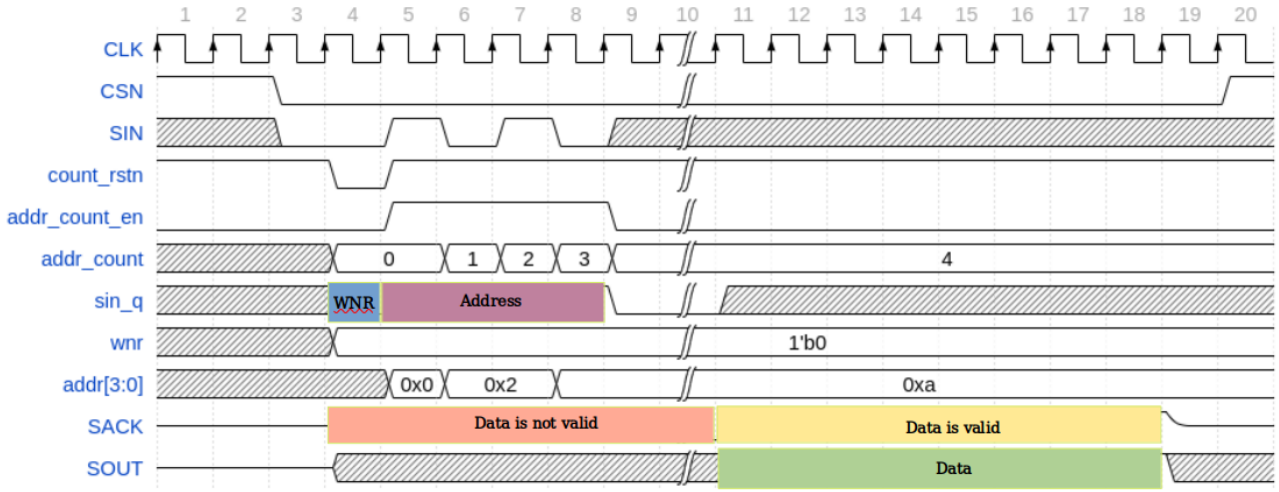
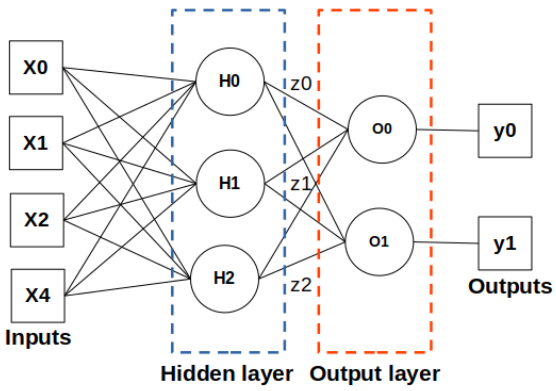


Figure 15: Read access

Fully-connected network emulation



$$H_j : z_j = f(x_0 w^h_{0j} + x_1 w^h_{1j} + x_2 w^h_{2j} + x_3 w^h_{3j})$$

$$O_k : y_k = f(z_0 w^o_{0k} + z_1 w^o_{1k} + z_2 w^o_{2k})$$

$$temp^1_0 = x_0 w^h_{00} + x_1 w^h_{01}$$
$$temp^3_2 = x_2 w^h_{02} + x_3 w^h_{03}$$
$$z_0 = f(temp^1_0 \cdot 1 + temp^3_2 \cdot 1)$$

Figure 16: Network emulation

	ui_in[7:6]			
uo_out	2'b00	2'b01	2'b10	2'b11
uo_out[0]	ena	loopback	start	chip_id[3]
uo_out[1]	overflow	ready	valid_out	chip_id[2]
uo_out[2]	soft_reset	WREQ	valid_out_latch	chip_id[1]
uo_out[3]	open_req	WACK	mul_start	chip_id[0]
uo_out[4]	addr_count_en	RREQ	mul_done	mul_overflow
uo_out[5]	data_count_en	RVALID	add_done	add_overflow
uo_out[6]	ni_wreq	rstn_i	bias_add_done	bias_add_overflow
uo_out[7]	ni_rreq	rdata_shift	act_done	act_overflow

Figure 17: Debug signals mux

Pinout

#	Input	Output	Bidirectional
0	FPGA clock	Shared debug output dbug_out[0]	SCI_CSN
1	Active-low FPGA reset	Shared debug output dbug_out[1]	SCI_REQ
2	Loopback data	Shared debug output dbug_out[2]	SCI_RESP
3		Shared debug output dbug_out[3]	SCI_ACK
4		Shared debug output dbug_out[4]	
5		Shared debug output dbug_out[5]	
6	Debug select [0]	Shared debug output dbug_out[6]	
7	Debug select [1]	Shared debug output dbug_out[7]	

TTRPG Dice + simple I2C peripheral [105]

- Author: Jonas Nilsson
- Description: TTRPG dice roller
- GitHub repository
- HDL project
- Mux address: 105
- Extra docs
- Clock: 32768 Hz

How it works

Press buttons to roll various types of TTRPG dice Playing table top role playing games (TTRPGs) often involves rolling dice of various types. This design is a combination of the most common types of dice used.

It has six button inputs, each corresponding to a certain type of die, and a two-digit seven segment display that shows the result of the roll when the button is released.

While a button is pressed, a counter is decremented every clock cycle. When the counter reaches 1, it is reloaded with the largest value of the corresponding die. When the button is released, the counter stops and the result is displayed on the seven segment display. Around 8 seconds later, the display is turned off.

The design outputs seven segment signals and 'common' drivers for two digit displays. It has configuration pins that set the active level of segment and common signals independently of each other, to allow the connection of either common-cathode or common-anode diplays, or displays with inverting or non-inverting drive buffers for segments and/or common signals. Similarly, the button inputs can be configured as active low or high.

Dice up to d10 can use a single seven segment display without a common driver like the one on the demo board. If such a display is used, it will toggle between showing the 1s digit and the blanked 10s digit. When the result is 10, it will show a 1 and 0 superimposed, which will look like a slightly wonky 0.

The design uses clock timing to debounce the buttons and is optimized to run at 32768 Hz, but it should work well at frequencies from 10kHz to 100kHz. At higher frequencies, the button debouncer may be unreliable and display muxing may not work properly. At lower frequencies, the higher valued dice will have a low cycle rate and it could be possible to cheat by using well-timed key presses. The clock frequency will also affect the timer that turns off the display.

The 'polarity' config pins sets the active level for the corresponding I/O signals. For instance: uio[7]=0 causes the digit mux signals to be active low, suitable for directly driving common cathode pins. When uio[6]=1, lit segments are high, suitable for direct segment drive of common cathode displays. Similarly, when uio[5]=0 button inputs are expected to be high when idle and low when pressed.

How to test

Set clock frequency to 32768 Hz (10-100 kHz). Configure uio[7:5] for the appropriate signal polarity:

Seven segment	uio[7:6]
Common cathode, direct segment drive (demo board)	01
Common anode, direct segment drive	10
Inverting common anode drive, direct segment drive	00

For the demo board, set uio[7:5] to 010

Press one of the buttons ui[6:0] (according to the selected button polarity) and release it. The dice roll is shown on the LED display for about 8 seconds.

- ui[0] rolls a d4 (four sided die).
- ui[1] rolls a d6
- ui[2] rolls a d8
- ui[3] rolls a d10
- ui[4] rolls a d12
- ui[5] rolls a d20
- ui[6] rolls a d100

External hardware

Pullups on ui[6:0] with pushbuttons closing to GND.

A two-digit LED display. Common anode and/or cathode are supported using the configuraiton pins. Segments are connected to uo[7:0] (DP, G, F, E, D, C, B, A in that order) Left cathode connected to uio[1] Right cathode to uio[0]

Static configuration inputs on uio[7:5] should be connected to VDD or GND.

The chip may struggle to supply common anode displays with enough current. If so, drive the common anode pin with an inverting transistor driver and change the active level of the 'common' output by setting uio[7] to 0.

But wait: There's more!

The die roller only used 1/3 of the available area, and I had a few spare pins, so I also added a simple I2C slave to experiment with. It contains an 8 byte memory and a GPIO unit that can read the ui pins, and a GPIO pin uio[0] that can be used as input or output, and also has PWM capabilities.

The slave has a 7-bit I2C address 0x70. Communicate with it as if it were an I2C memory with a one byte address: Make a write transaction where the first byte is the sub-adress, followed by any number of data bytes. The data bytes will be stored in successive locations. Make a read transaction by first making a dummy write without data bytes to set the sub-address, followed by a restart and a read transaction to read any number of consecutive bytes back.

Address Map The 7-bit I2C slave address is 0x70.

The address map of the peripheral is as follows:

Address	Function
0x0 - 0x7	Memory cells
0x8	IOCtrl
0x9	IO_oe
0xA	uio_in (r/o)
0xB	ui_in (r/o)

IOCtrl Write 0 to set uio[0] to 0
Write 0 to 128 to output a PWM waveform with duty cycle IOCtrl/128
Write >128 to output 1

IO_oe Bit 0 = 0 configures uio[0] as an input
Bit 0 = 1 configures uio[0] as an output

uio_in, ui_in Reads the current values of the uio and ui pins.
Remember that the dice roller is still active, so you will see things happening on the uio[4:3] pins, as well as the state of the I2C pins.

Testing the I2C slave Set the clock frequency to 10 MHz or above. It should be possible to access the I2C slave from the I2C1 interface of the RP2040 or from an I2C master connected to the PMOD interface J12, where you can also access he PWM output.

Pinout

#	Input	Output	Bidirectional
0	Btn4	segA	PWM capable GPIO pin (bidir)
1	Btn6	segB	SDA (bidir)
2	Btn8	segC	SCL (input)
3	Btn10	segD	1s digit common (output)
4	Btn12	segE	10s digit common (output)
5	Btn20	segF	Button polarity (input)
6	Btn100	segG	Segment polarity (input)
7	GPIO input	DP	Common polarity (input)

i4004 for TinyTapeout [107]

- Author: ISHI-KAI
- Description: i4004 for TinyTapeout by ISHI-KAI.
- GitHub repository
- HDL project
- Mux address: 107
- Extra docs
- Clock: 741000 Hz

How it works

This is i4004 Chip.

How to test

No Test.

External hardware

-MCS4 Memory System

Pinout

#	Input	Output	Bidirectional
0	data0_pad	cmrom_pad	
1	data1_pad	cmram0_pad	
2	data2_pad	cmram1_pad	
3	data3_pad	cmram2_pad	
4	clk1_pad	cmram3_pad	
5	clk2_pad	sync_pad	
6	poc_pad		
7	test_pad		

Moving average filter [108]

- Author: Alexander Hofer
- Description: 10-bit moving average filter designed to smooth input data streams.
- GitHub repository
- HDL project
- Mux address: 108
- Extra docs
- Clock: 0 Hz

How it works

The design implements a moving average filter using a series of registers and a finite state machine (FSM). The filter calculates the average of a set of recent values in a data stream, determined by the FILTER_POWER parameter. This smooths out short-term fluctuations and highlights longer-term trends or cycles. The master module handles input and output processing, including bidirectional IO handling and filter selection based on input control signals.

How to test

To test the moving average filter, provide a series of digital input values to the 'ui_in' port and observe the smoothed output on 'uo_out'. The 'uio_in' can be used to control the filter's width and operational parameters. Test with varying input patterns and filter widths to evaluate the filter's response.

External hardware

There is no external Hardware

Pinout

#	Input	Output	Bidirectional
0	Input for filter	Output for filter	Strobe input
1	Input for filter	Output for filter	Strobe output
2	Input for filter	Output for filter	Additional input bit
3	Input for filter	Output for filter	Additional input bit
4	Input for filter	Output for filter	Additional output bit

#	Input	Output	Bidirectional
5	Input for filter	Output for filter	Additional output bit
6	Input for filter	Output for filter	Filter width input
7	Input for filter	Output for filter	Filter width input

Synthesized Time-to-Digital Converter (TDC) v2 [109]

- Author: Harald Pretl
- Description: Synthesized TDC based on two Vernier delay rings
- GitHub repository
- HDL project
- Mux address: 109
- Extra docs
- Clock: 50000000 Hz

How it works

This is a synthesized time-to-digital converter (TDC), consisting of two wavefront delay rings with a slightly different delay forming a Vernier TDC.

The time between the rising edge of `start=ui_in[0]` and the rising edge of `stop=ui_in[1]` is measured by both rings and the output in 8b chunks. Based on analog simulation, the time resolution (typical process, room temperature) is on the order of 6ps.

How to test

Apply two signals to `ui_in[0]` and `ui_in[1]`.

After capturing (rising edge of `ui_in[1]`) the result (i.e., the time delay between rising edge of `ui_in[0]` and `ui_in[2]`) can be muxed-out to `uo_out[7:0]` using `ui_in[7:3]` as byte-wise selector. `ui_in[7:3]=0000` gives result byte 0, `ui_in[7:3]=0001` gives result byte 1, etc.

The input `ui_in[2]` selects the output of ring 0 or ring 1.

External hardware

Two signal generators generating the logical signals with a programmable delay (important is ns resolution).

Pinout

#	Input	Output	Bidirectional
0	Start signal of TDC	Result (LSB)	
1	Stop signal of the TDC	Result	
2	Select result of ring for output	Result	
3	output select (LSB)	Result	
4	output select	Result	
5	output select	Result	
6	output select	Result	
7	output select (MSB)	Result (MSB)	

SPI to RGBLED Decoder/Driver [110]

- Author: Andreas Scharnreitner
- Description: Control multiple RGB LEDs (WS2812B) via SPI
- GitHub repository
- HDL project
- Mux address: 110
- Extra docs
- Clock: 25000000 Hz

How it works

When nCS is pulled low, each clock pulse on SCLK shifts a bit from MOSI into an internal register. The internal register length is 240 bits long (10 LEDs with 3 colors and 8 bit per color). The contents of this register are then used to generate output pulses. The output pulses encode bits of the color data. They are 1.25us in length. A pulse representing a 1 has a high-time of 800ns and an low-time of 450ns. A pulse representing a 0 has a high-time of 400ns and a low-time of 850ns. Each LED consumes 24 bits. Subsequent bits are transmitted to LEDs further on the chain. When a full transmission (Every LED has received its 24 bits of color data) has occured, a reset occurs (output goes low for >= 50 us).

How to test

Connect the LED_DATA pin to the DIN pin of a string of WS2812B LEDs. Use a microcontroller to shift in color data via the SPI Interface.

External hardware

Any SPI Master (RPi, Arduino, MCU, etc.), and a String of 10 WS2812B LEDs.

Pinout

#	Input	Output	Bidirectional
0	MOSI	LED_DATA	
1	SCLK		
2	nCS		
3			

#	Input	Output	Bidirectional
4			
5			
6			
7			

8-bit CPU with Debugger (Lite) [111]

- Author: Sean Patrick O'Brien
- Description: 8-bit CPU with debugger accessible via I2C
- GitHub repository
- HDL project
- Mux address: 111
- Extra docs
- Clock: 0 Hz

How it works

The CPU is based on Ben Eater's 8-bit breadboard CPU. A built-in debugger allows pausing the CPU, loading programs, inspecting/modifying registers, etc.

How to test

The debugger is accessible over I2C at address 0x2A (0x54 write, 0x55 read). The provided dbg program can be used to load programs, inspect registers, etc.

External hardware

Optionally, data can be provided on the input pins and consumed on the output pins. They are accessible to the CPU as the IN and OUT registers.

Pinout

#	Input	Output	Bidirectional
0	Input Port	Output Port	
1	Input Port	Output Port	
2	Input Port	Output Port	SCL
3	Input Port	Output Port	SDA
4	Input Port	Output Port	HALTED
5	Input Port	Output Port	
6	Input Port	Output Port	
7	Input Port	Output Port	

FIR Filter with adaptable coefficients [128]

- Author: Markus Häusler
- Description: FIR Filter with configurable coefficients via serial interface
- GitHub repository
- HDL project
- Mux address: 128
- Extra docs
- Clock: 0 Hz

How it works

The FIR FILTER ADAPT is a project that performs a FIR Filter behaviour.

The Input of the filter is provided by the 8 input pins. These represent an signed integer. The Chip performs than a multiplikation with the filter-tap values and adds them up. The Filter needs as many clock cycles as taps to calculate the Filter output. After that the filtered signal is set to the output by using the output pins.

To optimize calculation and memory requirements, the symmetric step response property of a FIR Filter is used. That means only one half of the step response is stored and to calculate the whole filter characteristic, the first half of the step response is flipped.

In addition to the basic FIR Filter functionality, it is also possible to adapt the filter coefficients by enabling the configuration bit, then the input pins are interpreted as tap value and are shifted into the FIR tap memory

How to test

The Testbench consists of three testcases which can be compared to a real life measurement to ensure correct functionality. These Tests are:

- Impulse Response with the Initial FIR Coefficents.
- Configuration of new Filter Coefficients and testing them by performing another impulse response.
- Perfoming a step response with the new filter coefficients.

For more detailed information check out the test.py file.

External hardware

This can be done by any microcontroller or if one is interested in the functionlity of an FIR Filter, even using buttons as input and leds as output is sufficent. But i recommend using at least an arduino microcontroller because timing and reproducibility is much more easily using automated testing. Have fun!

Pinout

#	Input	Output	Bidirectional
0	FIR Input data Bit 0	FIR Output data Bit 0	FIR Output data Bit 8
1	FIR Input data Bit 1	FIR Output data Bit 1	FIR Output data Bit 9
2	FIR Input data Bit 2	FIR Output data Bit 2	FIR Output data Bit 10
3	FIR Input data Bit 3	FIR Output data Bit 3	not used
4	FIR Input data Bit 4	FIR Output data Bit 4	not used
5	FIR Input data Bit 5	FIR Output data Bit 5	not used
6	FIR Input data Bit 6	FIR Output data Bit 6	FIR CONFIG ENABLE
7	FIR Input data Bit 7	FIR Output data Bit 7	FIR TVALID Input

Karplus-Strong String Synthesis [132]

- Author: Chinmay Patil
- Description: A KS String Audio Synthesizer
- GitHub repository
- HDL project
- Mux address: 132
- Extra docs
- Clock: 256000 Hz

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 are sent out through the I2S transmitter 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 through the I2S transmitter interface.

External hardware

An I2S DAC The 8-bit signed sound samples are sent out at `f_sck = 256 kHz` through this interface.

Pinout

#	Input	Output	Bidirectional
0	~rst_n_prbs_15, ~rst_n_prbs_7		sck_i
1	load_prbs_15, load_prbs_7		sdi_i
2	freeze_prbs_15		sdo_o
3	freeze_prbs_7		cs_ni

#	Input	Output	Bidirectional
4	i2s_noise_sel		i2s_sck_o
5	~rst_n_ks_string		i2s_ws_o
6	pluck		i2s_sd_o
7			prbs_15

ADPCM Encoder Audio Compressor [136]

- Author: Charlie Hess, Emil Ivanov
- Description: Accepts a microphone PDM input and returns an ADPCM encoded/compressed ouput
- GitHub repository
- HDL project
- Mux address: 136
- Extra docs
- Clock: 0 Hz

How it works

This HDL block accepts a pulse density modulated (PDM) microphone signal and produces an encoded output at a lower sampling frequency while maintaining audio intelligibility.

Expected Inputs: clk (clk) slow_clk (ui_in[1]) for the ADPCM block at 1/8 frequency of clk Pulse Density Modulated input pdm_in (ui_in[2]) at clocked with clk block_enable (ui_in[3]) (active high): single bit enable for the entire block

Outputs: encPcm (uo_out[4:1]): the final 4 bit ADPCM encoded output outValid (uo_out[0]): Output Valid flag for the ADPCM block, goes high for one cycle of slow_clk each time a new valid adpcm value is output

How to test

Provide two synchronized external clocks, clk and slow_clk, slow_clk a frequency 8x slower than clk (e.g. clk = 512 kHz, slow_clk = 64 khz) Connect the data pin of a pdm microphone clocked with clk to pdm_in. Intially set block_enable to low, then when recording/compression should begin, set block_enable to high. Now, a CIC decimated and ADPCM encoded output of the microphone data will stream from encPcm, which can be stored to memory, or decoded for writing to an audio file or playback.

Pinout

#	Input	Output	Bidirectional
0	clk	outValid	
1	slow_clk		
2	block_enable		

#	Input	Output	Bidirectional
3	pdm_in		
4		encPcm[0]	
5		encPcm[1]	
6		encPcm[2]	
7		encPcm[3]	

Ternary 1.58-bit x 8-bit matrix multiplier [142]

- Author: ReJ aka Renaldas Zioma
- Description: Matrix multiplication block for 1.58 bit aka TERNARY weight LLMs
- GitHub repository
- HDL project
- Mux address: 142
- Extra docs
- Clock: 48000000 Hz

How it works

Matrix multiplication is implemented using a systolic array architecture.

How to test

Every cycle feed packed weight data to Input pins and input data to Bidirectional pins. Strobe Enable pin to start receiving results of the matrix multiplication on the Output pins.

External hardware

MCU is necessary to feed weights and input data into the accelerator and fetch the results.

Pinout

#	Input	Output	Bidirectional
0	packed weights LSB	result LSB	(in) activations LSB
1	packed weights	result	(in) activations
2	packed weights	result	(in) activations
3	packed weights	result	(in) activations
4	packed weights	result	(in) activations
5	packed weights	result	(in) activations
6	packed weights	result	(in) activations
7	packed weights MSB	result MSB	(in) activations MSB

32-Bit Fibonacci Linear Feedback Shift Register [160]

- Author: icaris lab
- Description: 32-bit Fibonacci libear feedback shift register with taps at (32, 30, 26, 25).
- GitHub repository
- Wokwi project
- Mux address: 160
- Extra docs
- Clock: 50000000 Hz

How it works

The project is a hardware implementation of a maximum-cycle 32-bit Fibonacci linear feedback shift register (LFSR) with taps at registers (R32, R30, R26, R25). The LFSR is defined with the least-significant bit (LSB) at the left-most register R1 and the most-significant bit (MSB) at the right-most register R32. The LFSR shifts bits from left to right (R_n -> R_n+1), with the LSB populated by XORing bits from the tapped registers (R1 = R32 ^ R30 & R26 ^ R25). The LFSR contains an initialization/fail-safe feedback that prevents the LFSR from entering an all-zero state. If the LFSR is ever in an all-zero state, a "1" value is inserted into R1.

A schematic of the circuit may be found at:

https://wokwi.com/projects/394704587372210177

The circuit has 10 inputs:

Input	Setting
CLK	Clock
RST_N	Not Used
01	Not Used
02	Manual R0 Input Value
03	Input Select
04	Not Used
05	Not Used
06	Not Used
07	Not Used
08	Not Used

The CLK sets the clocking for the flip-flop registers for latching the LFSR values. In the schematic shown in the Wokwi project, a switch is used to select either the system

clock or an externally provided or manual clock that allows the user to manually step through each latching event.

An 8-input DIP switch provides some flexibility to initalizing the LFSR. DIP03 (IN2) allows the user to toggle the Input Select function, which is a multiplexer that select whether the left-most register (R1) takes in as the input the LFSR feedback output (i.e., (R1 = R32 ^ R30 & R26 ^ R25) or a value that is manually selected by the user.

DIP02 (IN1) allows a the user to manually enter a 0 or a 1 value into the leftmost register.

The cicuit has 8 outputs. They output the values of the 8 right-most registers (R25, R26, R27, R28, R29, R30, R31, R32).

Output	Value in
01	R25
02	R26
03	R27
04	R28
05	R29
06	R30
07	R31
08	R32

How to test

The circuit can be tested by powering on the circuit, and first setting the Input Select switch (DIP03) to "1" to reset/initialize the entire LFSR to all-zeros. The Input Select switch can then be switched to "0" to allow the LFSR to run from its all-zero initialized value. The first 100 8-bit output values of the LFSR from this zeroized state may be observed using a logic analyzer, and should be:

[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0], [0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0], [0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0], [0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0], [0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0], [0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0], [0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[1 0 0 0 0 0 0 0],[0 1 0 0 0 0 0 0], [0 0 1 0 0 0 0 0],[0 0 0 1 0 0 0 0],[0 0 0 0 1 0 0 0],[0 0 0 0 0 1 0 0], [0 0 0 0 0 0 1 0],[0 0 0 0 0 0 0 1],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0], [0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0], [0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0], [0 0 0 0 0 0 0

0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0], [0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[1 0 0 0 0 0 0 0], [1 1 0 0 0 0 0 0],[0 1 1 0 0 0 0 0],[0 0 1 1 0 0 0 0],[0 0 0 1 1 0 0 0], [1 0 0 0 1 1 0 0],[0 1 0 0 0 1 1 0],[1 0 1 0 0 0 1 1],[0 1 0 1 0 0 0 1], [0 0 1 0 1 0 0 0],[0 0 0 1 0 1 0 0],[0 0 0 0 1 0 1 0],[0 0 0 0 0 1 0 1], [0 0 0 0 0 0 1 0],[0 0 0 0 0 0 0 1],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0], [0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0], [0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0], [1 0 0 0 0 0 0 0],[0 1 0 0 0 0 0 0],[1 0 1 0 0 0 0 0],[0 1 0 1 0 0 0 0], [0 0 1 0 1 0 0 0],[0 0 0 1 0 1 0 0],[0 0 0 0 1 0 1 0],[0 0 0 0 0 1 0 1], [0 0 0 0 0 0 1 0],[0 0 0 0 0 0 0 1],[1 0 0 0 0 0 0 0],[0 1 0 0 0 0 0 0], [0 0 1 0 0 0 0 0],[0 0 0 1 0 0 0 0],[1 0 0 0 1 0 0 0],[0 1 0 0 0 1 0 0], [0 0 1 0 0 0 1 0],[0 0 0 1 0 0 0 1],[0 0 0 0 1 0 0 0],[0 0 0 0 0 1 0 0], [0 0 0 0 0 0 1 0],[0 0 0 0 0 0 0 1],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0], [0 0 0 0 0 0 0 0],[1 0 0 0 0 0 0 0]

A python implementation of the 32-bit Fibonacci LFSR can be found at the link below. It may be used for testing the hardware for sequences longer than the initial 100 values.

https://github.com/icarislab/tt06_32bit-fibonacci-prng_cu/main/docs/32-bit-fibonacci-prng_pythong_simulation.py

External hardware

No external hardware is required.

Pinout

#	Input	Output	Bidirectional
0		r25_val	
1	data_in	r26_val	
2	load_en	r27_val	
3		r28_val	
4		r29_val	
5		r30_val	
6		r31_val	
7		r32_val_LSFR_out	

Some_LEDs [161]

- Author: marsPRE
- Description: Nothing special
- GitHub repository
- Wokwi project
- Mux address: 161
- Extra docs
- Clock: 0 Hz

How it works

I don't know how it works, yet.

How to test

connect the right things

External hardware

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

Pinout

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

RGB Mixer [162]

- Author: Zhe Wang
- Description: Zero to ASIC demo project
- GitHub repository
- HDL project
- Mux address: 162
- Extra docs
- Clock: 10000000 Hz

How it works

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

How to test

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

External hardware

Use 3 digital encoders attached to the first 6 inputs.

Pinout

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

Workshop Hackaday Juli [163]

- Author: Juli
- Description: TinyTapeOut Workshop Hackaday project
- GitHub repository
- Wokwi project
- Mux address: 163
- Extra docs
- Clock: 0 Hz

How it works

I simply refuse to do this now

How to test

I simply refuse to do this now

External hardware

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

Pinout

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

Animated 7-segment character display [164]

- Author: Aron Dennen
- Description: Displays 7-segment characters with animation
- GitHub repository
- HDL project
- Mux address: 164
- Extra docs
- Clock: 12500000 Hz

How it works

Animates the 7-segment display by reading in the input switches to create a custom 7-segment character. Nothing will be displayed until you toggle input 7 to start the character animation sequence.

Inputs 0 through 6 map to outputs 0 though 6 (display segments a through g). Output 7 becomes active while input 7 is active.

The uio inputs are used for an experimental pwm dimming feature, to enable pwm display dimming, set uio pin 7 active. uio inputs 0 through 6 set a 7-bit pwm dimming value on an 8-bit pwm unit. The pwm lsb input is tied to 0.

The circuit works by iterating over the character bit pattern, enabling segments sequentially at a speed of about 0.12 seconds per segment.

How to test

Toggle the input switches to create a character with inputs 0-6, toggle input 7 to start the character animation sequence.

Optionally dim the display by enabling the pwm feature described above.

External hardware

none

Pinout

#	Input	Output	Bidirectional
0	input0	seg0	pwm_bit1
1	input1	seg1	pwm_bit2
2	input2	seg2	pwm_bit3
3	input3	seg3	pwm_bit4
4	input4	seg4	pwm_bit5
5	input5	seg5	pwm_bit6
6	input6	seg6	pwm_bit7
7	enable display	seg7	usePwm

Keypad Decoder [165]

- Author: Slobodan Vrkacevic
- Description: A simple controller that detects a pressed key in 4x4 keypad matrix, and displays it on 7-seg. display
- GitHub repository
- Wokwi project
- Mux address: 165
- Extra docs
- Clock: 1000 Hz

How it works

The keypad rows are scanned one by one, and their state is stored into a local 16-bit register. Each bit in the register corresponds to one key on the keypad.

The output of the 16-bit register is then converted to the 7-segment display with some simple combinatorial logic.

There are no debouncing, latching or some other advanced features.

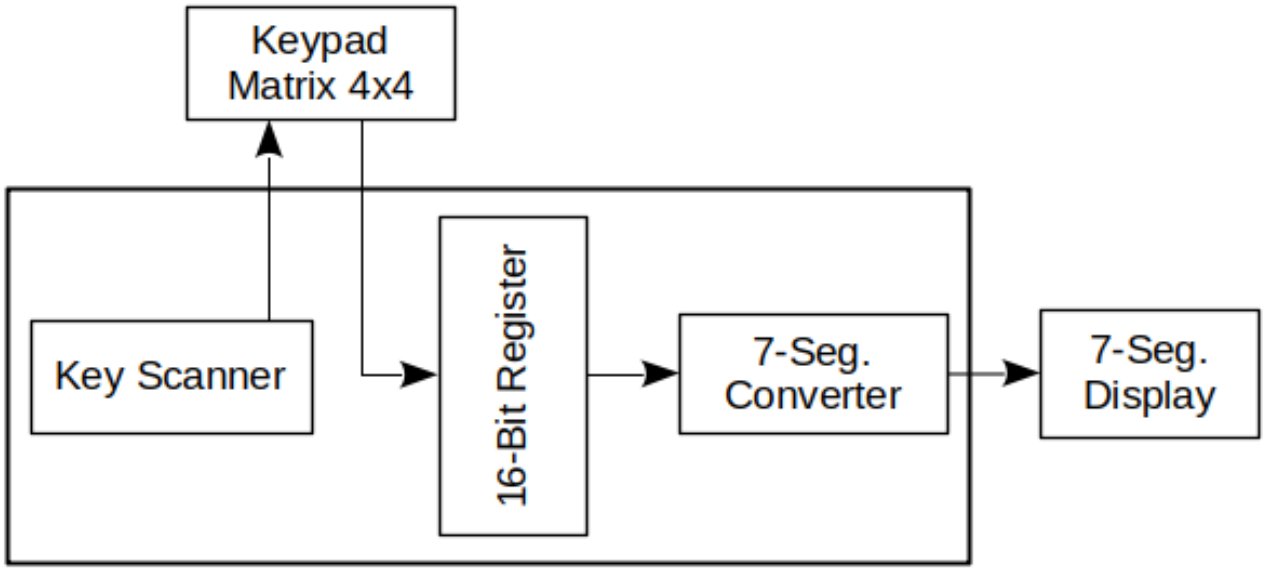


Figure 18: Block Diagram

How to test

Connect a keypad (take a look at the pinout table below), reset the hardware, and start pressing the keypad keys. The corresponding numbers, and characters, should be shown on the 7-segment display.

External hardware

Keypad matrix 4x4. For example:

Figure 19: Keypad matrix 4x4

Pinout

#	Input	Output	Bidirectional
0		segment a	col 4 (input)
1		segment b	col 3 (input)
2		segment c	col 2 (input)
3		segment d	col 1 (input)
4		segment e	row 4 (output)
5		segment f	row 3 (output)
6		segment g	row 2 (output)
7		dot	row 1 (output)

Tiny 8-bit CPU [166]

- Author: Ryota Suzuki
- Description: Tiny 8-bit CPU with SPI Flash/PSRAM controller
- GitHub repository
- HDL project
- Mux address: 166
- Extra docs
- Clock: 50000000 Hz

How it works

This design consists of these blocks:

- CPU
- Memory controller for SPI Flash (for instruction memory) / PSRAM (for data memory)
- GPIO (Output only x4, In/Out x4)
- SPI Tx (mode 0 only)

Dedicated macro assembler is also available at tt06-tmasm.

CPU This design has an 8-bit CPU that has a simple instruction set.

This CPU employs a Harvard architecture. So, it has an instruction bus and a data bus internally. Both buses have 16-bit address space.

External SPI Flash is mapped to 0x0000-0xFFFF on the instruction memory space. CPU will read an instruction from 0x0000 after reset.

PSRAM and some peripherals are mapped to the data memory space. Address map is below:

Address	Description
0x0000-0xEFFF	Mapped to SPI PSRAM
0xF000	GPIO Direction Set Register
0xF001	GPIO Output Data Register
0xF002	GPIO Input Data Register
0xF003	Reserved
0xF004	SPI Divider Value Register
0xF005	SPI CS Control Register
0xF006	SPI Status Register

Address	Description
0xF007	SPI Data Register
0xF008-0xFFFF	0xF000-0xF007 are mirrored in every 8 bytes

Peripherals This design has GPIO and SPI peripherals.

GPIO GPIO has 4x Output-only pins and 4x I/O pins. These pins are mapped 8-bit registers. Upper 4-bits represent output-only pins.

Address	Name	Description
0xF000	GPIO Direction	If bit is set, corresponding pin is configured as output, otherwise configured as input (Lower 4-bit only)
0xF001	GPIO Output Data	Output data value
0xF002	GPIO Input Data	Current pin status

SPI Tx SPI Tx only supports 8-bit data, mode 0. CS signal is not controlled automatically.

Address	Name	Description
0xF004	SPI Clock Divider Value	SPI SCLK frequency[Hz] = (Main Clock / 2) / (Value[3:0] + 1)
0xF005	SPI CS Value	CS pin output value (Valid lowest bit only)
0xF006	SPI Status	If bit[0] is set, transmission is ongoing
0xF007	SPI Tx Data	When write data to this register, SPI transmission will be started

How to test

Write program to SPI Flash (by using ROM Writer etc.) and connect it to the board (Please also see the Pinout section). SPI PSRAM is also needed if you need data

storage other than general-purpose regsiters.

When you negate rst_n, then CPU will load instruction from 0x0000 on SPI Flash.

External hardware

- SPI Flash Memory (W25Q128 etc.)
- SPI PSRAM (IPS6404 etc.) if you want to use external memory

Pinout

#	Input	Output	Bidirectional
0	MISO input from SPI Flash/PSRAM	SCLK output to SPI Flash/PSRAM	General purpose I/O
1		CS output to SPI Flash	General purpose I/O
2		CS output to PSRAM	General purpose I/O
3		MOSI output to SPI Flash/PSRAM	General purpose I/O
4		SCLK output for debugging	General purpose output
5		MOSI output for debugging	General purpose output
6		CS output for debugging	General purpose output
7		Fetch cycle indicator pulse for debbuging	General purpose output

NOT WORKING HP 5082-7500 Decoder [167]

- Author: Per Jensen
- Description: Trying to recreate the decoding logic in the HP 5082-7500-display. At this moment project is not finished.
- GitHub repository
- Wokwi project
- Mux address: 167
- Extra docs
- Clock: 0 Hz

How it works

This design should be able to recreate the old HP 5082-7500 display logic. This is a custom HP ASIC from the 70s, made on new silicon.

How to test

Connect inputs to 4-bit BCD input and LEDs to output x—y

External hardware

BCD switch or counter LED dot-matrix-display.

Pinout

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

LED PWM controller [168]

- Author: Mikkel Holm Olsen
- Description: Exponential LED PWM controller
- GitHub repository
- HDL project
- Mux address: 168
- Extra docs
- Clock: 32768000 Hz

How it works

This is an 8-channel PWM controller for LED brightness.

The PWM duty cycle is generated according to an X3 curve, so the "percieved brightness" changes linearly with the register setting. This design means we get the dynamic range of a 16-bit PWM but use only 8 bits to specify the desired output. With an input clock of 32.7 MHz, the PWM frequency is 500 Hz.

After reset, the controller is in UI mode, where the ui[7:0] set the 8 PWM value registers. The first PWM value register is set to ui[7:0], for the remaining PWM value registers they are set to 0 when ui[7:0] == 0, but in other cases their value is ui[7:0] XOR X, where X is 0x10 times the register number.

The individual registers can be accessed by I2C; SCL=UIO2, SDA=UIO1, which should allow accessing it from the Rpi2040 on the demo board. The slave address is 0x6C, and the 8 PWM channels are controlled by register addresses 0 thru 7. As soon as the first I2C write occurs, the controller is set to I2C mode, and the ui inputs no longer affect the registers.

How to test

Play with the DIP-switches to see different segments of the 7-segment LED display show different brightnesses.

External hardware

Currently no external hardware is supported.

Pinout

#	Input	Output	Bidirectional
0	duty[0]	PWM channel 0	
1	duty[1]	PWM channel 1	SDA
2	duty[2]	PWM channel 2	SCL
3	duty[3]	PWM channel 3	
4	duty[4]	PWM channel 4	
5	duty[5]	PWM channel 5	
6	duty[6]	PWM channel 6	
7	duty[7]	PWM channel 7	

8-Bit CPU In a Week [169]

- Author: Ramyad Hadidi
- Description: 8-bit Single-Cycle CPU
- GitHub repository
- HDL project
- Mux address: 169
- Extra docs
- Clock: 0 Hz

How it works

This project details the design and implementation of an 8-bit single-cycle microprocessor. The processor includes a register file and an Arithmetic Logic Unit (ALU). The design was crafted to handle a simple instruction set architecture (ISA) that supports basic ALU operations, load/store operations, and status checks for the ALU carry – all within less than a week. While the current version lacks a program counter and external memory, thus omitting any form of jump operations, it provides a solid foundation for understanding basic computational operations within a custom CPU architecture.

ISCA Overview The ISA is straightforward and is primarily focused on register operations and basic arithmetic/logic functions. Below is the breakdown of the instruction set:

```
// ISA ---------------------------------------------------------
//-- R level
`define MVR 4'b0000             // Move Register
`define LDB 4'b0001             // Load Byte into Regsiter
`define STB 4'b0010             // Store Byte from Regsiter
`define RDS 4'b0011             // Read (store) processor status
// 1'b0100 NOP
// 1'b0101 NOP
// 1'b0110 NOP
// 1'b0111 NOP

//-- Arithmatics
`define NOT {1'b1, `ALU_NOT}
`define AND {1'b1, `ALU_AND}
`define ORA {1'b1, `ALU_ORA}
`define ADD {1'b1, `ALU_ADD}
```

```
`define SUB {1'b1, `ALU_SUB}
`define XOR {1'b1, `ALU_XOR}
`define INC {1'b1, `ALU_INC}
// 1'b1111 NOP
```

How to test

The processor has been tested through a suite of 12 testbenches, each designed to validate a specific functionality or operation. These testbenches cover basic ALU operations, data movement between registers, and the load/store functionalities. Although basic operational tests are passing, timing interactions between instructions have not been exhaustively verified, and it is anticipated that a sophisticated compiler would handle these timing considerations effectively, reminiscent of approaches taken in historical computing systems.

External hardware

Currently, the processor does not interface with any external hardware components. It operates entirely within a simulated environment where all inputs and outputs are managed through testbenches. This setup is ideal for educational purposes or for foundational experimentation in CPU design.

Pinout

#	Input	Output	Bidirectional
0	Register 1 (R1) Address bit 0	Data out bit 0 (either register data / Processor stat)	Data in bit 0 / Register 3 (R3) Address bit 0
1	Register 1 (R1) Address bit 1	Data out bit 1 (either register data / 0)	Data in bit 1 / Register 3 (R3) Address bit 1
2	Register 1 (R1) Address bit 2	Data out bit 2 (either register data / 0)	Data in bit 2 / Register 3 (R3) Address bit 2

#	Input	Output	Bidirectional
3	Register 1 (R1) Address bit 3	Data out bit 3 (either register data / 0)	Data in bit 3 / Register 3 (R3) Address bit 3
4	Instruction ISA Opcode bit 0	Data out bit 4 (either register data / 0)	Data in bit 4 / Register 2 (R2) Address bit 0
5	Instruction ISA Opcode bit 1	Data out bit 5 (either register data / 0)	Data in bit 5 / Register 2 (R2) Address bit 1
6	Instruction ISA Opcode bit 2	Data out bit 6 (either register data / 0)	Data in bit 6 / Register 2 (R2) Address bit 2
7	Instruction ISA Opcode bit 3	Data out bit 7 (either register data / 0)	Data in bit 7 / Register 2 (R2) Address bit 3

Clock Domain Crossing FIFO [170]

- Author: Kenneth Wilke
- Description: This FIFO buffers 4-bits data asynchronously across clock domains
- GitHub repository
- HDL project
- Mux address: 170
- Extra docs
- Clock: 0 Hz

How it works

This is a FIFO that can pass data asynchronously across clock domains. This was a project I created when I was first learning logic design, and it took me a couple weeks to settle on a design that I felt was clean and reusable.

The FIFO can hold 32 4-bit values, or 16 bytes. So use them wisely and greatly!

The original design can be found at https://github.com/KennethWilke/sv-cdc-fifo

The architecture of this design was influenced by this paper written by Clifford E. Cummings of Sunburst Design by the implementation was fully written by me.

How to test

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

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

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

Pinout

#	Input	Output	Bidirectional
0	write_clock	empty	write_reset
1	write_increment	full	read_reset
2	read_clock		
3	read_increment		
4	write_data0	read_data0	
5	write_data1	read_data1	
6	write_data2	read_data2	
7	write_data3	read_data3	

Frequency to digital converters (asynchronous and synchronous) [171]

- Author: Eduardo Holguin
- Description: This chip combines asynchronous and synchronous frequency-to-digital converters, offering two options in a single package.
- GitHub repository
- HDL project
- Mux address: 171
- Extra docs
- Clock: 0 Hz

How it works

The system incorporates two Frequency-to-Digital Converters (FDCs), one synchronous and one asynchronous. A selector controls a multiplexer, which chooses between these two FDCs. Both frequency signals are sent to the inputs of the chip. Depending on the selected mode, either the asynchronous or synchronous FDC processes the input signal. The chosen FDC then converts the frequency signal into a digital value, representing the frequency, which can be further processed or transmitted as needed.

How to test

To test this chip, connect the inputs as follows: ui[0] (selec) controls the multiplexer selector, ui[1] (clk_ref) receives the reference clock signal or frequency, ui[2] (VCO) is the frequency beig measured, and ui[3] (reset) is the reset input. Ensure all connections are secure and provide appropriate signals to these inputs. Monitor the outputs uo[0] to uo[4], which represent the digital values of the frequency measurements. Apply power to the chip, vary input signals, and toggle the selector pin to observe the chip's behavior under different conditions. Analyze the digital output values to verify the accuracy and performance of the chip, comparing them against expected frequency measurements to ensure compliance with specifications and requirements.

External hardware

Waveform Generator and Logic Analyzer.

Pinout

#	Input	Output	Bidirectional
0	selec	out[0]	
1	clk_ref	out[1]	
2	VCO	out[2]	
3	reset	out[3]	
4		out[4]	
5			
6			
7			

Die Roller [172]

- Author: Nathan Gross
- Description: Generates a random number when rolled with input 1-99
- GitHub repository
- HDL project
- Mux address: 172
- Extra docs
- Clock: 0 Hz

How it works

Takes binary input die size from user and generates a random number from 1-die_size when input to roll is received. Uses clock counter for random number.

How to test

Select die size with inputs 0-6, see that die size is displaying. roll die with input 7 repeatedly, noting random numbers from 1 to die size.

External hardware

PMOD output splitter PMOD dual 7-segment display

Pinout

#	Input	Output	Bidirectional
0	Die Size bit 0	Dual 7 segment data 0	
1	Die Size bit 1	Dual 7 segment data 1	
2	Die Size bit 2	Dual 7 segment data 2	
3	Die Size bit 3	Dual 7 segment data 3	
4	Die Size bit 4	Dual 7 segment data 4	
5	Die Size bit 5	Dual 7 segment data 5	
6	Die Size bit 6	Dual 7 segment data 6	
7	Die Roll	Source selection	

4-bit Stochastic Multiplier Compact with Stochastic Resonator [173]

- Author: Spandan Kottakota and David Parent
- Description: This is a 4-bit stochastic multiplier with a more compact architecture. Also the Schmitt trigger on the input pins is used as a stochastic reonator.
- GitHub repository
- Wokwi project
- Mux address: 173
- Extra docs
- Clock: 0 Hz

How it works

two four-bit || binary weighted vectors are read in and converted to two 1-bit serial stochastic streams with a PRBS and a comparator. These signals are then fed into an AND gate, which multiplies the signal.
https://en.wikipedia.org/wiki/Stochastic_computing B. R. Gaines, "Origins of Stochastic Computing," in Stochastic Computing: Techniques and Applications, W. J. Gross and V. C. Gaudet, Eds., Cham: Springer International Publishing, 2019, pp. 13–37. doi: 10.1007/978-3-030-03730-7_2. C. F. Frasser et al., "Using Stochastic Computing for Virtual Screening Acceleration," Electronics, vol. 10, no. 23, p. 2981, Nov. 2021, doi: 10.3390/electronics10232981. M. Nobari and H. Jahanirad, "FPGA-based implementation of deep neural network using stochastic computing," Appl. Soft Comput., vol. 137, p. 110166, Apr. 2023, doi: 10.1016/j.asoc.2023.110166. P. K. Gupta and R. Kumaresan, "Binary multiplication with PN sequences," IEEE Trans. Acoust., vol. 36, no. 4, pp. 603–606, Apr. 1988, doi: 10.1109/29.1564.
A. Alaghi and J. P. Hayes, "Survey of Stochastic Computing," ACM Trans. Embed. Comput. Syst., vol. 12, no. 2s, pp. 1–19, May 2013, doi: 10.1145/2465787.2465794.

How to test

Use an ADLAM2000 and Python to control the reset and the clock. Hold A and B contents and watch the multiplier output. Use the DALM200 and Python to convert the signal back to binary weight signals. The number of ones at any given time is the number

External hardware

ADLAM2000

Pinout

#	Input	Output	Bidirectional
0	A0	CK	
1	A1	RST	
2	A2	PRBS0	
3	A3	SM	
4	B0	SS0	
5	B1	SS1	
6	B2	SR0	
7	B3		

ASG [174]

- Author: Steffen Reith
- Description: An Alternating Step Generator to generate (pseudo)random bit with huge period length
- GitHub repository
- HDL project
- Mux address: 174
- Extra docs
- Clock: 0 Hz

How it works

This is a naive implementation of an "Alternating Step Generator" (ASG) to produce bit sequences with a very long period. ASGs are characterized by their easy implementability in hardware, which is why they are a nice example for the use of SpinalHDL (the provided Verilog is generated by using SpinalHDL).

An ASG consists of three different "Linear Shift Feedback Registers" (LSFR), which must be coupled appropriately. The provided configuration is expected to have a period length of 226156424186320902518104893031800133178333732395566208938371914392362024 cycles. If the chip could be operated by 1GHz this would be reached after 3.89 10^38 years (approximately 2.78 10^28 times the age of the universe)!

The SpinalHDL based version (including more info about ASGs) can be found at https://github.com/SteffenReith/ASG

Used connection polynoms:

private val connPolyStrR1 = "x^{31+x}3+1" private val connPolyStrR2 = "x^63+x+1" private val connPolyStrR3 = "x^{89+x}38+1"

How to test

Simply load the registers R1 (loadit==1), R2 (loadit == 2) and R3 (loadit == 3) with non-null seed data. Set loadit to 0 and enable to 1. A new bit is generated every clock.

External hardware

No external hardware is used

Pinout

#	Input	Output	Bidirectional
0	loadIt[0]	newBit	
1	loadIt[1]		
2	enable		
3			
4			
5			
6			
7			

Silly 4b CPU v2 [175]

- Author: Tommy Thorn
- Description: A trivial little 4b CPU in the style of the PDP-8, 2nd try
- GitHub repository
- HDL project
- Mux address: 175
- Extra docs
- Clock: 50 Hz

How it works

With ~ 200 gates and 8-inputs, 8-outputs, can we make a full CPU? If we depend on external memory, we can do like the Intel 4004 and multiplex nibbles in and out. However for this submission, we keep the memory on-chip which puts some severe constaints on the size of everything. The only architeture that I could managed in that space is one that relies heavily on self-modifying code, like the beloved DEC PDP-8.

Features:

- Updatable code and data storage
- Instructions include load, store, alu, and conditional branches
- Can execute Fibonacci

CPU state

- 8 words of 6-bit memory, split into 2-bit of instruction and 4-bit of operand.
- 3-bit PC
- 4-bit Accumulator (ACC)

The Instruction Set All instructions have an opcode and an argument.

- `load` value (loads value to the accumulator)
- `store` address (stores the accumulator at the address, top bit ignored)
- `add` value (adds value to the accumulator)
- `brzero` address (branches to address if the accumulator is zero)

Obviously this is very limiting, but it does show the basic structure and could probably be tweaked to be more general with more time (but space is limited).

Example Fibonacci Program

```
int a = 1, b = 1;
for (;;) {
  int t = a;
  a += b;
  b = t;
  if (b == 8) for (;;)
}
```

```
0: load 1  // a is here
1: store 4 // store to t at address 4

2: add 1   // b is here
3: store 0 // Update a at address 0

4: load _  // t is here, value overwritten
5: store 2 // update b

6: add 8  //  -8 == 8
7: brzero 7 // if acc - 8 == 0 we stop
// otherwise roll over to 0
```

Execution trace:

```
$ make -C src -f test.mk | tail -50 | head -17
        Running 0 (insn 0,  8)
00500  pc 1 acc  8
        Running 1 (insn 1,  4)
00510  pc 2 acc  8
        Running 2 (insn 2,  5)
00520  pc 3 acc 13
        Running 3 (insn 1,  0)
00530  pc 4 acc 13
        Running 4 (insn 0,  8)
00540  pc 5 acc  8
        Running 5 (insn 1,  2)
00550  pc 6 acc  8
        Running 6 (insn 2,  8)
00560  pc 7 acc  0
        Running 7 (insn 3,  7)
```

```
Running 7 (insn 3,  7)
Running 7 (insn 3,  7)
```

We actually computed fib all the way to 13 (largest that will fit in 4-bits). Explain how your project works

How to test

Use ui_in[7:4] and cmdarg and ui_in[3:2] as cmd. Inputs are only registered on posedge of clock. Keep rst_n high. Then issue the sequence of commands as follows: (XXX please see tb in tt_um_tommythorn_4b_cpu_v2.v for now)

External hardware

Nothing required but without observing outputs it's a bit boring.

Pinout

#	Input	Output	Bidirectional
0	clock	acc[0]	
1	cmd[0]	acc[1]	
2	cmd[1]	acc[2]	
3		acc[3]	
4	cmdarg[0]	pc[0]	
5	cmdarg[1]	pc[1]	
6	cmdarg[2]	pc[2]	
7	cmdarg[3]		

ANS Encoder/Decoder [194]

- Author: Davide Asnaghi & Lenny Khazan
- Description: Asymmetric Numeral Systems Encoder/Decoder
- GitHub repository
- HDL project
- Mux address: 194
- Extra docs
- Clock: 25000000 Hz

How it works

Big data goes in, small data comes out

How to test

TBD

External hardware

None

Pinout

#	Input	Output	Bidirectional
0	input	output	cmd
1	input	output	cmd
2	input	output	in_vld
3	input	output	out_rdy
4			in_rdy
5			out_vld
6			
7			

Two ports USB CDC device [198]

- Author: Maximo Balestrini
- Description: USB CDC device with two ports each with a different application
- GitHub repository
- HDL project
- Mux address: 198
- Extra docs
- Clock: 48000000 Hz

How it works

The design works as Full Speed (12Mbit/s) USB communications device class (or USB CDC class). It implements the Abstract Control Model (ACM) subclass.
Most of the code is based on this repo: https://github.com/ulixxe/usb_cdc

When connected to a pc the device should appear as two virtual serial ports. (COMX on Windows, /dev/ttyACMx on Linux and /dev/cu.usbmodemxxxx on OSX)
(Linux requires that the user account belongs to the dialout group to grant permissions for virtual COM access)

Each port/channel has a different application:

USB CDC Channel 0 Application: Input to serial
Description: When the value from one of the inputs change from 0 to 1 or 1 to 0 it sends a character to the port.
USB Interfaces: 0 and 1
USB Endpoints: EP2 (IN, INTERRUPT), EP1 (IN, BULK), EP1 (OUT, BULK)

pin/character relation: | pin | rise | fall | | — | —- | —- | |input[0] | A | a | |input[1] | B | b | |input[2] | C | c | |input[3] | D | d | |input[4] | E | e | |input[5] | F | f | |input[6] | G | g | |input[7] | H | h |

USB CDC Channel 1 Application: Loopback
Description: Simple loopback application to test the port. Every character sent shall return.
USB Interfaces: 2 and 3
USB Endpoints: EP4 (IN, INTERRUPT), EP3 (IN, BULK), EP3 (OUT, BULK)

Port identification There's no warranty that the ports are going to be named in that order (not sure on Linux, on Windows they definitely can have the order reversed). Without any extra OS functions, the simple way to identify them is to open one of the ports and write something to it. If you are connected to the Channel 1 loopback you should receive the same char as response.

External hardware

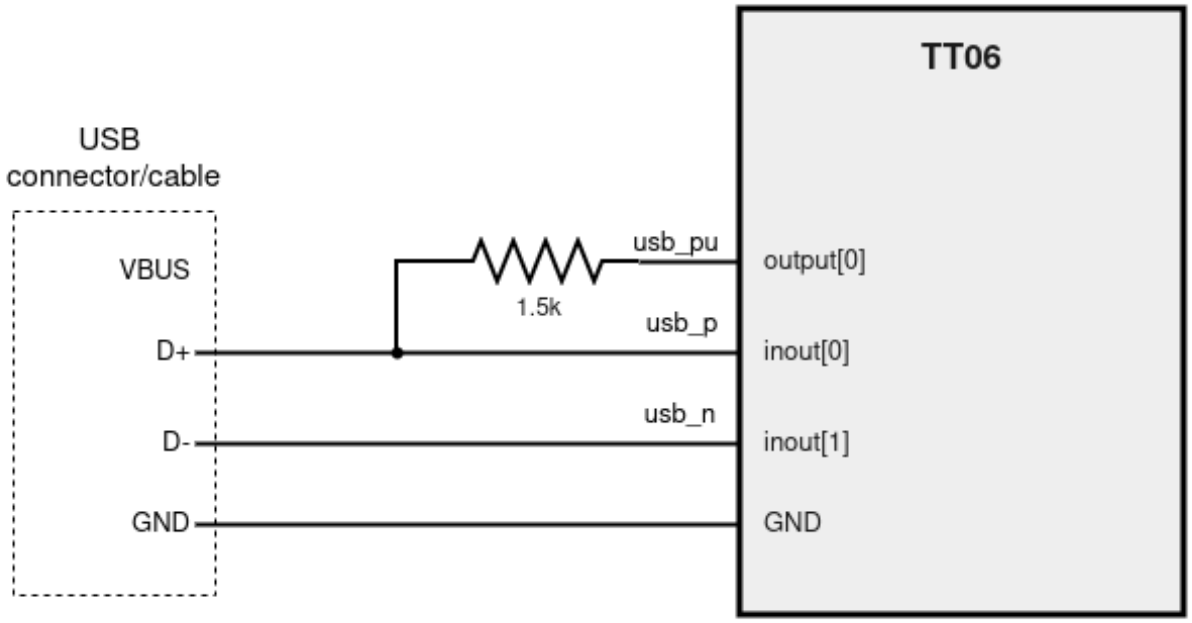
USB cable with internal cables exposed or USB connector + USB cable
1.5k resistor
Computer

Optional:
Buttons for the inputs or use the TT demo board switches

How to test

TT board clock needs to be set to 48MHz

Basic schematic:



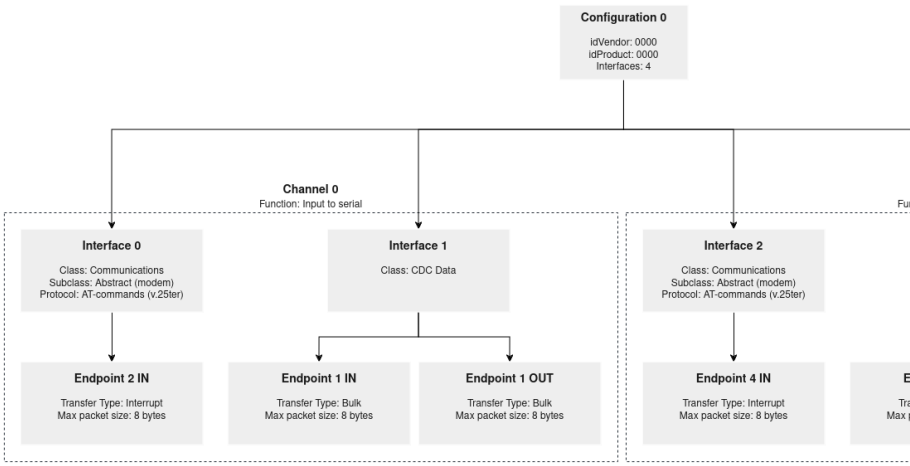
Cut output[0] LED display jumper on TT board? I haven't been able to test if this is necessary or not.

Once the USB cable is connected to the PC two virtual serial ports should be available to communicate: COMX on Windows, /dev/ttyACMx on Linux and /dev/cu.usbmodemxxxx on OSX

Linux requires that the user account belongs to the dialout group to grant permissions for virtual COM access. You can change udev rules to fix that or add the user to the group by running: `sudo usermod -a -G dialout $USER` and restart

Example connection on Linux using picocom:
`picocom /dev/ttyACM0` or `picocom /dev/ttyACM1`

Extra information



USB Interfaces/Endpoints:

The device has USB vendor ID and product ID = 0000

Useful scripts On the project repo there are some Linux scripts to get information about the USB devices:

- `list_usb_devices.sh`
  - list all USB devices connected. Look for ID 0000:0000
- `list_device_0000_0000.sh`
  - detailed USB descriptors configuration of device 0000:0000

This is how the device should look with the `lsusb -tv` or `list_usb_devices.sh` command:

```
    |__ Port 1: Dev 42, If 2, Class=Communications, Driver=cdc_acm, 12M
        ID 0000:0000
    |__ Port 1: Dev 42, If 0, Class=Communications, Driver=cdc_acm, 12M
        ID 0000:0000
    |__ Port 1: Dev 42, If 3, Class=CDC Data, Driver=cdc_acm, 12M
        ID 0000:0000
```

```
|__ Port 1: Dev 42, If 1, Class=CDC Data, Driver=cdc_acm, 12M
    ID 0000:0000
```

For deeper debugging and understanding of the USB protocol exchange between the device and the PC *Wireshark* app can be used

Debug pins: These are the extra pins used for development debugging: | port | name | description | | — | —- | —- | |output[1] | “debug_led” | once configured lights aprox. once per second| |output[2] | “debug_usb_configured” | USB configured on PC| |output[3] | “debug_usb_tx_en” | inout[0] and inout[1] as outputs| |output[4] | “debug_frame[0]” | USB frame number binary digit 0| |output[5] | “debug_frame[1]” | USB frame number binary digit 1| |output[6] | “debug_frame[2]” | USB frame number binary digit 2| |output[7] | “debug_frame[3]” | USB frame number binary digit 3|

Some USB information resources:

- USB in a NutShell:
  - https://www.beyondlogic.org/usbnutshell/usb1.shtml
- USB Made Simple
  - https://www.usbmadesimple.co.uk/index.html
- Understanding the Universal Serial Bus (USB)
  - https://github.com/DCC-Lab/PyHardwareLibrary/blob/939ffca7c8b3b214b77acada1-USB.md
- USB Device CDC ACM Class
  - https://docs.silabs.com/protocol-usb/1.2.0/protocol-usb-cdc/

Pinout

#	Input	Output	Bidirectional
0	input_0	usb_pu	usp_p
1	input_1	debug_led	usb_n
2	input_2	debug_usb_configured	
3	input_3	debug_usb_tx_en	
4	input_4	debug_frame[0]	
5	input_5	debug_frame[1]	

#	Input	Output	Bidirectional
6	input_6	debug_frame[2]	
7	input_7	debug_frame[3]	

Snake Game [200]

- Author: Stefan Hirschböck
- Description: Simple implementation of the game Snake with VGA Output
- GitHub repository
- HDL project
- Mux address: 200
- Extra docs
- Clock: 25179000 Hz

How it works

```
Simple implementation of the game "Snake" with VGA Output.
 Due to size limitations snake can only grow to 9 body parts.
 Game resets when snake touches border or any of its body parts
 Vga output is compatible with tiny vga pmod.
```

How to test

```
 After reset snake can be controlled though inputs. When collecting an a
 clock has to be set to 25.179 Mhz for vga sync signal generation to wor
 inputs should be done with push buttons. Not pressed is logic 0, presse
 So an external circuit with pull down resistors should be used for inpu
 If no tiny VGA pmod is available a vga dac like in this project:https:/
 could probably also be used.
```

External hardware

VGA Display, external buttons for input

Pinout

#	Input	Output	Bidirectional
0	none	R1	none
1	none	G1	none
2	none	B1	none
3	none	vsync	none
4	Right	R0	none

#	Input	Output	Bidirectional
5	Left	G0	none
6	Down	B0	none
7	Up	hsync	none

8-bit CPU with Debugger [202]

- Author: Sean Patrick O'Brien
- Description: 8-bit CPU with debugger accessible via I2C
- GitHub repository
- HDL project
- Mux address: 202
- Extra docs
- Clock: 0 Hz

How it works

The CPU is based on Ben Eater's 8-bit breadboard CPU. A built-in debugger allows pausing the CPU, loading programs, inspecting/modifying registers, etc.

How to test

The debugger is accessible over I2C at address 0x2A (0x54 write, 0x55 read). The provided dbg program can be used to load programs, inspect registers, etc.

External hardware

Optionally, data can be provided on the input pins and consumed on the output pins. They are accessible to the CPU as the IN and OUT registers.

Pinout

#	Input	Output	Bidirectional
0	Input Port	Output Port	
1	Input Port	Output Port	
2	Input Port	Output Port	SCL
3	Input Port	Output Port	SDA
4	Input Port	Output Port	HALTED
5	Input Port	Output Port	
6	Input Port	Output Port	
7	Input Port	Output Port	

The James Retro Byte 8 computer [204]

- Author: James Ridey
- Description: A 8bit microprocessor built from the ground up (nand2tetris)
- GitHub repository
- HDL project
- Mux address: 204
- Extra docs
- Clock: 0 Hz

Overview

This project is an 8-bit computer I originally designed in Logisim Evolution, which I am now porting to TinyTapeout for manufacturing. Below is the computer's general architecture as shown in Logisim Evolution; however, certain modifications were made to ensure compatibility with VHDL and TinyTapeout.

The primary change includes the addition of a B register, alongside adjustments to enable ROM and RAM communication via SPI. Detailed information on these modifications is provided below.

The computer supports the following operations:

Register Operations The computer features four main registers: `a`, `b`, `c`, and `d`. It supports:

- Moving (`mov`) data between all registers.
- Comparing (`cmp`) values between registers, or between a register and a constant (0, 1, -1, or 255).
- Jumping (`jmp`) to labels and performing conditional jumps (=, `!=`, `&lt;`, `&lt;=`, `&gt;`, `&gt;=`) and relative jumps.

ALU Operations The ALU (Arithmetic Logic Unit) offers the following operations:

- Bitwise NOT (~) and negation (-).
- Increment (+1) and decrement (-1).
- Addition (+) and subtraction (-).
- Multiplication (*) and division (/).
- Bitwise AND (`&amp;`) and OR (|).
- Signed mode operations and a carry flag for extended 8-bit addition/subtraction.

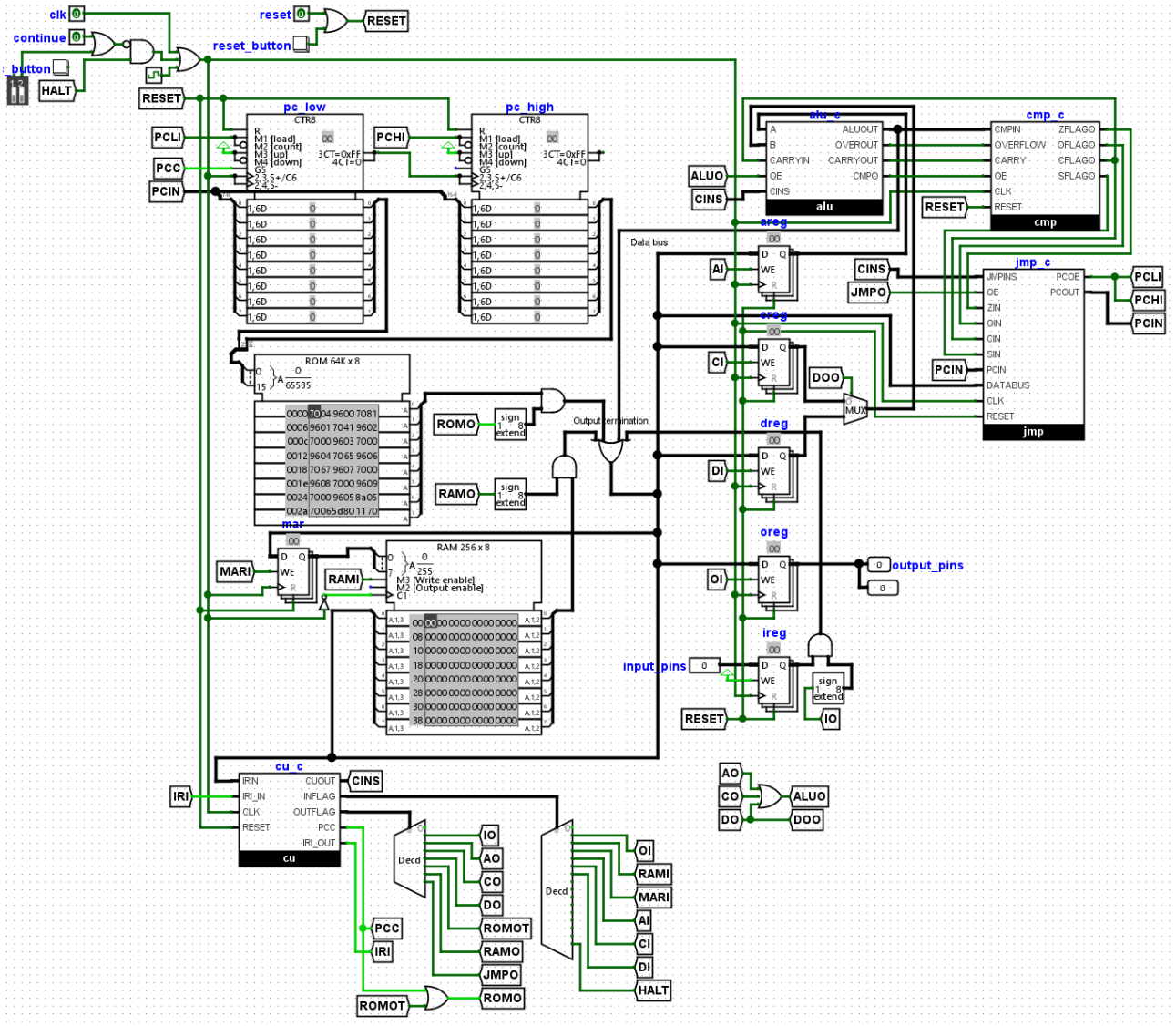


Figure 20: architecture

Memory and I/O

- Load data from ROM into a register.
- Load data from RAM into a register.
- Save data from a register to RAM.
- Read from the input (`in`) port.
- Write to the output (`out`) port.

Testing the Computer

The computer doesn't have any start signal and will begin read from the SPI ROM as soon as the clock signal (`clk`) starts ticking.

For examples of programs and a basic assembler, please see this repository (https://github.com/AeroX2/tt06-jrb8-computer/).
Clone the repository and use the assembler with the following command:

```
python3 ./example_programs/assembler.py
```

A file dialog will open, allowing you to select a `*.j` file.

Sample Program: 8-bit Fibonacci Sequence Below is a simple 8-bit Fibonacci program in the custom J format (`fibonacci.j`):

```
:start
load rom a 1
load rom b 0
:repeat
// Store the previous in c register
mov a c
// a = a + b
opp a+b
// This also corresponds to the carry flag being set
// So jump to start if a+b has overflowed
jmp < start
// Output the value to the output pins
out a
// Restore the previous value in b register
mov c b
jmp repeat
```

This program, when assembled, translates to the following hexadecimal format:

```
d0 01 d1 00 02 6c 33 00 00 f4 08 36 00 04
```

For a comprehensive guide on assembler instructions and their corresponding hex codes, refer to this document (https://docs.google.com/document/d/1ZVZw_Kt-KQHER0Wr5ty7JpUEeox_284Mih4rwE16FVM/edit?usp=sharing).
You can also look at `roms/cu_flags.csv` in the `tt06-jrb8-computer` repository

The input register or the `i` register is mapped to `ui_in`
The output register or the `o` register is mapped to `uo_out`

Memory Mapping To load data into the ROM, place it at offset 0. The address space is divided as follows:

- **ROM (Program Data):** 0x0000 to 0xFFFF
- **RAM:** 0x10000 to 0x1FFFF

RAM Addressing

RAM addressing is handled through two registers:

- **`mpage` Register:** Controls `0x1**00`
- **`mar` Register:** Controls `0x100**`

External Hardware Requirements

External SPI storage is required for this computer, with mappings compatible with spi-ram-emu (https://github.com/MichaelBell/spi-ram-emu/). The following `uio` mappings are used:

```
uio[0]: "cs rom"
uio[1]: "mosi"
uio[2]: "miso"
uio[3]: "sck"
```

#	Input	Output	Bidirectional

Pinout

#	Input	Output	Bidirectional
0	input bit 0	output bit 0	cs rom
1	input bit 1	output bit 1	mosi
2	input bit 2	output bit 2	miso
3	input bit 3	output bit 3	sck
4	input bit 4	output bit 4	cs ram
5	input bit 5	output bit 5	
6	input bit 6	output bit 6	
7	input bit 7	output bit 7	24 addressing bit mode

co processor for precision farming [206]

- Author: MITS ECE
- Description: The processor will detect the deviation in sensor data and the sensor fault
- GitHub repository
- HDL project
- Mux address: 206
- Extra docs
- Clock: 0 Hz

How it works

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

How to test

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

External hardware

8 bit ADC is needed to convert the sensor data

Pinout

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

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

Keypad controller [224]

- Author: Ian Tawileh
- Description: Reads a keypad and displays the number on the 7 segment
- GitHub repository
- HDL project
- Mux address: 224
- Extra docs
- Clock: 1000 Hz

How it works

This Project works by driving power to the Cols Columns one by one, then waits for any changes on the Rows (triggered by Human Input) and scans a case to find the combination between the row and col columns before finding the right combination and recording the corresponding key.

This key is passed on to a decode module that finds the correct Seven Segment combination and then passes it on to the 1 digit seven Segment Display where it is displayed.

How to test

Connect your keypad to the PMOD pins and experiment by clicking some buttons and seeing their outputs!

External hardware

Keypad PMOD: https://t.ly/ITZF0

Pinout

#	Input	Output	Bidirectional
0	row0	7 segment display outputs	col0
1	row1		col1
2	row2		col2
3	row3		col3
4			col counter 0
5			col counter 1
6			

#	Input	Output	Bidirectional
7			

multimac [226]

- Author: Jonny Edwards
- Description: a multi use multi-hit dot product accelerator
- GitHub repository
- HDL project
- Mux address: 226
- Extra docs
- Clock: 0 Hz

How it works

This is a simple circuit to calculate:

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

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

How to test

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

External hardware

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

#	Input	Output	Bidirectional

Pinout

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

TinyQV Risc-V SoC [227]

- Author: Michael Bell
- Description: A Risc-V SoC for Tiny Tapeout
- GitHub repository
- HDL project
- Mux address: 227
- Extra docs
- Clock: 64000000 Hz

How it works

TinyQV is a small Risc-V SoC, implementing the RV32EC instruction set, with a couple of caveats:

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

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

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

The SoC includes a UART and an SPI controller.

Address map

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

GPIO

Register	Address	Description
OUT	0x8000000 (W)	Control out0-7, if the corresponding bit in SEL is high
OUT	0x8000000 (R)	Reads the current state of out0-7
IN	0x8000004 (R)	Reads the current state of in0-7
SEL	0x800000C (R/W)	Enables general purpose output on the corresponding bit on out0-7

UART

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

Debug UART (Transmit only)

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

SPI

Register	Address	Description
DATA	0x8000020 (W)	Transmits the byte in bits 7-0, bit 8 is set if this is the last byte of the transaction, bit 9 controls Data/Command on out3

Register	Address	Description
DATA	0x8000020 (R)	Reads the last received byte
CONFIG	0x8000024 (W)	The low 2 bits set the clock divisor for the SPI clock to 2*(value + 1), bit 2 adds half a cycle to the read latency when set
STATUS	0x8000024 (R)	Bit 0 indicates whether the SPI is busy, bytes should not be written or read from the data register while this bit is set.

How to test

Load an image into flash and then select the design.

Reset the design as follows:

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

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

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

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

External hardware

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

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

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

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

Pinout

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

10-bit Linear feedback shift register [228]

- Author: Shivam Bhardwaj, Sachin Sharma, Pankaj Lodhi and Ambika Prasad Shah
- Description: This Verilog module implements a 10-bit Linear Feedback Shift Register (LFSR) for generating pseudo-random sequences with clock and reset inputs.
- GitHub repository
- HDL project
- Mux address: 228
- Extra docs
- Clock: 50000000 Hz

How it works

This Verilog module defines a 10-bit Linear Feedback Shift Register (LFSR). It features clock (`clk`) and reset (`rst`) input pins. The output pin (`out`) delivers a pseudo-random sequence based on clock edges and reset conditions. It's designed for digital applications requiring pseudo-random sequence generation and pattern generation.

How to test

We test it on Vivado and open sources (OpenROAD and OpenLane).

External hardware

defaults

Pinout

#	Input	Output	Bidirectional
0	clk	out	
1	rst		
2			
3			
4			
5			
6			

#	Input	Output	Bidirectional
7			

Analog 8bit R2R DAC [229]

- Author: Matt Venn
- Description: A simple 8 bit DAC with a sawtooth waveform driver
- GitHub repository
- Analog project
- Mux address: 229
- Extra docs
- Clock: 0 Hz

How it works

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

How to test

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

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

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

External hardware

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

Pinout

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

#	Input	Output	Bidirectional
5	bit 5		
6	bit 6		
7	bit 7		

Analog pins

ua#	analog#	Description
0	4	DAC output

Pulse Width Modulation [230]

- Author: Shivam Bhardwaj, Sachin Sharma and Ambika Prasad Shah
- Description: This Verilog module generates a Pulse Width Modulation (PWM) signal with adjustable duty cycle. It utilizes a 50MHz clock input and debounced buttons to increase or decrease the duty cycle, producing a 5MHz PWM output for various applications like motor speed control or LED brightness adjustment.
- GitHub repository
- HDL project
- Mux address: 230
- Extra docs
- Clock: 50000000 Hz

How it works

We want to design Pulse width Modulation (PWM) with 50MHz input Frequency. The Verilog code defines a module named `tt_um_shivam` responsible for generating a Pulse Width Modulation (PWM) signal. It takes a 50MHz clock input (`clk`) and provides inputs for increasing the assigned pin (`ui_in[0]`) and decreasing the assigned pin (`ui_in[0]`) in the duty cycle. The PWM signal is output through the assigned pin `PWM_OUT` at a frequency of 5MHz.

The code implements debouncing logic for the increase and decrease duty cycle buttons using D flip-flops (`DFF_PWM` modules) to prevent rapid fluctuations due to button bouncing. It also includes counters for generating slow clock enable signals to facilitate debouncing.

The duty cycle can be adjusted by pressing the increase or decrease buttons, which are debounced to ensure reliable operation. The duty cycle can vary from 0% to 90% (in 10% increments), and the PWM signal is generated based on this duty cycle.

Overall, the code provides a flexible and robust PWM signal generator with adjustable duty cycle control.

How to test

We check our design with the help of OpenROAD flow script (ORFS).

External hardware

default

Pinout

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

TT06 8-bit SAR ADC [231]

- Author: Carsten Wulff
- Description: 8-bit Successive Approximation Register ADC
- GitHub repository
- Analog project
- Mux address: 231
- Extra docs
- Clock: 4000000 Hz

Who

Carsten Wulff carsten@wulff.no

Why

Many years ago I made a compiler, and a state-of-the-art compiled ADC in 28 nm FDSOI, described in A Compiled 9-bit 20-MS/s 3.5-fJ/conv.step SAR ADC in 28-nm FDSOI for Bluetooth Low Energy Receivers.

Since then, I've ported the ADC to multiple closed PDKs (22 nm FDSOI, 22 nm, 28 nm, 55 nm, 65 nm and 130nm). A while ago I ported the SAR ADC to Skywater 130nm SUN_SAR9B_SKY130NM.

The fact that Tiny Tapeout now includes analog possibility inspired me to try and see if I could fit the SAR into the Tiny Tapeout area. The original 9-bit ADC did not fit, so I had to reduce it to 8-bit.

How to test

Apply a differential voltage with a common mode of around VDD/2 to ua[1] and ua[0]. If you want to measure the offset and noise of the ADC then connect ua[1] to ua[0] and provide 0.9 V to both.

A common mode of 0 V will not work. The comparator will not make a decision in time, and the asynchronous clock generation loop will be to slow.

Apply a 4 MHz clock to clk. Typical corner should be able to run faster.

Set ui_in[0] high to enable the ADC

The uo_out[7:0] is two's complement digital output. The MSB is [7].

The ADC will open the input switches to start sampling on the rising edge of the clock. The ADC will sample on the falling edge of the clock. When clock is low, then the asynchronous binary search algorithm tries to find the sampled analog input voltage, and convert the analog value to digital.

The uio_out[0] is the "done" signal from the asynchronous binary search algorithm. The digital outputs are sampled on the rising edge of this "done" signal.

If you want to capture the output of the ADC with a logic analyzer then I'd recommend you sample the digital outputs on the falling edge of the "done" signal.

Alternatively, you could sample on the rising edge of the clk, however, any insertion delay between the clk source and the ADC clk has to be taken into account.

If there is no "done" signal, then the clock is too fast, or the input common mode too low.

How it works

The differential input (ua[1:0]) is sampled onto a capacitor array. When the clk is high, the input switch is low resistance and the input voltage stabilizes on the capacitor array. When the clock goes low, the input switch will be high resistive, and the voltage on the capacitor array is sampled.

A strong arm comparator decides whether the differential voltage on the capacitor array is larger or smaller than zero.

Based on the comparator decision, parts of the capacitor array is switched from VPWR to VGND, or visa versa. A charge re-distribution will occur, which changes the differential voltage on the capacitor array.

A asynchronous custom digital logic performs a binary search to find the digital value.

The comparator input has the net name SARP and SARN. Observe those in a simulation to see how the SAR operates.

I would also recommend reading A Compiled 9-bit 20-MS/s 3.5-fJ/conv.step SAR ADC in 28-nm FDSOI for Bluetooth Low Energy Receivers, which explains the operation in detail. I've also added docs to sun_sar9b_sky130nm

Parameter	Min	Typ	Max	Unit

Key parameters

Parameter	Min	Typ	Max	Unit
Technology		SKY130A		
AVDD	1.7	1.8	1.9	V
Temperature	-40	27	125	C
Sampling frequency (CLK)			4	MHz
Average current VPWR		48		uA
SNDR_FS		47.7		dBFS
SFDR		49.7		dBc
ENOB_FS		7.63		bit

Implementation

If you just want to see the layout, then go to http://analogicus.com/tt06-sar/

To have a look locally, do the commands below. I assume you have xschem, magic and the Skywater 130 nm PDK installed.

```
git clone --recursive git@github.com:wulffern/tt06-sar.git
cd tt06-sar/ip/tt06_sar_sky130nm/work/
xschem -b ../design/TT06_SAR_SKY130NM/tt_um_TT06_SAR_wulffern.sch &
magic ../design/TT06_SAR_SKY130NM/tt_um_TT06_SAR_wulffern.mag &
```

How to simulate Install cicsim

```
python3 -m pip install cicsim
```

Navigate to the testbench and run a typical simulation (requires cicsim)

```
cd ip/tt06_sar_sky130nm/sim/TT06_SAR
make typical OPT="Debug"
```

The main testbench is `ip/tt06_sar_sky130nm/sim/TT6_06/tran.spi`

How to compile The SAR ADC is made with ciccreator and cicpy.

The sources for the ADC are

```
ip/sun_sar9b_sky130nm/cic
   ip.json                      # Object file, describes the object hiera
   ip.spi                       # Spice file, describes the connectivity
   capacitor.json               # Object file for capacitors
   dmos_sky130nm_core.json      # Object file for transistors
   sky130.tech                  # Technology file for Skywater 130 nm
```

The SAR is pre-compiled, so you don't really need to compile it. The compiled files are in the `ip/sun_sar9b_sky130nm/design/` directory.

If you want to try the compilation, then compile `ip/ciccreator` and install `ip/cicpy`, next

```
cd ip/sun_sar9b_sky130nm/work
make ip
```

Verification plan Testbench folder `ip/TT06_SAR_SKY130NM/sim/TT06_SAR/`

Purpose	Testbench	corner	Status	Notes
SNDR, SFDR, ENOB, active current	tran	tfs + C	OK	python3 tran.py to plot FFT
	tran	typ + RC	Not OK	RC extraction does not work yet
Check power down after 2 sample, clock running	pwrdwn	typ	OK	

Results at TT06_SAR

Below is a Power Spectrum of a sinusoidal input signal

Known issues

Nr	Issue	Solution	Discovery	Resolved
1	RC extraction removes coupling caps		2024-04-13	

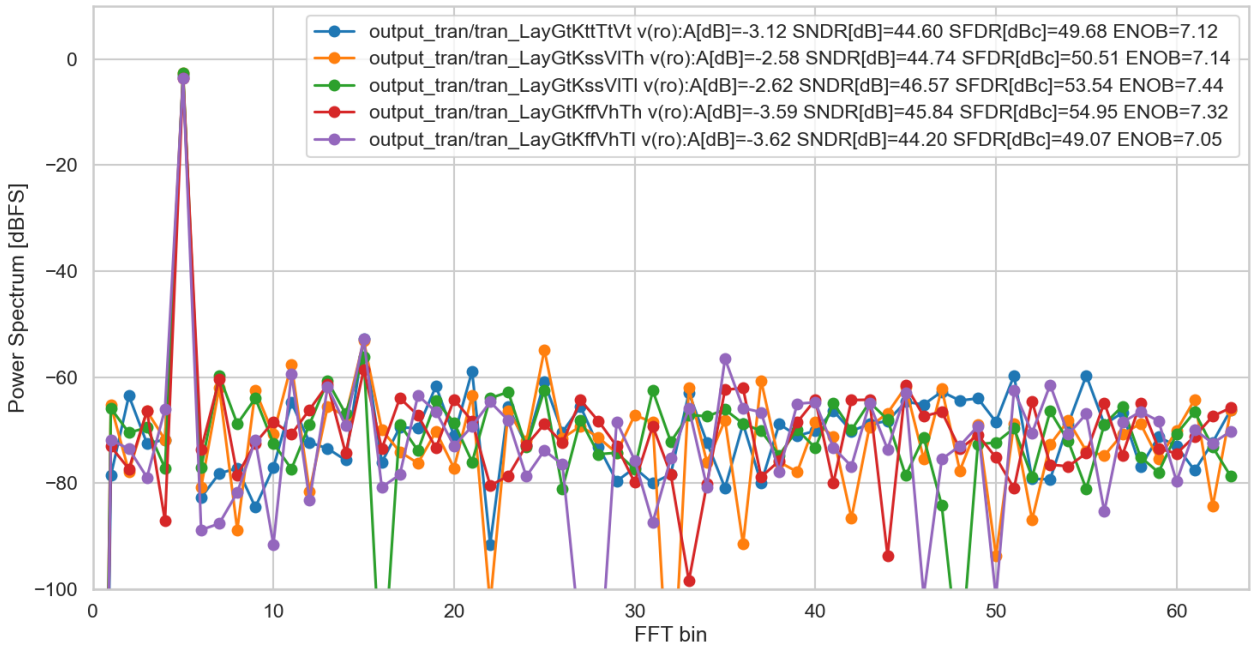


Figure 21: typical fast slow FFT

1: RC extraction removes coupling caps Extracting R and C seems to remove coupling caps, which removes both the cap in the bootstrapped switch, and the SAR. As a result, simulations don't work.

I make the RC extracted netlist with

```
cd ip/tt06_sar_sky130nm/work
make lper
```

In that command, I try to match the RC extracted netlist to the schematic netlist. First I remove all the parasitic Cs, the parasitic Rs and remove the R nets ((t|n)+). The resulting RC extracted netlist is not LVS clean. The m3 resistors in BSSW have been removed.

I've also tried to change all parasitic resistors to 0.1 Ohm (`make lowres`), but the simulation still does not work.

After a bit of digging it's clear that the cap between XCAP.B and XCAP.A in the BSSW is gone (it should be 0.3ish pF). There are almost no coupling caps, only caps to ground.

So I'm resonably sure it's not a real issue. It's a tool issue. Let's see when the IC comes back.

Pinout

#	Input	Output	Bidirectional
0	Enable ADC	ADC LSB	Conversion Done
1		ADC MSB-6	
2		ADC MSB-5	
3		ADC MSB-4	
4		ADC MSB-3	
5		ADC MSB-2	
6		ADC MSB-1	
7		ADC MSB (two's complement)	

Analog pins

ua#	analog#	Description
0	5	Negative ADC input
1	0	Positive ADC input

4-bit stochastic multiplier traditional [232]

- Author: Vedika Sharma and David PArent
- Description: Two 4-bit || vectors are converted into 1-bit serial stochastic signals and then multiplied with a two input and gate.
- GitHub repository
- Wokwi project
- Mux address: 232
- Extra docs
- Clock: 10000 Hz

How it works

two four-bit || binary weighted vectors are read in and converted to two 1-bit serial stochastic streams with a PRBS and a comparator. These signals are then fed into an AND gate, which multiplies the signal.
https://en.wikipedia.org/wiki/Stochastic_computing B. R. Gaines, "Origins of Stochastic Computing," in Stochastic Computing: Techniques and Applications, W. J. Gross and V. C. Gaudet, Eds., Cham: Springer International Publishing, 2019, pp. 13–37. doi: 10.1007/978-3-030-03730-7_2. C. F. Frasser et al., "Using Stochastic Computing for Virtual Screening Acceleration," Electronics, vol. 10, no. 23, p. 2981, Nov. 2021, doi: 10.3390/electronics10232981. M. Nobari and H. Jahanirad, "FPGA-based implementation of deep neural network using stochastic computing," Appl. Soft Comput., vol. 137, p. 110166, Apr. 2023, doi: 10.1016/j.asoc.2023.110166. P. K. Gupta and R. Kumaresan, "Binary multiplication with PN sequences," IEEE Trans. Acoust., vol. 36, no. 4, pp. 603–606, Apr. 1988, doi: 10.1109/29.1564.
A. Alaghi and J. P. Hayes, "Survey of Stochastic Computing," ACM Trans. Embed. Comput. Syst., vol. 12, no. 2s, pp. 1–19, May 2013, doi: 10.1145/2465787.2465794.

How to test

Use an ADLAM2000 and Python to control the reset and the clock. Hold A and B contents and watch the multiplier output. Use the DALM200 and Python to convert the signal back to binary weight signals. The number of ones at any given time is the number

External hardware

ADLAM2000

Pinout

#	Input	Output	Bidirectional
0	A0	SSA	
1	A1	SSB	
2	A2	PRBS0	
3	A3	PRBS1	
4	B0	PRBS2	
5	B1	PRBS3	
6	B2	S_M	
7	B3		

VCII [233]

- Author: Alfiero Leoni
- Description: Simple Voltage Conveyor
- GitHub repository
- Analog project
- Mux address: 233
- Extra docs
- Clock: 0 Hz

How it works

The VCII (second generation Voltage Conveyor) is an analog block that has a low impedance current input pin (y), a high-impedance current output pin (x) and a low impedance output voltage pin (z) plus a reference voltage input pin (Ref) to provide the virtual ground reference for the circuit, behing used in single supply (for this design, ref is 0.9 V to be provided with a power supply). The VCII presents to main parameters: alpha and beta. Beta is the current gain, so placing a resisntance between x and ref and injecting a current in y, we should have that I(x) = beta*I(Y). Aplha is the voltage gain, i.e. the voltage produced at the node x due to the current flowing will be amplified in z. The relationship is V(z)=alpha*V(x). In this design, aplha and beta should be equal to 1, more or less.

How to test

The VCII could be tested in TIA (transimpedance amplifier) configuration. A current should be injected into the y pin (if a current source is not available, a big resistor can be used in serias to a voltage supply) of few uA. Then an external resistor should be connected between x and Vref. The resistor will set the TIA gain e.g. a resistor of 10K with an input sine current of 2uA pp should produce an output sine voltage of 20 mVpp to the z pin.

External hardware

The transipedance gain resistor, oscilloscope, power supplies

Pinout

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

Analog pins

ua#	analog#	Description
0	4	output
1	1	output
2	3	input
3	2	input

8 Bit Digital QIF [234]

- Author: David Parent
- Description: The circuit will spike when the input is positive. It will reset when the signal exceeds a predetermined value
- GitHub repository
- HDL project
- Mux address: 234
- Extra docs
- Clock: 0 Hz

How it works

QIF

How to test

QIF

External hardware

ADALM2000

Pinout

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

Programmable Thing [235]

- Author: James Meech
- Description: One inverter and a programmable resistor with one terminal connected to ground
- GitHub repository
- Analog project
- Mux address: 235
- Extra docs
- Clock: 0 Hz

How it works

The project is a programmable resistor controlled by setting ui[0] to ui[7] and ui[0] to ui[7] high to connect a set of 58.218 k ohm resistors between pin ua[0] and ground in parallel using programmable analog switches. There is also an inverter with analog pin ua[5] as an input and analog pin ua[4] as an output. Try using the inverter as an amplifier as explained here: https://www.youtube.com/watch?v=03Ds1TnoMbA&ab_channel=MSMTUE and see if you get the same results when trying to use the inverters in my digital tile as an amplifier: https://github.com/JamesTimothyMeech/TT06/blob/main/info.yaml

How to test

Apply inputs to the inverters with a square wave or other signal generator and measure the output. To test the programmable resistor connect the supply voltage in series with an ammeter to pin ua[0]. Set ui[0] to ui[7] and ui[0] to ui[7] high to connect a set of 58.218 k ohm resistors to ground internally inside the chip. You should be able to measure differences in current as you connect each resistor to ground by setting the corresponding digital input pin high.

External hardware

TT06 printed circuit board, signal generator, an oscilliscope or similar to measure the input and output.

Pinout

#	Input	Output	Bidirectional
0	Control pin to connect one of the 16 resistors between ua[0] and ground using an analog switch	Not used, grounded internally	Control pin to connect one of the 16 resistors between ua[0] and ground using an analog switch
1	Control pin to connect one of the 16 resistors between ua[0] and ground using an analog switch	Not used, grounded internally	Control pin to connect one of the 16 resistors between ua[0] and ground using an analog switch
2	Control pin to connect one of the 16 resistors between ua[0] and ground using an analog switch	Not used, grounded internally	Control pin to connect one of the 16 resistors between ua[0] and ground using an analog switch
3	Control pin to connect one of the 16 resistors between ua[0] and ground using an analog switch	Not used, grounded internally	Control pin to connect one of the 16 resistors between ua[0] and ground using an analog switch

#	Input	Output	Bidirectional
4	Control pin to connect one of the 16 resistors between ua[0] and ground using an analog switch	Not used, grounded internally	Control pin to connect one of the 16 resistors between ua[0] and ground using an analog switch
5	Control pin to connect one of the 16 resistors between ua[0] and ground using an analog switch	Not used, grounded internally	Control pin to connect one of the 16 resistors between ua[0] and ground using an analog switch
6	Control pin to connect one of the 16 resistors between ua[0] and ground using an analog switch	Not used, grounded internally	Control pin to connect one of the 16 resistors between ua[0] and ground using an analog switch
7	Control pin to connect one of the 16 resistors between ua[0] and ground using an analog switch	Not used, grounded internally	Control pin to connect one of the 16 resistors between ua[0] and ground using an analog switch

Analog pins

ua#	analog#	Description
0	5	Internal programmable resistor connected to this pin
1	0	Analog pin not used
2	4	Analog pin not used
3	1	Analog pin not used
4	3	Analog inverter output
5	2	Analog inverter input

easy PAL [236]

- Author: Matthias Musch
- Description: This is a simple PAL device with shift-register based (re)configuration
- GitHub repository
- HDL project
- Mux address: 236
- Extra docs
- Clock: 0 Hz

How it works

This project is a PAL (programmable array logic device). It is programmed with a shift register.

Taped-out configuration and pin assignment Because I do not want to update the text below too often I write the configuration of the physical PAL device in terms of:

- Number of inputs
- Number of itermediatory stages
- Number of outputs ...only once. In the following this will be refered to however the exact number is only mentioned here. The numbers are:
- 8 inputs
- 11 intermediatory stages
- 4 outputs

There was a really convenient picture that unfortunatly I cannot include in the generated documentation. However if you check out the github repo of this project you can study it. It shows how the inputs, intermediate stages and output stages correlate to each other. Basically there is a matrix of inputs (N) and intermediate stages (P) with the size N*P. In the picture you can see numbers at the intersections of inputs/intermediate/output wires, which denote the indices of the shift register chain at this postion. The generated bitstream has a '1' at this positions if a connection is set and a '0' if no connection is set.

Pin assignment

- The eight inputs are connected to the eight `uio_in` wires.
- The enable pin to put the logic function on the outputs is connected to the `uio_in[1]` pin.
- The clock for the shift register is connected to `uio_in[2]`
- The configuration bit pin, which holds the data that is next shifted in is connected to the `uio_in[0]`. Aka here the bitstream is fed into - bit by bit!
- The outputs are displayed on the first four `uo_out[3:0]` bits.
- The rising edges are (clock for the shift register) are supplied via the `uio_in[2]` pin.

Programming At every rising edge of the programming-clock the shift register takes in a value from the config_bit pin. When the configuration is done the PAL implements the programmed combinatorial function(s). However in order to get the programmed function(s) to generate outputs the enable pin has to be asserted.

Generate bitstreams To generate bitstreams for the shift register a Python script is provided in this repository. It is important to set the right number of inputs, intermediate stages and outputs. This has to be exactly like the physical PAL-device you have at hand. A boolean logic function is denoted in the following way: `O0 = ~I0 | I1 &amp; ~(I2 &amp; I3)` It is important to declare the used variables before. See the Python script as it was done for O0, I1, I2, I3. You can add or remove variables. However keep in mind that the physical number of variables is limited. You can check the physical number that will be on the device in the `project.v` file.

At this point in time the bitstream generation in the Python script has some of limitations.

Using the PAL Okay now that you have transmitted the bitstream onto the PALs shift register you can set the enable pin (uio-pin) to output the programmed logic functions on the outputs.

How to test

By first shifting in a bitstream configuration into the device the AND/OR matrix of the device can be programmed to implement boolean functions with a set of inputs and outputs. You can test the design by clocking in a bitstream with a microcontroller (I will provide some example code for that) and by connecting buttons to the inputs and maybe LEDs to the outputs.

External hardware

No external HW is needed. However to see your glorious boolean functions come to life you might want to connect some switches to the inputs and LEDs to the outputs.

Pinout

#	Input	Output	Bidirectional
0	Combinatorial input 0	Combinatorial output 0	Config pin: This pin is used to apply the config bit that will be shifted in on a rising clock edge.
1	Combinatorial input 1	Combinatorial output 1	Enable pin: If HIGH (1) the result of the logic function is applied to all outputs.
2	Combinatorial input 2	Combinatorial output 2	Clock pin: Used for the shift register to sample in the [config pin] data (see uio[0]).
3	Combinatorial input 3	Combinatorial output 3	unused
4	Combinatorial input 4	Combinatorial output 3	unused
5	Combinatorial input 5	unused - tied to 0	unused
6	Combinatorial input 6	unused - tied to 0	unused
7	Combinatorial input 7	unused - tied to 0	unused

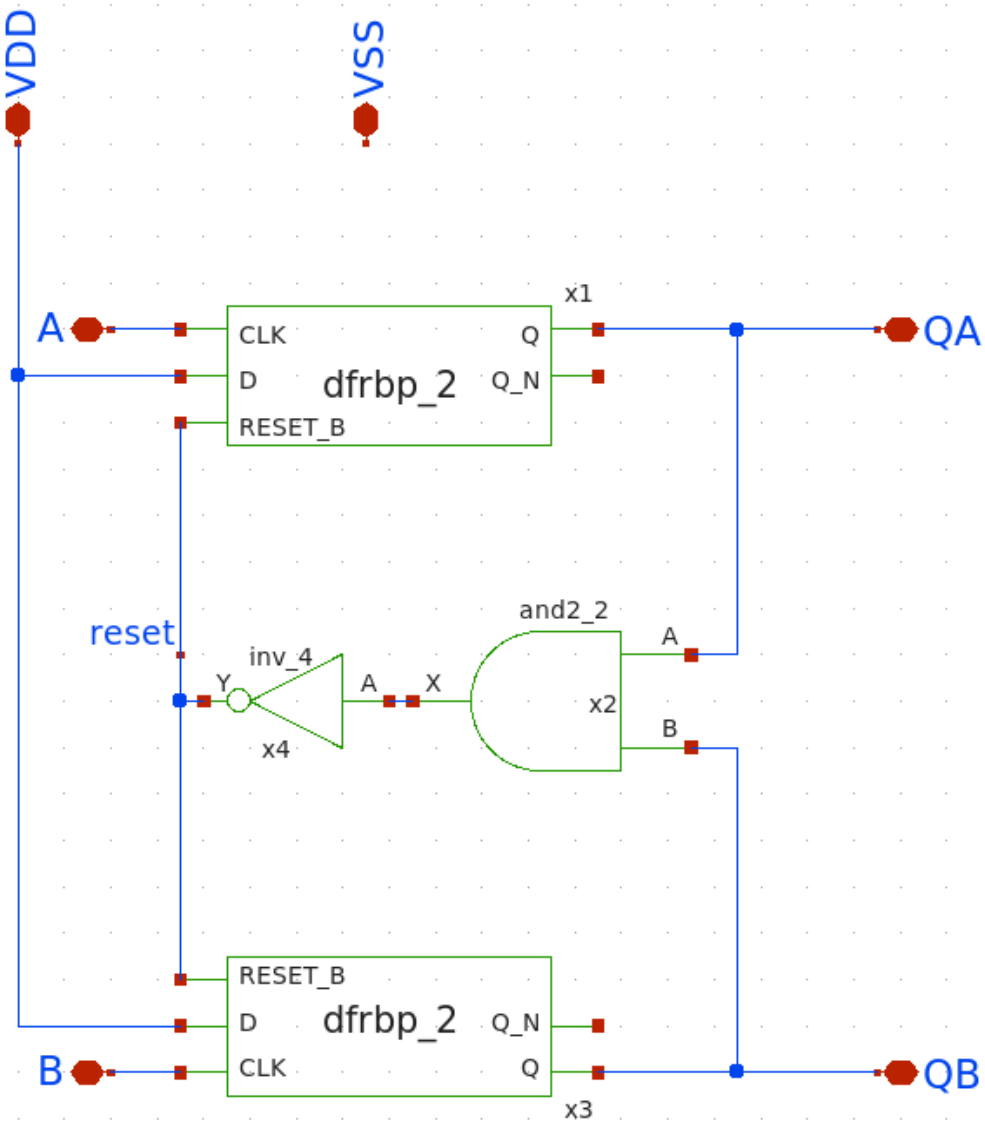
PLL blocks [237]

- Author: Vipul Sharma
- Description: This design contains blocks used in PLL circuit
- GitHub repository
- Analog project
- Mux address: 237
- Extra docs
- Clock: 0 Hz

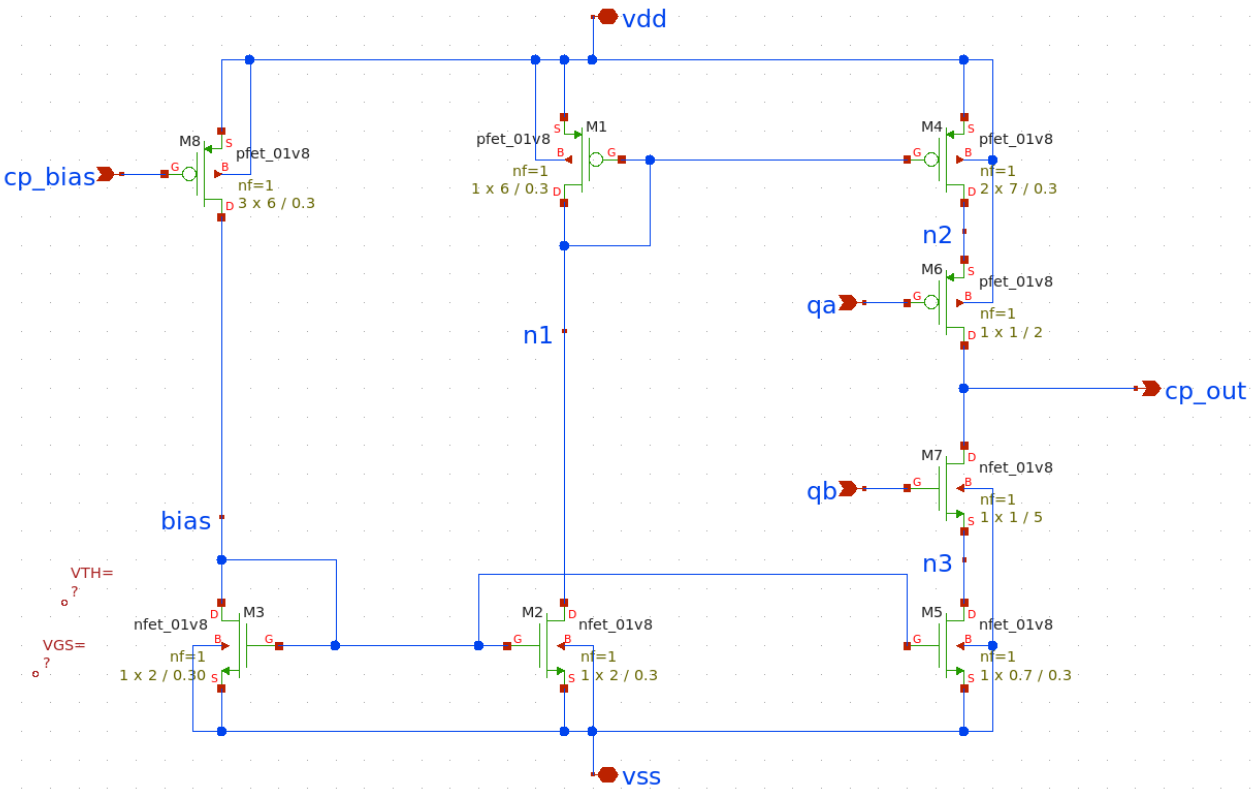
How it works

This design contains individual blocks used to realize a PLL circuit. These blocks are designed by participants under SSP (Saudi Semicondcutor Program) using open-source analog EDA tools. There are 4 blocks designed by multiple individuals and teams:

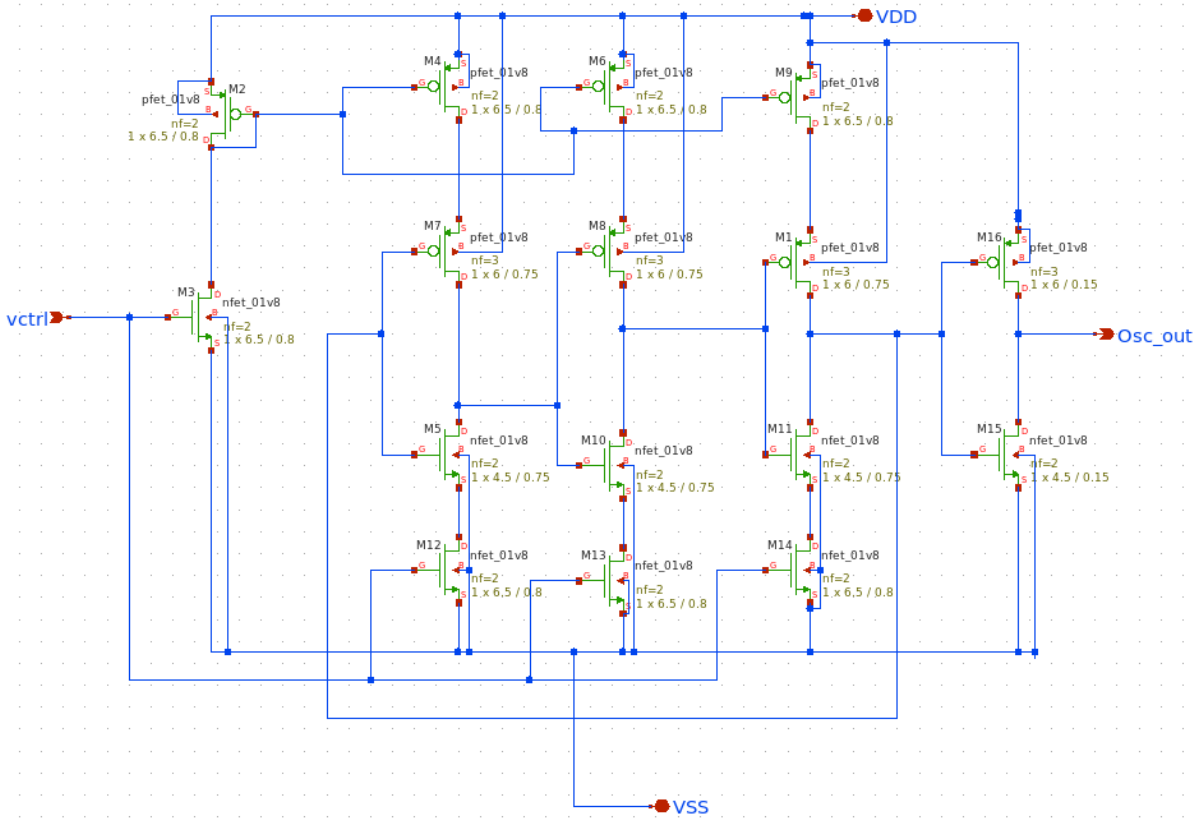
Phase Frequency Detector (PFD): D flip-flop-based phase frequency detector (PFD) with inputs A and B, and outputs QA and QB compares the phases of two input signals, A and B, and generates output pulses to indicate the phase difference between them. When A leads B, the output QA transitions high, while QB transitions low. Conversely, when B leads A, QA transitions low and QB transitions high. If both signals are in phase, neither QA nor QB transitions. PFD's output signals can be used to control the frequency and phase of a voltage-controlled oscillator (VCO) in a PLL system, thereby locking the output frequency and phase to the reference input. This is essential in applications such as clock synchronization, frequency synthesis, and communication systems, ensuring precise timing and synchronization.



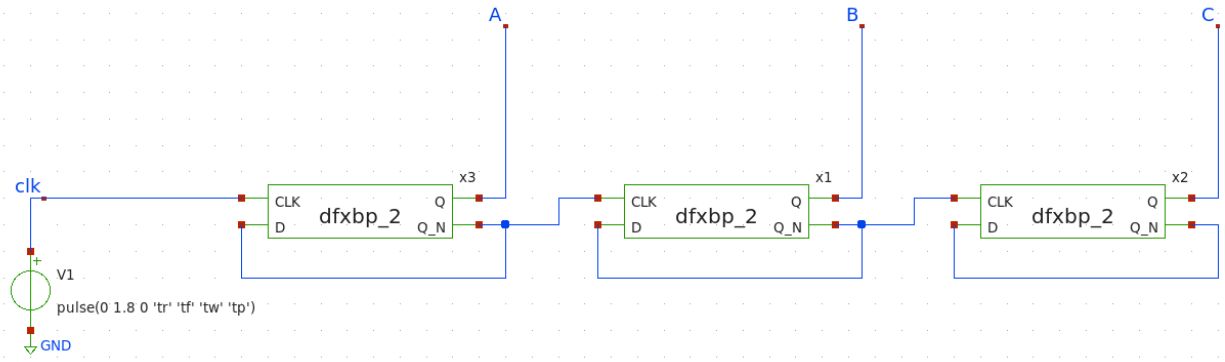
Charge Pump (CP): Charge pump circuit converts the output signals from the PFD into a control voltage for Voltage-Controlled Oscillator(VCO). Charge pump circuit consists of a pair of switches and a capacitor. When the PFD generates a positive pulse, one switch connects the capacitor to a reference voltage, charging it. Conversely, when the PFD generates a negative pulse, the other switch connects the capacitor to ground, discharging it. This creates a control voltage proportional to the phase difference between the input and reference signals.



Voltage Controlled Oscillator(VCO): A ring oscillator Voltage-Controlled Oscillator (VCO) consists of odd number of inverting stages connected in a ring configuration, generating an oscillating waveform. By controlling the bias voltage of the transistors within the stages, the oscillation frequency can be adjusted. This VCO serves as the controlled oscillator in the PLL, with its frequency locked to the reference signal through the feedback loop.



Frequency Divider(FD): A D flip-flop frequency divider divides the frequency of VCO output signal by a fixed integer ratio. This division process creates a feedback mechanism that compares the divided output frequency with the reference frequency. The D flip-flop's toggling action divides the frequency by 2/4/8, allowing for frequency multiplication or division within PLL loop.



	Designer Name	Block Name
1	Abdulrahman Alghamdi	Frequency Divider (FD-1)
2	Abdulrahman Alghamdi	PFD (PFD-1)
3	Baraa Musa Abdullah	PFD (PFD-2)
4	Faisal Tareq	Charge Pump (CP-1)

	Designer Name	Block Name
5	Khalid Abdulaziz	Frequency Divider (FD-2)
6	Khowla Alkhulayfi	Frequency Divider (FD-3)
7	Nawaf	VCO
8	Nawaf	Frequency Divider (FD-4)

How to test

PFD: Apply input pulses with phase difference between them and A and B pins at PFD input. Observe the output at QA and QB pins with output pulses based on +ve or -ve phase difference between signal applied at input pins of PFD.

CP: Apply input pulses at QA and QB input pins of charge pump replicating the output of PFD circuit. Based on whether QA or QB pulses are high, the output of charge pump circuit will demonstrate charging and discharging behaviour respectively. Charging and discharging rate can be controlled by changing bias voltage cp_bias to either increase of decrease current.

VCO: Apply a control voltage, vctrl=0.9V to VCO's input and measure the resulting output frequency. Verify that the output frequency varies with the applied control voltage within the specified range i.e. 0.75V to 1V. Check VCO frequency tuning range by sweeping the control voltage across and observing the output frequency response.

FD: Input a signal with a frequency (40 to 80MHz range) to input of frequency divider. Measure the output frequency using a oscilloscope. Verify that the output frequency is one-eighth of the input frequency.

External hardware

2-channel function/waveform generator with varying frequency and pulse time generation. 2-channel Oscilloscope to measure output signal waveforms.

Pinout

#	Input	Output	Bidirectional
0	CP-1:QA	PFD-2:QA	FD-1:fo
1	CP-1:QB	PFD-2:QB	FD-1:fo_by_8
2	FD-2:Clk	FD-2:fo_4	PFD-1:A
3	FD-3:fo	FD-2:fo_8	PFD-1:B

#	Input	Output	Bidirectional
4		FD-3:fo_4	PFD-1:QA
5		FD-3:fo_8	PFD-1:QB
6	ref	FD-4:Out_4	PFD-2:A
7		FD-4:Out_8	PFD-2:B

Analog pins

ua#	analog#	Description
0	5	CP_bias
1	0	vout
2	4	vctrl
3	1	Osc_out
4	3	cp_bias
5	2	out

Rule 30 Engine! [238]

- Author: andrewtron3000
- Description: Iterate Rule 30 Cellular Automaton
- GitHub repository
- HDL project
- Mux address: 238
- Extra docs
- Clock: 50000000 Hz

How it works

This project is designed to run the Rule 30 cellular automata with initial starting value defined on the input bus. Each iteration is output to the UART, which is operating as an 8N1 UART at 115,200 baud.

The Rule 30 field is 56 bits wide with a boundary condition of 0 surrounding these 56 bits.

How to build

The Rule 30 driver and logic are written in Bluespec. Install the Bluespec compiler to compile these sources into Verilog. The generated Verilog is also included in this repo as well in case you don't have the Bluespec compiler.

The Bluespec compiler can be found here: https://github.com/B-Lang-org/bsc.

Several Bluespec generated Verilog files need to be copied from the Bluespec library directory:

```
Counter.v
FIFO1.v
SizedFIFO.v
```

To compile the Bluespec, do the following:

```
cd src
make verilog
```

At this point all necessary Verilog files have been created.

External hardware

The initial condition used for the Rule 30 iteration is defined on the input bus. The output in `uo_out[4]` is used as the TX side of the UART.

Pinout

#	Input	Output	Bidirectional
0	Initial Value LSB		
1	Initial Value		
2	Initial Value		
3	Initial Value		
4	Initial Value	UART TX	
5	Initial Value		
6	Initial Value		
7	Initial Value MSB		

TT06 Analog Factory Test [239]

- Author: Sylvain Munaut
- Description: Test structures for TT06 analog support
- GitHub repository
- Analog project
- Mux address: 239
- Extra docs
- Clock: 0 Hz

How it works

FIXME

How to test

FIXME

External hardware

FIXME

Pinout

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

Analog pins

ua#	analog#	Description
0	5	ibias
1	0	vgnd_sense
2	4	vpwr_sense
3	1	loopback[0]
4	3	loopback[1]
5	2	loopback[2]

tt06-RV32E_MinMCU [258]

- Author: Weihao Liu
- Description: Microcontroller RV32E implementation. Supports inputs, outputs, GPIOs, UART and SPI.
- GitHub repository
- HDL project
- Mux address: 258
- Extra docs
- Clock: 24000000 Hz

How it works

RV32E implementation for a minimum microcontroller that is designed for small HW projects. The microcontroller interfaces with an external NOR flash for program memory and a external PSRAM for RAM over SPI.

The MCU has the following peripherals:

- 7 x input/output pins
- Up to 5 input only pins
- Up to 4 output only pins
- 1 x UART (flow control can be enabled)
- 1 x SPI bus
- Debug interface over SPI to read out registers and program counter

There is only 1 SPI controller in the design and this controller is used to interface with program memory and RAM. This SPI controller can be configured to interface with other SPI peripherals too.

Tested with Lattice ice40-HX8K breakout board at 24MHz clock.

Pin allocation

PIN	UI_IN	UO_OUT	UIO
0	IN0/UART-CTS	UART-RX	SPI-CS2
1	IN1	OUT0/UART-RTS	IO0
2	SPI-MISO	OUT1	IO1
3	IN2	SPI-MOSI	IO2
4	IN3	SPI-CS1	IO3
5	IN4	SPI-SCLK	IO4

PIN	UI_IN	UO_OUT	UIO
6	EN_DEBUG	OUT2	IO5
7	UART-TX	OUT3	IO6

Memory space

Memory address	Description
0x00000 - 0x0FFFF	Program memory, read-only (external memory)
0x10000 - 0x1FFFF	RAM (external PSRAM)
0x20000 - 0x2FFFF	Peripheral registers

Peripheral registers

Pin control registers

Address	Description
0x20000	Output values for the output pins (OUT0 to OUT3).
0x20001	Input values for the input pins (IN0 - IN4), read-only.
0x20002	Direction bits for the IO pins. Set to 1 for output, 0 for input.
0x20003	Input values for IO pins. The corresponding bit in the IO direction register has to be set to 0, for the input values to be set.
0x20004	Output values for IO pins. The corresponding bit in the IO direction register has to be set to 1, for the output value to be set.

SPI peripheral registers The SPI controller interfaces with program memory and RAM. It can additionally be configured to interface with other SPI devices by configuring the output pins (OUT0-OUT3) as CS pins.

0x20005 - SPI control register As the SPI controller is shared for program memory and RAM access, the entire CPU is blocked until the SPI transaction is completed.

Bit	Description
0	Set to 1 to start SPI transaction
1	Set to 1 to use OUT0 as CS pin
2	Set to 1 to use OUT1 as CS pin
3	Set to 1 to use OUT2 as CS pin
4	Set to 1 to use OUT3 as CS pin

Note: Only 1 CS pin can be configured each time. When the OUTn pin are is as CS, that pin in the output bits register (0x20000) will be ignored.

Address	Description
0x20006	SPI status register. Bit 0 is set to 1 when the SPI transaction is completed. This bit is cleared when an SPI transaction is started (by writing 1 to bit 0 of the SPI control register 0x20005).
0x20008	SPI TX byte. Byte to transmit to SPI peripheral
0x2000C	SPI RX byte. Byte received from SPI peripheral

UART peripheral registers

0x20010 - UART control register

Bit	Description
0	Set to 1 to start TX
1	Set to 1 to clear RX byte availabe bit in the UART status register
2	Set to 1 to enable flow control. OUT0 is used as CTS and IN0 is used as RTS.

0x20011 - UART status register

Bit	Description
0	When 1, the TX operation is completed
1	RX byte available bit, is set to 1 when there is a byte available in the RX buffer

Address	Description
0x20014	UART Tx byte. Set byte to be written to external UART device
0x20015	Stores byte that is received from external UART device

Debug mode To set the CPU into debug mode, set the EN_DEBUG pin to HIGH. In this mode, the CPU will continuosly output the program counter and all registers (excluding x0 register) over the SPI interface. OUT3 is used as the DEBUG_CS pin.

How to test

1. Load a program into the program memory
2. Assert and deassert rst_n pin
3. Interact with the program

External hardware

This project requires at minimum the following:

- PMOD for SPI flash (example, digilent PMOD SF3)
- PMOD for SPI PSRAM chip

Pinout

#	Input	Output	Bidirectional
0	IN0/UART-CTS	UART-RX	SPI-CS2
1	IN1	OUT0/UART-RTS	IO0
2	SPI-MISO	OUT1	IO1
3	IN2	SPI-MOSI	IO2
4	IN3	SPI-CS1	IO3

#	Input	Output	Bidirectional
5	IN4	SPI-SCLK	IO4
6	EN_DEBUG	OUT2	IO5
7	UART-TX	OUT3	IO6

Crossbar Array [263]

- Author: Kevin Guan
- Description: Analog Matrix Multiplication with 64x64 array
- GitHub repository
- Analog project
- Mux address: 263
- Extra docs
- Clock: 0 Hz

How it works

This is an analog crossbar array used as a placeholder for future ReRAM projects. Poly Resistors are used instead of ReRAM between Met1 and Met2. Thus, this project has a fixed weight matrix. This project performs 4x4 matrix multiplication in one run.

How to test

6 analog pins and 8-bit digital input bus are used. First 4 analog pins (i.e. ua[3:0]) are used as inputs to the crossbar. The ua[4] is the supply voltage 0-1.8V (default: keep at 1.8V). The ua[5] is the output analog pin used for observing the output current (summing junction). First 4 digital input pins (i.e. ui_in[3:0]) control the 4 4-bit muxes on the input side. The ui_in[4:7] is used to control the column selection.

External hardware

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

This project will need external off-the-shelf DACs and ADCs including a TIA or ADC that can convert output current into a readable voltage.

Pinout

#	Input	Output	Bidirectional
0	bit control 1		
1	bit control 2		
2	bit control 3		
3	bit control 4		
4	write/select control 1		

#	Input	Output	Bidirectional
5	write/select control 2		
6	write/select control 3		
7	write/select control 4		

Analog pins

ua#	analog#	Description
0	11	bit 1
1	6	bit 2
2	10	bit 3
3	7	bit 4
4	9	write 1
5	8	select 1

TinyRV1 CPU [264]

- Author: Prof. Dr. Matthias Jung, Jonathan Hager, Philipp Wetzstein
- Description: TinyRV1 compliant CPU that has to be attached to an external SPI memory. The ISA is described in the documentation
- GitHub repository
- HDL project
- Mux address: 264
- Extra docs
- Clock: 12000000 Hz

How it works

The project consist of a RISC-V VHDL Model and supports the Tiny RV1 ISA without MUL. In addition AND and XOR are supported.

How to test To test our design you will need to use external hardware.

External hardware To use our design you will need to use the provided spi_slave_tt06_with_memory and synthesize it for an 12 MHz FPGA.

Pinout

#	Input	Output	Bidirectional
0	SPI MISO	SPI MOSI	Register_1(5)
1	unused	SPI SCLK	Register_1(6)
2	unused	SPI CS	Register_1(7)
3	unused	Register_1(0)	Register_1(8)
4	unused	Register_1(1)	Register_1(9)
5	unused	Register_1(2)	Register_1(10)
6	unused	Register_1(3)	Register_1(11)
7	unused	Register_1(4)	Register_1(12)

WoWA [265]

- Author: Pat Deegan
- Description: Is it really the World Worst ADC? Maybe it'll be a wow-ADC instead... we'll see!
- GitHub repository
- Analog project
- Mux address: 265
- Extra docs
- Clock: 0 Hz

How it works

This project is a mixed signal design that glues together a few bits to create a simple ADC. It uses

- An analog comparator, based on the design done by Stefan Schippers in Analog schematic capture and simulation, re-captured in xschem and laid out with magic;
- The analog blob from the R2R DAC in Matt Venn's R2R DAC TT06 submission;
- A digital signal processor and front end, created using Amaranth; and
- A few analog switches and a 2:1 analog mux, created and laid out for the project.

While at it, I also laid out a version of the p3 opamp from this project and embedded it in for testing purposes.

The ADC uses the DAC to set the threshold on the comparator and see what it says about the input signal–is it higher or lower–to perform a search and hone in on a digital value to output. Doing it in this way, it manages to determine a value in about 60 clock cycles).

The analog output of the comparator, R2R DAC and p3 opamp are all provided through analog pins for testing and experimentation.

A few options are available:

- The system can perform a comparator calibration before each reading (which increases the processing time but should make things more reliable). Enable this by holding the enable calibrations pin high;
- Rather than feed the R2R DAC output to the comparator, it can receive input from an analog pin instead. Set "use external threshold" input pin HIGH for this, and feed into appropriate analog pin.

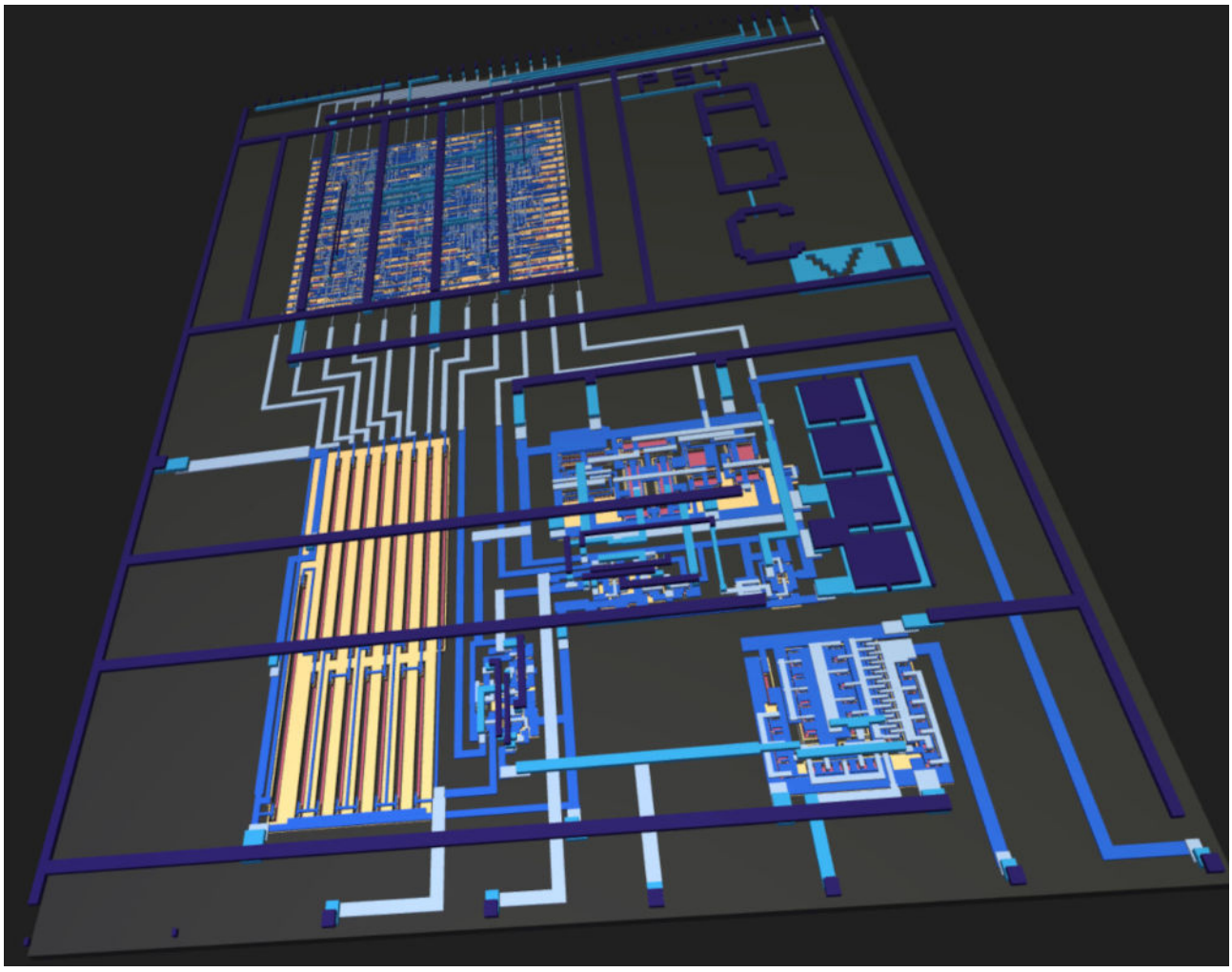
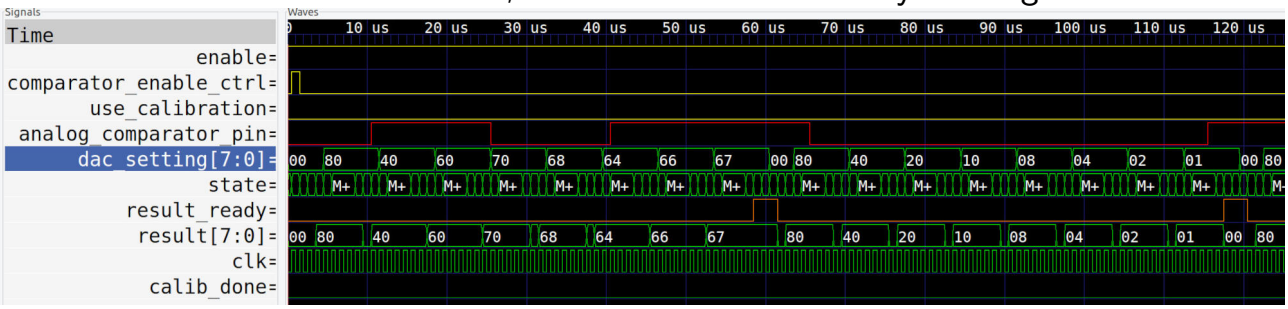


Figure 22: wowa ADC

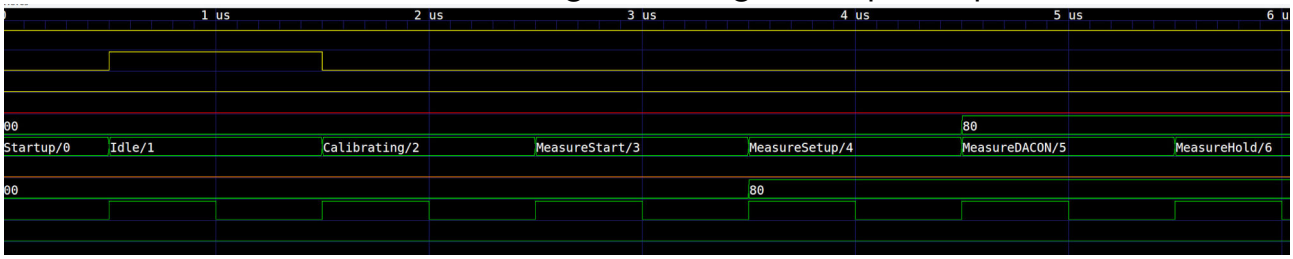
Internals Some details about how it's built...

Digital With no calibration enabled, it takes about 60 clock cycles to get a valid result



In the above, there is a simulated analog comparator output (in red). The DAC is setting each bit, starting with the high bit, in turn. If the comparator says HIGH when the output is checked, it means that the input voltage is higher than what we're comparing to so that bit is preserved. Each bit is set, if the comparator is low when we check, that bit is cleared from the eventual result. When we reach the LSB, the result ready signal goes high–it's held that way for 3 clocks. Then the process repeats.

This all works with an FSM that goes through multiple steps, zoomed in here:



For simplicity, we alway pass through the calibrating state, though it only lasts a single clock cycle when the use_calibration pin is low.

For things to work, the comparator does need some calibration, at least sometimes. How often? I don't know yet. When use_calibration is high, the cycle has a period at the start where:

- the DAC is set mid-range
- the comparator receives the same input on both inputs and is put into calibrate mode

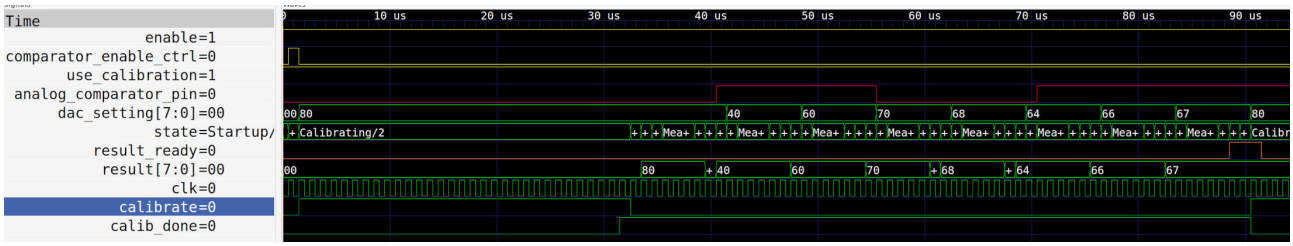


Figure 23: With calib readings

This actually charges up a capacitor internally which is used to adjust the comparator output. Because this takes a finite amount of time, which was found to be around 400ns for reliable operation in simulation, I gave it 28 clock cycles of calib time to support a theoretical clock of up to 70MHz.

Using a conservative 50MHz clock, this means that readings can be done in about 1.75 microseconds with calibration, and 1.2 us without. Assuming we need to cal every 3 samples, this gives the ADC a throughput of around 720k samples per second, or more than 560ksps if you want to keep it simple and calibrate before every measurement, on a 50MHz clock.

These are *theoretical* maxima, of course. I haven't even tried to drive things this fast in sim, though that is planned.

And this is for the digital side only. From the analog side, a lot depends on the comparator (more below). Short version is that it can react quite quickly.

Here's a sim of the full analog side, looking at the comparator (a number of runs, gaussian distribution of temperatures):

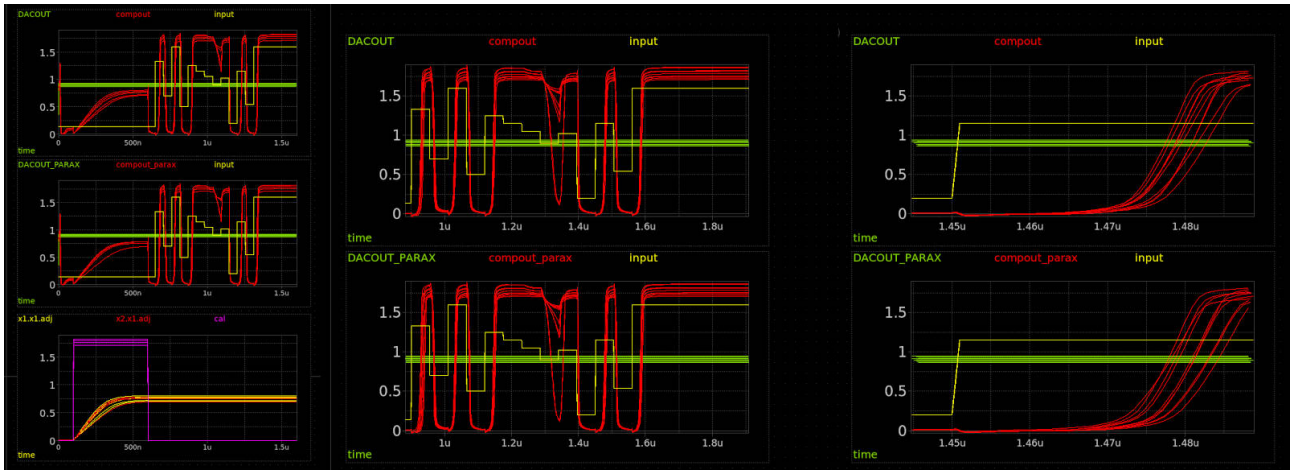


Figure 24: comparator sim 1

Green is the threshold we're comparing to (DAC output), yellow is the input signal (so clean), and red is the comparator output. On the left, you can see the whole sequence including a calibration step at the beginning.

Things get a little fuzzy when the input is very close to the threshold, but otherwise you can see the comparator does a good job comparing, and reacts quickly too. About 40 nanoseconds after the input goes above the threshold: bam, comparator is logic HIGH, even in the worst cases.

So we're talking being able to deal in the tens of MHz for this wonderful circuit.

However I did find some instances, like here where I suddenly set the DAC above the input (so comparator should go low):

Figure 25: comparator slow reactions

That's not pretty. Suddenly we're looking at 200-400ns sometimes. Ok, so let's revise that down to the low single digit MHz. Say we always want to leave 400ns before sampling the comparator output, well it'll take us 3.2us To get a full byte of comparisons, so we're down to 312ksps. I'd be ok with that.

A final *however* on this front: if we really want 400ns for things to settle after the DAC is set, the digital side has 4 clock cycles between setting the DAC value, and the cycle when it actually captures the compator output. I think that means we'd have to clock at 10MHz, thereby reducing our best effective max sample rate to 144ksps – booo. Ok, not that bad if it actually works and I am being quite conservative (hopefully) with that 400ns settle time.

Analog The analog side of things is at the service of the digital. The main ADC block is

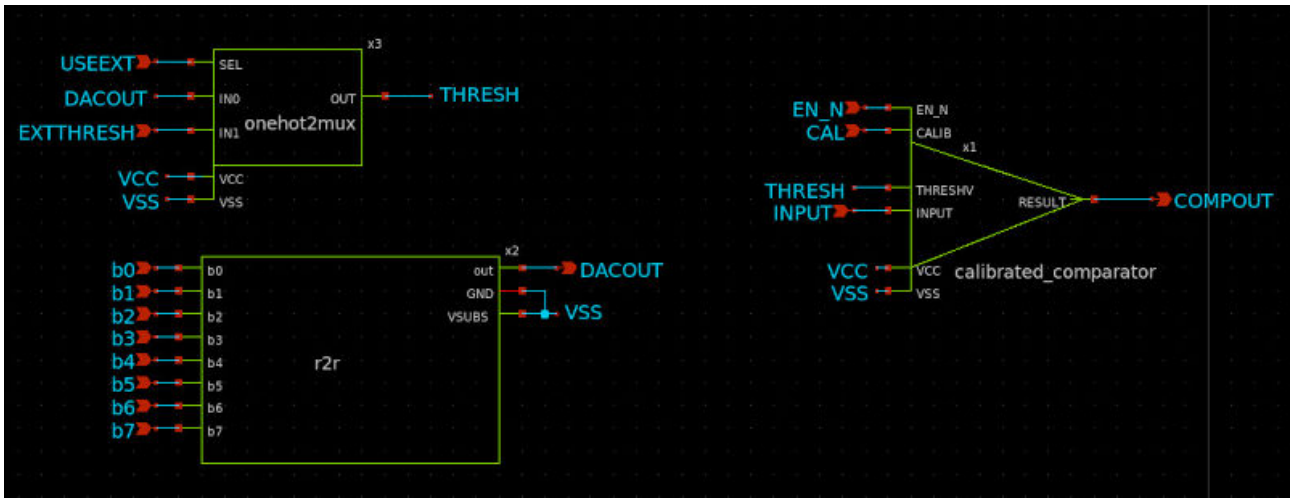


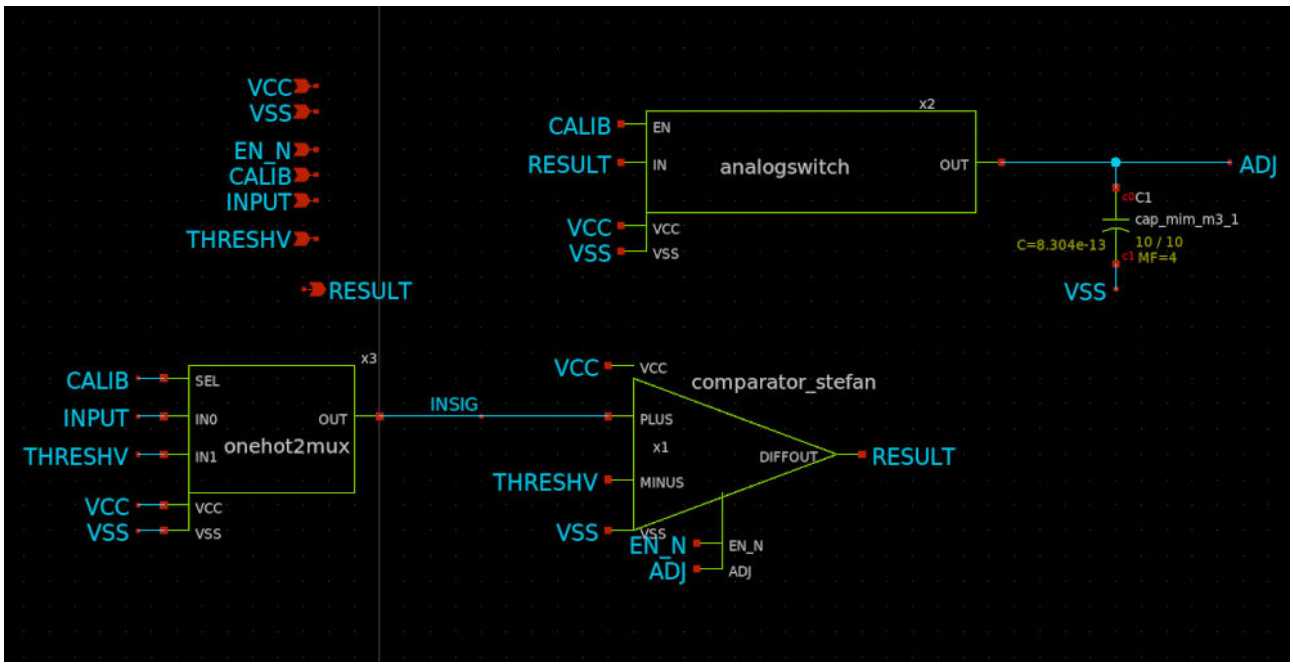
Figure 26: WoWA Analog

The r2r DAC is simply controlled using 8 bits, the end. Its output goes to a 2:1 analog mux–called one-hot here because I intended to make 4:1 and maybe other versions that'd just be one-hot but that's for later.

External input goes to one side of the comparator, and the output of this mux goes to the other – that lets us send either the DAC output or whatever's coming in from an external pin to the other side for comparison.

That calibrated comparator has a CAL input. That's because, inside of that symbol is

this:



Another mux and and analog switch, in addition to the comparator. That lets us choose between sending the input or the threshold to the plus side of the comparator. Sending the threshold to both inputs of the comparator seems useless, but that only happens when CALIB is HIGH, which also trips the analog switch and pipes the output of the comparator to the capacitor and the adjust pin on said comparator. This is why it's a *calibrated* comparator: Hold CALIB high for a little bit, and the results come out looking a lot better.

Finally, inside that triangle is the actual comparator circuit, which I re-did watching a video of Stefan Schippers teaching some xschem design.

It's a bunch of FETs doing FET things.

TODO Well, I should probably have put a sample-and-hold system in there–would've been easy with the analog switch and a cap... c'est la vie.

Also, I didn't know how big the digital side would turn out, so I went the route of premature optimization (the "route" of all evil), and just used the output result bits as my scratch for the DAC twiddling. The upside is that we get to have a good look at what the DAC is actually doing. The downside is that it's a bit noisy, you only get a valid result during the result_ready blip (which I actually stretched with an additional FSM state).

The analog and digital together are all LVS clean and the digital side is tested in sim and with formal verification methods (a bit), but I never got a full simulation at the gate level because I couldn't coax verilator into handling my python-generated verilog... caused a weird bug, close to deadline. Too bad: fingers crossed.

Would be nice to have a mode to control the DAC manually.

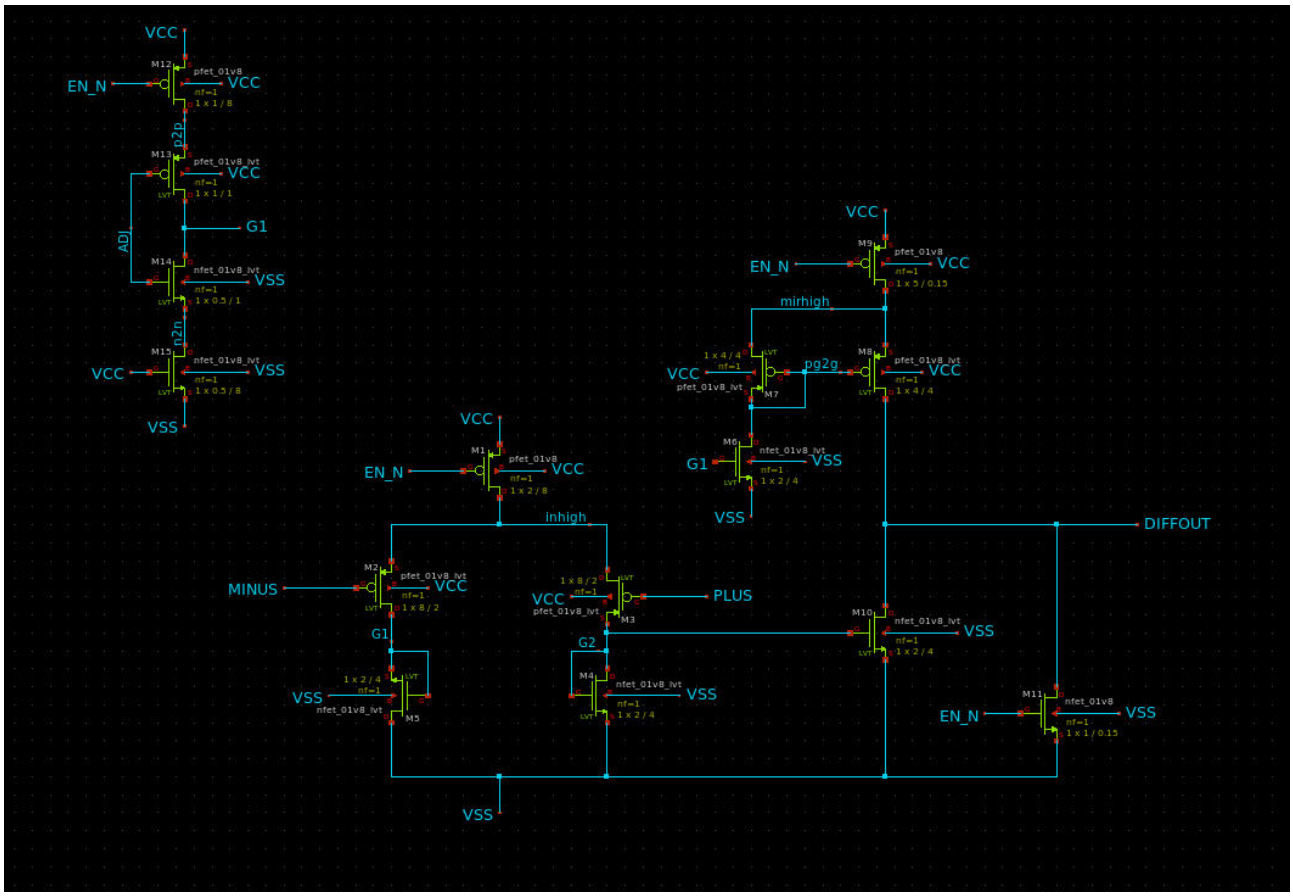


Figure 27: wowa stefan comparator

I wasn't exactly certain how the digital reset side would behave–I got scared that I'd wind up with reversed logic and the demoboard holding everything in reset forever. So I left it unconnected and sacrificed a digital in to be the new reset pin. Sadness. But safe. Just remember it needs to be connected, "right" (probably low to go but, yeah, not sure, hah).

More grounding, and try to get in some bypass/bulk (for tiny values of bulk) capacitance. We don't get filler, so it's DIY I did not D.

Before all that: build up testbench, get full simulation working, see if it'll even be worth the wait and have something to compare to when it does come back in physical form.

How to test

Bring enable comparator, and reset pin low (?), feed a target voltage (less than 1v8) into appropriate analog input pin, clock the device and watch the output bits on the digital side.

When result ready output pin pulses high, the output bits are a calculated result.

Seems like it will be safe to clock at 10MHz, maybe more, uncertain as of yet.

For your testing pleasure, there are addition inspection and manipulation ports through the analog pins. These are

- ua[0]: The raw output from the comparator at the core of the design
- ua[1]: p3 opamp out
- ua[2]: p3 opamp plus side
- ua[3]: p3 opamp minus/ext threshold for comp
- ua[4]: Analog input to ADC
- ua[5]: A probe into Matt's R2R DAC output (internal threshold for comparator while running the ADC

So the ADC basically only needs you to feed a signal into ua4. You can see it in operation through ua0 and ua5. You can use the comparator ignoring the ADC function, by setting ui[3] (use external threshold, digital input) HIGH and feeding a threshold voltage to the comparator on ua3.

Finally, there's a whole opamp in there, designed by Sai, which I laid out and included as a second test of this design. We'll be able to see if there's any measurable difference and play with opamps, which is fun.

External hardware

Voltage source for analog input. Some way to look at outputs, nominally through the RP2040 on the demoboard–will be writing a script for that.

Pinout

#	Input	Output	Bidirectional
0	reset	result bit 0	result ready
1	enable calibrations	result bit 1	0
2	enable comparator	result bit 2	0
3	use external threshold	result bit 3	0
4		result bit 4	1
5		result bit 5	1
6		result bit 6	1
7		result bit 7	1

Analog pins

ua#	analog#	Description
0	11	Comparator out
1	6	p3 opamp out
2	10	p3 opamp plus
3	7	p3 opamp minus/ext threshold for comp
4	9	Analog input to ADC
5	8	Matt's DAC output

A 555-Timer Clone for Tiny Tapeout 6 [267]

- Author: Vincent Fusco
- Description: Blinks an LED the hard way
- GitHub repository
- Analog project
- Mux address: 267
- Extra docs
- Clock: 0 Hz

How it works

It duplicates the functionality of a 555 timer. Try configuring it in the "astable" configuration using external resistors and capacitors.

How to test

Find a 555 timer datasheet and attempt some of the suggested circuits.

Connect pins to a breadboard with jumper wire.

Construct circuit shown in Figure 7-5 at: https://www.ti.com/lit/ds/symlink/lmc555.pdf?ts=171

Test 2:

Duplicate circuit in Figure 6-2 at: https://www.ti.com/lit/ds/symlink/lmc555.pdf?ts=17117380
Compare resulting maximum frequency. The CMOS-based TI 555-Timer has a maximum frequency of 3.0MHz. Compare.

External hardware

1. Wires for breadboard.
2. Through-hole resistors and capacitors of various values, LEDs, etc.
3. Breadboard.
4. Oscilloscope (for maximum frequency test)

#	Input	Output	Bidirectional

Pinout

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

Analog pins

ua#	analog#	Description
0	7	V_THRESH_I
1	9	V_TRIG_B_I
2	8	V_DISCH_O

Dickson Charge Pump [269]

- Author: Uri Shaked
- Description: Pumps the input voltage up to ~5.4V
- GitHub repository
- Analog project
- Mux address: 269
- Extra docs
- Clock: 0 Hz

How it works

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

How to test

Apply a clock signal of 2 MHz to the `clk` input. In TT06, the analog pin voltage is limited to 1.8 V, so the output voltage will be divided by 6. You can measure the divided output voltage at the `ua[0]` (vout_div) pin.

Simulation results

Post layout simulation showing the output voltage `x1.vout` and the divided output voltage on ta `ua[0]` pin, with ~16.8 mega ohms load (the internal voltage divider). The output voltage stabilizes at ~5.07 V, and the divided output voltage at ~0.85 V. The current draw is about 355 nA.

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

Silicon measurements

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

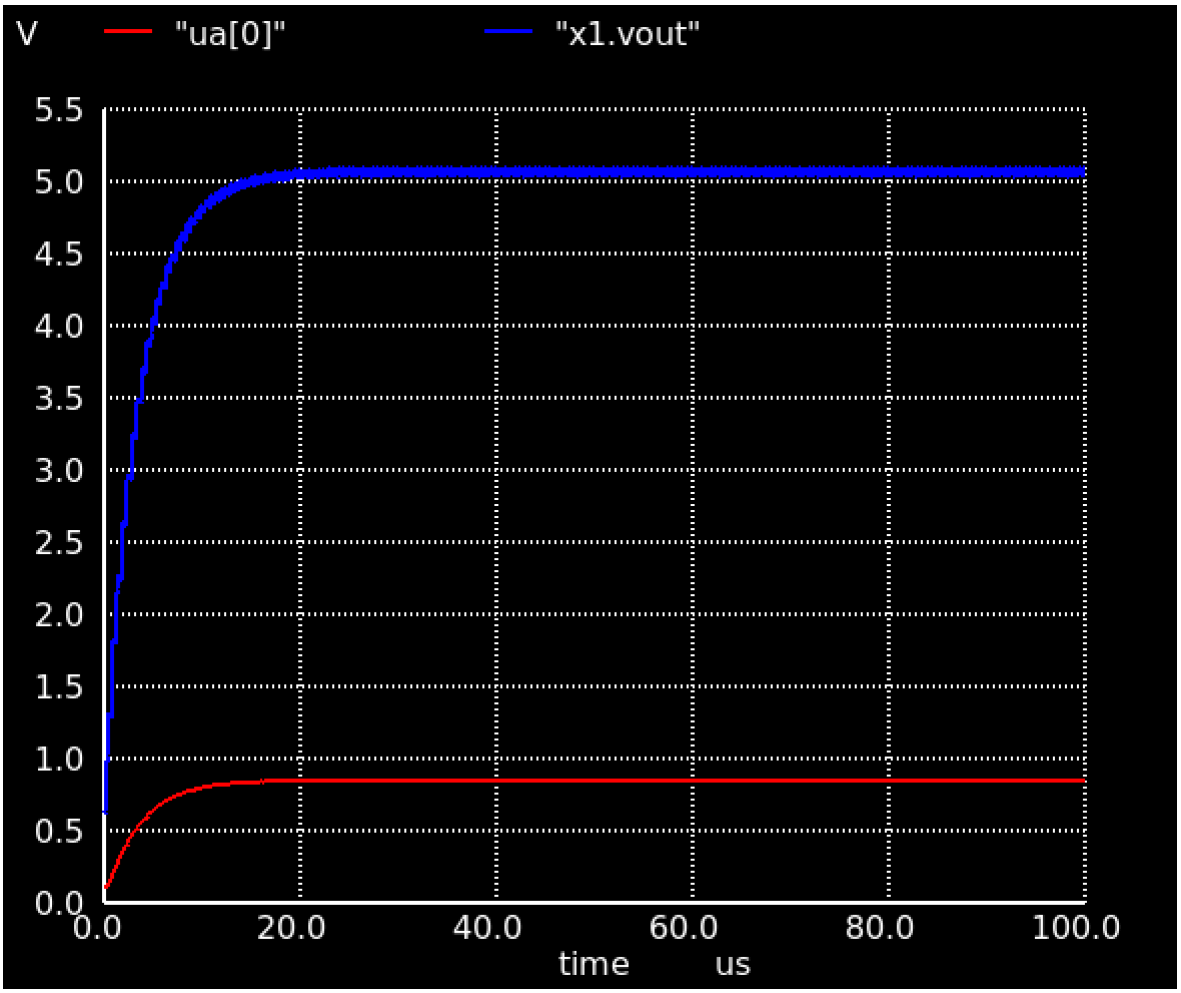


Figure 28: output voltage and divided voltage

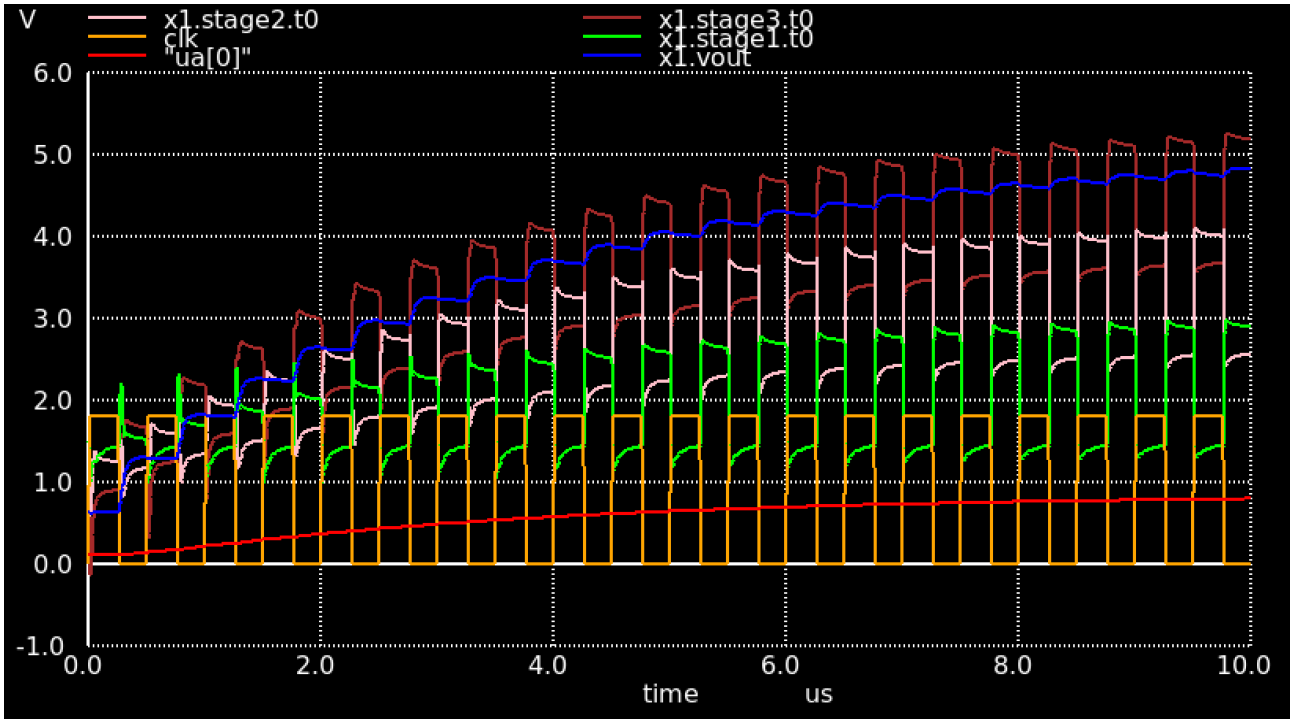


Figure 29: output voltage and intermediate voltages

Input Frequency (KHz)	ua[0] Voltage	Charge Pump Voltage *
0	0.090	0.540
10	0.107	0.642
50	0.171	1.026
100	0.267	1.602
250	0.462	2.772
500	0.604	3.624
1000	0.673	4.038
2000	0.704	4.224
5000	0.716	4.296
7500	0.716	4.296
10000	0.716	4.296
15000	0.714	4.284
20000	0.712	4.272
40000	0.698	4.188
62000	0.676	4.056

- The charge pump voltage is the ua[0] voltage measurement multiplied by 6. This is because the analog pin voltage is limited to 1.8 V, so the output voltage will be divided by 6.

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

Overall, it seems that the charge pump works as expected, with the output voltage peaking at around 4.3 V when the input frequency is in the 5-10 MHz range.

Pinout

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

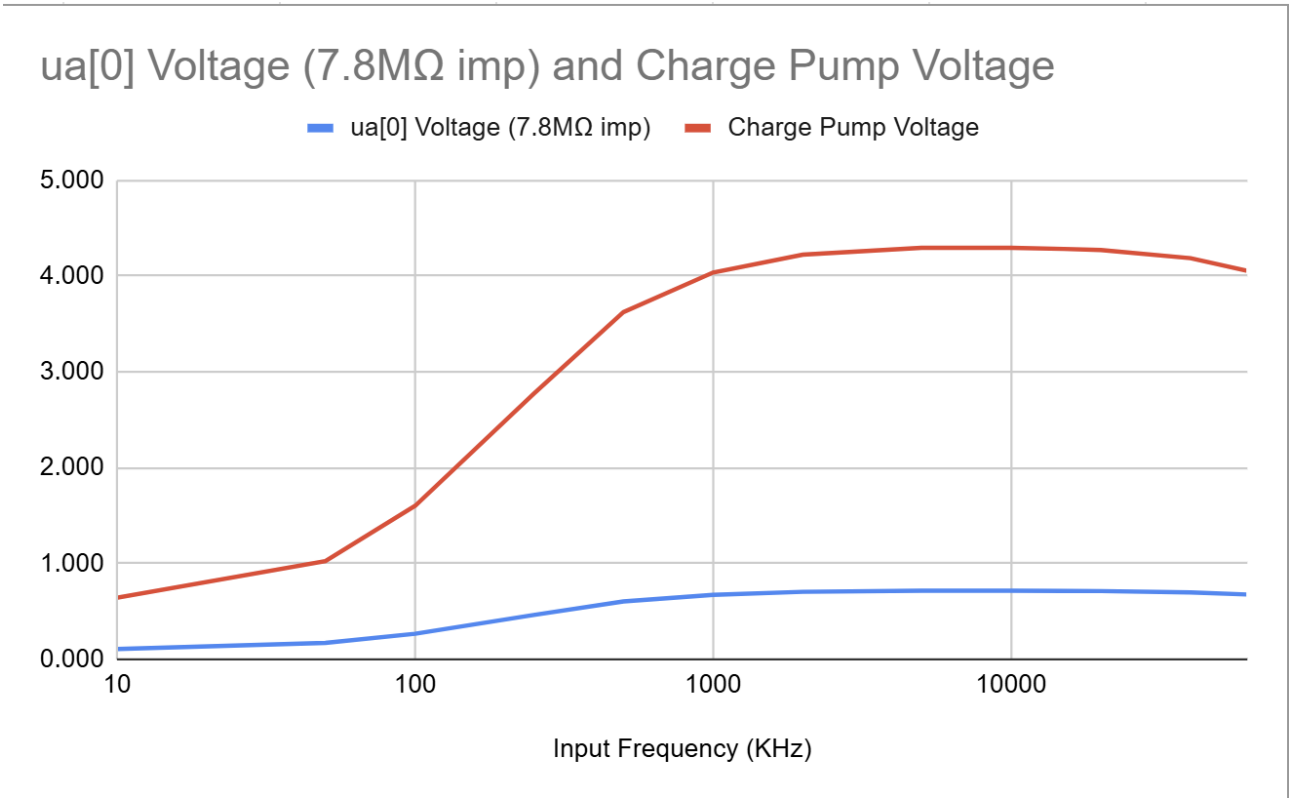


Figure 30: output voltage vs frequency

Analog pins

ua#	analog#	Description
0	8	vout_div

Neurocore [270]

- Author: Kyrylo Kalashnikov
- Description: 2x2 systolic array multiplier using 16bit floats
- GitHub repository
- HDL project
- Mux address: 270
- Extra docs
- Clock: 20000000 Hz

How it works

This is a an implementation of a classic systolic array multiplier with uart interface.

This implimention can multiply 2x2 matrix by another 2x2 matrix. The calculations are done in float16.

How to test

Your "How to Test" section outlines a clear testing protocol for your systolic array multiplier with a UART interface, designed to multiply 2x2 matrices using float16 representations. To make it more structured and clear, I've refined your instructions below:

How to Test

1. **Initialization Sequence**: Begin by sending the initialization sequence `11111110` to the device. This sequence prepares the device to start receiving data for matrix multiplication.
2. **Sending Matrix Data**:
    - You will need to send the elements of the two matrices you wish to multiply. Each matrix element is a float16 number, which must be transmitted as two separate 8-bit frames (high byte first).
    - Since you are multiplying 2x2 matrices, you must send a total of 8 float16 numbers, equating to 16 data frames of 8 bits each.

- For clarity, send the matrix elements in row-major order. For example, if your matrices are A and B, with elements `a11, a12, a21, a22` for A and similarly for B, send them in the order `a11, a12, a21, a22, b11, b12, b21, b22`.

3. **Receiving the Result**:

   - Upon completion of the data processing, the device will first send back an acknowledgment sequence `11111110`, indicating that the multiplication process is complete and the device is about to send the result.
   - Following this, expect to receive the result of the matrix multiplication in a similar format to the input. The device will send 4 float16 numbers (representing the resulting matrix elements) as 8-bit frames (high byte first), which you will need to interpret accordingly.

4. **Interpreting the Results**:

   - Collect the 8-bit frames received from the device and reconstruct them into float16 numbers to obtain the resulting matrix elements.
   - These elements represent the resultant matrix from the multiplication of the two input matrices.

External hardware

No external hardware is used for this project.

Pinout

#	Input	Output	Bidirectional
0	RX input	RX output	Block multiply done status
1			Calculation start signal
2			Send State bit 0
3			Send State bit 1
4			Send State bit 2
5			Send State bit 3
6			Done send signal
7			Send data signal

Tiny Opamp [271]

- Author: argunda
- Description: Super simple two stage opamp without miller compensation
- GitHub repository
- Analog project
- Mux address: 271
- Extra docs
- Clock: 0 Hz

How it works

This opamp has VDD=1.8V and VSS=0V. It's input common mode range is not very good so make sure your AC input signal is centered around 0.9V. The opamp is internally biased so you just need to apply a differential input.

It should be able to hit 60dB gain at low frequencies. Please do not connects loads requiring more than a few mA.

How to test

Power up the chip, test opamp in closed loop configuration only. VOUT is analog pin 0. PLUS is a differential input on analog pin 1. MINUS is a differential input on analog pin 2.

External hardware

At the bare minimum a resistor at the output is needed to test the opamp as a source-follower. Use multimeter or oscilloscope to probe the output.

Pinout

#	Input	Output	Bidirectional
0	pause	blue	
1	new_game	green	
2	down_key	red	
3	up_key	hsync	
4		vsync	
5		speaker	

#	Input	Output	Bidirectional
6		col0	
7		row0	

Analog pins

ua#	analog#	Description
0	11	VOUT1
1	6	PLUS1
2	10	MINUS1
3	7	VOUT2
4	9	PLUS2
5	8	MINUS2

test for tiny tapeout hackaday [288]

- Author: ivo-tt
- Description: a useless test
- GitHub repository
- Wokwi project
- Mux address: 288
- Extra docs
- Clock: 0 Hz

How it works

It does not. Don't use it.

How to test

If you are disappointed, it's your own fault.

External hardware

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

Pinout

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

Triple Watchdog [289]

- Author: Ignacio Chechile
- Description: A redundant watchdog to be part of a larger fault-tolerant supervisor/failover manager later on
- GitHub repository
- HDL project
- Mux address: 289
- Extra docs
- Clock: 0 Hz

How it works

The circuit is composed of three watchdog modules working in lockstep. They share a common clock, a common reset line (active low) and a common 8-bit input word. An external system must write a value in ui_in in order to kick the watchdog, and the value must be different than the previous one each time. The timeout is fixed and set in 1ms. Once the timeout happens, the WD module goes to IDLE mode until a reset is issued and each internal watchdog will set an output pin high.

Pinout

Pin	Direction	Comment
clk	input	
rst_n	reset	Active low
ui_in[8]	input	System must write a new value to kick the watchdog
watchdog_expired1	output	1: wd has expired; 0: wd has not expired
watchdog_expired2	output	1: wd has expired; 0: wd has not expired
watchdog_expired3	output	1: wd has expired; 0: wd has not expired

How to test

- Provide a 10ns period clock in its clk input
- Set rst_n to low
- Write a value in ui_in before 1ms after reset is released
- Write a different value in ui_in to prevent the watchdog expiring

External hardware

None

Pinout

#	Input	Output	Bidirectional
0	ui_in[0]	watchdog_expired1	
1	ui_in[1]	watchdog_expired2	
2	ui_in[2]	watchdog_expired3	
3	ui_in[3]		
4	ui_in[4]		
5	ui_in[5]		
6	ui_in[6]		
7	ui_in[7]		

1-Bit ALU 2 [290]

- Author: FW
- Description: Test Tapeout
- GitHub repository
- Wokwi project
- Mux address: 290
- Extra docs
- Clock: 0 Hz

How it works

1-Bit ALU

How to test

1-Bit ALU

External hardware

None List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Pinout

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

Minibyte CPU [291]

- Author: Zach Frazee
- Description: A super simple 8-bit CPU
- GitHub repository
- HDL project
- Mux address: 291
- Extra docs
- Clock: 50000000 Hz

How it works

The Minibyte CPU is a simple "toy" 8-bit CPU that uses a custom RISC instruction set

The CPU also has some built in DFT (Design For Test) features and a Demo ROM that can be enabled for easy testing (some external hardware required)

This was created mostly as a learning/reference project to get more familiar with Verilog

Specs

```
Max CLK Frequency: 50Mhz (untested)

Data Buss Width:    8 bits
Address Buss Width: 8 bits (only 7 bits usable due to limited IO)

Registers:
    A   - 8 bits wide - Accumulator
    M   - 8 bits wide - Memory Address Pointer
    PC  - 8 bits wide - Program Counter
    IR  - 8 bits wide - Instruction Register
    CCR - 2 bits wide - Condition Code Register

Memory Mapped Registers:
    R0  - 8 bits wide - Gen Purpose Reg 0
    R1  - 8 bits wide - Gen Purpose Reg 1
    R2  - 8 bits wide - Gen Purpose Reg 2
    R3  - 8 bits wide - Gen Purpose Reg 3
    R4  - 8 bits wide - Gen Purpose Reg 4
```

```
    R5  - 8 bits wide - Gen Purpose Reg 5
    R6  - 8 bits wide - Gen Purpose Reg 6
    R7  - 8 bits wide - Gen Purpose Reg 7

Number of Instructions: 37

ALU:
    Data Inputs: 2x 8 bit inputs
    Data Output: 8 bits (result) + 2 bits (flags)

    Operations Supported:
        PASSA - Passthrough input A
        PASSB - Passthrough input B
        ADD   - Add A and B
        SUB   - Subtract B from A
        AND   - Logical and of A, B
        OR    - Logical or of A, B
        XOR   - Logical xor of A, B
        LSL   - Logical shift A left by B
        LSR   - Logical shift A right by B
        ASL   - Arithmetic shift A left by B
        ASR   - Arithmetic shift A right by B
        RSL   - Rotary shift A left by B
        RSR   - Rotary shift A right by B

    Flags:
        Z - Set if result is 0, otherwise clear
        N - Set if result is a negative signed int, otherwise clear
```

Pinout

```
uio[7:0]    - DATA IN/OUT BUSS
ui_in[7:0]  - DFT Test and Configuration Select
uo_out[7]   - WE (Write Enable Signal)
uo_out[6:0] - ADDR OUT BUSS
```

Architecture The Minibyte CPU uses a very traditional architecture where most data is manipulated via a single accumulator (A Register)

The ALU operates on data from the A Register and either direct data (from memory indexed by the M register), or immediate data (from the current instruction's operand indexed by the PC register)

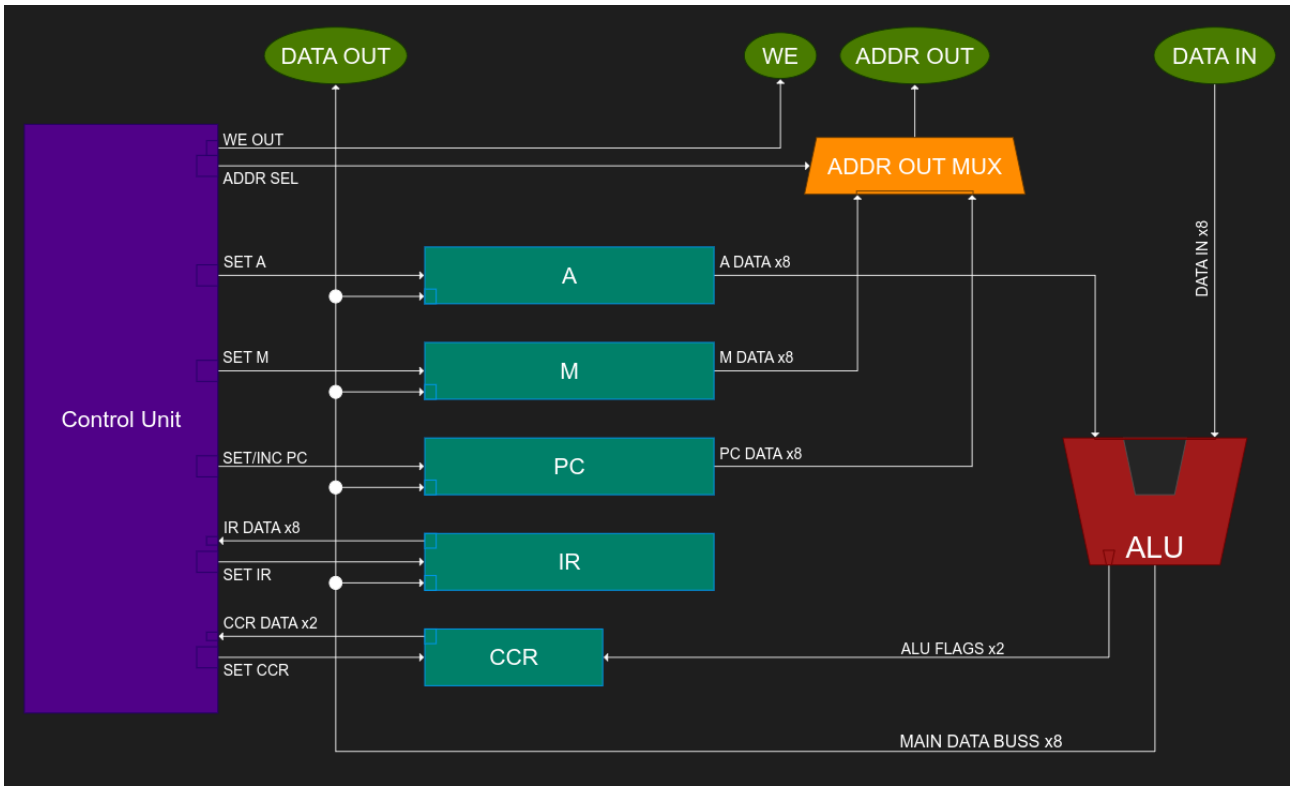


Figure 31: Minibyte Block Diagram

*Note that DFT and testing features are not represented in the above block diagram

Power Up State Upon reset, the device will be initialized with all registers cleared to 0x00. This includes the program counter (PC register). It is expected that the program memory will start at address 0x00 to begin execution.

Instruction Set The Minibyte CPU has 4 instruction format types. The program memory is chunked into bytes. Some instructions only occupy a single byte, while others occupy 2 bytes for an opcode and a following operand

Type	Length	Desc
Inherent	8 - bits	IR with no operand
Immediate	16 - bits	IR with an operand containing DATA

Type	Length	Desc
Direct	16 - bits	IR with an operand containing an ADDRESS
Indirect	16 - bits	IR with an operand containing an ADDRESS that points to another ADDRESS

As a visual reference, here is how we would expect a basic program to be structured in memory. Please note that all programs start execution from address 0x00 as shown.

ADDRESS	0x00	0x01	0x02	0x03	0x04	0x05	0x06
DATA	NOP	LDA_IMM	0x05	ADD_IMM	0x03	JMP_DIR	0x00

Figure 32: Example Program Memory

The above program adds the numbers 0x05 and 0x03 together, and then loops back to the starting IP of 0x00

Inherent IR:

Type	OP[7:0]
Inherent	IR OPCODE

Immediate/Direct IR:

Type	OP[15:8]	OP[7:0]
Immediate	IR OPCODE	OPERAND DATA
Direct	IR OPCODE	OPERAND ADDRESS
Indirect	IR OPCODE	OPERAND ADDRESS

Opcode Table:

OPCODE	HEX	Operand	CCR
NOP	0x00	N/A	N/A
LDA_IMM	0x01	Immediate	N/A
LDA_DIR	0x02	Direct	N/A
STA_DIR	0x03	Direct	N/A
STA_IND	0x04	Indirect	N/A
ADD_IMM	0x05	Immediate	N/A
ADD_DIR	0x06	Direct	N/A
SUB_IMM	0x07	Immediate	N/A
SUB_DIR	0x08	Direct	N/A
AND_IMM	0x09	Immediate	N/A
AND_DIR	0x0A	Direct	N/A
OR_IMM	0x0B	Immediate	N/A
OR_DIR	0x0C	Direct	N/A
XOR_IMM	0x0D	Immediate	N/A
XOR_DIR	0x0E	Direct	N/A
LSL_IMM	0x0F	Immediate	N/A
LSL_DIR	0x10	Direct	N/A
LSR_IMM	0x11	Immediate	N/A
LSR_DIR	0x12	Direct	N/A
ASL_IMM	0x13	Immediate	N/A
ASL_DIR	0x14	Direct	N/A
ASR_IMM	0x15	Immediate	N/A
ASR_DIR	0x16	Direct	N/A
RSL_IMM	0x17	Immediate	N/A
RSL_DIR	0x18	Direct	N/A
RSR_IMM	0x19	Immediate	N/A
RSR_DIR	0x1A	Direct	N/A
JMP_DIR	0x1B	Direct	N/A
JMP_IND	0x1C	Indirect	N/A
BNE_DIR	0x1D	Direct	Z==CLEAR
BNE_IND	0x1E	Indirect	Z==CLEAR
BEQ_DIR	0x1F	Direct	Z==SET
BEQ_IND	0x20	Indirect	Z==SET
BPL_DIR	0x21	Direct	N==CLEAR
BPL_IND	0x22	Indirect	N==CLEAR
BMI_DIR	0x23	Direct	N==SET
BMI_IND	0x24	Indirect	N==SET

OPCODE	Desc
NOP	No Operation
LDA_IMM	Load A with immediate operand data
LDA_DIR	Load A with the data stored at the operand addr
STA_DIR	Store A at the operand addr
STA_IND	Store A at the addr contained at the operand addr
ADD_IMM	Add the immediate operand data to A
ADD_DIR	Add the data stored at the operand addr to A
SUB_IMM	Subtract the immediate operand data from A
SUB_DIR	Subtract the data stored at the operand addr from A
AND_IMM	And the immediate operand data with A
AND_DIR	And the data stored at the operand addr with A
OR_IMM	Or the immediate operand data with A
OR_DIR	Or the data stored at the operand addr with A
XOR_IMM	Xor the immediate operand data with A
XOR_DIR	Xor the data stored at the operand addr with A
LSL_IMM	Logical shift A left by the immediate operand data
LSL_DIR	Logical shift A left by the data at the operand addr
LSR_IMM	Logical shift A right by the immediate operand data
LSR_DIR	Logical shift A right by the data at the operand addr
ASL_IMM	Arithmetic shift A left by the immediate operand data
ASL_DIR	Arithmetic shift A left by the data at the operand addr
ASR_IMM	Arithmetic shift A right by the immediate operand data
ASR_DIR	Arithmetic shift A right by the data at the operand addr
RSL_IMM	Rotate A left by the immediate operand
RSL_DIR	Rotate A left by the data stored at the operand addr
RSR_IMM	Rotate A right by the immediate operand data
RSR_DIR	Rotate A right by the data stored at the operand addr
JMP_DIR	Jump PC to the operand addr
JMP_IND	Jump PC to the addr stored at the operand addr
BNE_DIR	Jump PC (if Z is clear) to the operand addr
BNE_IND	Jump PC (if Z is clear) to the addr stored at the operand addr
BEQ_DIR	Jump PC (if Z is set) to the operand addr
BEQ_IND	Jump PC (if Z is set) to the addr stored at the operand addr
BPL_DIR	Jump PC (if N is clear) to the operand addr
BPL_IND	Jump PC (if N is clear) to the addr stored at the operand addr
BMI_DIR	Jump PC (if N is set) to the operand addr
BMI_IND	Jump PC (if N is set) to the addr stored at the operand addr

DFT and Extra Features The Minibyte CPU has a few DFT features that might be useful on live silicon for debug/testing. All DFT functions are enabled by an active high signal on one of the ui_in[7:0] pins (ui_in[7:0] should be tied to zero during normal operation)

ui_in Bit	Feature
ui_in [7]	Enable Memory Mapped Gen Purpose Registers
ui_in [6:5]	Unused
ui_in [4]	Enable Demo ROM
ui_in [3]	Halt Control Unit on Next Fetch
ui_in [2:0]	Debug Output Signal Select

Gen Purpose Registers: The Gen Purpose Registers are a set of 8 memory mapped general purpose registers that can be accessed at the following addresses as long as ui_in[7] is held high

Reg Name	Mem Address
Register R0	0x78
Register R1	0x79
Register R2	0x7A
Register R3	0x7B
Register R4	0x7C
Register R5	0x7D
Register R6	0x7E
Register R7	0x7F

Debug Out Select: The CPU has an extra mux between the normal addr out mux and the uo_out pins. To leverage this ui_in[2:0] can be used to select a debug signal to output on the uo_out[6:0] pins in place of the normal address buss

Debug Out Select	Function
ui_in[2:0] = 0b000	Normal Operation
ui_in[2:0] = 0b001	Output A[6:0] to uo_out[6:0]
ui_in[2:0] = 0b010	Output A[7] to uo_out[0]
ui_in[2:0] = 0b011	Output M[6:0] to uo_out[6:0]
ui_in[2:0] = 0b011	Output PC[6:0] to uo_out[6:0]
ui_in[2:0] = 0b011	Output IR[6:0] to uo_out[6:0]
ui_in[2:0] = 0b011	Output CCR[1:0] to uo_out[1:0]

Debug Out Select	Function
ui_in[2:0] = 0b011	Output CU_STATE[6:0] to uo_out[6:0]

How to test

Simulation The Minibyte CPU has fairly exhaustive cocotb test suite that is able to test and verify most of the device's intended functionality.

To run the test suite, cd into the ./test directory of the project and run "make"

```
**************************************************************************************
** TEST                          STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
**************************************************************************************
** test.test_tm_debug_out_a      PASS      390000.00           0.01   41321104.51  **
** test.test_nop                 PASS      400000.00           0.00   135321955.49  **
** test.test_lda_imm_sta_dir     PASS      760000.00           0.01   114924867.30  **
** test.test_lda_dir_sta_ind     PASS      920000.00           0.01   101107288.99  **
** test.test_alu_imm_dir         PASS   660120000.00           7.39   89368986.17  **
** test.test_alu_ccr             PASS   660120000.00           7.20   91631501.54  **
** test.test_jmp_dir             PASS      360000.00           0.00   94971346.89  **
** test.test_jmp_ind             PASS      440000.00           0.00   114166023.15  **
** test.test_bne_beq             PASS     1160000.00           0.01   96877715.83  **
** test.test_bpl_bmi             PASS     1160000.00           0.01   104239799.55  **
** test.test_demorom             PASS   137380000.00           1.21   113202537.93  **
**************************************************************************************
** TESTS=11 PASS=11 FAIL=0 SKIP=0          1463210000.01          15.89   92101070.49  **
**************************************************************************************
```

Figure 33: Simulation Results

On Live Silicon The easiest way to test the Minibyte CPU on live silicon is to use the built-in Demo ROM

To enable the Demo ROM, make sure that ui_in[4] and ui_in[7] are held high on reset, and remain high while the program runs

Holding ui_in[4] high will enable the Demo Rom

Holding ui_in[7] high will enable the General Purpose Registers, which are required/utilized by the Demo ROM program

The Demo ROM will run the following program

```
PSEUDOCODE:
    WHILE FOREVER{
        //Part 1: Binary Count
        SET A to 0
```

```
        WHILE A <= 255 {
            INCREMENT A

            WRITE A to ADDRESS 0x40
        }

        //Part 2: Walking 1
        SET A to 1

        WHILE A > 0 {
            LEFT SHIFT A by 1

            WRITE A to ADDRESS 0x40
        }

        //Part 3: Deadbeef to RAM/Gen Purpose Registers and back out
        LOAD 0xDEADBEEF into R0->R3

        WRITE R0 to ADDRESS 0x40
        WRITE R1 to ADDRESS 0x40
        WRITE R2 to ADDRESS 0x40
        WRITE R3 to ADDRESS 0x40
    }
```

To capture the output of the program with LEDs, it is recommended to add a D-Flip Flop (such as a 74x273 series chip) on the output of the data buss (uio[7:0]). See External Hardware section below for more details

External hardware

Demo Setup Something similar to the above schematic is recommended when running the Demo ROM. Note that an inverter (such as a 74x04 series chip) should be used as shown on the CLK input of the DFF. We want data to be latched when WE falls to 0 (after the data has had time to set up and make its way out of the chip). Please also note that you will probably need to run the CPU at a fairly low CLK frequency in order to see any LED activity with the naked eye.

Other Setups The sky is the limit as far as as what devices you attach to the CPU. If you are writing your own programs, you probably are going to want to attach some

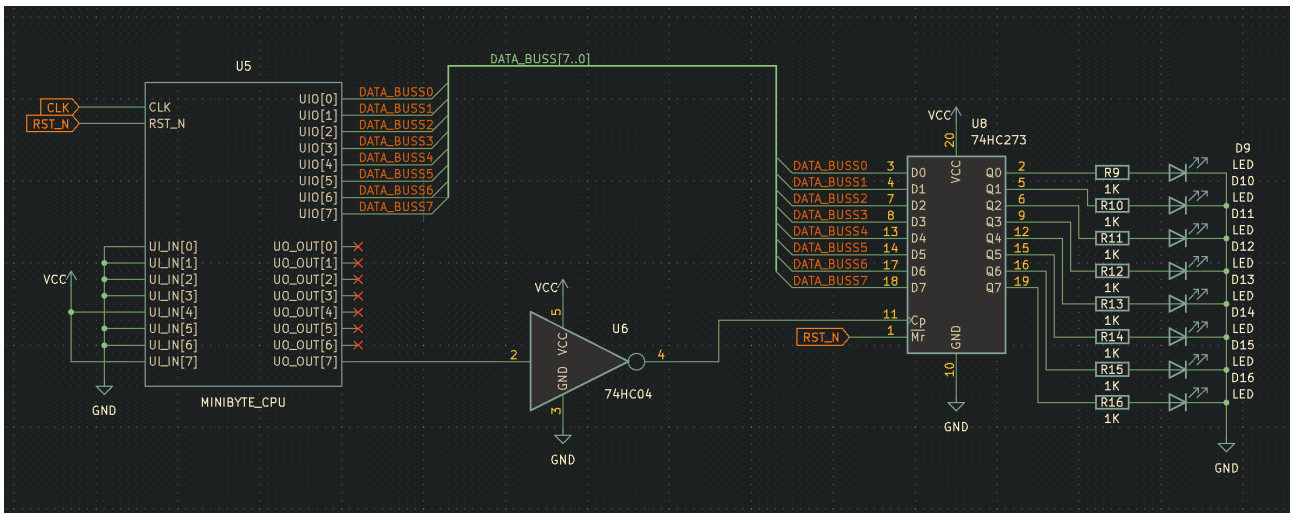


Figure 34: Demo Schematic

sort of external ROM to the main address and data buss. Here is a recommended setup adding an external EEPROM.

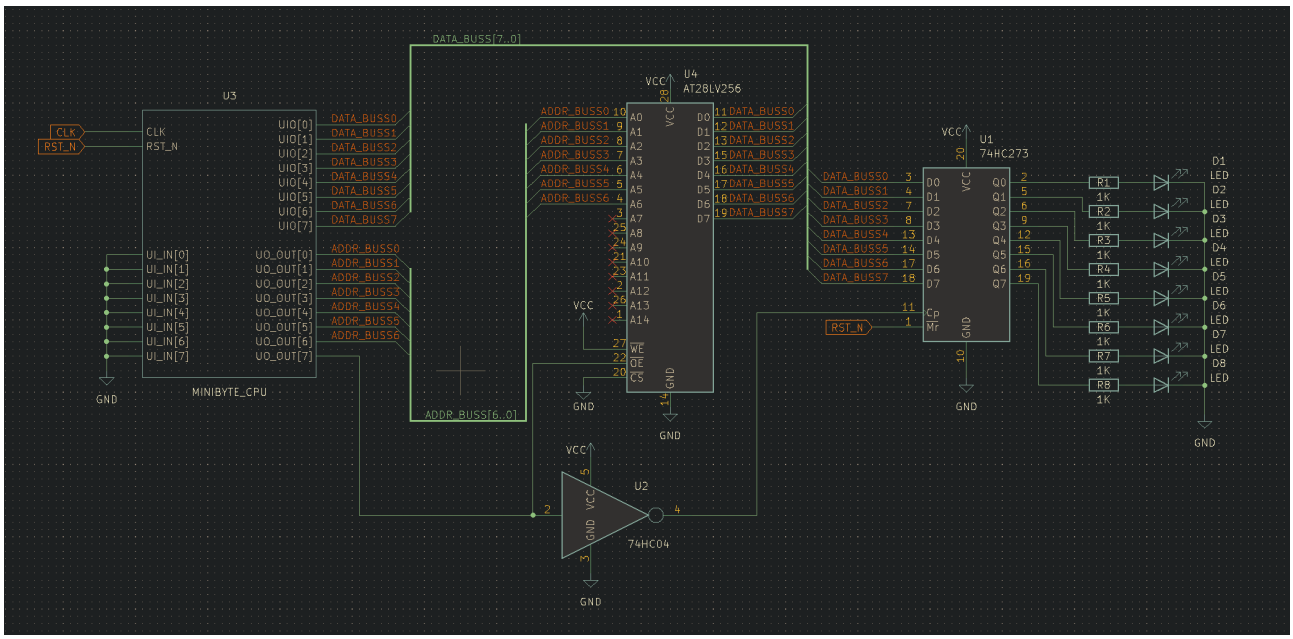


Figure 35: External EEPROM Schematic

Beyond this, you will hopefully find that the Minibyte CPU can be paired with a wide variety 3.3V compatible parallel ROM/EPROM/EEPROM, SRAM, and IO expander modules.

Pinout

#	Input	Output	Bidirectional
0	DEBUG_OUT_SELECT_0	ADDR_OUT_0	DATA_0
1	DEBUG_OUT_SELECT_1	ADDR_OUT_1	DATA_1
2	DEBUG_OUT_SELECT_2	ADDR_OUT_2	DATA_2
3	HALT_CU	ADDR_OUT_3	DATA_3
4	DEMO_ROM_ENABLE	ADDR_OUT_4	DATA_4
5		ADDR_OUT_5	DATA_5
6		ADDR_OUT_6	DATA_6
7	ENABLE_GEN_REGS	WE_OUT	DATA_7

Bestagon LED matrix driver [292]

- Author: Marijn
- Description: Driver for a hexagonal charlieplexed LED matrix
- GitHub repository
- Wokwi project
- Mux address: 292
- Extra docs
- Clock: 1000 Hz

We all know the hexagon is the bestagon. If the cells in your eyes that catch light are hexagonal, it doesn't make sense that all displays use a square grid. This has to end now. That is why the Bestagon LED matrix driver is born.

How it works

This circuit can drive a charlieplexed hexagonal LED matrix. This matrix has columns with 3-4-5-4-3 pixels.

How to test

- Connect the display shown under "External hardware" (or if you want to see if the circuit is functioning: connect a few LEDs between the display output pins randomly)
- Set the Display Enable pin low.
- The Data pin is now sampled on each rising clock edge. The data shall be entered column wise, bottom to top, right to left (in the schematic below, "1" represents the first bit entered). The following data may make you smile: 10010101000010101000
- Now set the Display Enable pin high.
- Keep pulsing the clock pin (at least 100Hz is recommended)

External hardware

Charlieplexed hexagonal display:

(Maybe add some resistors depending on what LEDs you chose. I recommend blue LEDs because they look cool.

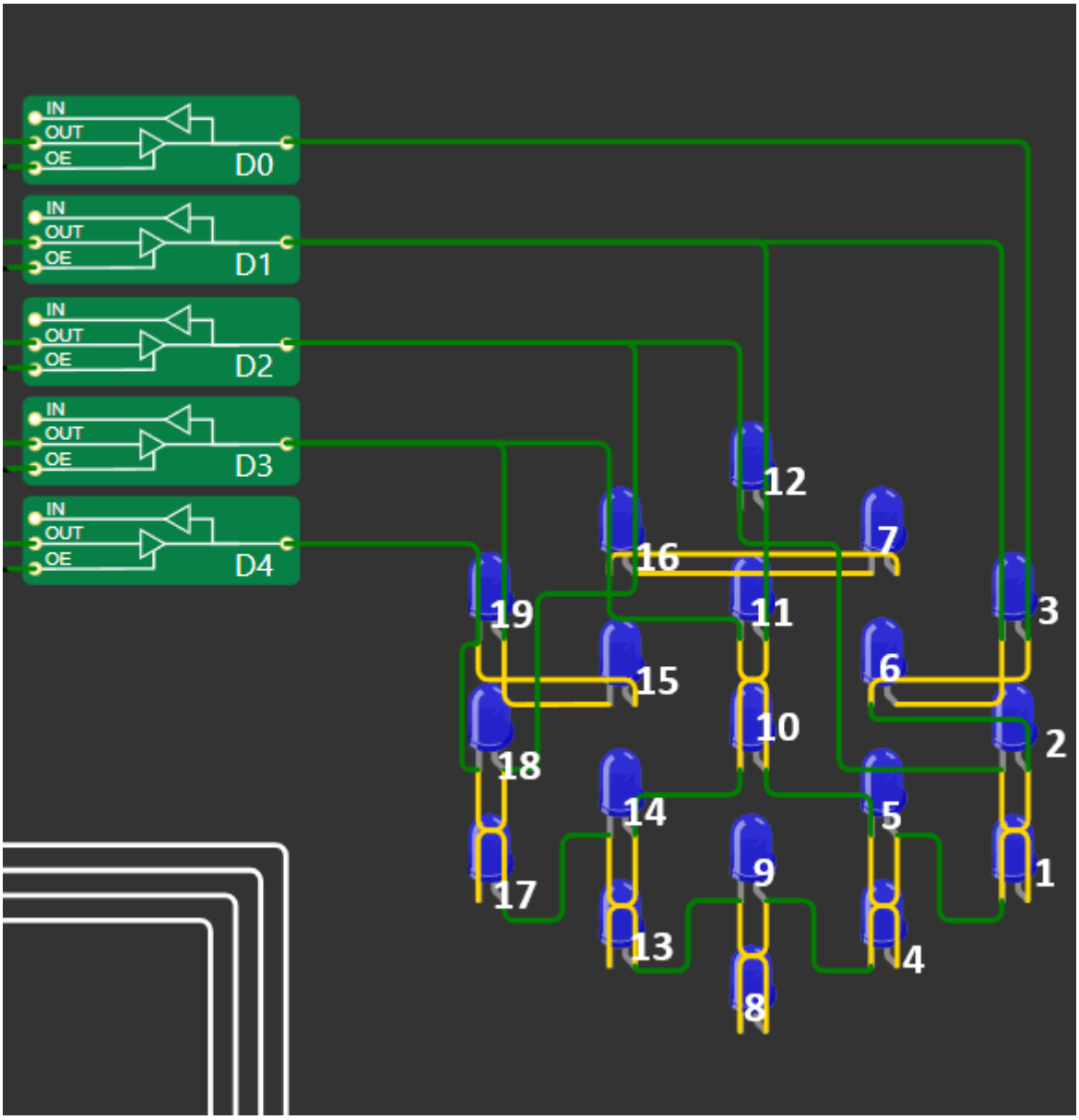


Figure 36: Schematic with LED numbering

Pinout

#	Input	Output	Bidirectional
0	Data		Display pin 0
1	Display Enable		Display pin 1
2			Display pin 2
3			Display pin 3
4			Display pin 4
5			
6			
7			

My Chip [293]

- Author: Ani Hakobyan
- Description: Design of my chip
- GitHub repository
- Wokwi project
- Mux address: 293
- Extra docs
- Clock: 0 Hz

How it works

It is a microchip that lights up an LED digital number and an LED light.

How to test

Test the project by starting the simulation in Wokwi then flicking switches.

External hardware

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

Pinout

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

8-bit PRNG [294]

- Author: Jakub Duchniewicz
- Description: Pure Random Noise Generator using Linear Feedback Shift Register with 2 halves of the 16-bit internal states shifted in different directions and xor'ed
- GitHub repository
- HDL project
- Mux address: 294
- Extra docs
- Clock: 0 Hz

How it works

The project is based on 16-bit Linear Feedback Shift Register but with a small twist - at each clock cycle the LFSR combines it's output from 2 halves, upper half (bits 15 to 8) is rotated left and the lower (bits 7 to 0) are rotated right and XOR'ed at the end.

Inspired by this StackOverflow post.

How to test

You can experiment with different initialization seeds and see how it changes the generated sequence - all 0 initialization does not work, the PRNG always returns 0s from such seed. The proposed usage of this project is as a noise generator that could be fed to e.g. musical synthesizer or be used as a non-cryptographic randomness generator.

Pinout

#	Input	Output	Bidirectional
0	Bit 0 initial PRNG seed	Bit 0 output noise	
1	Bit 1 initial PRNG seed	Bit 1 output noise	
2	Bit 2 initial PRNG seed	Bit 2 output noise	
3	Bit 3 initial PRNG seed	Bit 3 output noise	
4	Bit 4 initial PRNG seed	Bit 4 output noise	
5	Bit 5 initial PRNG seed	Bit 5 output noise	
6	Bit 6 initial PRNG seed	Bit 6 output noise	
7	Bit 7 initial PRNG seed	Bit 7 output noise	

Tiny Shader [295]

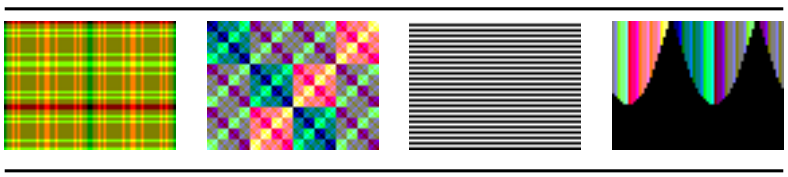
- Author: Leo Moser
- Description: With Tiny Shader you can write a small program to create different images and even animations.
- GitHub repository
- HDL project
- Mux address: 295
- Extra docs
- Clock: 25175000 Hz

How it works

Modern GPUs use fragment shaders to determine the final color for each pixel. Thousands of shading units run in parallel to speed up this process and ensure that a high FPS ratio can be achieved.

Tiny Shader mimics such a shading unit and executes a shader with 10 instructions for each pixel. No framebuffer is used, the color values are generated on the fly. Tiny Shader also offers an SPI interface via which a new shader can be loaded. The final result can be viewed via the VGA output at 640x480 @ 60 Hz, although at an internal resolution of 64x48 pixel.

Examples These images and many more can be generated with Tiny Shader. Note, that shaders can even be animated by acessing the user or time register.



The shader for the last image is shown here:

```
# Shader to display a rainbow colored sine wave

# Clear R3
CLEAR R3

# Get the sine value for x and add the user value
GETX R0
GETUSER R1
```

```
ADD R0 R1

# Set default color to R0
SETRGB R0

# Get the sine value for R0
SINE R0
HALF R0

# Get y coord
GETY R1

# If the sine value is greater
# or equal y, set color to black
IFGE R1
SETRGB R3
```

Architecture Tiny Shader has four (mostly) general purpose registers, REG0 to REG3. REG0 is special in a way as it is the target or destination register for some instructions. All registers are 6 bit wide.

Input The shader has four sources to get input from:

- `X` - X position of the current pixel
- `Y` - Y position of the current pixel
- `TIME` - Increments at 7.5 Hz, before it overflow it counts down again.
- `USER` - Register that can be set via the SPI interface.

Output The goal of the shader is to determine the final output color:

- `RGB` - The output color for the current pixel. Channel R, G and B can be set individually. If not set, the color of the previous pixel is used.

Sine Look Up Table Tiny Shader contains a LUT with 16 6-bit sine values for a quarter of a sine wave. When accesing the LUT, the entries are automatically mirrored to form one half of a sine wave with a total of 32 6-bit values.

Instructions The following instructions are supported by Tiny Shader. A program consists of 10 instructions and is executed for each pixel individually. The actual resolution is therefore one tenth of the VGA resolution (64x48 pixel).

Output

Instruction	Operation	Description
SETRGB RA	RGB <= RA	Set the output color to the value of the specified register.
SETR RA	R <= RA[1:0]	Set the red channel of the output color to the lower two bits of the specified register.
SETG RA	G <= RA[1:0]	Set the green channel of the output color to the lower two bits of the specified register.
SETB RA	B <= RA[1:0]	Set the blue channel of the output color to the lower two bits of the specified register.

Input

Instruction	Operation	Description
GETX RA	RA <= X	Set the specified register to the x position of the current pixel.
GETY RA	RA <= Y	Set the specified register to the y position of the current pixel.
GETTIME RA	RA <= TIME	Set the specified register to the current time value, increases with each frame.
GETUSER RA	RA <= USER	Set the specified register to the user value, can be set via the SPI interface.

Branches

Instruction	Operation	Description
IFEQ RA	TAKE <= RA == R0	Execute the next instruction if RA equals R0.
IFNE RA	TAKE <= RA != R0	Execute the next instruction if RA does not equal R0.
IFGE RA	TAKE <= RA >= R0	Execute the next instruction if RA is greater then or equal R0.
IFLT RA	TAKE <= RA < R0	Execute the next instruction if RA is less than R0.

Arithmetic

Instruction	Operation	Description
DOUBLE RA	RA <= RA * 2	Double the value of RA.
HALF RA	RA <= RA / 2	Half the value of RA.
ADD RA RB	RA <= RA + RB	Add RA and RB, result written into RA.

Load

Instruction	Operation	Description
CLEAR RA	RA <= 0	Clear RA by writing 0.
LDI IMMEDIATE	RA <= IMMEDIATE	Load an immediate value into RA.

Special

Instruction	Operation	Description
SINE RA	RA <= SINE[R0[4:0]]	Get the sine value for R0 and write into RA. The sine value LUT has 32 entries.

Boolean

Instruction	Operation	Description
AND RA RB	RA <= RA & RB	Boolean AND of RA and RB, result written into RA.
OR RA RB	RA <= RA	RB
NOT RA RB	RA <= ~RB	Invert all bits of RB, result written into RA.
XOR RA RB	RA <= RA ^ RB	XOR of RA and RB, result written into RA.

Move

Instruction	Operation	Description
MOV RA RB	RA <= RB	Move value of RB into RA.

Shift

Instruction	Operation	Description
SHIFTL RA RB	RA <= RA « RB	Shift RA with RB to the left, result written into RA.
SHIFTR RA RB	RA <= RA » RB	Shift RA with RB to the right, result written into RA.

Pseudo

Instruction	Operation	Description
NOP	R0 <= R0 & R0	No operation.

How to test

First set the clock to 25.175 MHz and reset the design. For a simple test, simply connect a Tiny VGA to the output Pmod. A shader is loaded by default and an image should be displayed via VGA.

For advanced features, connect an SPI controller to the bidir pmod. If ui[0], the mode signal, is set to 0, you can write to the user register via SPI. Note that only the last 6 bit are used.

If the mode signal is 1, all bytes transmitted via SPI are shifted into the shader memory. This way you can load a new shader program. Have fun!

External hardware

- Tiny VGA or similar VGA Pmod
- Optional: SPI controller to write the user register and new shaders

Pinout

#	Input	Output	Bidirectional
0	mode	R[1]	CS
1	debug_i[0]	G[1]	MOSI
2	debug_i[1]	B[1]	MISO
3		vsync	SCK
4		R[0]	next_vertical
5		G[0]	next_frame
6		B[0]	debug_o[0]
7		hsync	debug_o[1]

Workshop_chip [296]

- Author: Inne Lemstra
- Description: My first chip design, to be honest I'm not sure how far I will get.
- GitHub repository
- Wokwi project
- Mux address: 296
- Extra docs
- Clock: 0 Hz

How it works

This is the example project in Wokwi, used in the Hackaday Europe workshop. It does not have any changes in it at this point.

How to test

Flip switches 0-7 to changes to switch on the segments on the display

External hardware

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

Pinout

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

PCKY´s Successive Approximation Game [297]

- Author: pcky
- Description: Try to retrieve a pseudo-random 8-bit number by successive approximation.
- GitHub repository
- HDL project
- Mux address: 297
- Extra docs
- Clock: 10000 Hz

How it works

In this little game the player guesses a 8-bit unsigned number by setting a binary number on digital input port 'ui_in'. The player can manually follow the 'successive approximation' algorithm like its implemented in SAR ADCs to find the value. A 7-segment LED connected to 'uo_out' tells the player if his provided value is above, below or matching the wanted number.

How to test

Put the 'reset' port low (press the reset button on the demo board) and hold it for about a second in order to generate a secret number. Then set the DIP switches of the demo board in order to input a 8-bit value to the 'ui-in' port. The 7-segment LED will give immediate feedback if the player's number is above, below or exactly machting the 'secret' number. The player can continuously adjust the DIP switches until the wanted number is found.

To play another game just press the reset button again.

External hardware

This game utilizes the DIP switch and the 7-segment LED of the demo board.

#	Input	Output	Bidirectional

Pinout

#	Input	Output	Bidirectional
0	Binary Number Input 0 (LSB)	7-segment-LED 0	
1	Binary Number Input 1	7-segment-LED 1	
2	Binary Number Input 2	7-segment-LED 2	
3	Binary Number Input 3	7-segment-LED 3	
4	Binary Number Input 4	7-segment-LED 4	
5	Binary Number Input 5	7-segment-LED 5	
6	Binary Number Input 6	7-segment-LED 6	
7	Binary Number Input 7 (MSB)	7-segment-LED 7	

Dice [298]

- Author: Mastro Gippo
- Description: Roll a dice
- GitHub repository
- Wokwi project
- Mux address: 298
- Extra docs
- Clock: 10000 Hz

How it works

Pull IN0 high, dice will roll. Release and it will stop.

How to test

Please don't

External hardware

7seg display mapped a-g to OUT0-6, a button to VCC on IN0

Pinout

#	Input	Output	Bidirectional
0	btn_r	da	
1		db	
2		dc	
3		dd	
4		de	
5		df	
6		dg	
7			

Display test 1 [299]

- Author: Mastro Gippo
- Description: just a test
- GitHub repository
- HDL project
- Mux address: 299
- Extra docs
- Clock: 10000 Hz

How it works

I just copied Uri's project and messed around with its verilog

How to test

Connect 0-7 outputd to LCD pins: D4 D5 D6 D7 E

External hardware

Character LCD

Pinout

#	Input	Output	Bidirectional
0	s1	D4	
1	s2	D5	
2	e1	D6	
3	e2	D7	
4	e3	RS	
5	e4	E	
6	e5		
7	e6		

First TT Project [300]

- Author: t4m
- Description: First TinyTakeout Project at Hackaday
- GitHub repository
- Wokwi project
- Mux address: 300
- Extra docs
- Clock: 0 Hz

How it works

The default project just displays three segments lighting up.

How to test

You can use the project as a template to build more complicated 7 segment project.

External hardware

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

Pinout

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

Tiny_Tapeout_6_Frank [301]

- Author: Frank Hellmann
- Description: 7Seg Around the Clock / FH display
- GitHub repository
- Wokwi project
- Mux address: 301
- Extra docs
- Clock: 0 Hz

How it works

This project was created during the Hackaday Conference 2024 and shows two simple animations on the 7seg display. It takes the clock input (10kHz) and divides it down depending on the input 7. The mode input 0 either shows the letters F and H or is switched to the rotation animation.

Inputs

SW1 - IN0 = Mode (FH or Rotation Animation)

SW2 - IN1 = Blinking (turns dot on/off)

SW3 - IN2 = unused

SW4 - IN3 = unused

SW5 - IN4 = unused

SW6 - IN5 = Pause (if switched on, animation will freeze)

SW7 - IN6 = Debug (if switched on, divider will stop)

SW8 - IN7 = Divider (fast an slow)

How to test

Apply clock (10khz) and watch the 7seg display

If all inputs are off the 7seg display will show alternating letters F and H

If switch SW1 is on the 7seg display will show the rotation animation

Toggle SW2 to disable the dot blinking

Toggle SW6 to pause

Toggle SW8 to change speed

SW7 is for debugging divider

External hardware

The 7Seg LED display is used on outputs

Pinout

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

Hack a day Tiny Tapeout project [302]

- Author: Manuel
- Description: Circuit designed during the Hack a day workshop
- GitHub repository
- Wokwi project
- Mux address: 302
- Extra docs
- Clock: 0 Hz

How it works

TODO

How to test

TODO

External hardware

TODO

Pinout

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

Simple NCO [303]

- Author: s1Pu11i
- Description: Simple NCO which can also output sine, square or sawtooth.
- GitHub repository
- HDL project
- Mux address: 303
- Extra docs
- Clock: 0 Hz

How it works

Simple NCO to generate a sine, square or sawtooth output. Sine wave is generated from a table and square and sawtooth from the phase accumulator. Note: Output is unsigned 8bit.

How to test

Modeselection is done with the uio[1:0]: 0: NONE, output is 0 1: SINE 2: SQUARE 3: SAWTOOTH Frequency word is 16bit and is given as split into upper and lower part. Lower part is given with by uio[2]=‘1’ and ui[7:0]=word and the upper part by uio[3]=‘1’ and ui[7:0]=word.

External hardware

None.

Pinout

#	Input	Output	Bidirectional
0			ModeSelectionBit0
1			ModeSelectionBit1
2			FrequencyWordLower8BitUpdateEnable
3			FrequencyWordUpper8BitUpdateEnable
4			
5			
6			
7			

SiliconJackets_Systolic_Array [324]

- Author: SiliconJackets
- Description: a tiny systolic array capable of row stationary operation
- GitHub repository
- HDL project
- Mux address: 324
- Extra docs
- Clock: 25000000 Hz

How it works

This is a systolic array capable of matrix multiplication and 2D convolution using 9 processing elements

How to test

this project needs to be connected to an external FPGA to feed in the data to compute on

External hardware

FPGA connected to all 24 IO

Pinout

#	Input	Output	Bidirectional
0	readA[0]	write[0]	readB[0]
1	readA[1]	write[1]	readB[1]
2	readA[2]	write[2]	readB[2]
3	readA[3]	write[3]	readB[3]
4	readA[4]	write[4]	readB[4]
5	readA[5]	write[5]	readB[5]
6	readA[6]	write[6]	readB[6]
7	readA[7]	write[7]	readB[7]

ChatGPT designed Recurrent Spiking Neural Network [330]

- Author: Paola Vitolo, Michael Tomlinson, ChatGPT-4, Gian Domenico Licciardo, Andreas Andreou - pvitolo1@jh.edu
- Description: Programmable recurrent spiking neural network with 9 recurrent LIF neurons (3 input - 3 output) with fully programmable weights (8-bit)
- GitHub repository
- HDL project
- Mux address: 330
- Extra docs
- Clock: 0 Hz

How it works

This project implements 9 programmable digital recurrent LIF neurons. The neurons are arranged in 3 layers (3 in each). Spikes_in directly maps to the inputs of the first layer neurons. When an input spike is received, it is first multiplied by an 8 bit weight, programmable from a custom interface, 1 per input neuron. This 8 bit value is then added to the membrane potential of the respective neuron. When the first layer neurons activate, its pulse is routed to each of the 3 neurons in the next layer. There are 9x3 programmable weights describing the connectivity between the input spikes and the first layer (9 weights=3x3), the first and second layers (9 weights=3x3), and second and third layers (9 weights=3x3). Output spikes from the 3nd layer drive spikes_out.

How to test

After reset, program the neuron threshold, leak rate, feedback_scale and refractory period. Additionally program the first, 2nd, 3rd layer weights. Once programmed activate spikes_in to represent input data, track spikes_out synchronously (1 clock cycle pulses).

External hardware

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

Memory

The memory block stores 39 words of 8 bits. The first 12 words represent Feedback scale, Refractory Period, Decay rate, Membrane Threshold of each layer (4 parameters x 3 layers). The last 27 words represent the network weights (3 ins x 3 neurons x 3 layers).

Address	Address	Data Index	Data Index	Data Description	Data Description
dec	hex	MSB	LSB	Description	LYR #
0	0	7	0	Feedback scale	1
1	1	15	8	Refractory Period	1
2	2	23	16	Decay rate	1
3	3	31	24	Membrane Threshold	1
4	4	39	32	Feedback scale	2
5	5	47	40	Refractory Period	2
6	6	55	48	Decay rate	2
7	7	63	56	Membrane Threshold	2
8	8	71	64	Feedback scale	3
9	9	79	72	Refractory Period	3
10	A	87	80	Decay rate	3
11	B	95	88	Membrane Threshold	3
12	C	103	96	weight1_0	3
13	D	111	104	weight1_1	3
14	E	119	112	weight1_2	3
15	F	127	120	weight2_0	3
16	10	135	128	weight2_1	3
17	11	143	136	weight2_2	3
18	12	151	144	weight3_0	3
19	13	159	152	weight3_1	3
20	14	167	160	weight3_2	3
21	15	175	168	weight1_0	2
22	16	183	176	weight1_1	2
23	17	191	184	weight1_2	2
24	18	199	192	weight2_0	2
25	19	207	200	weight2_1	2
26	1A	215	208	weight2_2	2
27	1B	223	216	weight3_0	2
28	1C	231	224	weight3_1	2
29	1D	239	232	weight3_2	2
30	1E	247	240	weight1_0	1
31	1F	255	248	weight1_1	1

Address	Address	Data Index	Data Index	Data Description	Data Description
32	20	263	256	weight1_2	1
33	21	271	264	weight2_0	1
34	22	279	272	weight2_1	1
35	23	287	280	weight2_2	1
36	24	295	288	weight3_0	1
37	25	303	296	weight3_1	1
38	26	311	304	weight3_2	1

ChatGPT Transcripts:

- TT6 - LIF Neuron
- TT6 - LIF Neuron: overflow managing
- TT6- LIF Neuron Test Bench
- TT6 - RLIF Neuron
- TT6 - RLIF Neuron Test bench
- TT6-RLIF Layer
- TT6-RLIF Layer overflow managing
- TT6-RLIF Layer Test Bench
- TT-6 RSNN
- TT-6 RSNN Test Bench
- TT-6 FIPO Memory
- TT-6 FIFO Memory Test bench
- TT-6 RegN
- TT-6 Control Memory
- TT-6 Control Memory Test bench
- TT-6 Top Module
- TT-6 Top Module Test Bench

Pinout

#	Input	Output	Bidirectional
0	Input Spike 0	Output Spike 0	out_test 0
1	Input Spike 1	Output Spike 1	out_test 1
2	Input Spike 2	Output Spike 2	out_test 2

#	Input	Output	Bidirectional
3	Spike Input Register Enable	End of Writing Parameters into Memory	out_test 3
4	RSNN enable	Parameter Data Written	out_test 4
5	Serial Data IN		out_test 5
6	Parameter Load		out_test 6
7	Test selection		out_test 7

Izhikevich Neuron [334]

- Author: ExAI Dmitri Lyalikov
- Description: ASIC Digital Implementation of Izhikevich Neuron Model with configurable A, B Parameters
- GitHub repository
- HDL project
- Mux address: 334
- Extra docs
- Clock: 50000000 Hz

How it works

This is a simple Izhikevich model of a neuron. The Izhikevich model is a spiking neuron that is able to replicate the behavior of many different types of neurons. The model is described by the following equations: v' = 0.04v^2 + 5v + 140 - u + I u' = a(bv - u) if v >= 30 then v = c, u = u + d

The model has four parameters: a, b, c, and d. These parameters can be adjusted to replicate the behavior of different types of neurons. The model also has an input current I that can be used to stimulate the neuron. The model is implemented in the `izhikevich_neuron` module.

The `izhikevich_neuron` module has the following ports:

- `clk`: The clock signal
- `reset`: The reset signal
- `uo_out`: The output voltage of the neuron
- `ui_in`: Configuration input for the neuron
- `uio_in`: Configuration input for the neuron

The followiing parameters and IO are exposed through these module pins: | Name | Bits | Direction | Pins | Description | | —————–|——| ———-| ————–|———————————–| | `Input Current` | 5 | Input | ui[0-4] | Input current (mA) | | `Neuron Mode` | 3 | Input | uio[0-2] | See Table Below | | `A Param` | 4 | Input | ui[5-7], uio[3] | 4-bit custom A-parameter | | `B Param` | 4 | Input | uio[4:7] | 4-bit custom B-parameter | | `Membrane Potential` | 8 | Output | uo[0:7] | Signed 8-bit voltage (mV) |

Neuron Mode	Behavior	A	B	C	D
0	RS (Regular Spiking)	.02	.02	-65	8
1	IB (Intrinsically Bursting)	.02	.02	-55	4

Neuron Mode	Behavior	A	B	C	D
2	CH (Chattering)	.02	.02	-50	2
3	FS (Fast Spiking)	0.1	0.2	-65	2
4	TC (Thalamo-Cortical)	.02	0.25	-65	.05
5	RZ (Resonator)	0.1	0.25	-65	2
6	LTS (Low Threshold Spiking)	.02	0.25	-65	2
7	Custom	A	B	MRU	MRU

MRU: Most Recently Used Value. For example in custom mode, if the user was previously in RS mode, the C and D values will be set to -65 and 8 respectively, but A-B will be set to the custom values.

Default state: RS mode

How to test

The cocotb test bench provides a sweep across all neuron modes and a sweep across all A and B parameters. The test bench also provides a sweep across all input currents. The test bench checks that the output voltage of the neuron is within the expected range for each configuration. This can be used to plot the output voltage of the neuron for different configurations.

External hardware

This module requires a driver to interface with the neuron. The driver should be able to set the input current and the neuron mode, and read the output voltage. The driver should also be able to reset the neuron and provide a clock signal.

Pinout

#	Input	Output	Bidirectional
0	Input Current [0]	Membrane Potential [0]	Neuron Select [0]
1	Input Current [1]	Membrane Potential [1]	Neuron Select [1]
2	Input Current [2]	Membrane Potential [2]	Neuron Select [2]
3	Input Current [3]	Membrane Potential [3]	A Parameter [3]
4	Input Current [4]	Membrane Potential [4]	B Parameter [0]
5	A Parameter [0]	Membrane Potential [5]	B Parameter [1]
6	A Parameter [1]	Membrane Potential [6]	B Parameter [3]

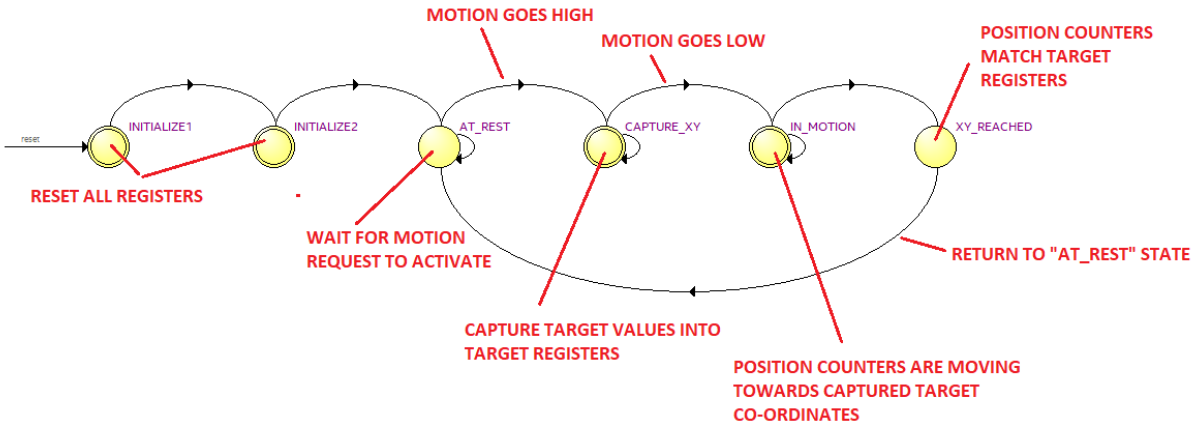
#	Input	Output	Bidirectional
7	A Parameter [2]	Membrane Potential [7]	B Parameter [4]

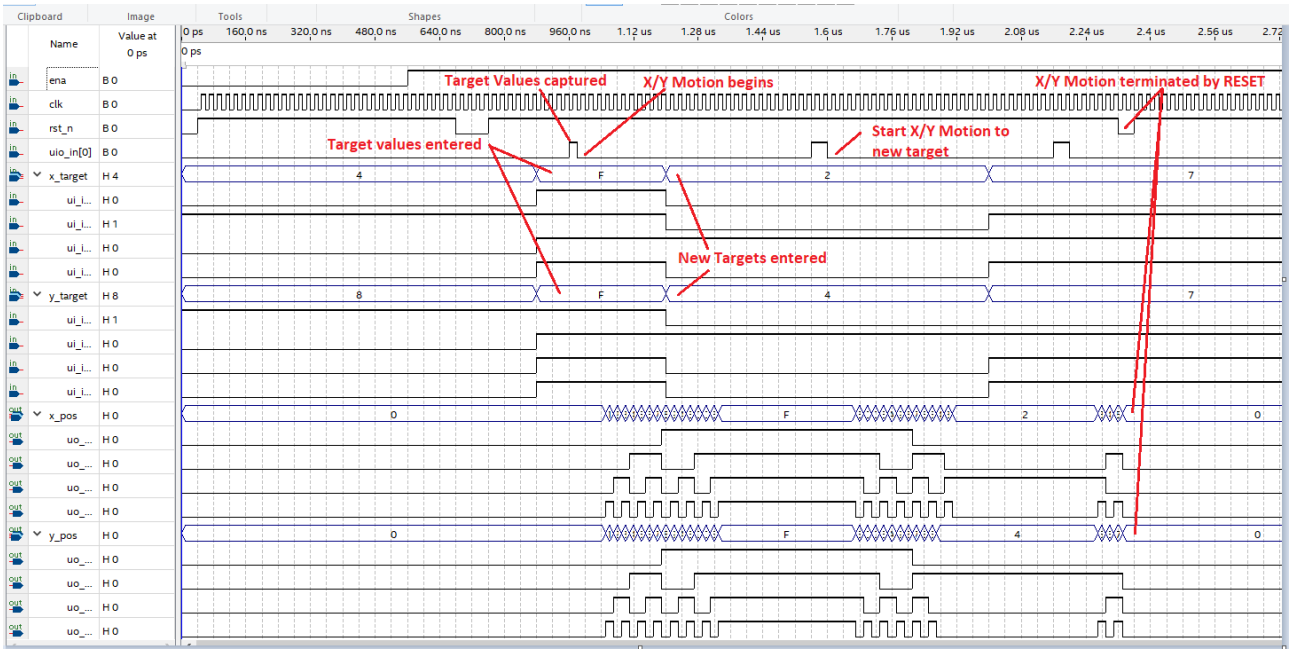
X/Y Controller [416]

- Author: Charles Pope
- Description: Two-Axis position Controller (4 bits of range per axis)
- GitHub repository
- HDL project
- Mux address: 416
- Extra docs
- Clock: 50000000 Hz

How it works

This design is a simple two-dimensional axis controller. It uses 4-bit wide counter, 4-bit Magnitude Comparator and a 4-bit wide target co-ordinate register for each dimension. A simple State Machine is used to acquire the target value for each dimension and then control the movement to the target co-ordinate value for each dimension in parallel operation. Comparisons of the current position and the target value for each dimension along the way and when the Traget is reached, the controller sits in a "AT-REST" state and waits for another motion directive. The Target values can change at any time after the Motion begins. New Target values will not be captured until the NEXT MOTION request. The controller can be RESET at any time with the rst_n input.





Inputs: Target X co-ordinate value (4 bits) on ui_in[7:4] Target Y co-ordinate value (4 bits) on ui_in[3:0] Motion Input (1 bit) on uio_in[0]; This signal is input after the target values are set. The movement begins after the signal is deactivated.

TT Infrastructure Signals: ena : to enable the design operation rst_n : to reset the design clk : to operate the pipelines of the design (up to 50 MHz). uio_oe[7:0] : are all configure for Input mode only uio_out[7:0]: are not used

Outputs: Current X position value (4 bits) on uo_out[7:4] Current Y position value (4 bits) on uo_out[3:0]

How to test

Set the X/Y Target values on ui_in[7:4] for X and ui_in3:0] for Y. Press the Motion button and release. The Controller will advance the X and Y potions towards the target values by one increment for each clock. If one dimension's target value is reached before the other, the controller will hold the current position for that one while the other one continues to its destination. You may update the Target values at anytime after the motion has started. The controller will move towards the NEW target values onlyif the Motion button is pressed again.

External hardware

Connect input switches to the ui_in[7:0] pins for the target X/Y co-ordiate inputs. Connect a push-button to the uio_in[0] pin for the Motion button. Each input should have a resistor pull-down of about 10 KOhms to GND of the TT06 chip. Each switch

or push-button must connect the TT06Chip VDD to the input when closed. Connect either low-current (5-10 ma) LEDS or a LED Driver device a Logic Analyzer to the uo_out[7:0] pins for X and Y position values (4-bit binary for each axis) to be displayed.

Pinout

#	Input	Output	Bidirectional
0	y_target0	y_pos0	motion_inp
1	y_target1	y_pos1	
2	y_target2	y_pos2	
3	y_target3	y_pos3	
4	x_target0	x_pos0	
5	x_target1	x_pos1	
6	x_target2	x_pos2	
7	x_target3	x_pos3	

Digital Temperature Monitor [417]

- Author: Priyansu Sahoo and Saroj Rout
- Description: This projects reads 8-bit temperature data using SPI from a LM70 sensor.
- GitHub repository
- HDL project
- Mux address: 417
- Extra docs
- Clock: 10000 Hz

Objective

This is an educational project for undergraduate engineering students with the objective of exposing them to real-world product design. In the process, the students learn a wide variety of engineering principles including product design, digital system design, mixed-signal modeling, digital design using Verilog, design verification, ASIC design flow, FPGA design flow, and documentation using gitHub.

Project Brief

This project implements a digital temperature monitor by connecting a temperature sensor (LM70 [`docs/datasheet-LM70-TI-tempSensor.pdf`]) and a three-segment display to measure and display a range of $0 - 99^{\circ}C$ or $0 - 99^{\circ}F$ with an accuracy of $\pm 2^{\circ}C$.

How it Works

Mode of Operation

MODE	`ui_in[0]`	`ui_in[1]`	`ui_in[2]`	DESCRIPTION
1	0	X	0	External Display in deg-C
2	0	X	1	External Display in deg-F
3	1	0	X	MSB Onboard Display in deg-C
4	1	1	X	LSB Onboard Display in deg-C

Figure 1 shows the general block diagram of the complete system. The temperature sensor (Texas Instrument LM70) has a dynamic range of 11 bits with a resolution of $\pm 0.25^{\circ}C$. In this project, we will only use MSB 8 bits with a resolution of $\pm 2^{\circ}C$.

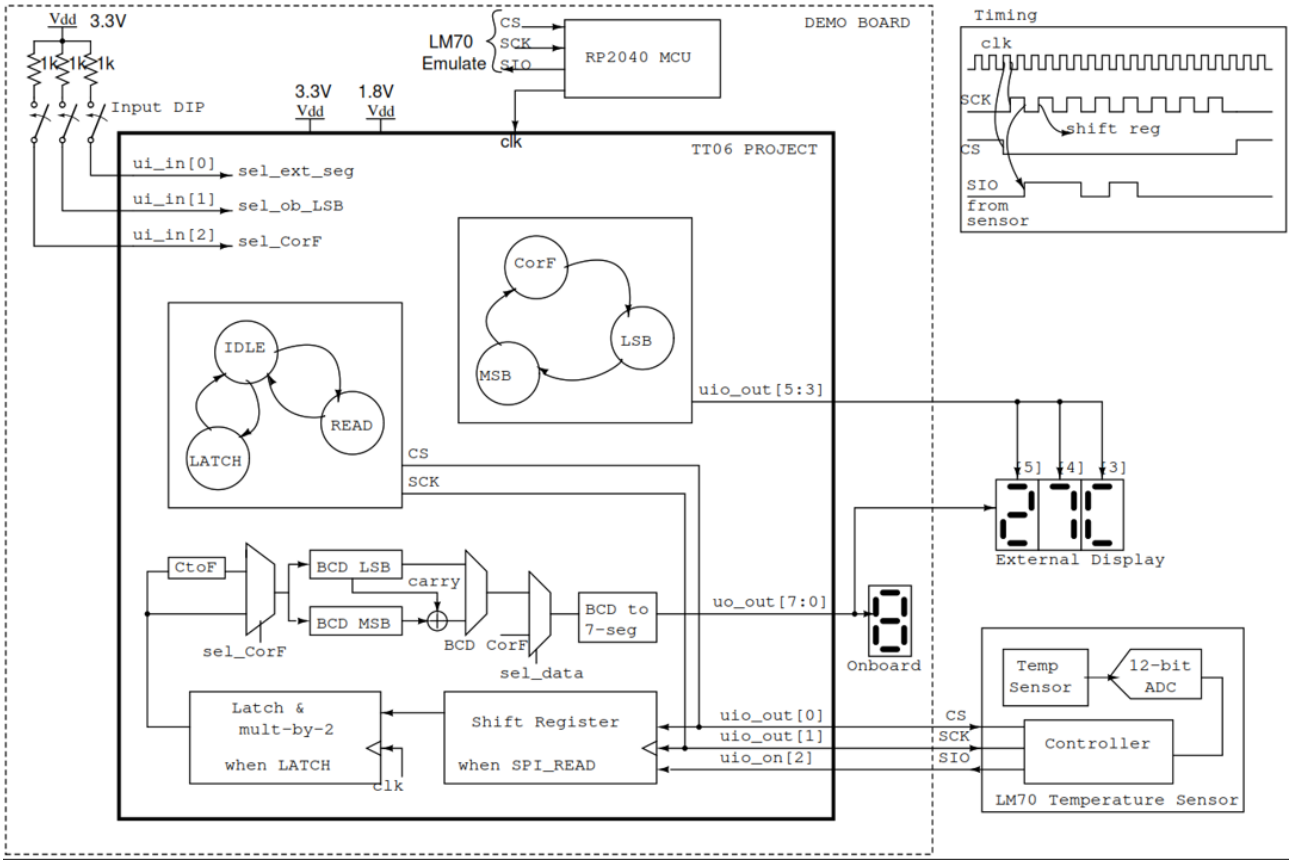


Figure 37: Block diagram of the complete system.

As shown in the timing diagram in the top right corner of the Figure 1, the LM 70 is configured as an SPI *peripheral*, with communication initiated by choosing the chip (CS) low. While CS is low, the data is clocked out of the sensor every *negative edge* of the SPI clock (SCK) and the design reads those data at the following *positive edge*. The design provides eight SCK clock pulses, and then the CS is pulled high to stop the communication.

The serial 8-bit data are captured in a shift register, and the data is *latched* after 8 SCK clock pulses. Before the data are latched, it is multiplied by 2 (left shift by 1). This multiplication captures the fact that the LSB of the data is $2^\circ\ C$.

he exact equation to convert temperature in centigrade to Fahrenheit is $T_F = T_C 9/5 + 32$. To keep the hardware simple, the implemented equation is approximated to $T_F = T_C 2 + 32$. By approximating $9/5$ by 2, the hardware is simply a left shift by 1. But this approximation results in an error in the output that is a function of temperature: 0.62 error at $0^\circ C$ and 9.43 error at $100^\circ C$. Based on the input `ui_in[2]`, a MUX selects the temperature in Celsius or Fahrenheit.

The data are then converted to binary coded decimal (BCD) decimal for the two temperature digits to be displayed. The BCD data are then converted to 8-bit 7-segment display format to drive an external display. To save output pins, the 7-segment

for all three displays are connected to the ports `uo_out[7:0]` and the displays are *time multiplexed* using the select lines `uio_out[5:3]`. If the displays are switched fast enough (but not too fast), all three displays appear steady without any appearance of flicker.

Since the demo PCB board has one 7-segment display, a provision in the design is made for test purpose where the temperature can be displayed on the onboard display, LSB and MSB one at a time. Kindly refer to the aforementioned 'Mode of Operation' table for further elucidation.

How to test

This project is designed with testability in mind so it can be tested with barebone PCB without any external hardware. The table below suggests different test modes for testing the design without any external hardware.

TestNo.	Mode	uio_in[2]	Ext. H/W	RP2040	7-seg Ouput
1	3	0	None	clk~10kHz	0
2	4	0	None	clk~10kHz	0
3	3	SIO from RP2040	None	clk~10kHz and SIO	MSB of data sent by RI
4	4	SIO from RP2040	None	clk~10kHz and SIO	LSB of data sent by RP

For the first two tests, the `uio_in[2]` port is grounded and a clock frequency of approximately 10 kHz is provided to the design from the RP2040 as shown in Figure 1. And when the inputs (`ui_in[2:0]`) are configure in Mode 3 or 4, the single 7-segment display should display `0` in both modes.

Test 3 and 4 in the table above will use the RP2040 as a SPI peripheral and micro-python code will be written to emulate the temperature sensor LM70. This will allow us to test the entire design without connecting the external temperature sensor or display.

External hardware

Needs a LM07 interfaced on the PCB. Detail hardware plan will be updated when we get close to receiveing the PCB.

Pinout

#	Input	Output	Bidirectional
0	Ext/Int	7seg-A	CS (O)
1	Int-LSB	7seg-B	SCK (O)
2	Ext-CorF	7seg-C	SIO (I)
3		7seg-D	7seg-sel0 (O)
4		7seg-E	7seg-sel1 (O)
5		7seg-F	7seg-sel2 (O)
6		7seg-G	
7		7seg-DP	

32-Bit Galois Linear Feedback Shift Register [418]

- Author: icaris lab
- Description: 32-bit Galois linear feedback shift register with taps at (32, 30, 26, 25).
- GitHub repository
- Wokwi project
- Mux address: 418
- Extra docs
- Clock: 50000000 Hz

How it works

The project is a hardware implementation of a maximum-cycle 32-bit Galois linear feedback shift register (LFSR) with taps at registers (R32, R30, R26, R25). The LFSR is defined with the most-significant bit (MSB) at the left-most register R32 and the least-significant bit (LSB) at the right-most register R01. The LFSR shifts bits from left to right (R_n+1 -> R_n), with the R30, R26, and R25 populated by XORing bits from R_n+1 with R1, the LFSR output. The LFSR contains an initialization/fail-safe feedback that prevents the LFSR from entering an all-zero state. If the LFSR is ever in an all-zero state, a "1" value is inserted into R32.

A schematic of the circuit may be found at:

https://wokwi.com/projects/394707429798790145

The circuit has 10 inputs:

Input	Setting
CLK	Clock
RST_N	Not Used
01	Not Used
02	Manual R0 Input Value
03	Input Select
04	Not Used
05	Not Used
06	Not Used
07	Not Used
08	Not Used

The CLK sets the clocking for the flip-flop registers for latching the LFSR values. In the schematic shown in the Wokwi project, a switch is used to select either the system

clock or an externally provided or manual clock that allows the user to manually step through each latching event.

An 8-input DIP switch provides some flexibility to initalizing the LFSR. DIP03 (IN2) allows the user to toggle the Input Select function, which is a multiplexer that select whether the left-most register (R32) takes in as the input the LFSR feedback from R01, or a value that is manually selected by the user. If manual input is selected, the taps on R30, R26, and R25 are turned off and their inputs are shifted in from R_n+1.

DIP02 (IN1) allows a the user to manually enter a 0 or a 1 value into the leftmost register.

The cicuit has 8 outputs. They output the values of the 8 right-most registers (R08, R07, R06, R05, R04, R03, R01, R01).

Output	Value in
01	R08
02	R07
03	R06
04	R05
05	R04
06	R03
07	R02
08	R01

How to test

The circuit can be tested by powering on the circuit, and first setting the Input Select switch (DIP03) to "1" to reset/initialize the entire LFSR to all-zeros. The Input Select switch can then be switched to "0" to allow the LFSR to run from its all-zero initialized value. The first 100 8-bit output values of the LFSR from this zeroized state may be observed using a logic analyzer, and should be:

[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0], [0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0], [0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0], [0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0], [0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0], [0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0], [0 0 0 0 0 0 0 0],[1 0 0 0 0 0 0 0],[0 1 0 0 0 0 0 0],[0 0 1 0 0 0 0 0], [0 0 0 1 0 0 0 0],[0 0 0 0 1 0 0 0],[0 0 0 0 0 1 0 0],[0 0 0 0 0 0 1 0], [0 0 0 0 0 0 0 1],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0], [0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0], [0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0], [0 0 0 0 0 0 0

0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0], [0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[1 0 0 0 0 0 0 0],[1 1 0 0 0 0 0 0], [0 1 1 0 0 0 0 0],[0 0 1 1 0 0 0 0],[0 0 0 1 1 0 0 0],[1 0 0 0 1 1 0 0], [0 1 0 0 0 1 1 0],[1 0 1 0 0 0 1 1],[0 1 0 1 0 0 0 1],[0 0 1 0 1 0 0 0], [0 0 0 1 0 1 0 0],[0 0 0 0 1 0 1 0],[0 0 0 0 0 1 0 1],[0 0 0 0 0 0 1 0], [0 0 0 0 0 0 0 1],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0], [0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0], [0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[1 0 0 0 0 0 0 0], [0 1 0 0 0 0 0 0],[1 0 1 0 0 0 0 0],[0 1 0 1 0 0 0 0],[0 0 1 0 1 0 0 0], [0 0 0 1 0 1 0 0],[0 0 0 0 1 0 1 0],[0 0 0 0 0 1 0 1],[0 0 0 0 0 0 1 0], [0 0 0 0 0 0 0 1],[1 0 0 0 0 0 0 0],[0 1 0 0 0 0 0 0],[0 0 1 0 0 0 0 0], [0 0 0 1 0 0 0 0],[1 0 0 0 1 0 0 0],[0 1 0 0 0 1 0 0],[0 0 1 0 0 0 1 0], [0 0 0 1 0 0 0 1],[0 0 0 0 1 0 0 0],[0 0 0 0 0 1 0 0],[0 0 0 0 0 0 1 0], [0 0 0 0 0 0 0 1],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 0], [1 0 0 0 0 0 0 0]

A python implementation of the 32-bit Galois LFSR can be found at the link below. It may be used for testing the hardware for sequences longer than the initial 100 values.

https://github.com/icarislab/tt06_32bit-fibonacci-prng_cu/main/docs/32-bit-fibonacci-prng_pythong_simulation.py

External hardware

No external hardware is required.

Pinout

#	Input	Output	Bidirectional
0		r08_val	
1	data_in	r07_val	
2	load_en	r06_val	
3		r05_val	
4		r04_val	
5		r03_val	
6		r02_val	
7		r01_val_LSFR_out	

DJ8 8-bit CPU [419]

- Author: DaveX
- Description: DJ8 8-bit CPU with parallel Flash / RAM interface
- GitHub repository
- HDL project
- Mux address: 419
- Extra docs
- Clock: 14000000 Hz

How it works

DJ8 is a 8-bit CPU implemented in VHDL, originally developped for XCS10XL featuring:

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

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

Other implementations:

- TT07 DJ8 8-bit CPU w/ DAC - Verilog, Mixed-signal, 8-bit DAC
- TTIHP0P2 DJ8 8-bit CPU - Verilog

Memory Map

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

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

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

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

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

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

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

ALU reg, imm8: Immediate ALU operation

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

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

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

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

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

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

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

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

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

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

- A : ALU operation

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

- D : dest register
- M: memory mode

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

- F : final shift operation

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

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

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

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

Jxx imm12: Conditional or unconditional jump to absolute address

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

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

JMP GH: Unconditional jump to address GH

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

Pinout Due to TT06 IO constraints, pins are shared between *Address bus LSB* and *Data bus OUT*. It means that during memory write instructions, the address space is only 128 bytes.

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

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

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

How to test

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

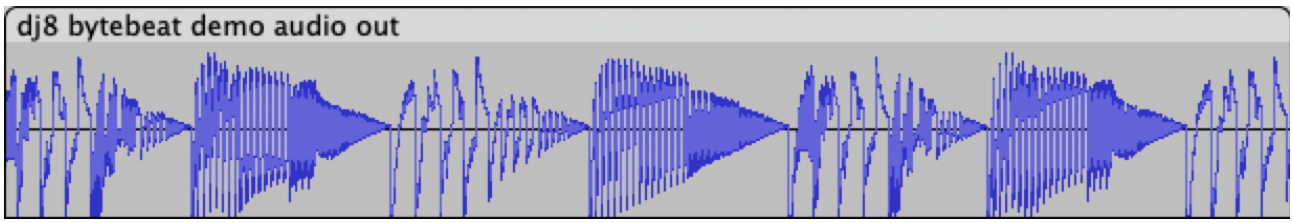
Demo 1: Rotating LED indicator

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

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

Demo 2: Bytebeat Synthetizer

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



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

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

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

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

External hardware

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

Pinout

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

Servo Signal Tester [420]

- Author: Holunder
- Description: If you provide a 4kHz Clock and 8 LED's as Output, the LED's will light up according to the Servo Signal on Input 0
- GitHub repository
- Wokwi project
- Mux address: 420
- Extra docs
- Clock: 4000 Hz

How it works

If you apply a Servo Signal it will be processed via Flip-Flops so that the Servo Signal controls the Pin's on the Output. If the Servo Pulse is 1ms (0 degree) no LED will light up and if the Servo Pulse is 2ms (180 degrees) all the LED's will light up.

How to test

Add 8 LED's to the Output's and connect a Servo Signal to Input 0

External hardware

8 LED's and resistors. (If your LED's need more Output Power then the Chip can provide, use a driver) Maybe the resistors can have a low value, because the LED's are only ON when the Servo Signal is HIGH. So the maximum is 20% duty cicle.

How to use

Add 8 LED's to the Output's and connect a Servo Signal to Input 0

Pinout

#	Input	Output	Bidirectional
0	Servo Signal	LED 0	
1		LED 1	
2		LED 2	

#	Input	Output	Bidirectional
3		LED 3	
4		LED 4	
5		LED 5	
6		LED 6	
7		LED 7	

Bivium-B Non-Linear Feedback Shift Register [421]

- Author: icaris lab
- Description: Bivium-B stream cipher used as a non-linear feedback shift register.
- GitHub repository
- Wokwi project
- Mux address: 421
- Extra docs
- Clock: 50000000 Hz

How it works

The project is a hardware implementation of the Bivium-B stream cipher used as a non-linear feedback shift register (NLFSR). The NLFSR is defined with the least-significant bit (LSB) at the left-most register R0 and the most-significant bit (MSB) at the right-most register R176. The LFSR circular shifts bits from left to right (R_n -> R_n+1), with the two feedback taps:

R93 = (R90 * R91) + (R65 + R92) + R170 R0 = (R174 * R175) + (R161 + R176) + R68

The output of the NLFSR is:

z = R65 + R92 + R161 + R176

The NLFSR contains an initialization/fail-safe feedback that prevents the LFSR from entering an all-zero state. If the LFSR is ever in an all-zero state, a "1" value is inserted into R0.

A schematic of the circuit may be found at:

https://wokwi.com/projects/395263962779770881

The circuit has 10 inputs:

Input	Setting
CLK	Clock
RST_N	Not Used
01	Not Used
02	Manual R0 Input Value
03	Input Select
04	Not Used
05	Not Used
06	Not Used

Input	Setting
07	Not Used
08	Not Used

The CLK sets the clocking for the flip-flop registers for latching the NLFSR values. In the schematic shown in the Wokwi project, a switch is used to select either the system clock or an externally provided or manual clock that allows the user to manually step through each latching event.

An 8-input DIP switch provides some flexibility to initalizing the NLFSR. DIP03 (IN2) allows the user to toggle the Input Select function, which is a multiplexer that select whether the left-most register (R0) takes in as the input the NLFSR feedback value or a value that is manually selected by the user. The switch also controls whether R93 takes in a NLFSR feedback or a value directly from R91.

DIP02 (IN1) allows a the user to manually enter a 0 or a 1 value into the leftmost register.

The cicuit has 8 outputs. They output the following values:

Output	Value in
01	R0 (NLFSR input)
02	R68
03	(R174 * R175) + (R161 + R176)
04	R65
05	R92
06	R161
07	R176
08	z (NLFSR output)

The output allows for some self-testing, where OUT01 = OUT02 + OUT03 and OUT08 = OUT004 + OUT05 + OUT06 + OUT07.

How to test

The circuit can be tested by powering on the circuit, and first setting the Input Select switch (DIP03) to "1" to reset/initialize the entire LFSR to all-zeros. The Input Select switch can then be switched to "0" to allow the LFSR to run from its all-zero initialized value. The output values of the NLFSR from this zeroized state may be observed using a logic analyzer, and can be compared with the values obtained for the python simulation:

https://github.com/icarislab/tt06_biviumb-prng_cu/blob/main/docs/biviumb-prng_python_simulation.py

External hardware

No external hardware is required.

Pinout

#	Input	Output	Bidirectional
0		r000_val	
1	data_in	r068_val	
2	load_en	INTERM_fb	
3		r065_val	
4		r092_val	
5		r161_val	
6		r176_val	
7		NLSFR_out	

Servotester [422]

- Author: Jonas Wuehr
- Description: Generate a 50 Hz test signal for RC servos.
- GitHub repository
- HDL project
- Mux address: 422
- Extra docs
- Clock: 10000000 Hz

How it works

It uses a 10 MHz clock to create a counter, which has it's carry signal every 50Hz. Then there is a second counter to create the 1 - 2 millisecond high signal according to the user input.

How to test

Connect a servo PWM signal to biderectional pin 7. According to the user input on the inputs it's position will change and be indicated on the 7-segment display.

External hardware

A RC servo is required for testing.

Pinout

#	Input	Output	Bidirectional
0	position bit 0		
1	position bit 1		
2	position bit 2		
3	position bit 3		
4	position bit 4		
5	position bit 5		
6	position bit 6		
7	position bit 7		servo pulse

Cyclic Redundancy Check 8 bit [423]

- Author: EconomIC Engineers
- Description: Error detecting circuit commonly used in digital networks and storage devices to detect accidental changes to digital data.
- GitHub repository
- Wokwi project
- Mux address: 423
- Extra docs
- Clock: 0 Hz

How it works

https://en.wikipedia.org/wiki/Cyclic_redundancy_check Explain how your project works

How to test

https://quickbirdstudios.com/blog/validate-data-with-crc/ https://thepiandi.blogspot.com/2014 redundancy-check-crc-of-ds18b20.html

Explain how to use your project

External hardware

Needed will be LEDs, a clock or button to step clock, a switch for reset and inputs. List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Pinout

#	Input	Output	Bidirectional
0	Data Stream input	8th bit	
1		7th bit	
2		6th bit	
3		5th bit	
4		4th bit	
5		3rd bit	
6		2nd bit	

#	Input	Output	Bidirectional
7	Set for all flip flops	1st bit	

DEFAULT [424]

- Author: Beau Ambur
- Description: Displays 'dEFAULt123' on 7-segment LED
- GitHub repository
- HDL project
- Mux address: 424
- Extra docs
- Clock: 10000000 Hz

How it works

You mash buttons until the build doesn't fail.

How to test

Plug it in and turn on. If there's now smoke call it a success

External hardware

N/A

Pinout

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

Anomaly Detection using Isolation trees [425]

- Author: Eleftherios Batzolis
- Description: Uses an isolation tree to check for anomalies during the operation of a device
- GitHub repository
- HDL project
- Mux address: 425
- Extra docs
- Clock: 10000000 Hz

How it works

This projects is implementing Isolation forest algorithm using ML methods.

The i_tree design encapsulates a Verilog implementation for detecting (infering) anomalies in sensor data, specifically tailored for integration into System on Chips (SoCs) with a focus on AI accelerators. This design comprises three primary modules: the InputBuffer, the IsolationTreeStateMachine, and the i_tree top module that orchestrates their interaction.

InputBuffer Module: This module collects incoming sensor data bit-by-bit until it accumulates a full byte. It utilizes a double-buffering mechanism to manage data efficiently, ensuring that data processing by downstream components does not block incoming sensor data collection. The buffer toggles between collecting new data and allowing the processed data to be consumed, controlled by internal logic that responds to the data processing status.

IsolationTreeStateMachine Module: Once a complete byte of data is ready, this state machine takes over. It processes the data to determine if an anomaly is present based on predefined criteria (currently, a simplistic check against a set byte pattern, but intended to be expanded to more complex algorithms). It operates in several states: IDLE, CHECK_ANOMALY, and PROCESS_DONE, transitioning between these states based on the presence of valid data and completing the processing cycle.

Top Module (i_tree): This module integrates the InputBuffer and IsolationTreeStateMachine, routing signals between them. It feeds sensor data into the InputBuffer, takes the processed output, and directs it into the IsolationTreeStateMachine. It also handles the overall reset and clock signals for synchronization and system stability.

Together, these modules form a robust system for real-time anomaly detection, designed with scalability and efficiency in mind, making it suitable for embedded applications where performance and space are critical constraints.

How to test

1st 8bit value are the data used for the anomaly detection.

External hardware

Binary output sensor used for anomaly detection on workload of devices.

Pinout

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

Inverters [426]

- Author: James Meech
- Description: A set of inverters
- GitHub repository
- HDL project
- Mux address: 426
- Extra docs
- Clock: 0 Hz

How it works

I made this design with the goal of seeing what different measurements I could perform on an inverter with an analog Tiny Tapeout tile: https://github.com/JamesTimothyMeech/tt06-programmable-thing compared to this digital one. This project contains a set of inverters of different sizes connected between the input and output pins. Input and output zero is a D Flip Flop. All other pins have an inverter connected between them. Please see the Wokwi circuit diagram: https://wokwi.com/projects/395134712676183041 or the info.yaml: https://github.com/JamesTimothyMeech/TT06/blob/main/info.yaml to see which pins are inverter inputs and which are inverter outputs.

How to test

Apply inputs to the inverters with a square wave or other signal generator and measure the output. Experiment by putting inverters in parallel and see if you can measure any differences in their speed. Try connecting a large capacitor to the input and a resistor between the input and the output to use the inverters as an amplifier: https://www.youtube.com/watch?v=03Ds1TnoMbA&ab_channel=MSMTUE does this work in the same way as the inverter on the analog tile? If not, why?

External hardware

TT06 printed circuit board, signal generator, an oscilliscope or similar to measure the input and output.

Pinout

#	Input	Output	Bidirectional
0	D Flip Flop Input	D Flip Flop output	Bidirectional inverter 1 input
1	Inverter 1 input	Inverter 1 output	Bidirectional inverter 1 output
2	Inverter 2 input	Inverter 2 output	Bidirectional inverter 2 input
3	Inverter 3 input	Inverter 3 output	Bidirectional inverter 2 output
4	Inverter 4 input	Inverter 4 output	Bidirectional inverter 3 input
5	Inverter 5 input	Inverter 5 output	Bidirectional inverter 3 output
6	Inverter 6 input, also the output enable for all bidirectional pins connected to inverter inputs	Inverter 6 output	Bidirectional inverter 4 input
7	Inverter 7 input, also the output enable for all bidirectional pins connected to inverter output	Inverter 7 output	Bidirectional inverter 4 output

Lipsi: Probably the Smallest Processor in the World [427]

- Author: Martin Schoeberl
- Description: A tiny 8-bit accumulator based microprocssor.
- GitHub repository
- HDL project
- Mux address: 427
- Extra docs
- Clock: 50000000 Hz

How it works

This is the Lipsi processor. It executes a hardcoded program counts up at a readable frequency. That number is displayed on the 7-segment display. Additionally, the DP blinks (in hhardware).

How to test

ChiselTest is used for waveform generation. Currently, we use cocotb, this shall change to ChiselTest. But that test is disabled

External hardware

non by default.

Pinout

#	Input	Output	Bidirectional
0	input for Lipsi, also switch of blinking LED	segment a	
1	input for Lipsi	segment b	
2	input for Lipsi	segment c	
3	input for Lipsi	segment d	
4	input for Lipsi	segment e	
5	input for Lipsi	segment f	
6	input for Lipsi	segment g	
7	input for Lipsi	dp (blinking)	

Chisel Hello World [428]

- Author: Martin Schoeberl
- Description: A Chisel Hello World with Counting on the 7-segment display and showing/playing Morse Code of hello world
- GitHub repository
- HDL project
- Mux address: 428
- Extra docs
- Clock: 50000000 Hz

How it works

This is a simple `Hello World` project for Chisel. It is a simple counter with 7-segment display output. And a Morse code generator writing out `hello world` in Morse code.

The project displays a counter on the 7-segment display. It also writes out `hello world` in Morse code on the DP of the 7-segment display. Furthermore, it also playes the Morse code with PWM on the BIDIR PMOD, connected to a PmodAMP2.

To better see the Morse code, the counter display can be disabled with switch 0.

How to Test

Currently, we use cocotb, this shall change to ChiselTest.

External Hardware

Audio PMOD (PmodAMP2) for audio output on the lower row of the BIDIR PMOD.

Pinout

#	Input	Output	Bidirectional
0	switch on 7-segment	segment a	
1		segment b	
2		segment c	
3		segment d	
4		segment e	audio

#	Input	Output	Bidirectional
5		segment f	gain
6		segment g	
7		dot: morse out	nshutdown

Signed Unsigned multiplyer [429]

- Author: Ole Henrik Moller
- Description: Do a 4x4 multiplication
- GitHub repository
- HDL project
- Mux address: 429
- Extra docs
- Clock: 0 Hz

How it works

The combinational multiplier takes as input a 4-bit multiplicand and a 4-bit multiplier and produces as output an 8-bit product. The numbers may all be either unsigned or signed integers as controlled by the signed_mode input. All 4+4+8+1 signals are active high.

The multiplier in unsigned mode uses an array of 4 x 4 cells that each consists of a full adder and an AND-gate (NAND-gate for a few cells in signed mode). The columns and rows of the array distribute the bits of the multiplicand and multiplier, respectively, to the two inputs of the AND-gate of each cell. The partial sums are fed from cell to cell diagonally (from upper left to lower right), while the carries are fed from cell to cell vertically. The sum that emanate at the right edge of the array constitute the lower half of the product, while the sum and carries at the lower edge of the array proceed to a ripple carry adder (with a carry-in of 0) that produces the upper part of the product.

For signed multiplication the multiplier above is modified by employing the Modified Baugh-Wooley multiplication algoritm, which avoids sign-extension of the multiplicand by flipping product bits in MSB positions of both operands (cancel out for cell that combine MSBs of both operands) with NAND-gates, and adding ones at the least and most significant bit positions of the final ripple-carry adder.

For a more detailed explanation of the Modified Baugh-Wooley algoritm see the book Computer Arithmetic, Algoritms and Hardware Designs by Behrooz Parhami, Oxford University Press, 2000, or the original article by C. R. Baugh and B. A. Wooley, A Two's Complement Parallel Array Multiplication Algoritm, IEEE Trans. Computers, Vol 22, pp. 1045-1047, December 1973.

How to test

The multiplier may be tested using a TinyTapeout demoboard with various combinations of multiplicand and multipliers using the input switches and checking the expected product with the 7-segment LED display (with decimal point).

External hardware

None beyound the TinyTapeout demoboard.

Pinout

#	Input	Output	Bidirectional
0	multiplier[0]	product[0]	signed_mode
1	multiplier[1]	product[1]	
2	multiplier[2]	product[2]	
3	multiplier[3]	product[3]	
4	multiplicand[0]	product[4]	
5	multiplicand[1]	product[5]	
6	multiplicand[2]	product[6]	
7	multiplicand[3]	product[7]	

EFAB Demo 2 [430]

- Author: Anton Maurovic
- Description: Displays 'EFAB' on 7seg display
- GitHub repository
- Wokwi project
- Mux address: 430
- Extra docs
- Clock: 2 Hz

How it works

Wokwi project uses a D flip-flop ring to display "EFAB" on the 7-segment display.

How to test

Works best with a slow clock (say, 2~10Hz) or manually stepping with the demo board's "step" button.

Turn off all input switches, then power on the board. Expect to see "F" blinking on the display with each clock pulse.

Switch on input switch 3, and "E" should start blinking.

Switch on input switch 1, and it should cycle through "EFAb".

Switch on input switch 8, and letter blinking should be disabled.

These are the inputs:

In	Signal	Function
SW1	`IN0`	**Run**: Off = Reset; On = Run
SW2	`IN1`	(Unused)
SW3	`IN2`	`state_init[0]`; Normally ON
SW4	`IN3`	`state_init[1]`; Normally off
SW5	`IN4`	`state_init[2]`; Normally off
SW6	`IN5`	`state_init[3]`; Normally off
SW7	`IN6`	(Unused)
SW8	`IN7`	**Blink control**: On = no blink

`state_init` specifies the initial state for the sequencing flip-flops during reset, and for normal operation the first (SW3) would be switched ON, and the other three (SW4..6) would be switched off.

External hardware

None, besides the TT demo board.

Pinout

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

Dual Deque [431]

- Author: Andrew Dona-Couch
- Description: Dual byte-width double-ended queues
- GitHub repository
- HDL project
- Mux address: 431
- Extra docs
- Clock: 0 Hz

Two independent double-ended queues in one tiny footprint.

How it works

Each deque is an array of flip flops with a pointer to the top. The empty and full status flags for each are directly available on pins. The push and pop inputs as well as data bus lines are multiplexed using the deque select line.

Hold `end_select` low to operate as a stack. Tie `end_select` to `push` to operate as a queue.

How to test

To push (if `full` is low):

- Put the data byte on `data_in`
- Select which deque to push to with `deque_select`
- Select which end to push to with `end_select`
- Bring `push` high for one cycle

To pop (if `empty` is low):

- Select which deque to push to with `deque_select`
- Select which end to push to with `end_select`
- Bring `pop` high for one cycle

To replace the last element of the deque (if `empty` is low):

- Select which deque to push to with `deque_select`
- Select which end to push to with `end_select`
- Put the new data byte on `data_in`
- Bring both `push` and `pop` high for one cycle

To read the end of the deque:

- Select which deque to push to with `deque_select`
- Select which end to push to with `end_select`
- Wait one cycle
- Read end of deque from `data_out`

External hardware

You would probably want to connect this to other devices that would find it useful.

Pinout

#	Input	Output	Bidirectional
0	Data In 0	Data Out 0	Deque Select
1	Data In 1	Data Out 1	End Select
2	Data In 2	Data Out 2	Push
3	Data In 3	Data Out 3	Pop
4	Data In 4	Data Out 4	Deque 0 Empty
5	Data In 5	Data Out 5	Deque 0 Full
6	Data In 6	Data Out 6	Deque 1 Empty
7	Data In 7	Data Out 7	Deque 1 Full

DFFRAM Example (128 bytes) [452]

- Author: Uri Shaked
- Description: 128 bytes DFFRAM module
- GitHub repository
- HDL project
- Mux address: 452
- Extra docs
- Clock: 0 Hz

How it works

It uses a 32x32 1RW DFFRAM macro to implement a 128 bytes (1 kilobit) RAM module.

Reseting the project **does not** reset the RAM contents.

How to test

Set the `addr` pins to the desired address, and set the `in` pins to the desired value. Then, set the `wen` pin to `1` to write the value to the RAM, or set it to `0` to read the value from the RAM, and pulse `clk`.

The `out` pins will contain the value read from the RAM.

Pinout

#	Input	Output	Bidirectional
0	addr[0]	out[0]	in[0]
1	addr[1]	out[1]	in[1]
2	addr[2]	out[2]	in[2]
3	addr[3]	out[3]	in[3]
4	addr[4]	out[4]	in[4]
5	addr[5]	out[5]	in[5]
6	addr[6]	out[6]	in[6]
7	wen	out[7]	in[7]

Retro Console [458]

- Author: Toivo Henningsson
- Description: 8½ bit retro console with sprite and tile graphics + synth
- GitHub repository
- HDL project
- Mux address: 458
- Extra docs
- Clock: 50350000 Hz

Overview

AnemoneGrafx-8 is a retro console containing

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

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

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

Features:

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

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

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

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

Contents:

Design rationale

The design target was

- PPU with 2 bpp graphics, with

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

Design considerations:

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

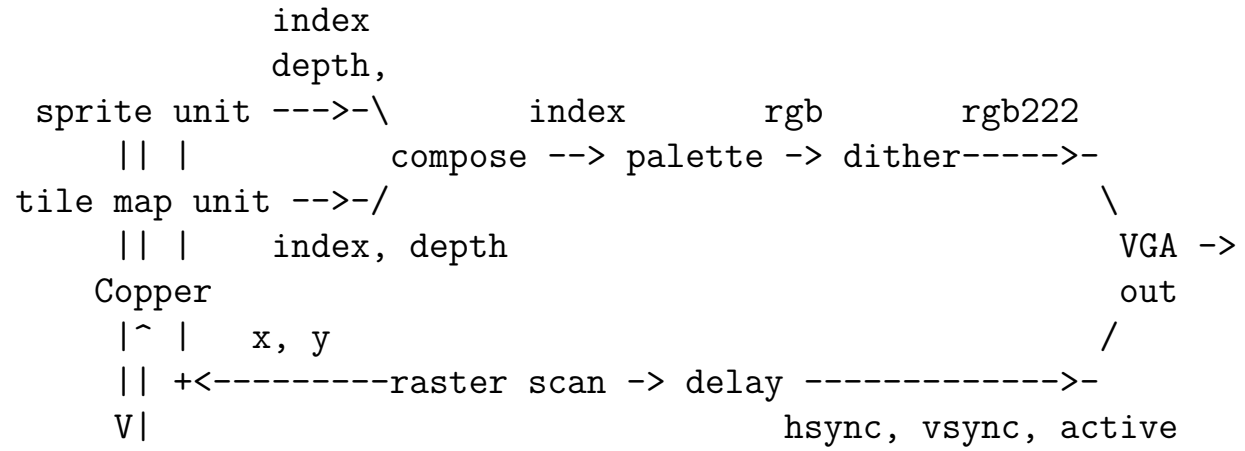
        * 16 bits/pixel read bandwidth gives space to read in new sprites during scan line
            · keeping track of only 4 sprites at a time to reduce on-chip storage
        * Overhead to keep track of each sprite means that it might as well use 32 bits of pixel data per scan line
        * Palette registers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

How it works

The console consists of two parts:

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

PPU



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

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

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

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

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

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

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

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

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

Sprite data is stored in memory in two structures:

- The sorted buffer
- The object attribute buffer

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

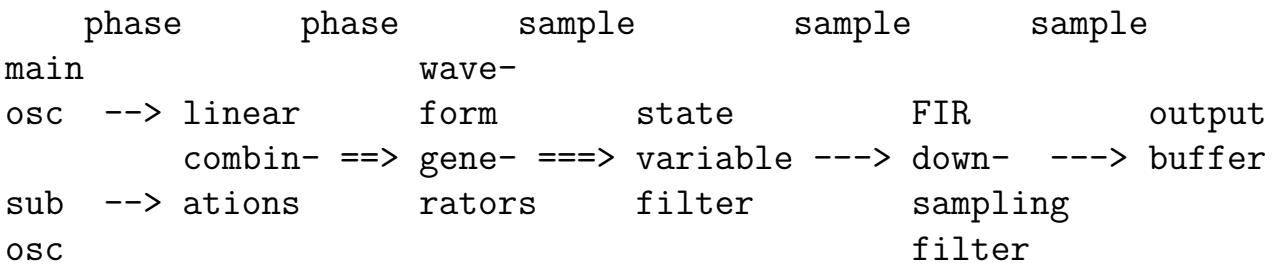
Sprite processing proceeds in three steps, each with its own buffers and head/tail pointers:

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

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

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

AnemoneSynth



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

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

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

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

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

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

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

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

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

IO interfaces

AnemoneGrafx-8 has four IO interfaces:

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

The pins also have additional functions:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- a message is initiated with one cycle of `tx_out[1:0] = 1` (low bit set, high bit clear),
- during the next cycle, `tx_out[1:0]` contains the 2 bit *TX header*, specifying the message type,
- during the following 8 cycles, a 16 bit payload is sent through `tx_out[1:0]`, from lowest bits to highest.

The `rx_in[1:0]` pins are used for messages to the console:

- a message is initiated with one cycle when `rx_in[1:0] != 0`, specifying the *RX header*, i e, the message type,
- during the following 8 cycles, a 16 bit payload is sent through `rx_in[1:0]`, from lowest bits to highest.

TX message types:

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

RX message types:

- 1: Context switch response with data.

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

Available registers:

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

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

Using the PPU

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

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

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

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

Initial values:

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

Each PPU register is described in the appropriate section:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The format of each line is

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

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

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

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

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

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

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

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

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

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

where the tile is read from word address

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

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

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

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

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

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

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

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

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

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

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

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

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

where

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

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

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

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

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

where

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

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

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

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

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

Copper The Copper executes simple instructions, which can

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

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

Each Copper instruction is 16 bits:

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

where

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

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

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

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

The raster position for x intially goes from 24 + r_x0_fp to 32 + r_xe_hsync as the raster scan goes through the front porch and horizontal sync (counted as the first part of the scan line). Due to a bug, during the back porch and active regions, it is then calculated as

```
x_raster = {x[X_BITS-1:7], x[6:5] - 2'd3, x[4:0]}
```

where x goes from 96 + r_x0_bp to xe_active. This makes x_raster non-monotonic, making it harder to wait for some x positions. A partial workaround for waiting for an x_raster value that is lower than a previous value in the scan line is to start with a write to cmp_x of the highest value expected before reaching the given position, followed directly with a write to cmp_x of the actual value.

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

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

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

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

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

The fast_mode bit

- Should be set to zero
  - at least three instructions before a write to any of the `copper_ctrl` registers,
  - for instructions that follow a write to `cmp_x` or `cmp_y`.
- Can be set to one by an instruction that writes to `jump_high` (but not the other `copper_ctrl` registers) unless it needs to be zero due to any of the above.

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

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

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

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

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

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

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

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

Horizontal timing Possible horizontal timings include

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

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

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

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

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

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

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

The contents are

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

with initial value `0b01011`. Functions:

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

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

Using AnemoneSynth

AnemoneSynth has four identical voices, each with

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

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

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

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

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

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

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

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

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

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

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

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

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

The period value can be calculated as

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

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

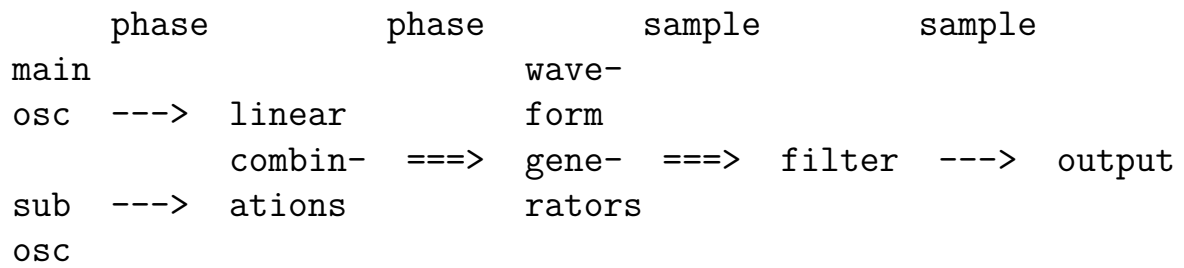
```
osc_freq[i] = output_freq * 32 / osc_period[i]
```

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

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

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

Signal path



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

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

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

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

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

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

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

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

Waveform generator The wf parameter selects between 8 waveforms:

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

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

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

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

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

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

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

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

(see Frequency representation for the definition of `mod_freq`).

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

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

The volume feeding into the filter is generally given by

```
gain = fvol / cutoff
```

but for `filter_mode=3`,

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

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

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

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

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

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

```
Q = cutoff / f_damp
```

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

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

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

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

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

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

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

The sweep value can have two formats:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Context switching To perform a context switch, the synth

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

How to test

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

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

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

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

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

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

TODO: example (in the RAM emulator code?)

External hardware

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

Pinout

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

FazyRV-ExoTiny [462]

- Author: Meinhard Kissich
- Description: A minimal SoC based on FazyRV that uses external QSPI ROM and RAM.
- GitHub repository
- HDL project
- Mux address: 462
- Extra docs
- Clock: 50000000 Hz

How it works

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

Features

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

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

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

How to test

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

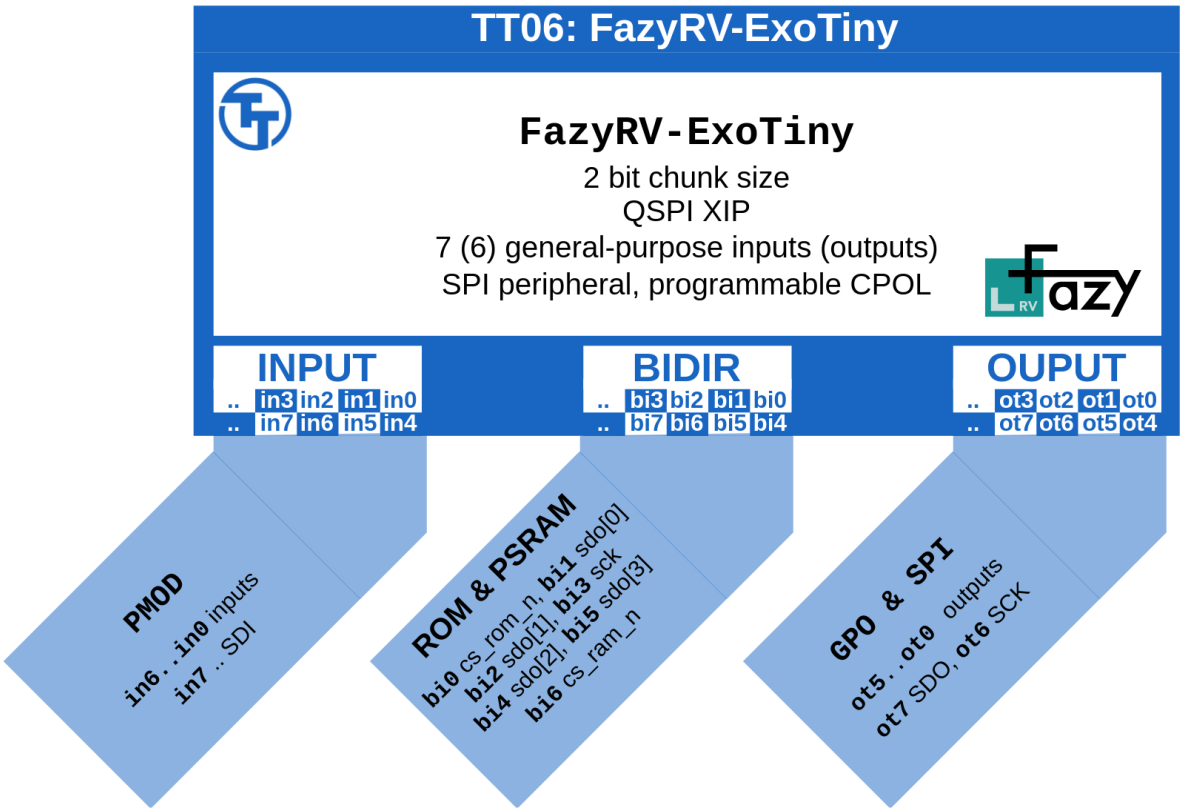


Figure 38: Pinout overview

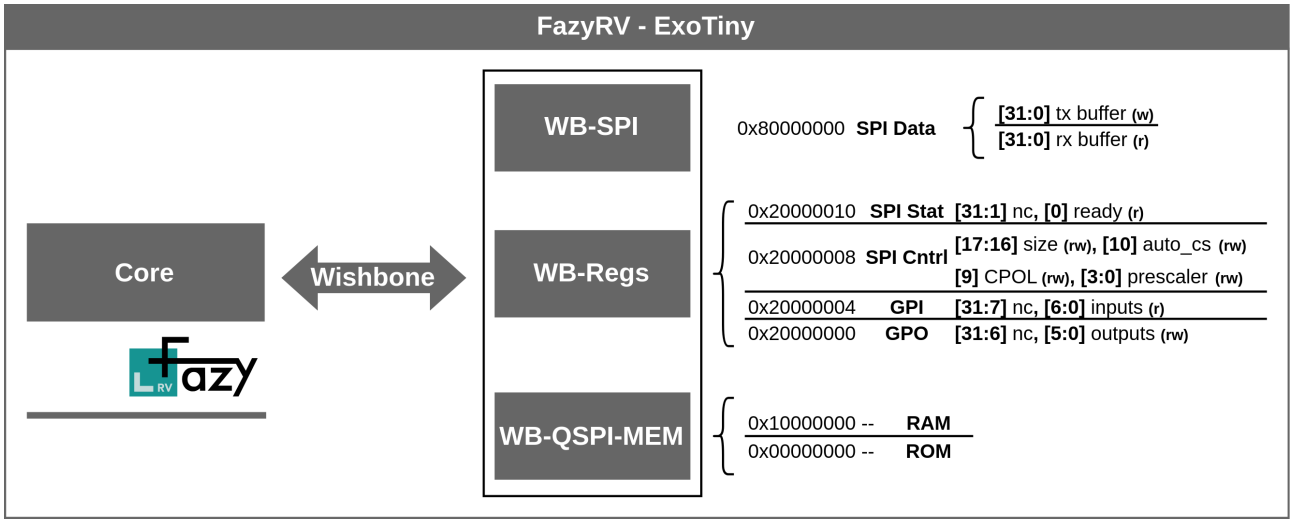


Figure 39: Block diagram

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

External hardware

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

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

Pinout

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

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

HELP for tinyTapeout [481]

- Author: Ariella Eliassaf
- Description: Use 7segment to show 'HELP'
- GitHub repository
- Wokwi project
- Mux address: 481
- Extra docs
- Clock: 0 Hz

How it works

Every flipflop is one letter of "HELP" (from top to buttom) the last flip flop darkens all 7 segments.

How to test

push the reset button and enjoy.

External hardware

none.

Pinout

#	Input	Output	Bidirectional
0		seg_a	
1		seg_b	
2		seg_c	
3		seg_d	
4		seg_e	
5		seg_f	
6		seg_g	
7			

1st passive Sigma Delta ADC [482]

- Author: Joerg Vollrath
- Description: External R1 and R2 and C2 realize a ADC
- GitHub repository
- HDL project
- Mux address: 482
- Extra docs
- Clock: 1000 Hz

How it works

A 1st order passive sigma delta modulator can be realized by attaching R1, R2 and C to a digital input. Further information is found here: https://personalpages.hs-kempten.de/~vollratj/InEl/SigmaDelta_ADC_real.html

A high level simulator: https://personalpages.hs-kempten.de/~vollratj/InEl/SigmaDelta.html

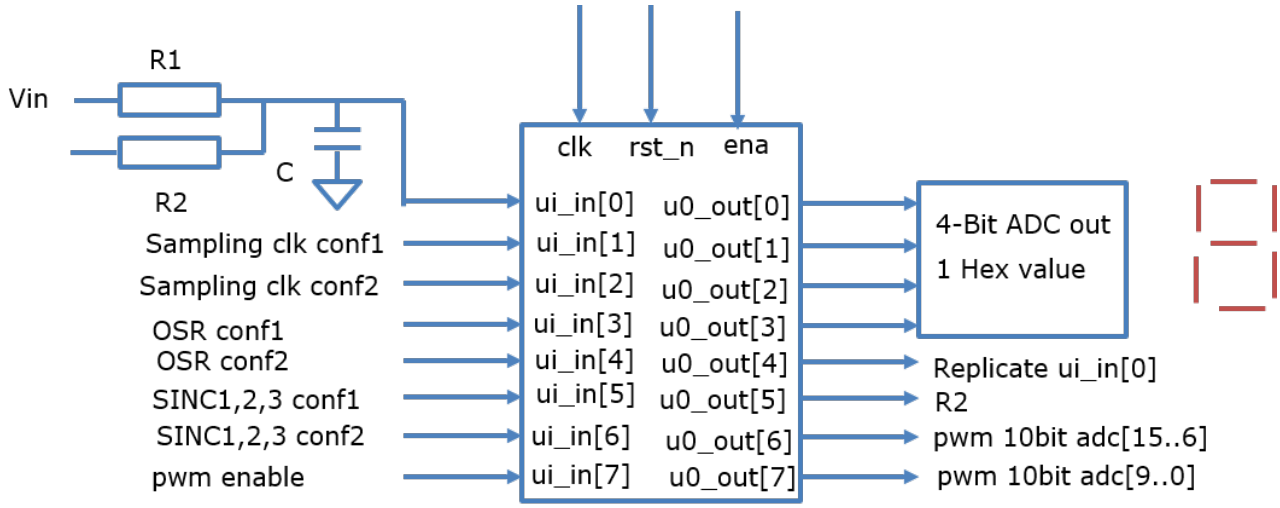


Figure 40: Tiny Tapeout Tile

How to test

Add the RC network and apply a DC voltage at the input in0 and out5. Select sampling, oversamplingrate and filter in1..6. The 4 output lines 0..3 should give a 4-Bit value. The out6,7 give a pwm signal changing with the input voltage.

All subcircuits were tested in one testfile tb_sigdel do be able to observe all signals.

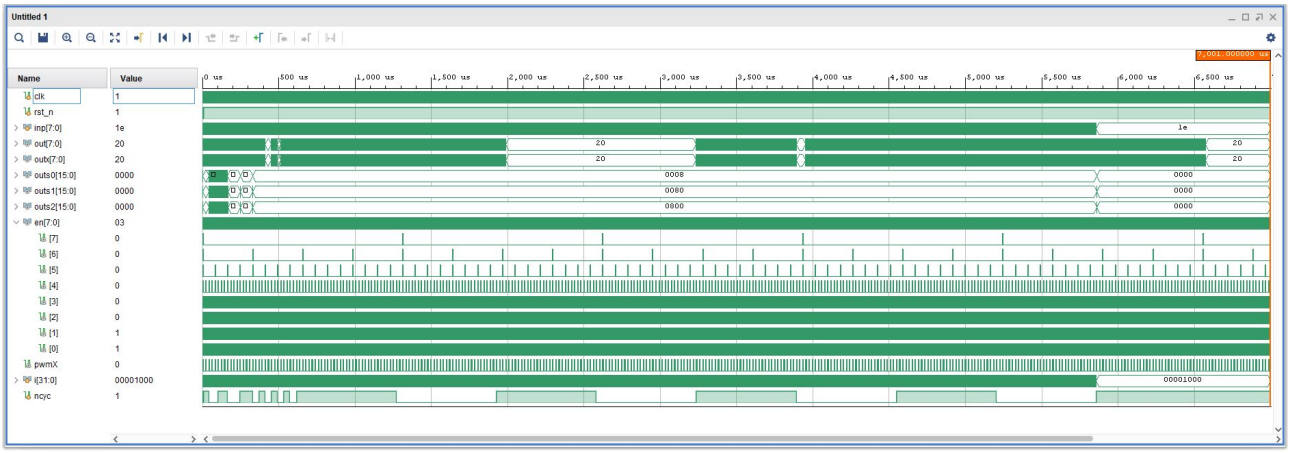


Figure 41: Figure: Circuit simulation

BASYS3 board measurements

22k resistors were used with 100 and 560 pF capacitances.

The signals at the capacitor and the digital signal inx were measured with an Electronic Explorer board.

Measurement showed a missing enable signal for inx sampling.

The table shows valid configuration options.

inp[6] inp[5] inp[4] inp[3] inp[2] inp[1] Cint fCLK=50MHz T=20ns OSR Ldmax Bits

0 1 0 0 0 0 100pF SINC1 fsCLK 40ns 256 LD7 8

0 1 0 0 0 1 SINC1 fsCLK 160ns 64 6

0 1 0 1 0 0 100 pF SINC1 fsCLK 40ns 1024 LD9 10

0 1 0 1 0 1 SINC1 fsCLK 160ns 256 8

0 1 0 1 1 0 SINC1 fsCLK 640ns 64 6

0 1 1 0 0 0 SINC1 fsCLK 40ns 4096 12

0 1 1 0 0 1 SINC1 fsCLK 160ns 1024 10

0 1 1 0 1 0 560 pF SINC1 fsCLK 640ns 256 8 ok

0 1 1 0 1 1 560pF SINC1 fsCLK 2560ns 64 LD5 6 ok

0 1 1 1 0 0 SINC1 fsCLK 40ns 16384 14

0 1 1 1 0 1 560 pF SINC1 fsCLK 160ns 4096 12 ok

0 1 1 1 1 0 560 pF SINC1 fsCLK 640ns 1024 10 ok

0 1 1 1 1 1 560 pF SINC1 fsCLK 2560ns 256 8 ok

1 0 0 0 0 1 SINC2 fsCLK 64 12

1 0 0 0 1 0 SINC2 fsCLK 16 8

1 0 0 1 1 0 SINC2 fsCLK 64 12

1 0 0 1 1 1 560pF SINC2 fsCLK 16 LD7 8

1 0 1 0 1 1 560 pF SINC2 fsCLK 64 LD11 12

1 1 0 0 1 0 SINC3 fsCLK 16 12

1 1 0 0 1 1 560 pF SINC3 fsCLK 4 LD5 6

1 1 0 1 1 1 560 pF SINC3 fsCLK 16 LD11 12

A better configuration scheme should be chosen in the next design. Higher fsCLK have lower capacitance.

A better multiplexing to the 4 Bit output with a case statement was done at the FPGA and the routing of out[3:0] done to led[3:0].

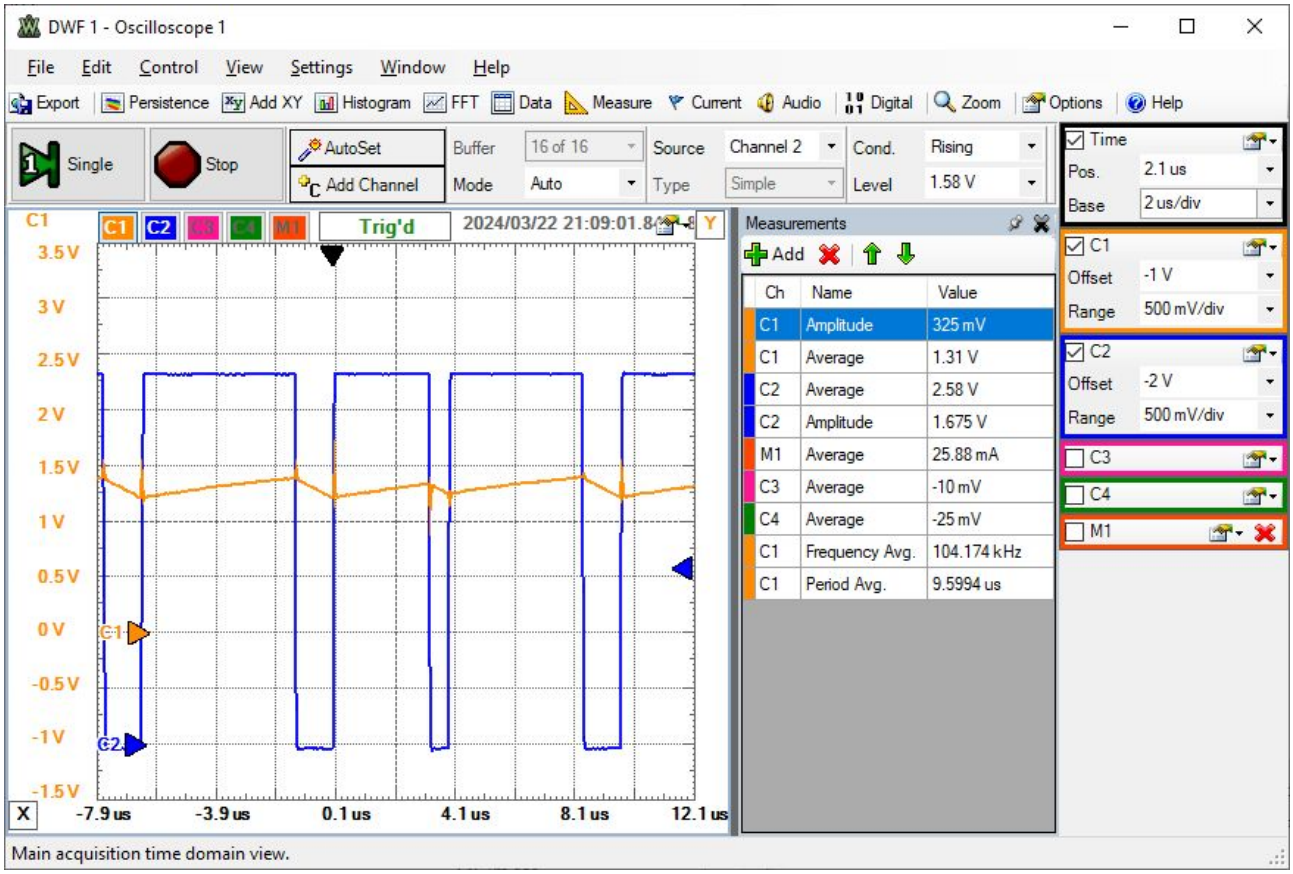


Figure 42: Oscilloscope picture BASYS3

Figure: Oscilloscope picture BASYS3 FPGA not(inx)(blue) and inp0

Summary

It is possible with this circuit to look at the influence of R, C and oversampling on the accuracy of a 1st order sigma delta ADC.

Bad R,C values can cause non linearities or signal limitation to VDD and ground.

The order of the SINC filter can lead to less resolution (SNR) than expected.

The order of the SINC filter should be at least one more than the sigma delta modulator.

The pwm signal has 10 bits and can be used for more precise output values.

References

Martin Knauer, Jörg Vollrath, 'Implementation and Testing of a FPGA Based Sigma Delta Analog to Digital Converter', 58. MPC Workshop, Reutlingen July 2017

Pinout

#	Input	Output	Bidirectional
0	Input voltage input voltage R1, uo5 R2, C attached	ADC 0 LSB	
1	Sampling clock conf1	ADC 1	
2	Sampling clock conf2	ADC 2	
3	OSR conf1	ADC 3 MSB	
4	OSR conf2	replicate ui0	
5	SINC1,2,3 conf1	invert ui0 R2	
6	SINC1,2,3 conf2	pwm upper	
7	pwm output enable	pwm lower	

Parallel / SPI modulation tester [483]

- Author: Chris Merrill
- Description: PDM/PWM/PFM waveform output based on digital data in
- GitHub repository
- HDL project
- Mux address: 483
- Extra docs
- Clock: 50000000 Hz

How it works

This is an attempt to build a DAC with multiple digital modulation schemes on the output. It was originally intended to use the analog pins to output the analog waveform as well, but I ran out of time.

The device will output four types of digital modulation to represent the analog input:

- PDM on `uo[0]`
- PFM (fixed width pulse, variable spacing) on `uo[2]`
- PFM (variable frequency, 50% duty cycle) on `uo[3]`
- PWM on `uo[4]`

The PDM signal is based on tracking an error accumulator, and has the fastest response to changing input. The PWM signal has a frequency that is 1/256th of the input clock frequency (~200kHz at 50MHz input).

The two PFM modes are less useful for actual modulation, but they can effectively do a frequency sweep of the filter on the output.

How to test

After connecting the external RC filter, you need to set the clock rate and program the DAC.

The modulation output rate can be divided down from the main clock input by up to 15. The `uio[0:3]` pins set the clock divisor.

There are two ways to program the DAC, and they can be selected between using the `uio[7]` pin.

`uio[7]` = 0: Parallel data input

- The 8-bit DAC level is input via `ui[0:7]`

- The data is latched onto the output on the rising edge of `uio[4]`

`uio[7]` = 1: SPI data input

- There is a SPI but with the pinout
  - `uio[5]` -> SCLK
  - `uio[6]` -> SDI
  - `uio[4]` -> CS_L
- The 8-bit DAC level is input as 8 bits of data sent over the SPI bus when CS_L is low.
- The SCLK signal should be slower than the main CLK signal to the IC.
- Data is latched onto the output on the rising edge of CS_L
- This path is less validated than the parallel path.

External hardware

The modulation outputs are all digital pulses of some sort. In order to get meaningful analog levels, you'll need to add an RC filter on the output pin that you are using. The cuttoff will depend on what your chosen frequency is and the type of modulation.

Pinout

#	Input	Output	Bidirectional
0	DAC Parallel Input, bit 0	PDM Waveform Output	CLK_DIV[0]
1	DAC Parallel Input, bit 1		CLK_DIV[1]
2	DAC Parallel Input, bit 2	PFM Output, Single cycle pulse	CLK_DIV[2]
3	DAC Parallel Input, bit 3	PFM Output, 50% duty cycle	CLK_DIV[3]
4	DAC Parallel Input, bit 4	PWM Waveformn Output	SPI CS_L / Parallel Latch
5	DAC Parallel Input, bit 5		SPI SCLK
6	DAC Parallel Input, bit 6		SPI SDI
7	DAC Parallel Input, bit 7		Parallel/SPI Select (0 => Par

Flash ADC [484]

- Author: htfab
- Description: 4-bit flash ADC with binary encoder
- GitHub repository
- Analog project
- Mux address: 484
- Extra docs
- Clock: 0 Hz

How it works

A resistor ladder between power and ground is used to generate 15 reference voltages at regular intervals. The input signal is compared with each of them in turn to get a unary ADC output. To avoid loading the input too much, some voltage followers are added before the comparators. Finally, a digital encoder circuit with error correction converts the unary output to binary using a tree of majority gates and multiplexers. The digital circuit also implements some debugging logic.

How to test

For basic operation, set the first two digital inputs to 0 and apply a voltage between 0 and 1.8 V to the analog input pin. The first four output pins should give a binary readout.

The second set of four output pins samples the unary output at bits 1, 5, 9 and 13 to indicate if the input voltage is over the reference voltages 0.18 V, 0.66 V, 1.14 V and 1.62 V respectively.

Bidirectional pins are configured as outputs where the first four is an XOR of some unary pins and the second four multiplexes unary pins using input pins 2 and 3 as a selector. Both features allow checking when the input signal crosses one of the intermediate reference thresholds.

Input range	Unary	Binary	Sample	XOR	Mux00	Mux01	Mux10	Mux1
0 V to 0.06 V	000000000000000	0000	0000	0000	0000	0000	0000	0000
0.06 V to 0.18 V	000000000000001	0001	0000	0001	0001	0000	0000	0000
0.18 V to 0.3 V	000000000000011	0010	0001	0011	0011	0000	0000	0000
0.3 V to 0.42 V	000000000000111	0011	0001	0111	0111	0000	0000	0000
0.42 V to 0.54 V	000000000001111	0100	0001	1111	1111	0000	0000	0000
0.54 V to 0.66 V	000000000011111	0101	0001	1110	1111	0001	0000	0000

Input range	Unary	Binary	Sample	XOR	Mux00	Mux01	Mux10	Mux1
0.66 V to 0.78 V	000000000111111	0110	0011	1100	1111	0011	0000	0000
0.78 V to 0.9 V	000000001111111	0111	0011	1000	1111	0111	0000	0000
0.9 V to 1.02 V	000000011111111	1000	0011	0000	1111	1111	0000	0000
1.02 V to 1.14 V	000000111111111	1001	0011	0001	1111	1111	0001	0000
1.14 V to 1.26 V	000001111111111	1010	0111	0011	1111	1111	0011	0000
1.26 V to 1.38 V	000011111111111	1011	0111	0111	1111	1111	0111	0000
1.38 V to 1.5 V	000111111111111	1100	0111	1111	1111	1111	1111	0000
1.5 V to 1.62 V	001111111111111	1101	0111	1110	1111	1111	1111	0001
1.62 V to 1.74 V	011111111111111	1110	1111	1100	1111	1111	1111	0011
1.74 V to 1.8 V	111111111111111	1111	1111	1000	1111	1111	1111	0111

The circuit also provides debug functionality to independently check the ADC and the encoder.

If the first input pin is set to 1, the circuit is in *encoder debug mode* where the rest of the input pins as well as the bidirectional pins (which are now turned into inputs) are used instead of the ADC. The output pins function as described above, the bidirectional pins obviously cannot provide output in this case.

Otherwise, if the second input pin is set to 1, the circuit is in *ADC debug mode* where the raw unary output from the ADC is directly sent to the output and bidirectional pins.

External hardware

You can use your favourite microcontroller to generate an analog input by outputting a PWM signal and adding an external capacitor to ground that together with the microcontroller's built-in resistance makes a simple low-pass RC filter.

Pinout

#	Input	Output	Bidirectional
0	debug encoder (skip ADC)	binary bit 0	xor of unary bits 0, 4, 8, 12
1	debug ADC (skip encoder)	binary bit 1	xor of unary bits 1, 5, 9, 13
2	unary selector bit 0	binary bit 2	xor of unary bits 2, 6, 10, 14
3	unary selector bit 1	binary bit 3	xor of unary bits 3, 7, 11
4	(debug mode only)	unary bit 1	unary bit 4*sel
5	(debug mode only)	unary bit 5	unary bit 4*sel+1

#	Input	Output	Bidirectional
6	(debug mode only)	unary bit 9	unary bit 4*sel+2
7	(debug mode only)	unary bit 13	unary bit 4*sel+3

Analog pins

ua#	analog#	Description
0	0	ADC input

CSIT-Luks [485]

- Author: CSIT Team (Jan Furlan, Jurica Gašpar, Marko Marinović, Tin Sorić, Ivan Štignedec, Dino Terman, Jurica Kundrata)
- Description: Camera lighting settings recommender.
- GitHub repository
- HDL project
- Mux address: 485
- Extra docs
- Clock: 1000 Hz

How it works

This project implements a settings recommender for photography. The ISO, shutter speed and focal ratio values are inputed using a rotational encoder and a four-digit seven-segment display. After inputing the values, an external luxmeter is read via SPI interface and all of the values are used to retrieve the recommended setting from a LUT in an SPI Flash. The recommended value is displayed on the four-digit seven-segment display.

How to test

This project uses a user interface consisting of a rotational encoder and four-digit seven-segment display. After reset or power-up, first the ISO value is selected by rotating the encoder. The current value is displayed on the four-digit seven-segment display and it is confirmed by a short press on the rotational encoder. Next, the shutter speed is selected by rotating and confirmed by a short press of the encoder. Finally, the focal ratio is selected by rotating the encoder and it is confirmed by a medium press of the encoder. After reading the luxmeter and the flash-based LUT, the recommended settings value is shown on the four-digit seven-segment display.

External hardware

External hardware comprises of a rotational encoder, a four-digit seven-segment display, SPI luxmeter (e.g. Pmod ALS) and SPI flash (e.g. MX25L3233FMI-08G).

Pinout

#	Input	Output	Bidirectional
0	A (rot_encoder)	a (seven_seg)	an[0] (seven_seg)
1	B (rot_encoder)	b (seven_seg)	an[1] (seven_seg)
2	PB (rot_encoder)	c (seven_seg)	an[2] (seven_seg)
3	MISO (spi_flash)	d (seven_seg)	an[3] (seven_seg)
4	MISO (spi_sensor)	e (seven_seg)	SCLK (spi_flash, spi_sensor)
5		f (seven_seg)	SS (spi_flash)
6		g (seven_seg)	SS (spi_sensor)
7		dp (seven_seg)	MOSI (spi_flash)

Double Inverter [486]

- Author: Matt Venn
- Description: A little inverter followed by a bigger one
- GitHub repository
- Analog project
- Mux address: 486
- Extra docs
- Clock: 0 Hz

How it works

Initially I just wanted to do an inverter, but then I thought I'll make the inverter much more powerful. The problem with much bigger output transistors is that their gate capacitance increases, and so the inverter is slower. My adding a primary, smaller inverter in front, I get much faster rise times on the big transistors.

How to test

Apply a pulse to analog pin 1 and see it replicated on analog pin 0.

External hardware

Oscilloscope

Pinout

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

Analog pins

ua#	analog#	Description
0	5	output
1	0	input

Trivium Non-Linear Feedback Shift Register [487]

- Author: icaris lab
- Description: Trivium stream cipher used as a non-linear feedback shift register.
- GitHub repository
- Wokwi project
- Mux address: 487
- Extra docs
- Clock: 50000000 Hz

How it works

The project is a hardware implementation of the Trivium stream cipher used as a non-linear feedback shift register (NLFSR). The NLFSR is defined with the least-significant bit (LSB) at the left-most register R0 and the most-significant bit (MSB) at the right-most register R287. The LFSR circular shifts bits from left to right (R_n -> R_n+1), with the three feedback taps:

R177 = (R174 * R175) + (R161 + R176) + R263 R93 = (R90 * R91) + (R65 + R92) + R170 R0 = (R285 * R286) + (R242 + R287) + R68

The output of the NLFSR is:

z = R65 + R92 + R161 + R176 + R242 + R287

The NLFSR contains an initialization/fail-safe feedback that prevents the LFSR from entering an all-zero state. If the LFSR is ever in an all-zero state, a "1" value is inserted into R0.

A schematic of the circuit may be found at:

https://wokwi.com/projects/395357890431011841

The circuit has 10 inputs:

Input	Setting
CLK	Clock
RST_N	Not Used
01	Not Used
02	Manual R0 Input Value
03	Input Select
04	Not Used
05	Not Used
06	Not Used

Input	Setting
07	Not Used
08	Not Used

The CLK sets the clocking for the flip-flop registers for latching the NLFSR values. In the schematic shown in the Wokwi project, a switch is used to select either the system clock or an externally provided or manual clock that allows the user to manually step through each latching event.

An 8-input DIP switch provides some flexibility to initalizing the NLFSR. DIP03 (IN2) allows the user to toggle the Input Select function, which is a multiplexer that select whether the left-most register (R0) takes in as the input the NLFSR feedback value or a value that is manually selected by the user. The switch also controls whether R93 and R177 takes in a NLFSR feedback or a value directly from R92 or R176, respectively.

DIP02 (IN1) allows a the user to manually enter a 0 or a 1 value into the leftmost register.

The cicuit has 8 outputs. They output the following values:

Output	Value in
01	R0 (NLFSR input)
02	R65
03	R92
04	R161
05	R176
06	R242
07	R287
08	z (NLFSR output)

The output allows for some self-testing, where OUT08 = OUT02 + OUT03 + OUT04 + OUT05 + OUT06 + OUT07.

How to test

The circuit can be tested by powering on the circuit, and first setting the Input Select switch (DIP03) to "1" to reset/initialize the entire LFSR to all-zeros. The Input Select switch can then be switched to "0" to allow the LFSR to run from its all-zero initialized value. The output values of the NLFSR from this zeroized state may be observed using a logic analyzer, and can be compared with the values obtained for the python simulation:

https://github.com/icarislab/tt06_biviumb-prng_cu/blob/main/docs/(TBD)

External hardware

No external hardware is required.

Pinout

#	Input	Output	Bidirectional
0		r000_val	
1	data_in	INTERM_fb	
2	load_en	r092_val	
3		r093_val	
4		r176_val	
5		r177_val	
6		r287_val	
7		NLSFR_out	

Analog Test Circuit ITS: VCO [488]

- Author: Astria Nur Irfansyah
- Description: Voltage controlled ring oscillator.
- GitHub repository
- Analog project
- Mux address: 488
- Extra docs
- Clock: 0 Hz

How it works

The circuit is a voltage-controlled ring oscillator (VCRO) using transmission gates as the delay control element. It is based on the paper:

Retdian, N., Takagi, S., & Fujii, N. (2002). Voltage controlled ring oscillator with wide tuning range and fast voltage swing. 2002 IEEE Asia-Pacific Conference on ASIC, AP-ASIC 2002 - Proceedings, 201–204. https://doi.org/10.1109/APASIC.2002.1031567.

How to test

Pinouts:

- input pins: v_control_n, v_control_p
- output pin: out

External hardware

To test, apply control voltage v_control_n and v_control_p, where the sum of the two voltages should ideally be equal to the supply voltage of 1.8V.

Pinout

#	Input	Output	Bidirectional
0		out5	
1		out4	
2		out3	
3		out2	
4		out1	

#	Input	Output	Bidirectional
5		out0	
6			
7			

Analog pins

ua#	analog#	Description
0	1	out
1	3	vcon_n
2	2	vcon_p

SADdiff_v1 [489]

- Author: Daniel Burke
- Description: digital neuron component test
- GitHub repository
- HDL project
- Mux address: 489
- Extra docs
- Clock: 10 Hz

How it works

Takes two 8bit values in and performs addition/subtraction

How to test

Supply two 8bit numbers and get back sum

External hardware

None needed.

Pinout

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

Simple FET OpAmp with Sky130. [490]

- Author: Diego Satizabal
- Description: A simple FET OpAmp, credits to Fulgor Foundation as I used Diegos and Julias design as a base available at https://github.com/diegohernando/caravel_fulg
- GitHub repository
- Analog project
- Mux address: 490
- Extra docs
- Clock: 0 Hz

Sky130 FET OpAmp

This is the implementation of a simple Operational Amplifier (OpAmp) buit with FET from the Sky130 PDK. The design is based on Diego's and Julia's design from Fulgor Foundation.

How it works Basically works as an OpAmp, that is, a high-impedance high-gain on open-loop amplifier block, without feedback loop acts as a comparator, with feedback loop may work as inverting and non-inverting amplifier with gain setteable with feedback network as with any OpAmp like LM317, not too much more to add here.

Internal schematics The following is a capture of Xschem of the internals of OpAmp:

A note on the R1/R2 network: This is to generate a virtual ground and provide a negative voltage as VDD. In TinyTapeout 6 the only available power is 1.8 VDC referenced to a ground, for the OpAmp we considered convenient to have a negative voltage, so we implemented this network in a way that the external signal ground (ZREF) be the reference ground and the actual power ground became a -0.9 Volts point from the OpAmp point of view.

Magic Layout Next we show a view of the Layout created in Magic for the OpAmp cell only:

The OpAmp cell integrated with TinyTapeout top cell shows less detail and is shown as reference:

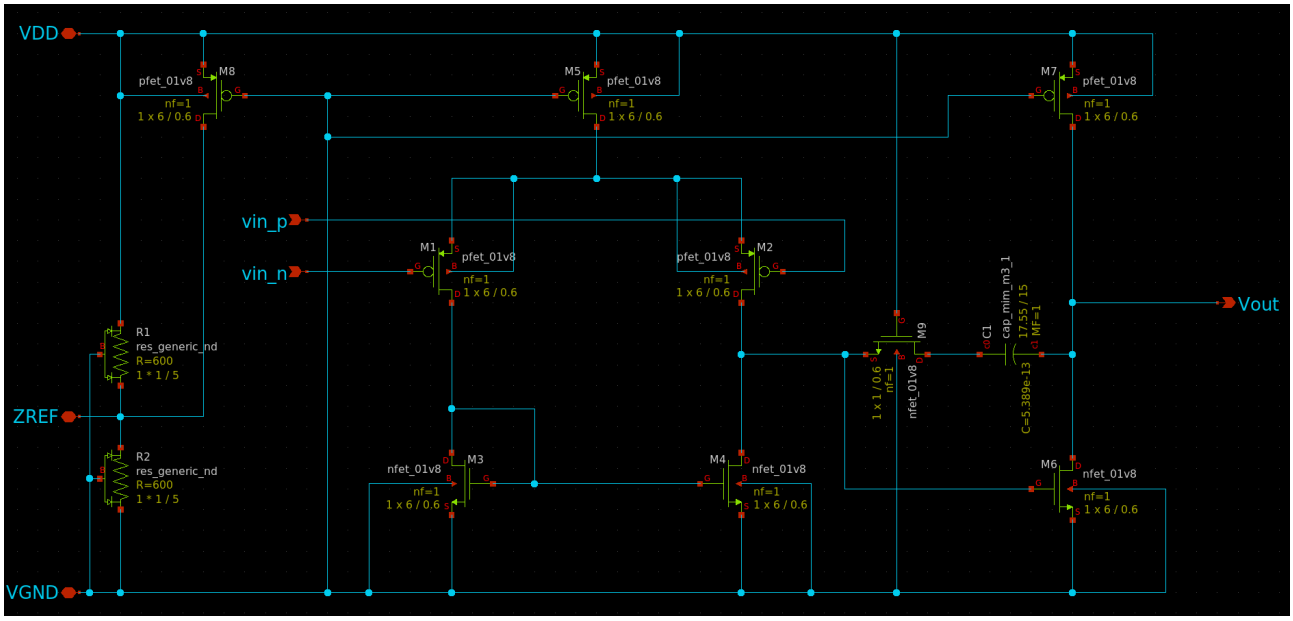


Figure 43: OpAmp schematics

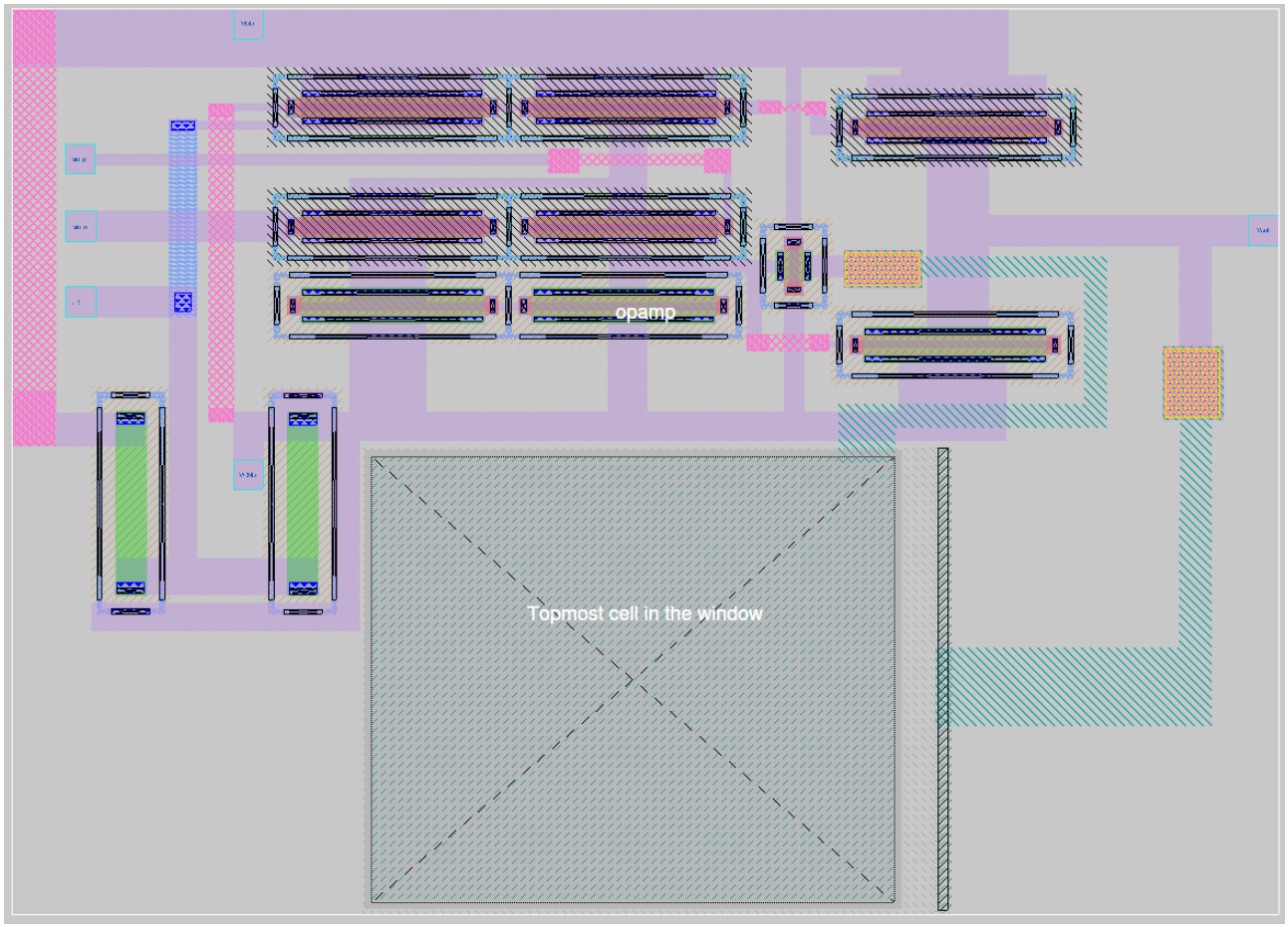


Figure 44: Magic layout

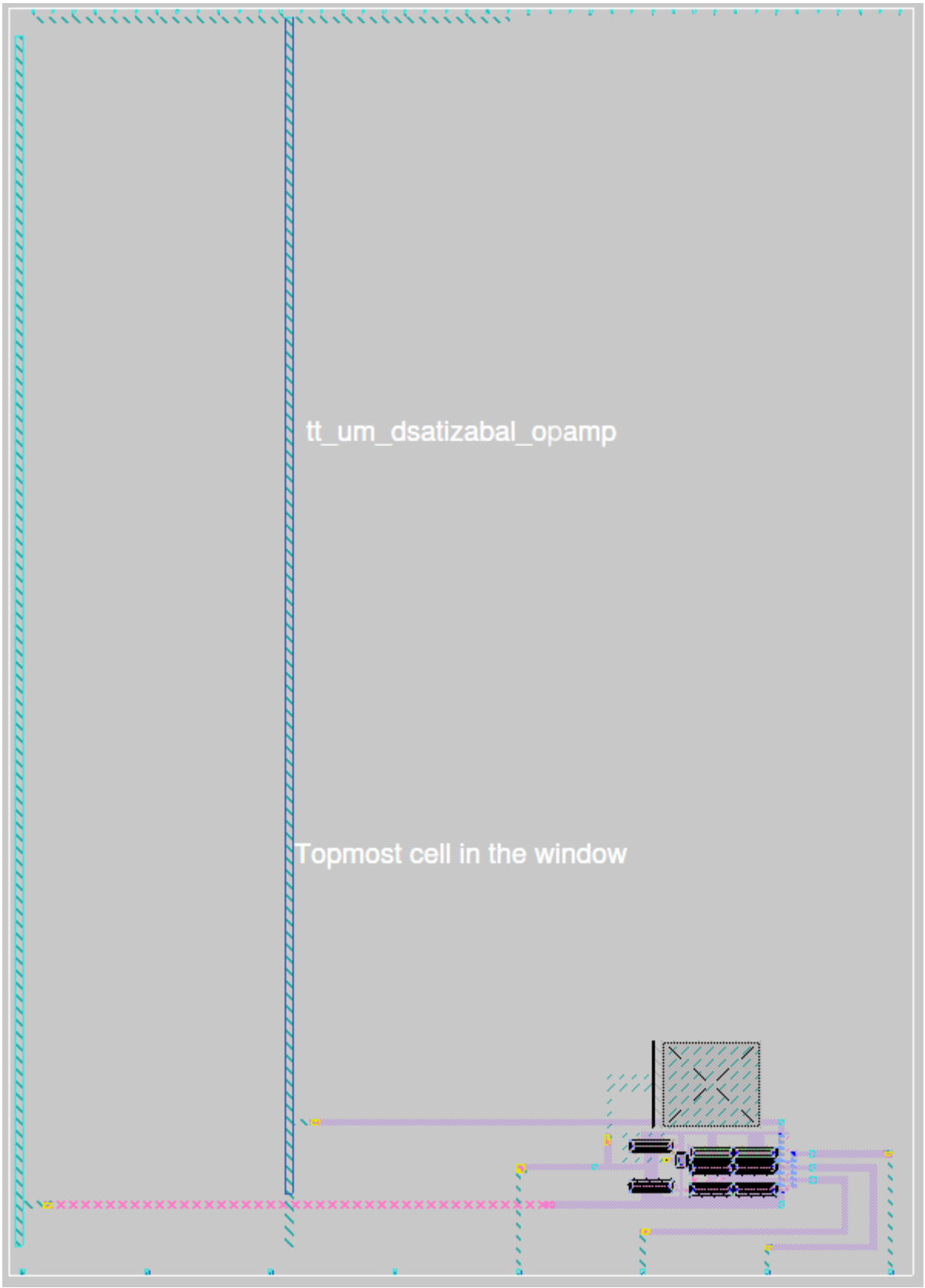


Figure 45: Magic layout

How to test Two testbenches are included for the OpAmp: a Non-Inverting amplifier and an Inverting amplifier

To test the Operational Amplifier any of those circuits, that are very easy to setup, can be utilized and check the corresponding output gains.

/! Beware that the input signal ground (Vin) must not be connected to the same ground as Chip power, it must be connected to ZREF pin. /!
/! Observe that output signal will have a DC offset due to the use of a virtual ground inside the OpAmp. /!

External hardware Power source, resistors, signal generator, oscilloscope.

Pinout

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

Analog pins

ua#	analog#	Description
0	4	ZEF
1	1	V-
2	3	V+
3	2	Vout

BF Processor [491]

- Author: Ivan Pancheniak
- Description: Implementation of a Brainf*ck processor in hardware
- GitHub repository
- HDL project
- Mux address: 491
- Extra docs
- Clock: 10000000 Hz

How it works

This is a 75% implementation (the IO operations of . and , weren't implemented) of the esoteric language Brainfuck as small factor processor. It works as any "regular" microprocessor would, executing the given ASCII values of each character as an opcode, following this state machine:

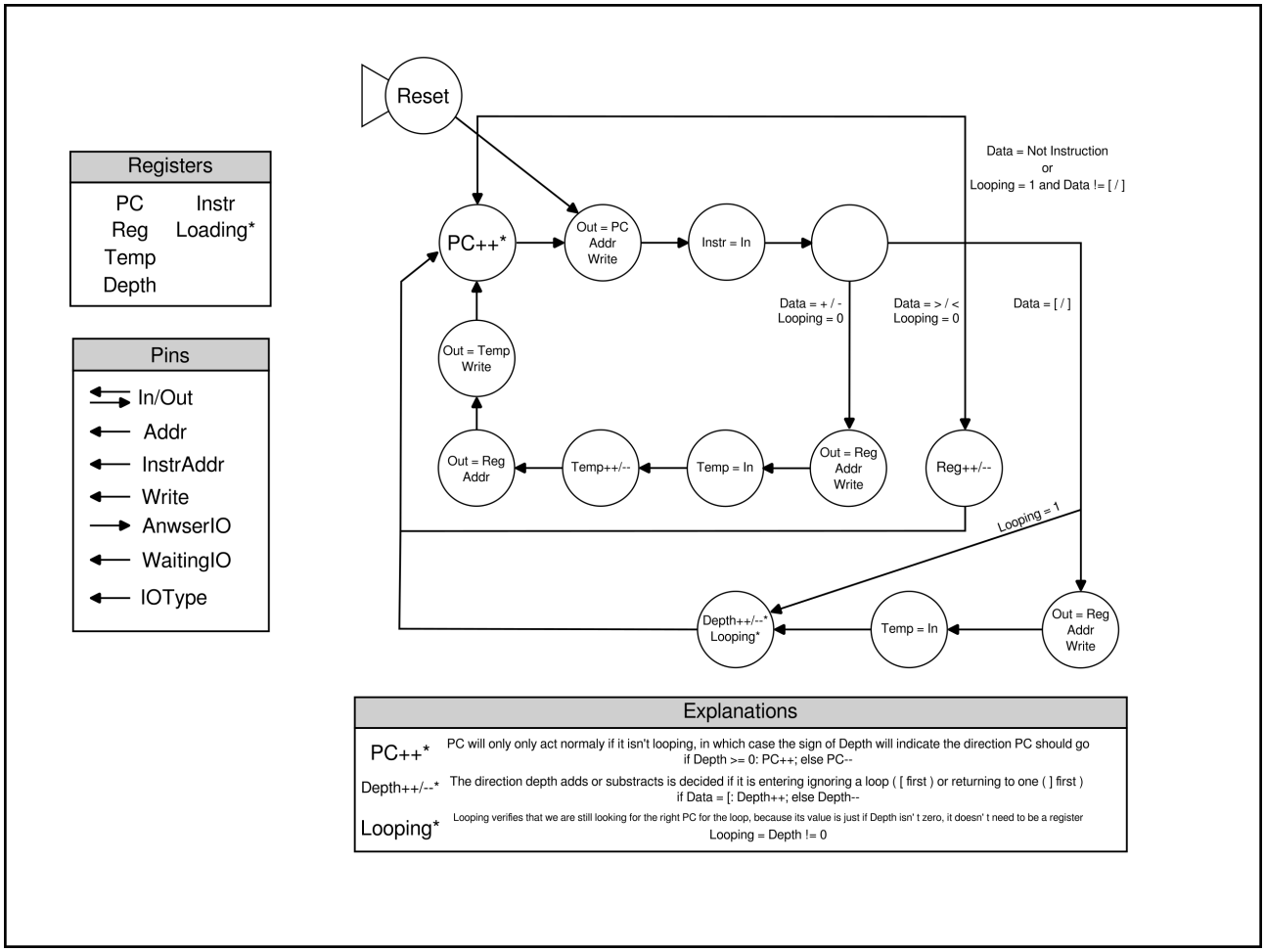


Figure 46: fsm

With an internal implementation close to the following one:

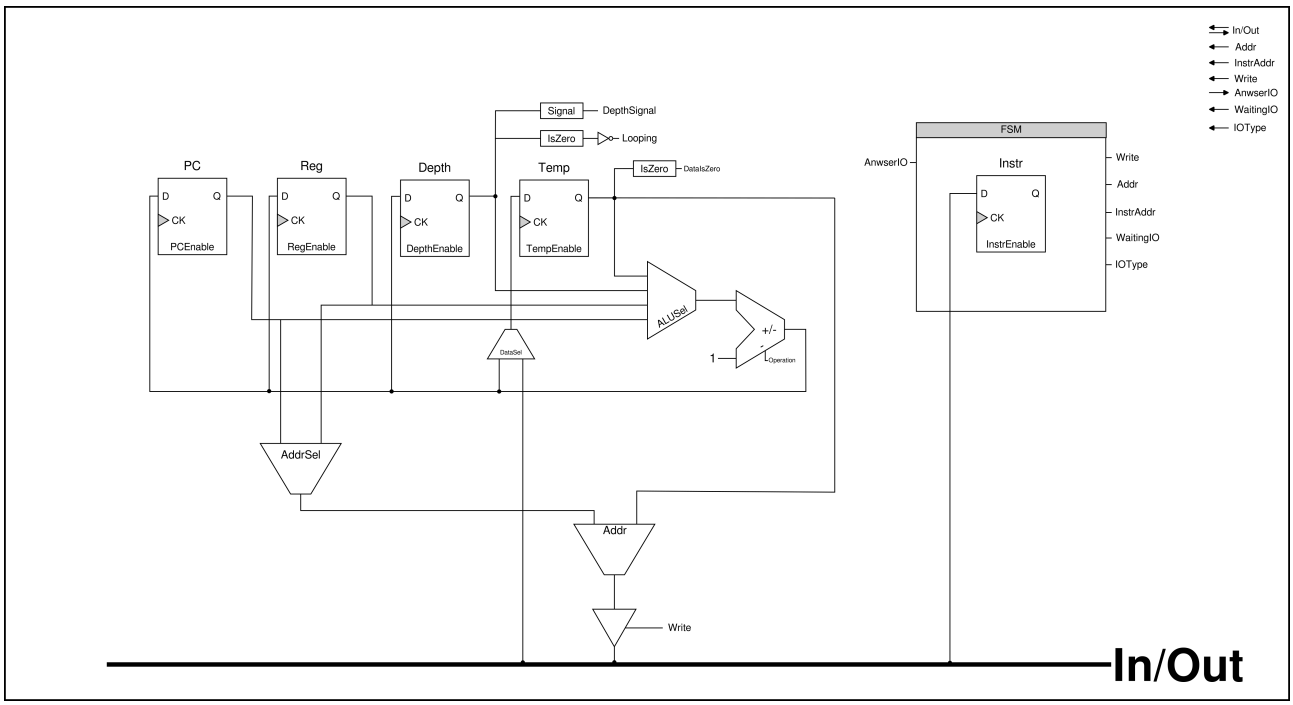
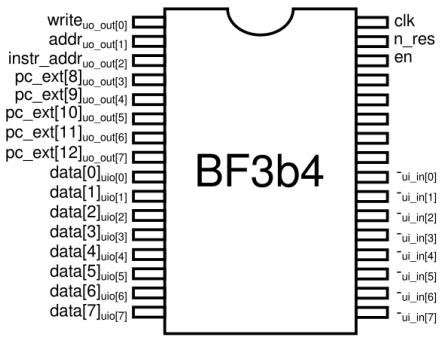


Figure 47: diagram

How to test



write - Enabled if the tristat

addr - Enabled if the data i

instr_addr - Enabled if the instruction

pc_ext[12:8] - Extension of to the upper [12:8] section enabled

data[7:0] - 8 bit in/out data reading mode, if enabled, i enabled, the data being wr request *on the following cy*

clk - The clock (up until 20

n_res - Resets when disab

en - Enables the processor

ui_in[7:0] - Not used

This circuit has the following pinout:
To drive it, you need to have an external address register, as it requests reads/writes data in alternating cycles, some memory organization to support, at the minimum, 256 x 8 of program memory and 256 x 8 of instruction memory (this can be expanded until 1KB memories by also using the *pc_ext* pins), and a clock, preferably at 20MHz. If you don't want for the program to access instruction memory, the pin *instr_addr* is

only set if the address being requested is for instructions, so you can use that to avoid it.

External hardware

These are some components that you can use for interfacing with the processor:

- 256 x 8 SRAM
- 8 bits to 13 bits Register
- 256 x 8 to 1K x 8 ROM
- LED bars with 8 segments to show the current value exiting the processor on the data bus (*uio_out*)

Pinout

#	Input	Output	Bidirectional
0		Write	Data_0
1		Addr	Data_1
2		Instr_Addr	Data_2
3		PC_Ext_8	Data_3
4		PC_Ext_9	Data_4
5		PC_Ext_10	Data_5
6		PC_Ext_11	Data_6
7		PC_Ext_12	Data_7

TT06 Grab Bag [492]

- Author: algofoogle (Anton Maurovic)
- Description: A few analog/mixed-signal experiments with a 24-bit VGA pattern generator as the highlight
- GitHub repository
- Analog project
- Mux address: 492
- Extra docs
- Clock: 25000000 Hz

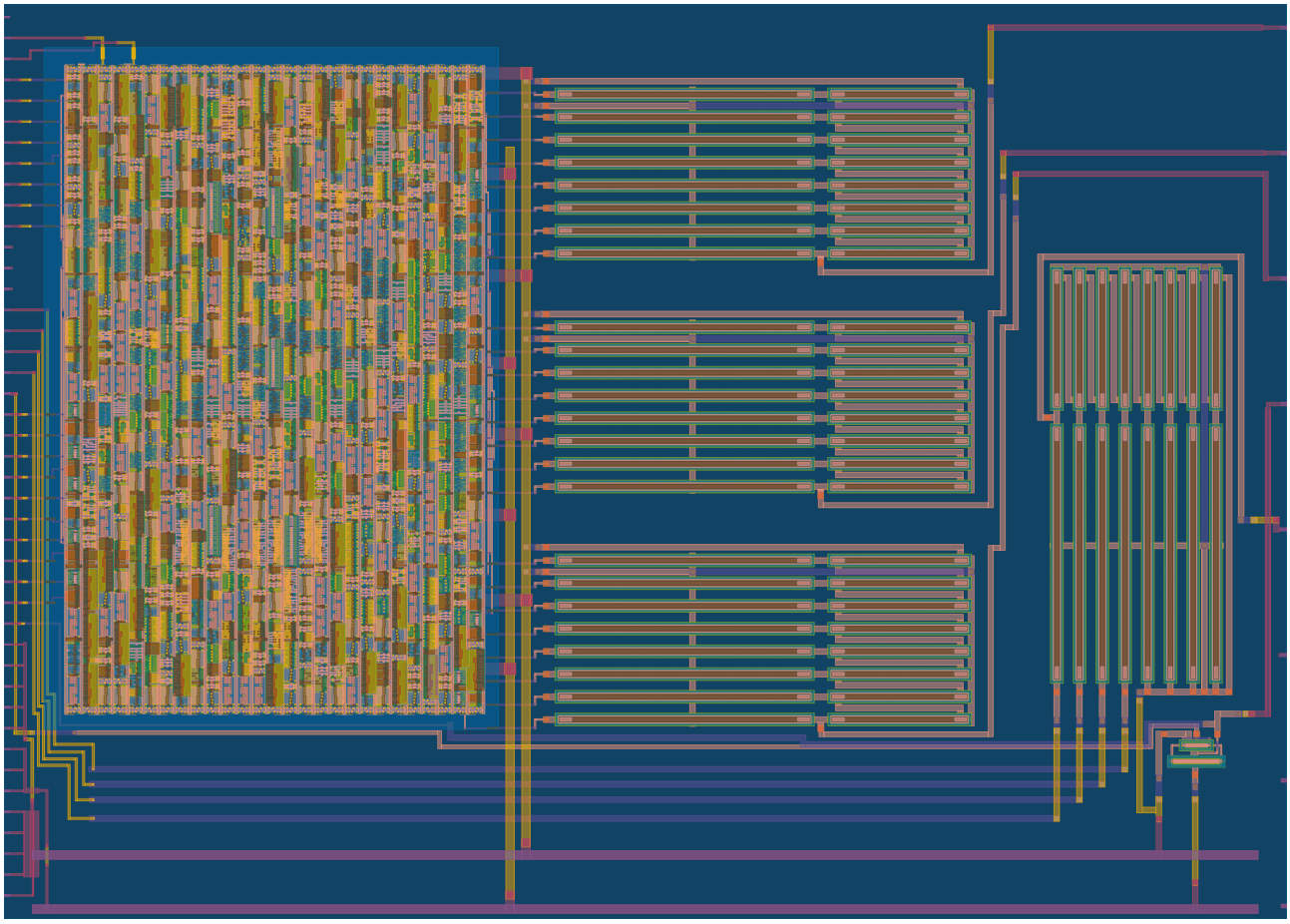


Figure 48: tt06-grab-bag GDS layout showing digital block, 4 DACs, and 1 inverter

What is this thing?

A simple analog/mixed-signal project I created in the 1st round of Matt Venn's Zero to ASIC **Analog Course** beta. **This design has been demonstrated to work in silicon.** For silicon test results, see my journal entry: https://algo.org/journal/0226

The design comprises:

- Simple CMOS inverter. That was my very first custom layout attempt.
- Digital block generating a few basic 24b-colour (RGB888) VGA test patterns.
- Analog RGB out (digital block VGA outputs via 3x 8-bit R2R DACs).
- An extra 4-bit R2R DAC.

VGA test pattern outputs

The design's *main* purpose is to generate VGA test patterns that were hoped to look as good as these simulations:

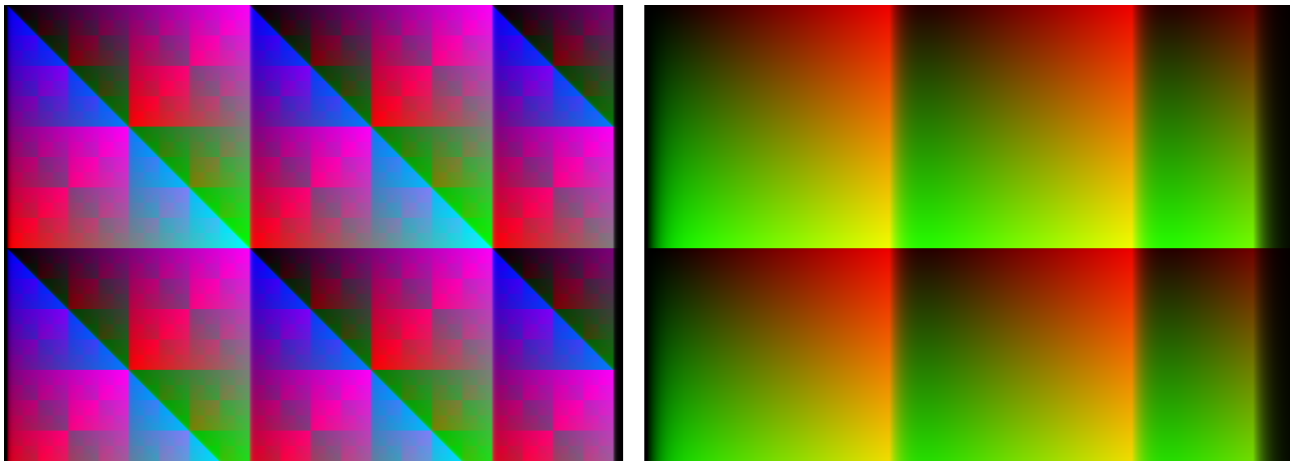


Figure 49: Simulated VGA outputs, XOR pattern and RAMP pattern

The left-hand pretty pattern is "MODE_XORS" (`ui_in==8'b0011_0000`) while the right-hand gradients pattern is "MODE_RAMP" (`ui_in==8'b0001_0000`).

Notice there is some horizontal smearing (more exaggerated in the right-hand image of the red/green mixes).

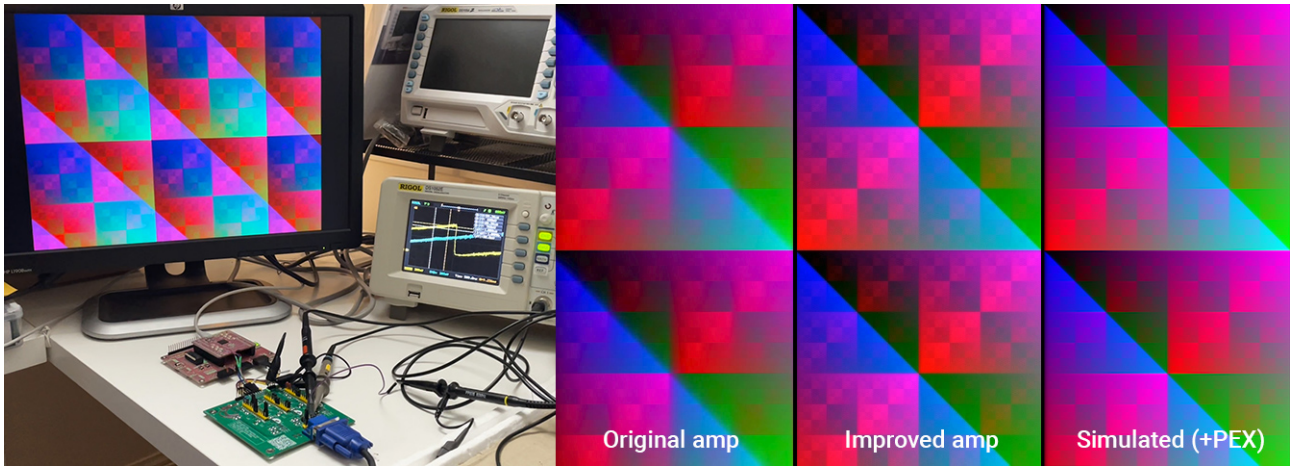


Figure 50: Real silicon, working, driving a VGA monitor

Actual results from silicon testing (seen above) are pleasing.

CMOS inverter

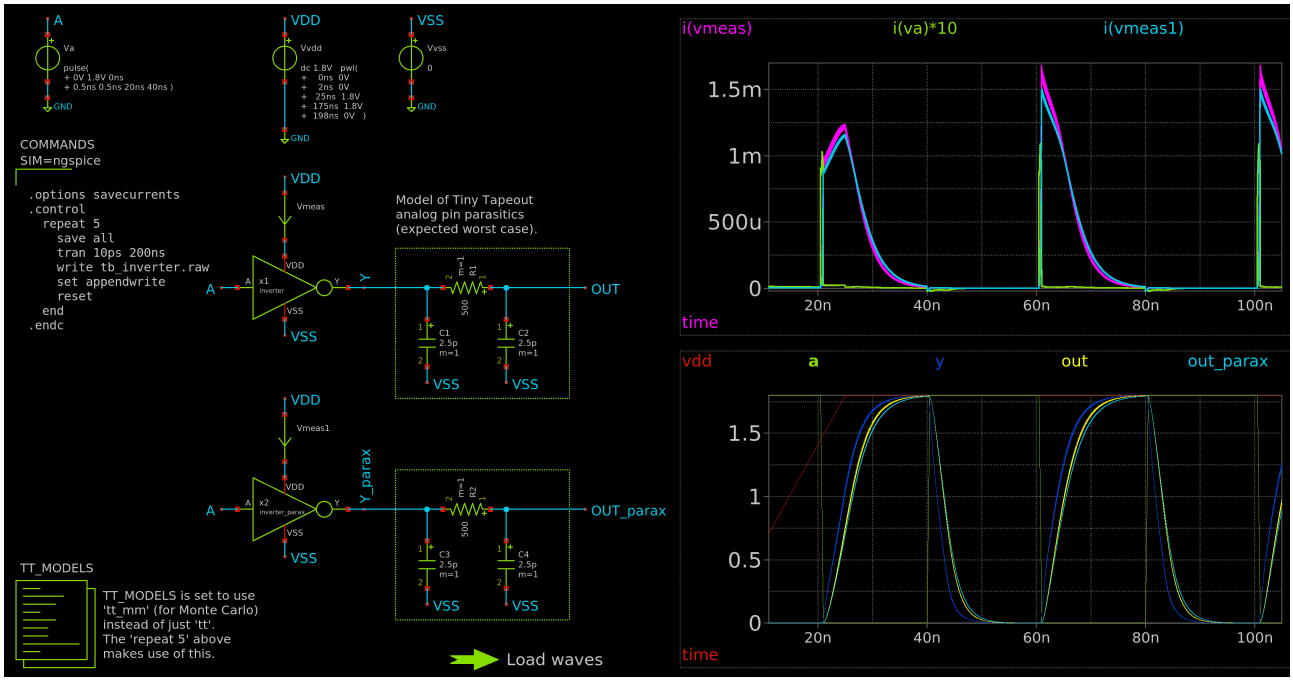


Figure 51: Xschem simulation of my CMOS inverter

Pretty simple:

- Its input is `uio_in[7]` (bidir 7).
- Its output goes to two places: `ua[3]` (analog) and `uio_out[2]` (digital).
- I'd expect its digital out performance to be better: the TT digital mux has more buffering & less loading than the TT analog mux.

The graphs accompanying the schematics simulate *analog* out is expected to be stable (enough) within 10ns; relatively poor performance characteristic of the TT analog mux loading. I expect bigger transistors could drive this harder and make it faster.

Extra 4-bit R2R

4th instance of my 8-bit R2R DAC cell grounds the 4 LSB, connecting the 4 MSB to spare bidir inputs (`uio_in[6:4]`) with DAC output via `ua[4]`.

How it works

TBC!

The internal R2R DACs for each of the RGB outputs just go directly (unbuffered) to the analog output pins, where they are subject to the loading of the TT06 analog

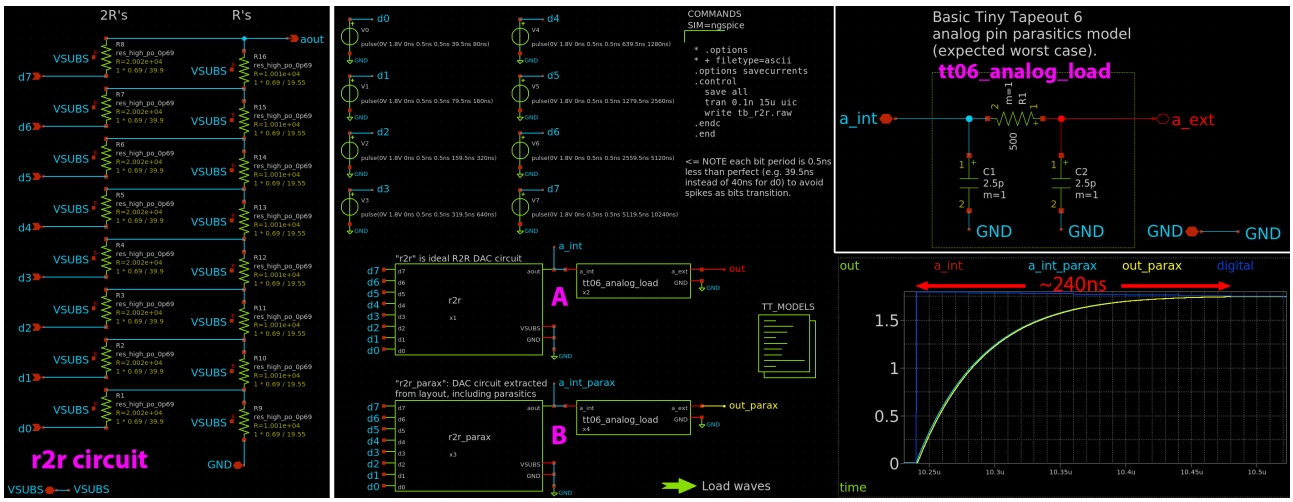


Figure 52: Combined VGA DACs schematics

mux (estimated to be about 500Ω and 5pF). This combination means their slew rate was expected to be pretty bad (at least by VGA timing standards): On the order of 240~360ns (or 6~9 horizontal pixels) going from 0V to full 1.8V. In the chips I received, it was a little better than that.

In a future design I plan to implement better internal buffering to help mitigate some of the TT analog mux load.

Select from a few simple test patterns in the VGA controller by having different `ui_in` values asserted while coming out of reset. the VGA controller digital block generates 8-bit digital outputs per each of red, green, and blue channels. These go into 3 basic RDACs to generate analog voltage outputs on `ua[2:0]` (`{B,G,R}`) in the range 0-1.8V (probably ~10kΩ impedance).

How to test

TBC!

1. Supply a 25MHz clock
2. Set `ui_in` to `8'b0001_0000`
3. Assert reset – NOTE: I didn't put a synchroniser on it, so it might (?) do a dirty reset – if that happens, it could be worked around by slowly/manually clocking around the reset pulse, I guess.
4. With a scope, trigger on the `uo_out[3]` rising edge (VSYNC) and hopefully see `ua[0]` ramp from 0V to 1.8V within 10.24us
5. With this mode (as selected in step 2 above), `ua[1]` will also ramp, but per line (instead of per pixel), as will `ua[2]` (per frame).
6. For the pretty XORs pattern, set `ui_in` to `8'b0011_0000` and assert reset again.

NOTE: For actual VGA output, you need the VSYNC and HSYNC signals (connecting them each with a 1kΩ resistor in series with their respective VGA cable connection should do). You will almost certainly need some sort of output buffering between this design and a VGA display, because the design outputs a high-impedance (~10kΩ but maybe a little worse) 0~1.8V range, while a VGA display expects 0~0.7V at 75Ω. See https://algo.org/pcb/tt06i for an example board that covers all this.

Other notes for testing:

- Digital block's mode selection is asserted via `ui_in` *during reset*
- For safety, initial test should be done with no analog output loading, and with all of `ui_in` pulled low (which selects pass-thru mode AND ensures all DAC *inputs* internally are low, so hopefully no current).
- RGB222 digital outputs compatible with the Tiny VGA PMOD.

External hardware

This is if you want to see an actual analog VGA display:

- 10MHz-capable (or better; preferably 25MHz) opamps on each of the R, G, B outputs, to both make them into low-impedance (matching 75Ω typical VGA termination), and also to level-shift from 0~1.8V to 0~0.7V. See https://algo.org/pcb/tt06i for an example "interposer board" (aka "sandwich board") which uses an OPA3355 video op-amp circuit with VGA connector, and is intended to sit between the TT06 chip's breakout board (aka "carrier") and the TT06 demo board.
- Optionally the Tiny VGA PMOD plugged into the dedicated output port (`uo_out`).

Pinout

#	Input	Output	Bidirectional
0	mode[0] / dac_in[0]	r7	vblank_out
1	mode[1] / dac_in[1]	g7	hblank_out
2	mode[2] / dac_in[2]	b7	inv_dout
3	mode[3] / dac_in[3]	vsync	dac4_in[4]
4	mode[4] / dac_in[4]	r6	dac4_in[5]
5	mode[5] / dac_in[5]	g6	dac4_in[6]
6	mode[6] / dac_in[6]	b6	dac4_in[7]
7	mode[7] / dac_in[7]	hsync	inv_in

Analog pins

ua#	analog#	Description
0	5	r_out
1	0	g_out
2	4	b_out
3	1	inv_aout
4	3	dac4_aout
5	2	

It's Alive [493]

- Author: Jonathan Anderson, Qubitbytes Ltd
- Description: plays a cool tune
- GitHub repository
- HDL project
- Mux address: 493
- Extra docs
- Clock: 100000 Hz

How it works

Why is it called It's Alive?
Because it was made within the last few days of the shuttle and only learnt Verilog within 24 hours.

If it works, it should play a cool tune over a speaker or piezo buzzer

How to test

The clock must be set to 100khz or else the speed and pitch of the song will be affected.

The reset button will restart the song.

The LED segment number will indicate the current song part, which starts at 4, finishes at 6, and loops around.
The LED segment dot, indicates song/processor activity.

If the LED segment displays `0` (zero), press the reset button.

Connect a speaker or piezo buzzer to `uio[0]` bi-directional pin (output)

```
Clock Speed: 100Khz
Reset Button: restarts tune
LED Segment: song/processor status
LED Segment == 0: press reset button
```

External hardware

speaker or piezo buzzer

Pinout

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

Analog loopback [494]

- Author: Matt Venn
- Description: Analog loopback test
- GitHub repository
- Analog project
- Mux address: 494
- Extra docs
- Clock: 0 Hz

How it works

Each pair of analog pins are shorted together.

How to test

Measure the resistance, step response through each pair to characterise the analog mux.

External hardware

n/a

Pinout

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

Analog pins

ua#	analog#	Description
0	5	connected to 1
1	0	connected to 0
2	4	connected to 3
3	1	connected to 2
4	3	connected to 5
5	2	connected to 4

BCD to single 7 segment display Converter [495]

- Author: Kelvin Kung
- Description: BCD to single 7 segment display
- GitHub repository
- Wokwi project
- Mux address: 495
- Extra docs
- Clock: 0 Hz

How it works

It uses multiple logic gates to transfrom binary input to a decimal number to display in a 7 segment display

How to test

Input any binary number between 0-9. 2^3=IN0 (most significant bit) 2^2=IN1 2^1=IN2 2^0=IN3 (least significant bit)

External hardware

For input use a dip switch with at least 4 outputs. For output use a single 7 segment display with common cathode.

Pinout

#	Input	Output	Bidirectional
0	B3	A	
1	B2	B	
2	B1	C	
3	B0	D	
4		E	
5		F	
6		G	
7			

AudioChip_V2 [514]

- Author: Thorsten Knoll
- Description: The AudioChip plays waveforms on PWM outputs. The inputs alter these waveforms in many ways.
- GitHub repository
- HDL project
- Mux address: 514
- Extra docs
- Clock: 12000000 Hz

How it works

This is an AudioChip that outputs two Audiosignals as PWM. It can be used as a audio generating device for electronic instruments, namely modular synthesizers. It is planned to build a Eurorack module for a modular synthesizer around this mikrochip. The inputs and outputs are designed to fit into the concept of such instruments. The source code of AudioChip is written in spinalHDL and generates verilog. The SpinalHDL source resides in this repository: Add link here.

How to test

Attach a lowpass filters to the PWM outputs and you get analog audio signal waveforms. The inputs alter the waveforms.

External hardware

Lowpass filters for the PWM outputs.

Pinout

#	Input	Output	Bidirectional
0	freq_bit_in_0	pwm_1_out	adsr_choice_in_0
1	freq_bit_in_1	pwm_2_out	adsr_choice_in_1
2	freq_bit_in_2		adsr_choice_in_2
3	freq_bit_in_3		adsr_switch_in
4	freq_bit_in_4		freq_bit_in_8
5	freq_bit_in_5		freq_bit_in_9

#	Input	Output	Bidirectional
6	freq_bit_in_6		freq_bit_in_10
7	freq_bit_in_7		freq_bit_in_11

Relaxation oscillator [516]

- Author: Matt Venn
- Description: A relaxation oscillator
- GitHub repository
- Analog project
- Mux address: 516
- Extra docs
- Clock: 0 Hz

How it works

In electronics a relaxation oscillator is a nonlinear electronic oscillator circuit that produces a nonsinusoidal repetitive output signal, such as a triangle wave or square wave.

The R&C have been chosen to make a ~2MHz signal. An inverter after the oscillator makes a full swing square wave.

How to test

Measure the oscillator out on pin 0 (tbc, might cause issues due to the analog mux parasitics). Measure the square wave out on digital output pin 0.

Pinout

#	Input	Output	Bidirectional
0		inverted output of oscillator	
1			
2			
3			
4			
5			
6			
7			

Analog pins

ua#	analog#	Description
0	10	analog oscillator output

Integrated Distorion Pedal [518]

- Author: Nanik Adnani
- Description: A simple distortion pedal circuit - taped out! (hopefully)
- GitHub repository
- Analog project
- Mux address: 518
- Extra docs
- Clock: 0 Hz

How it works

The design consists of a simple opamp in a non-inverting configuration. We achieve distortion simply by overdriving the opamp - because VDD is only 1.8V we very quickly run out of headroom and the transistors are pushed into the triode region which causes the output to be distorted, and usually look more square. There are 8 of these amplifiers in series, you can turn them on or bypass them by toggling inputs 0 through 7 - more amplifiers on means more distortion! In theory, we will see if it works.

How to test

You need to provide an input signal to UA[0] and an output will come from UA[1] - this unfortunately requires a little extra hardware. On the input you will need a large capacitor in series with the signal followed by a voltage divider (two large resistors of the same value) to bias in the input in the middle of the operating range of the opamp. On the output we will need to place another large capacitor in series before running it to an amp - don't connect directly to a speaker, this circuit can't drive a speaker on its own it needs to go through an amp first.

External hardware

- 2 resistors (large, same value - for a voltage divider on the input)
- 2 large capacitors (bigger the better, but around 10uf should be fine)
- 2 1/4 inch jacks (to make plugging a guitar in easy, you could probably figure it out without these)

Pinout

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

Analog pins

ua#	analog#	Description
0	11	IN
1	6	OUT

Leaky Integrate and fire neuron(LIF) [520]

- Author: Vyshnav P Dinesh
- Description: Single node of integrate and fire neuron (LIF)
- GitHub repository
- Analog project
- Mux address: 520
- Extra docs
- Clock: 0 Hz

How it works

This is an analog implementation of rate based encoding technique using leaky integrate and fire neuron.

How to test

The time varying input is applied to vin terminal, connect a threshold voltage in the Vth terminal and a rate encoded input single will produced in the Out terminal

External hardware

Function generator, CRO, Multimeter.

Pinout

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

Analog pins

ua#	analog#	Description
0	7	Out
1	9	Vth
2	8	Vin

IDAC8 based on divide current by 2 [522]

- Author: Emilian Miron
- Description: IDAC8 based on divide current by 2.
- GitHub repository
- Analog project
- Mux address: 522
- Extra docs
- Clock: 0 Hz

How it works

IDAC8 is a Digital to Analog conversion based on string of basic cells dividing the current by two repeatedly and optionally sourcing some current into the output pin.

idac1cell takes the following ports:

- VREF_IN - bias voltage for a 4/1 W/L transistor resulting in an InputCurrent
- OE - controls whether the output is enabled.
- IOUT - source current of InputCurrent/2 into here.
- VDD / VSS - power supply 1.8V
- VREF_OUT - output bias voltage for a 4/1 W/L transistor that results in InputCurrent/2.

The circuit uses current mirrors and identical transistors to divide the current by two. See tb_idac1cell.sch and SVG output for sample outputs and the schematic inside the xschem directory.

8 cells of idac1cell are chained together into one IDAC8. Essentially the OE pins correspond to digital inputs and the IOUT

How to test

- UI[7..0] are the digital inputs for currents of I, I/2, I/4, ...
- UA0 is the voltage output for the I/256 current (can be ignored.. only present for chaining or testing).
- UA1 is the current output (can hold a ~10K resistor to ground).
- UA2 is the voltage input (around 0.7V) corresponding to the high bit current.

Set UI to 0b10000000 and adjust UA0 so that the current through UA1 to the resistor is around 0.1mA. Then you can modify UI and observe the output changing into UA1 according to the 8 bit DAC output.

External hardware

- Multimeter on the UA1 10K resistor to ground.
- Potentiometer for setting UA0 bias voltage.
- set UI inputs via switches.

Pinout

#	Input	Output	Bidirectional
0	bit0	n/a	n/a
1	bit1	n/a	n/a
2	bit2	n/a	n/a
3	bit3	n/a	n/a
4	bit4	n/a	n/a
5	bit5	n/a	n/a
6	bit6	n/a	n/a
7	bit7	n/a	n/a

Analog pins

ua#	analog#	Description
0	11	VREF_OUT for current level/256
1	10	IOUT - output current
2	6	VREF_IN for current level

Analog Current Comparator [524]

- Author: Renaldas Zioma
- Description: An analog current comparator with Excitatory (+) and Inhibitory (-) currents formed by summing up digital inputs
- GitHub repository
- Analog project
- Mux address: 524
- Extra docs
- Clock: 0 Hz

How it works

A current comparator that compares 2 currents each formed by a sum of 8 digital pins.

How to test

Set the 8 inputs to form current A and 8 bi-directional inputs to form current B. The ouptut analog voltage is the result of the comparison operator (sigmoid).

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	Inhibitory current bit		Excitatory current bit
1	Inhibitory current bit		Excitatory current bit
2	Inhibitory current bit		Excitatory current bit
3	Inhibitory current bit		Excitatory current bit
4	Inhibitory current bit		Excitatory current bit
5	Inhibitory current bit		Excitatory current bit
6	Inhibitory current bit		Excitatory current bit
7	Inhibitory current bit		Excitatory current bit

Analog pins

ua#	analog#	Description
0	10	Main comparator output
1	7	Summed currents (debug)
2	9	2nd comparator input (debug)
3	8	2nd comparator output (debug)

Analog Sigmoid [526]

- Author: aleena
- Description: Activation functions for neuromorphic computing
- GitHub repository
- Analog project
- Mux address: 526
- Extra docs
- Clock: 0 Hz

How it works

ReLu and sigmoid activation functions are included

How to test

For ReLu:The output is same as the input value if it is positive and zero otherwise. For Sigmoid:Used for binary classification and predict the probability as output.

External hardware

A voltage source at analog pin 1 and analog pin 4 which is an output of an ANN

Pinout

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

Analog pins

ua#	analog#	Description
0	11	in1
1	6	out1
2	10	gnd1
3	7	in2
4	9	out2

TT06 OTP Encryptor [544]

- Author: , Aimee Kang, Alexander Schaefer
- Description: Encryption and Decryption Unit through Utilization of Psuedorandom One Time Pads
- GitHub repository
- HDL project
- Mux address: 544
- Extra docs
- Clock: 50000 Hz

How it works

8 bit data inputs come through ui_in in the form of either plaintext or ciphertext. Bit 0 of uio_in is used to determine whether the chip will perform encryption or decryption. When it is high, decryption will be performed. When it is low, decryption will be performed. However, the enable signal must be high in both cases for either encryption or decryption to be performed. In the case of encryption, the chip will take an 8 bit value from an internal pseudorandom number generator to use as a one time pad. To create the ciphertext, the chip will xor the bits of the one time pad with their relative bits in the plaintext to create ciphertext. In order to later recover the plaintext, the one time pad is stored in an internal register file, and the index of the register of which the pad is stored in is outputed to bits 6 through 4 of uio_out. There are 8 registers in the register file (0-7). To decrypt ciphertext, the decrypt signal must be high (bit 0 of uio_in) and the index that the associated one time pad is stored in must be inputted to uio_in bits 3 through 1. Then, the one time pad will be recovered from the indexed register, the pad will be xored with the ciphertext, and the plaintext will be produced as output.

How to test

Testing can be performed by ensuring that inputted plaintext can be recovered by taking the encrypted output and register index and feeding it into the system.

External hardware

External hardware with basic memory, wiring, and data displaying functionality should be suitable to test this chip. Verilog testing was performed by using one external register for data output, one external register for index output, and a means to read the data from the registers. Basic binary instrumentation may be needed if direct

access to wires is not possible in order to shift the register index from the output bits of 6 through 4 to the input bits of 3 through 1.

Pinout

#	Input	Output	Bidirectional
0	data[0]	out[0]	decrypt
1	data[1]	out[1]	r_num[0]
2	data[2]	out[2]	r_num[1]
3	data[3]	out[3]	r_num[2]
4	data[4]	out[4]	index_out[0]
5	data[5]	out[5]	index_out[1]
6	data[6]	out[6]	index_out[2]
7	data[7]	out[7]	

Convertidor de Tiempo a Digital (TDC) [545]

- Author: Juan Vargas Ferrer & Luis Carlos Alvarez Simón
- Description: El proyecto consiste en el diseño de un circuito Front end o interfaz para convertir a digital la señal proveniente de un sensor con salida en tiempo.
- GitHub repository
- HDL project
- Mux address: 545
- Extra docs
- Clock: 50000000 Hz

Convertidor de Tiempo a Digital (TDC)

How it works

El proyecto consiste en el diseño de un circuito Front end o interfaz para convertir a digital la señal proveniente de un sensor con salida en tiempo. La industria nos proporciona un sinfín de sensores para medir o monitorear diferentes variables físicas, dichos sensores pueden proporcionar su señal en diferentes formas; voltaje, corriente, frecuencia, tiempo (ancho de pulso), entre otras. El bloque que se propone se enfoca en la conversión de tiempo (definido entre el flanco de subida y bajada de un pulso) a un formato digital, también conocidos como circuitos **TDC (Time to Digital Converter)**, para posteriormente enviarlo vía RS232 para que pueda ser monitoreado en una PC o dispositivo compatible con el protocolo RS232.

En la *figura 1* se muestra el diagrama a bloques del sistema, como se observa se compone de un bloque llamado **contadorTDC**, el cual se encarga de realizar el conteo una vez que se recibe un pulso de entrada, cuando el pulso finaliza se guarda el dato generado en un registro y posteriormente se envía en al exterior en forma serial mediante el bloque **RS232_TX**. El bloque RS232_TX funciona a una velocidad de 9600 baudios con 8 bits de datos y paridad impar, de tal manera que en la PC o dispositivo usado para visualizar la información debe configurarse de la misma manera. El funcionamiento del sistema está controlado por la **máquina de estados**, cuyo funcionamiento es de la siguiente manera: cuando se recibe el flanco de subida del pulso de entrada el contador se activa y, al finalizar el pulso se pasa a un estado que hace que se almacene el valor del contador, posteriormente este valor es enviado al transmisor para que sea enviado de manera serial por el módulo **RS232_TX**, finalmente el sistema regresa al estado inicial para poder recibir un nuevo pulso.

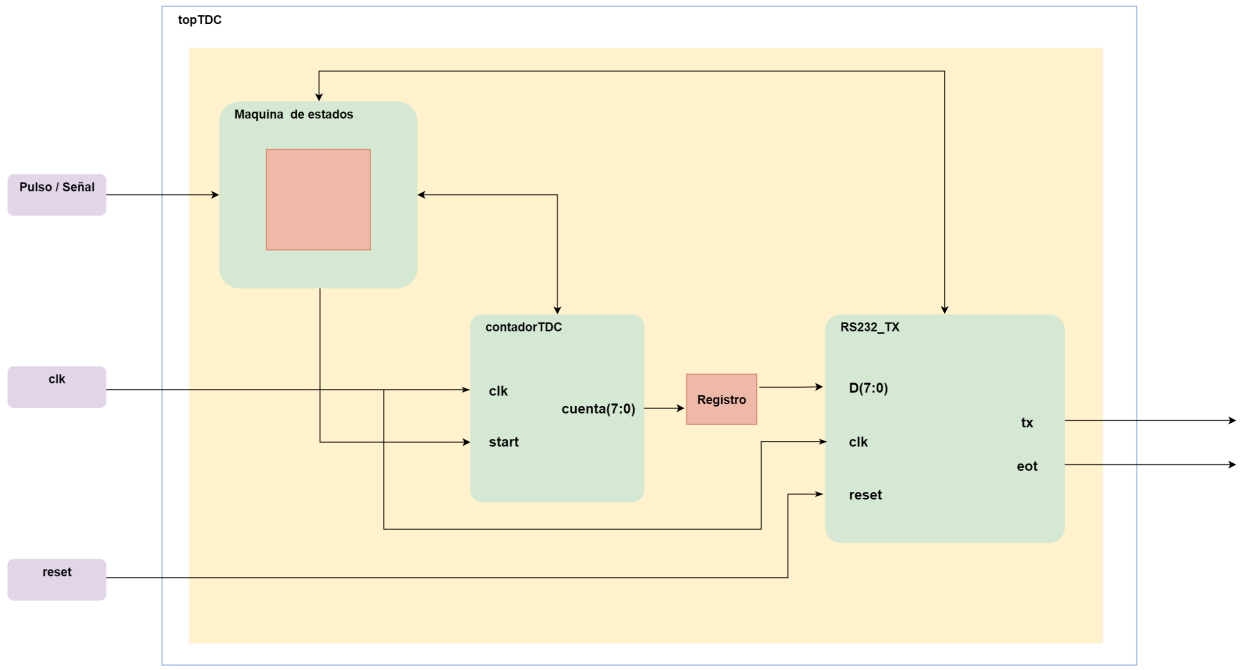


Figura 1. *Diagrama a bloques del TDC.*

[!NOTE] En la *figura 2* se muestra el diagrama de conexión del sistema con los pines del Frame del chip y a continuación se proporciona una descripción de las señales en cada una de ellas.

- **clk** -> **clk**.- Señal de reloj del sistema, la cual será de 50MHz.
- **in_out[7]** -> **Pulso/señal**.- En este pin de entrada se introducirá el pulso de entrada que será convertido a digital por el sistema. El ancho del pulso debe estar entre 40ns a 5.1us. Este pulso puede ser proporcionado por un *generador de funciones* o la señal proveniente de un *sensor en forma de pulsos*.
- **in_out[6]** -> **reset**.- Este pin de entrada produce el *reset del módulo transmisor*, el cual debe ser activado para inicializar dicho múdulo, se recomienda conectar un *push button* configurado en pull down.
- **in_out[1]** -> **tx**.- Pin de salida por el cual se envía el dato digital de manera serial, dicho dato corresponde al valor digital del ancho del pulso de entrada, con una resolución igual al periodo de la señal de reloj. Para poder capturar el dato en una computadora mediante un monitor serial, se deberá conectar este pin a la terminal RX del módulo RS232 que recibirá el dato; el cual deberá tener la siguiente configuración:
- Baud rate: 9600

- Data bits: 8 bits
- Paridad: impar (odd)

- **in_out[0]** -> **eot**.- Pin de salida que permite monitorear la bandera que indica el final de una transmisión, en este pin puede ser conectado un *led*. Sin embargo, debido a la velocidad de transmisión éste será casi imperceptible, por lo que es opcional. Aún así podemos conectar un *osciloscopio* para su mejor visualización.

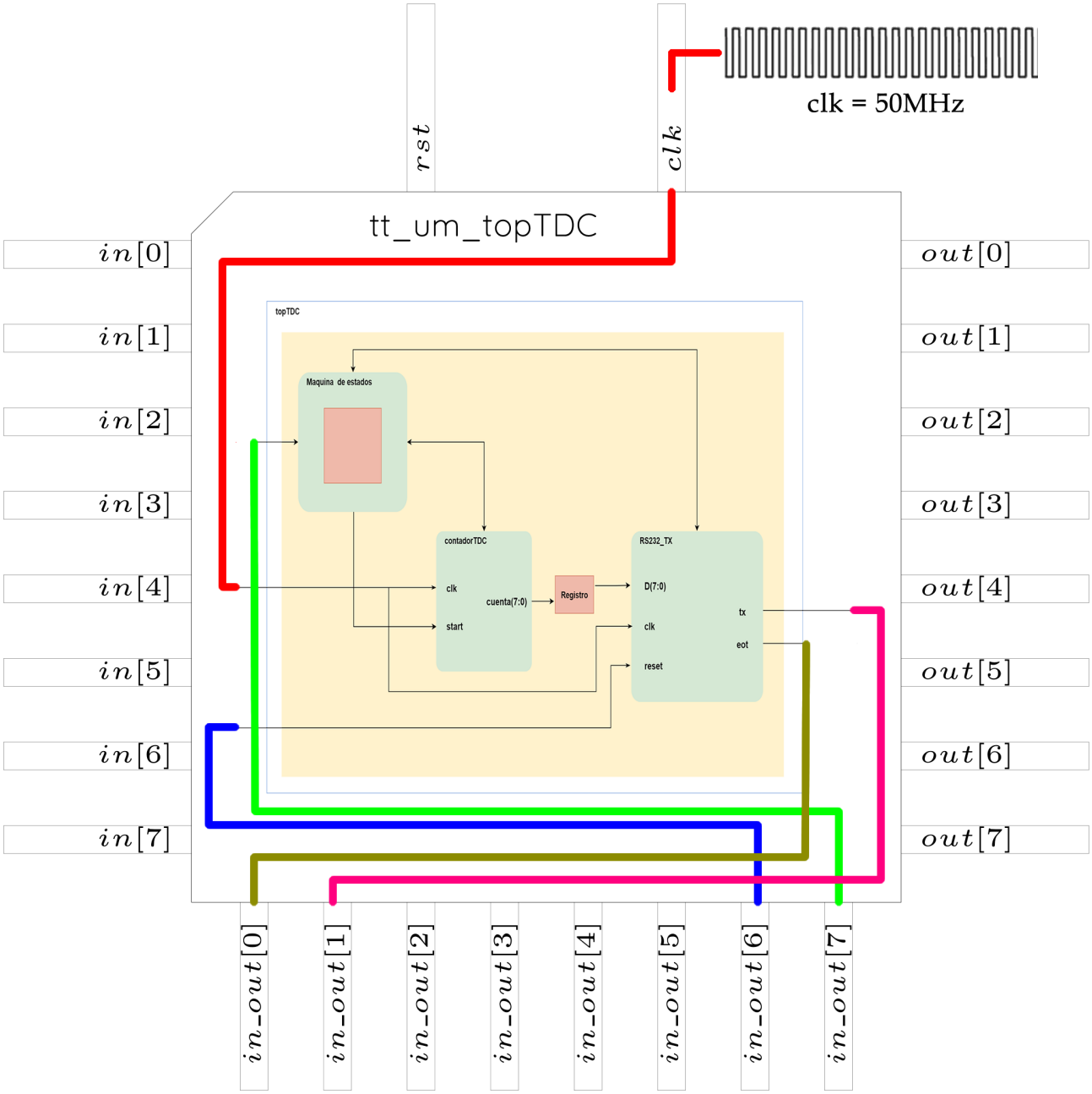


Figura 2. *Diagrama a bloques del TDC dentro del área defindа.*

How to test

Las pruebas se realizaron en **Modelsim** en su versión gratuita, para ello se hizo una adecuación generando un pequeño modulo PWM dentro de la aplicación para simular lo que sería la señal de un sensor, para este caso se opto por generar cuatro valores de PWM los cuales generan cuatro valores distintos que se transmiten por RS232, en la *figura 3* se muestra la primer combinación de PWM que corresponde a una combinación 00 y que genera un valor binario 00010011.

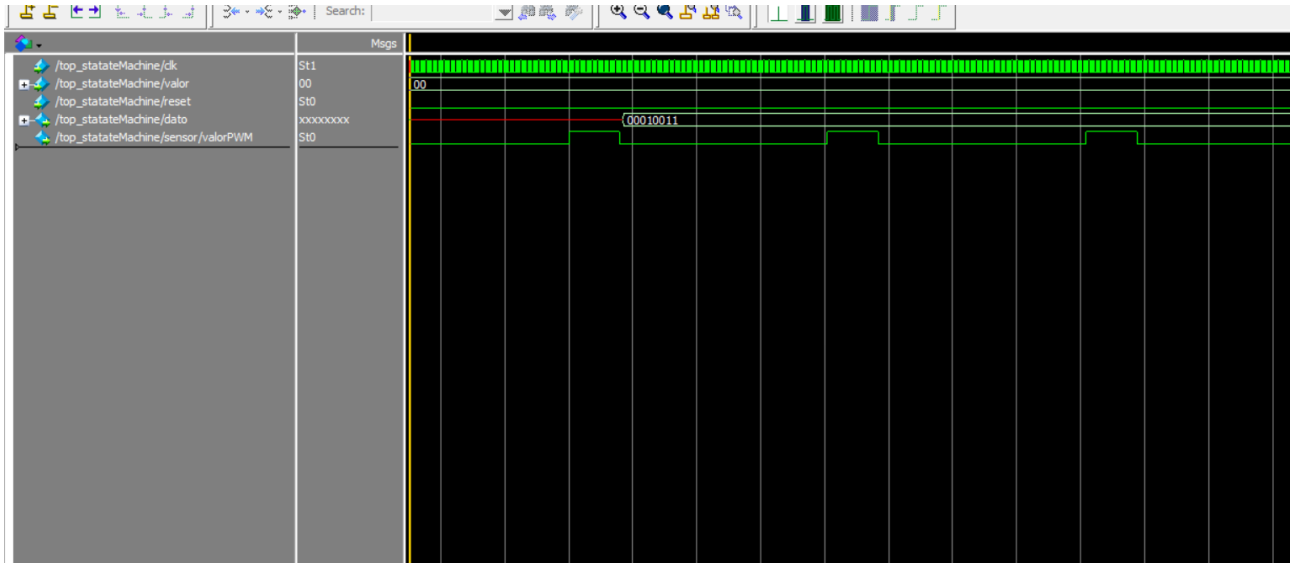


Figura 3. *Combinación 00 que genera un valor binario 00010011.*

El siguiente valor de prueba fue la combinación 01 la cual genero un valor binario 00100111 y dicha simulación se puede observar en la *figura 4*, en dicha figura se puede observar como cambia el ancho de pulso que hace que se genere dicho valor binario.

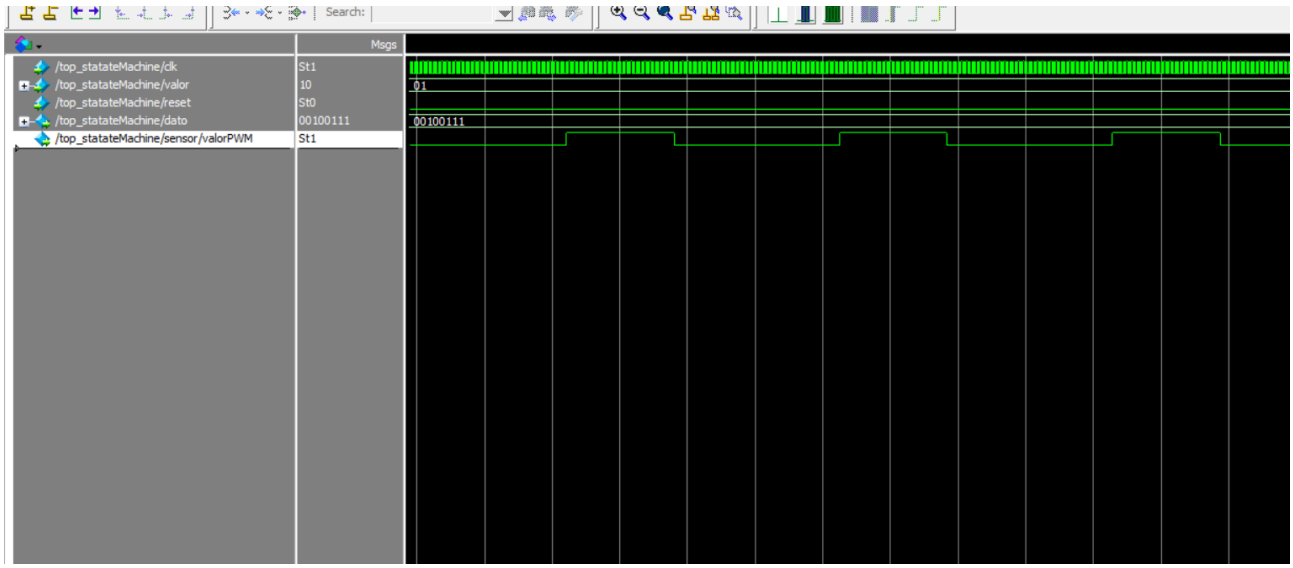


Figura 4. *Combinación 01 que genera un valor binario 00100111.*

A continuación el siguiente valor de prueba fue la combinación 10 la cual genero un valor binario 00111011 y dicha simulación se puede observar en la *figura 5*, en dicha

figura se puede observar como cambia el ancho de pulso que hace que se genere dicho valor binario.

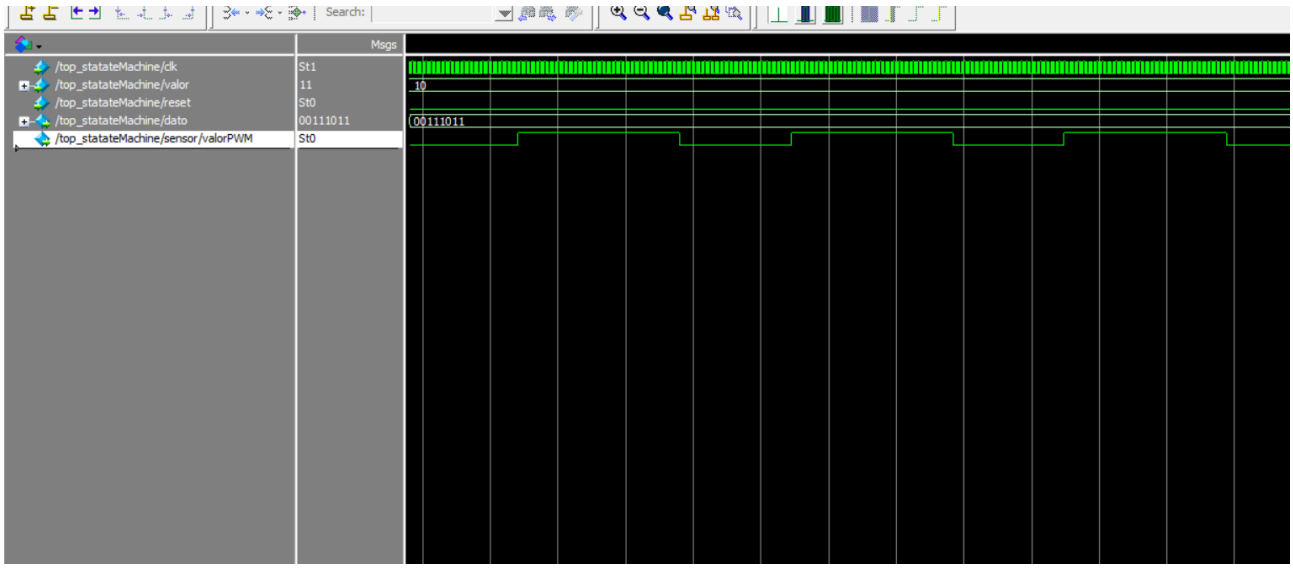


Figura 5. *Combinación 10 que genera un valor binario 00111011.*

Para finalizar la última combinación 11 genero un valor binario 01001111 y dicha simulación se puede observar en la *figura 6*, en dicha figura se puede observar al igual que las anteriores como cambia el ancho de pulso que hace que se genere dicho valor binario.

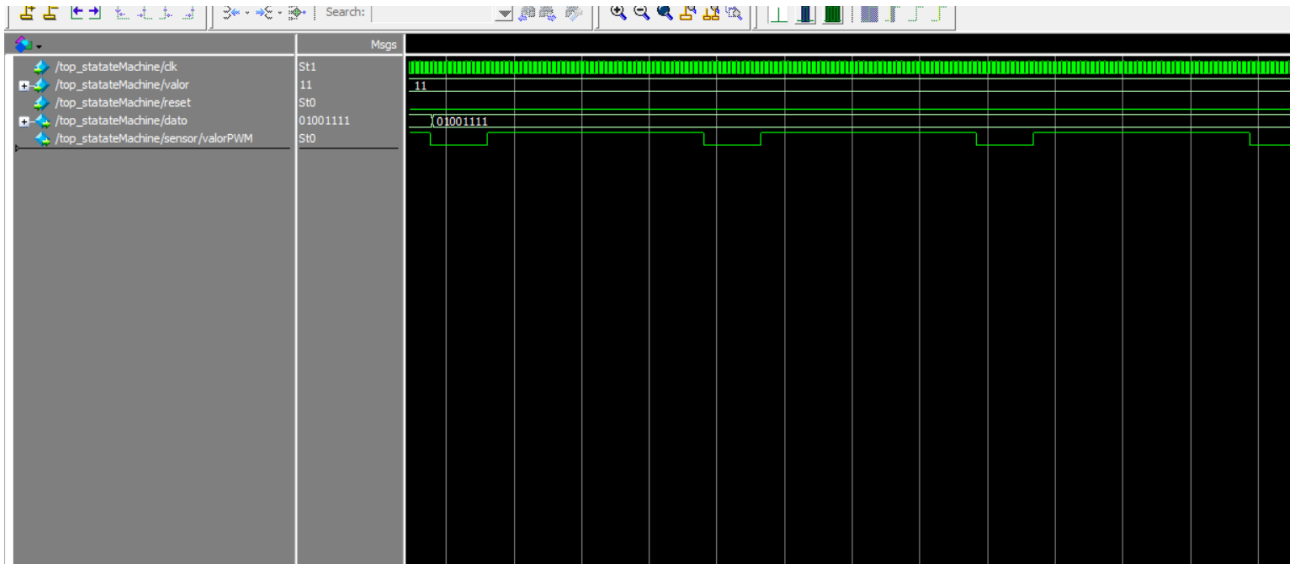


Figura 6. *Combinación 11 que genera un valor binario 01001111.*

Para finalizar la etapa de pruebas se opto por realizar una prueba en una **tarjeta de desarrollo AMIBA 2**, la cual cuenta con un FPGA Spartan 6 XC6SLX9, 216/576 Kb de Block RAM, un oscilador de 50 MHz, convertidor USB/RS232 (FTDI FT2232HL), leds de propósito general, switch de dos posiciones de propósito general, etc. En el siguiente enlace se podrá observar un video en el cual se muestran las distintas combinaciones simuladas anteriormente y además se puede ver el valor enviado por el puerto serial, el cual es monitoreado mediante la aplicación **Serial Debug Assistant**,

como recurso extra se hizo uso de los leds de propósito general como apoyo para poder visualizar el valor generado y a su vez poder ver este valor en el monitor serial, que en nuestro caso se muestra en hexadecimal corroborando lo generado con lo enviado.

Pinout

#	Input	Output	Bidirectional
0			eot
1			tx
2			
3			
4			
5			
6			reset
7			stop

SynchMux [546]

- Author: bc2kaneda
- Description: A 2 bit synchronous mux with output enable
- GitHub repository
- Wokwi project
- Mux address: 546
- Extra docs
- Clock: 0 Hz

How it works

If OE == 0 then Y[0:1], YP, Z[0:1] and ZP are in a high-z state.

Output Y[0:1] = (SEL == 0) ? A[0:1] : B[0:1] Output Z[0:1] = (SEL == 0) ? ~A[0:1] : ~B[0:1]

Output YP and ZP are the parity of Y and Z respectively.

POPCNT_Y[0:1] and POPCNT_Z[0:1] are the population count of {Y[0:1],YP} and {Z[0:1],ZP} respectively. These pins are never in high-z state.

Pinout

#	Input	Output	Bidirectional
0	CLK	POPCNT_Y0	Y1
1	RST	POPCNT_Y1	Y2
2	A1	POPCNT_Z0	YP
3	A2	POPCNT_Z1	Z1
4	B1		Z2
5	B2		ZP
6	SEL		
7	OE		

3-bit ALU [547]

- Author: José Raña Gámez
- Description: This device is a 3-bit ALU that generates 5 operations in parallel. The operations that the ALU performs are: addition, subtraction, multiplication, division and modulo operation. The device has 2 inputs; A[3-bit] and B[3-bit] along with a 3-bit selector (Selector[3-bit]). It also has a single 6-bit output (OutPut[6-bit]). In the end, the design entails 15 pins in total; 9 input and 6 output pins. The operation of this ALU is simple: At input A and B, the values are set using switches, for example, A= 111 and B=101. To obtain the 5 different results through the different 5 operations that the ALU performs, the 3-bit selector (Selector[3-bits]) is used, therefore, using 3 switches we will place the result that we want to observe at the output.
- GitHub repository
- HDL project
- Mux address: 547
- Extra docs
- Clock: 0 Hz

How it works

This device is a 3-bit ALU that generates 5 operations in parallel. The operations that the ALU performs are: addition, subtraction, multiplication, division and modulo operation. The device has 2 inputs; A[3-bit] and B[3-bit] along with a 3-bit selector (Selector[3-bit]). It also has a single 6-bit output (OutPut[6-bit]). In the end, the design entails 15 pins in total; 9 input and 6 output pins. The operation of this ALU is simple: At input A and B, the values are set using switches, for example, A= 111 and B=101. To obtain the 5 different results through the different 5 operations that the ALU performs, the 3-bit selector (Selector[3-bits]) is used, therefore, using 3 switches we will place the result that we want to observe at the output. The selector works as follows: since the selector is 3-bit, we will have 2^3= 8 combinations, but in this case we will only use the first 5 combinations in this way:

Combination 1 for sum: 000

Combination 2 for subtraction: 001

Combination 3 for multiplication: 010

Combination 4 for division: 011

Combination 5 for module: 100

The remaining combinations do not have an assigned operating function, therefore, the remaining combinations will not generate any result at the output of the device.

How to test

Choose two values of 3-bits for the inputs A and B by using switches, for example: A=111 and B=101. Then, use the 3-bit selector that has 3 switches to choose one of the 5 combinations to select an operation (sum[000], substraction[001], multiplication[010], division[011] and module[100]) so that it can be obtained the wanted results observed at the output (Leds).

For example: we choose the values A=111 and B=101. Then, if the 3-bit selector has the combination 011 then the operation will be A=111 / B=101 (division).

Another example, we choose the values A=111 and B=101. Then, if the 3-bit selector has the combination 010 then the operation will be A=111 * B=101 (multiplication).

Another example, we choose the values A=111 and B=101. Then, if the 3-bit selector has the combination 000 then the operation will be A=111 + B=101 (sum).

The results will be shown at the 6-bit output that uses 6 Leds to demonstrate the results of any of the 5 operations available in the ALU.

External hardware

3-bit input "A": uses 3 switches.

3-bit input "B": uses 3 switches.

3-bit input "Selector": uses 3 switches.

6-bit output "Results": uses 6 Leds.

The mentioned inputs and outputs are respectively connected to the pins of the project circuit as follows:

Inputs

ui[0]: "First bit for input 'A'(input of 3-bits)"

ui[1]: "Second bit for input 'A'(input of 3-bits)"

ui[2]: "Third bit for input 'A'(input of 3-bits)"

ui[3]: "First bit for input 'B'(input of 3-bits)"

ui[4]: "Second bit for input 'B'(input of 3-bits)"

ui[5]: "Third bit for input 'B'(input of 3-bits)"

ui[6]: "Unused input bit"

ui[7]: "Unused input bit"

Outputs

uo[0]: "First bit for output 'Leds'(output of 6-bits)"

uo[1]: "Second bit for output 'Leds'(output of 6-bits)"

uo[2]: "Third bit for output 'Leds'(output of 6-bits)"

uo[3]: "Fourth bit for output 'Leds'(output of 6-bits)"

uo[4]: "Fifth bit for output 'Leds'(output of 6-bits)"

uo[5]: "Sixth bit for output 'Leds'(output of 6-bits)"

uo[6]: "Unused output bit"

uo[7]: "Unused output bit"

Bidirectional pins

uio[0]: "First bit for input 'ctrl'(input of 3-bits)"

uio[1]: "Second bit for input 'ctrl'(input of 3-bits)"

uio[2]: "Third bit for input 'ctrl'(input of 3-bits)"

uio[3]: "Unused bidirectional I/O bit"

uio[4]: "Unused bidirectional I/O bit"

uio[5]: "Unused bidirectional I/O bit"

uio[6]: “Unused bidirectional I/O bit”

uio[7]: “Unused bidirectional I/O bit”

For a better visualization, see Figure 1.

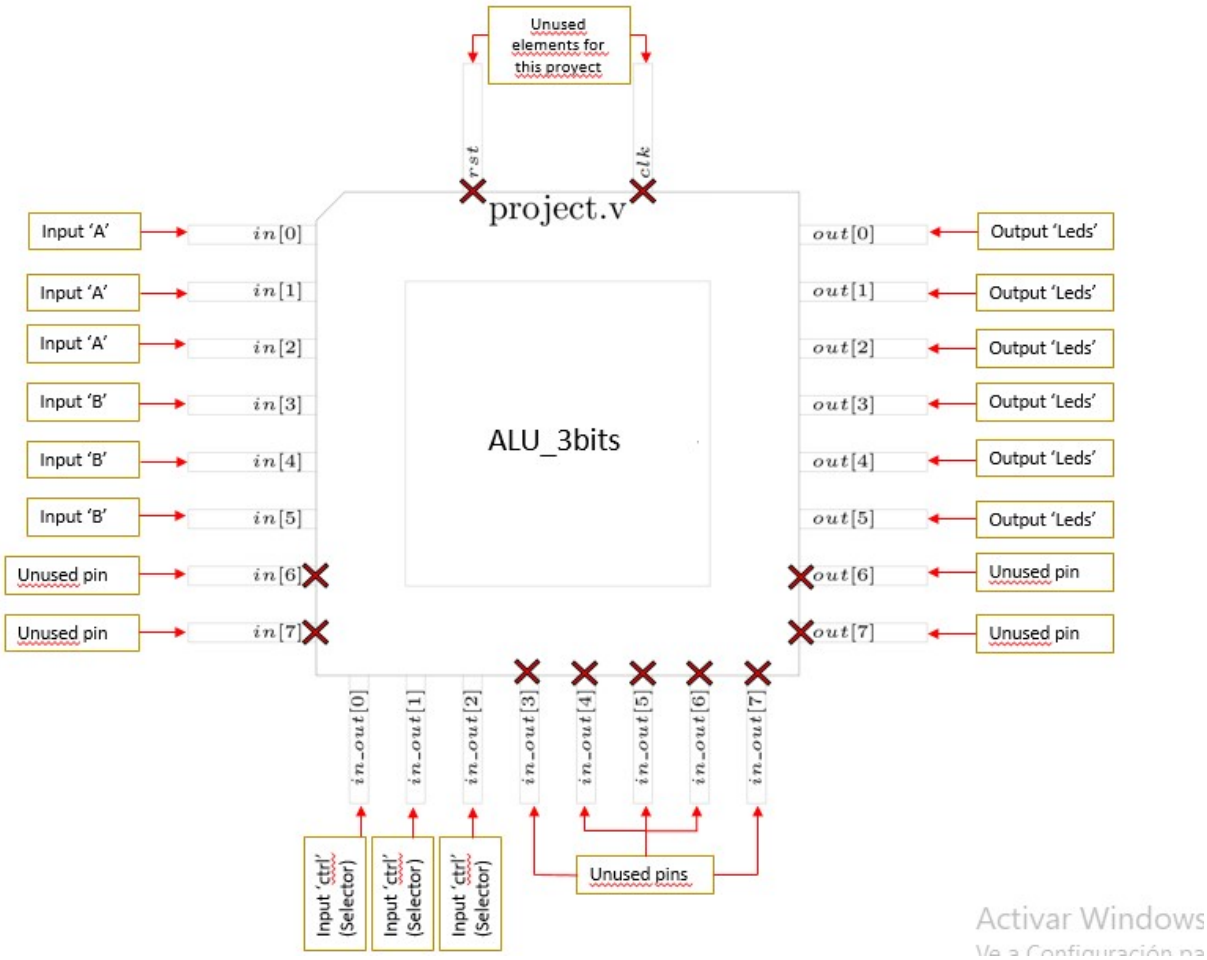


Figure 53: ALU_3bits

Figure 1: ‘External Hardware pins conections visualization’

Pinout

#	Input	Output	Bidirectional
0	First bit for input ‘A’(input of 3-bits)	First bit for output ‘Leds’(output of 6-bits)	First bit for input ‘ctrl’(input of 3-bits)

#	Input	Output	Bidirectional
1	Second bit for input 'A'(input of 3-bits)	Second bit for output 'Leds'(output of 6-bits)	Second bit for input 'ctrl'(input of 3-bits)
2	Third bit for input 'A'(input of 3-bits)	Third bit for output 'Leds'(output of 6-bits)	Third bit for input 'ctrl'(input of 3-bits)
3	First bit for input 'B'(input of 3-bits)	Fourth bit for output 'Leds'(output of 6-bits)	Unused bidirectional I/O bit
4	Second bit for input 'B'(input of 3-bits)	Fifth bit for output 'Leds'(output of 6-bits)	Unused bidirectional I/O bit
5	Third bit for input 'B'(input of 3-bits)	Sixth bit for output 'Leds'(output of 6-bits)	Unused bidirectional I/O bit
6	Unused input bit	Unused output bit	Unused bidirectional I/O bit
7	Unused input bit	Unused output bit	Unused bidirectional I/O bit

8-bit Binary Counter [548]

- Author: Aryan kannaujiya, Shivam Bhardwaj and Ambika Prasad Shah
- Description: This Verilog module defines a synchronous 8-bit counter, where the count increments on each rising edge of the clock input (clk). Additionally, it features an asynchronous reset input (rst_n), which,0 when activated, sets the counter output (out) to zero regardless of the clock signal.
- GitHub repository
- HDL project
- Mux address: 548
- Extra docs
- Clock: 50000000 Hz

How it works

The Verilog module implements a 8 bit binary counter with clock (`clk`), reset (`rst_n`), up count (`ui_in[2]`), down count(`ui_in[3]`), hold (`ui_in[4]`) ,output pins for binary (`out`), hexa decimal (`hex`) and decimal (`dec`). Upon a clock rising edge or reset assertion, it resets the output to 0 or increments it by 1, respectively. This design facilitates counting operations in digital systems, maintaining a 8-bit output range.

How to test2

We test it on Vivado and open sources (OpenROAD and OpenLane).

External hardware

defaults

Pinout

#	Input	Output	Bidirectional
0	clk	out	
1	rst_n	hex	
2	ui_in[2]	dec	
3	ui_in[3]		
4	ui_in[4]		
5			

#	Input	Output	Bidirectional
6			
7			

SumLatchUART_System [549]

- Author: Gilberto Ramos Valenzuela
- Description: 4 bit adder
- GitHub repository
- HDL project
- Mux address: 549
- Extra docs
- Clock: 50000000 Hz

How it works

The system operates by receiving a 4-bit input and storing it in one of two registers, designated as Register A or Register B. Following this, the Arithmetic Logic Unit (ALU) receives a 4-bit operation selection code which dictates the specific operation to be executed on the input data. These operations can include addition, subtraction, bitwise AND, bitwise OR, and other logical or arithmetic operations depending on the design of the ALU.

Once the operation is performed, the output is routed to a Universal Asynchronous Receiver-Transmitter (UART) transmitter. The UART transmitter facilitates the communication of the result to either a microcontroller or a standalone UART interface. This allows for seamless integration with larger systems or external devices, enabling the processed data to be utilized for various applications.

How to test

Hardware Components To test the hardware, you will need the following components:

1. Two push buttons with pull-up resistors, used for saving data to Register A and Register B respectively.
2. Eight switches, designated for Data_input and OP_select operations.
3. One LED indicator to signify the functioning of the Arithmetic Logic Unit (ALU).
4. An output pin configured to transmit the Tx signal.
5. A microcontroller or UART-capable device operating at a baud rate of 9600.

To conduct testing, you'll need to connect a 50MHz clock signal to the clk pin. Begin by selecting operations according to the Operation table provided in the README section of this repository. The operands to be saved in registers range from 0000 to 1111, corresponding to decimal values 0 to 15.

Given the utilization of an 8-bit output signal in the block diagram, no overflow is expected for most operations. However, when using the multiplication Op code, it's important to note that the maximum numbers to be multiplied are 1111 times 1111, resulting in 11100001, which equals 255 in decimal.

External hardware

1. LED for UARTBUSY indicator.
2. UART resiver to get data out.
3. 2 push buttons with pull-up resistor.
4. 50MHz ocilator or function generator

Pinout

#	Input	Output	Bidirectional
0	data_input [0]		clk
1	data_input [1]		reset_n
2	data_input [2]		save_a_n
3	data_input [3]		save_b_n
4	Op_select [4]		uart_tx_en
5	Op_select [5]		uart_txd
6	Op_select [6]		uartbusy
7	Op_select [7]		

Power Management IC [550]

- Author: Matthew Wong
- Description: Creates a half bridge PWM duel output from ADC input
- GitHub repository
- HDL project
- Mux address: 550
- Extra docs
- Clock: 10000000 Hz

How it works

This uses a set of state machines to generate 2 ADC controlled pwms. A heartbeat signal periodically pings the ADC to update the pwm's duty cycle.

The heartbeat sends the conversion start (convStart) high signal to begin the conversion. The ADC sets up and then replies with a busy high when the ADC is ready to be read. When the busy high is read, the chip responds with a read and chip select (rd_cs) which are tied together. The parallel 8bit conversion is sent to the chip. See AD7819YNZ datasheet for details on ADC conversion. The conversion is Mode 2.

After the ADC is read, the duel pwms' duty cycles are updated. 0 is the min voltage and 255 is the max voltage. The duel pwms are 180 deg out of phase and should never overlap. Otherwise this could lead to shoot-thru which could destroy the FETs. A dead zone was built into the state machine to prevent this overlap.

How to use

After reset, the syncRectifierLs and syncRectifierHs outputs will produce a pwm signal based on the 8bit parallel ADC input. You need to build the circuit shown in the PMIC.png. You could also just hook up an oscilloscope to the syncRectifierLs and syncRectifierHs and see the 180 deg out of phase square waves.

External hardware

https://www.analog.com/media/en/technical-documentation/data-sheets/ad7819.pdf (AD7819NZ 8-bit parallel output ADC) https://www.digikey.com/en/htmldatasheets/production (MCP14700 Highside Driver)

Future Versions

I plan to build the ADC internally so I'm not tying up 8 GPIO pins with the ADC parallel output. I also would like to implement current sensing, current feedforward, prebiasing, soft-switching, PID and other more advanced features if I have time.

Pinout

#	Input	Output	Bidirectional
0	adcVoltage[0]	convStart	busy
1	adcVoltage[1]	rd_cs	
2	adcVoltage[2]	syncRectifierLs	
3	adcVoltage[3]	syncRectifierHs	
4	adcVoltage[4]		
5	adcVoltage[5]		
6	adcVoltage[6]		
7	adcVoltage[7]		

BIT COMPARATOR [551]

- Author: FELIPE SD
- Description: Compare two bits
- GitHub repository
- Wokwi project
- Mux address: 551
- Extra docs
- Clock: 0 Hz

How it works

This test consist in compare two bits determinating wich one is bigger or are equals

How to test

To testo the project just insert 2 input bits (A and B) and then make a verification with the possible inputs 00 01 10 11 you should check the three outputs, fist one when both are equal, and second and thrid when any other input is bigger. OUTPUT 1 = 1 when equals OUTPUT 2 = 1 when bigger OUTPUT 3 = 1 when bigger

External hardware

Use leds to check the output Used input with any desired devices.

Pinout

#	Input	Output	Bidirectional
0	BIT A	OUTPUT A	
1	BIT B	OUTPUT B	
2		OUTPUT C	
3			
4			
5			
6			
7			

2 bit Binary Calculator [552]

- Author: Nikhil Jindal
- Description: This is a 2 bit calculator that can multiply, subtract or even add 2 bit binary numbers.
- GitHub repository
- Wokwi project
- Mux address: 552
- Extra docs
- Clock: 0 Hz

How it works

This is a 2 bit calculator that can multiply,add or subtract 2 bit binary numbers.

How to test

In this, user enters 2 numbers A and B of 2 bit each and select the operation at a time and it calculates the answer and display it on 7 segment display.(A should be greater than B to check subtract if not then do 2's compliment)

External hardware

Dip switch and 7 segment display screen.

Pinout

#	Input	Output	Bidirectional
0	A0	a	
1	A1	b	
2	B0	c	
3	B1	d	
4	Multiply	e	
5	Sum	f	
6	Subtract	g	
7			

IFSC Keypad Locker [553]

- Author: Roddy Romero; Gabriel Mota; Luis Davi Paganella; Vinicius Westphal De Paula;
- Description: 4 digit Locker of an ordinary 4x4 contact keypad (based on Arduino keypad)
- GitHub repository
- Wokwi project
- Mux address: 553
- Extra docs
- Clock: 100 Hz

How it works

The circuit consists of 8 input and output pins. 4 input and output pins are reserved for the 4 x 4 matrix keypad. The rows are connected to the output (0,1,2,3) while the columns are connected to the inputs (3,4,5,6). The circuit is powered at input pin 0. A high clock signal is recommended to prevent delays when keys are pressed, so the frequency was set as 100 Hz. Pins 1 and 2 are used for setting and resetting the circuit's flip-flops. This circuit does not require setting any flip-flops, so this input should be grounded. The Reset should be connected to a button because every time the circuit is started for the first time, the flip-flops may start with random states, which can impede the correct operation of the circuit. Therefore, the button is used to reset it the first time it is used and to be able to register a new password. Output pin 6 is the signal indicating whether a password is registered. Output pin 5 indicates the state of the lock; if the entered password matches the registered password, the signal is positive. Pin 7 is a verification signal indicating that the circuit is operating correctly.

Basic Operating Principle:

The circuit must receive a signal from a 4x4 matrix contact keypad to register and receive password attempts. For this, the circuit must energize each row of the keypad so that when a button is pressed, the contact of that row x and column y sends a signal (x,y) to the circuit. Additionally, the circuit must automatically register 4 digits and then enter the "password attempt" mode, where all subsequent digits are used to attempt to enter the password.

Therefore, there are 8 registers to receive the Row x Column coordinates of the keypad. Since the password consists of 4 digits, there are 4 sets of these 8 registers connected in series. There are two main states: "Register password" and "Password attempt". Each state has 32 registers, grouped into 4 sets in series, each with 8 registers in parallel. All registers are connected with the clock in parallel.

How to test

After turning on the circuit and restarting it for the first time, the circuit has 3 logical operating states: No key pressed, Key pressed, Key released.

When no key is pressed, the clock signal is sent in parallel to 4 registers in series (Shift registers). These registers sequentially transfer only one positive signal in a loop. This allows the circuit to energize only one row of the keypad at a time.

When a key is pressed, the signal travels through the 4 registers until it reaches the respective row that had contact with the column. When this row is reached, the closed contact of this row with the column of the pressed key energizes that column. The column sends the signal back to the circuit, which triggers a clock gating that blocks the clock signal to the registers. This way, the shift circuit is "paused" while the column is energized. To ensure that only one row is connected, there is a verification step. If it is confirmed that only one row is connected and the column is activated, a "button pressed" flip-flop is set. This flip-flop serves as a small delay to allow a clock step for the password registers. This permission is achieved with an AND gate, which connects the delay flip-flop and a control flip-flop, responsible for controlling the "password attempt" and "password registration" states. While no password is registered, the circuit first feeds the "password registration" registers.

When the key is released, the column is de-energized, and then the "button pressed" flip-flop is automatically reset, sending a low signal to the clock of the registers. Thus, when another key is pressed, the clock of the registers goes to the rising edge, and the coordinates data are shifted until reaching the 4th and last register.

The last register of the "Password Register" sends a signal to the state control flip-flop, which is then set and starts blocking any clock from that set of registers. This activates the "Password Attempt" mode, where now the clock step is only for this other set of registers. When the same 4 digits are pressed in the exact order as those registered, a high signal is sent at the output, indicating that the password is correct.

External hardware

- 1 Arduino 4x4 Matrix membrane Keyboard
- 1 Step Button
- Couple of LEDs

Pinout

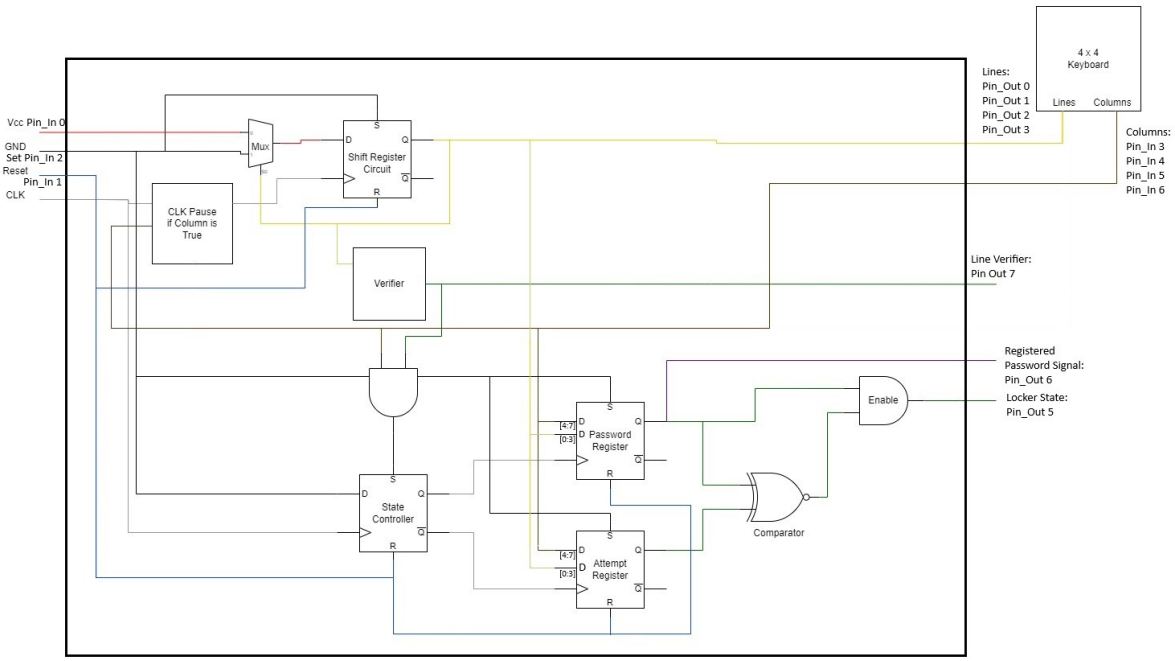


Figure 54: Diagram

#	Input	Output	Bidirectional
0	VCC	Keypad Row pin 1	
1	RESET	Keypad Row pin 2	
2	GND	Keypad Row pin 3	
3	Keypad Column pin 1	Keypad Row pin 4	
4	Keypad Column pin 2		
5	Keypad Column pin 3	Locker State (LED 1)	
6	Keypad Column pin 4	Registered Password Signal (LED 2)	
7		Line Verifier (LED 3)	

Measurement of CMOS VLSI Design Problem 4.11 [554]

- Author: Eric Smith
- Description: Measure the delay of each design in the problem with varying load. See project Readme for details.
- GitHub repository
- HDL project
- Mux address: 554
- Extra docs
- Clock: 0 Hz

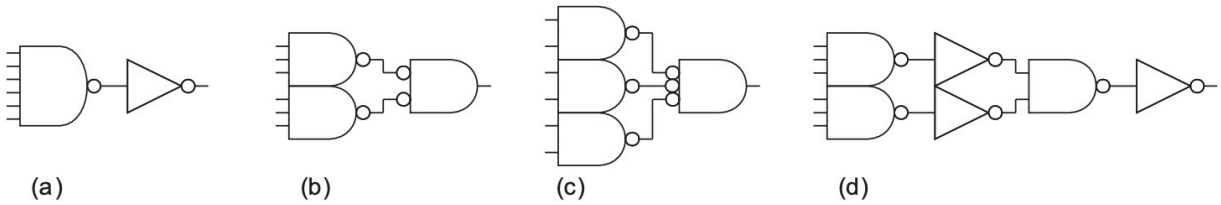
Project

Measure the delay of the implementations of problem 4.11 in Weste & Harris

https://pages.hmc.edu/harris/cmosvlsi/4e/index.html

Problem 4.11

Consider four designs of a 6-input AND gate shown below. Develop an expression for the delay of each path if the path electrical effort is H. What design is fastest for H = 1? For H = 5? For H = 20?



For this problem, H measures the capacitance the AND gate needs to drive. It's the output capacitance divided by the input capacitance of one inverter. So H=20 means the output capacitance is the same as 20 inverters.

For an example of when this could occur, consider this AND gate as a row decoder, and H is the number of columns.

The Theory of Logical Effort

For more details see

- Weste & Harris Chapter 4 Section 4 (4th Edition)

- https://www.ece.ucdavis.edu/~vojin/CLASSES/EPFL/Papers/LE-orig-paper.pdf
- https://shop.harvard.com/book/9781558605572

This is the linear delay model and the basic theory is the delay of a gate can be determined by the equation:

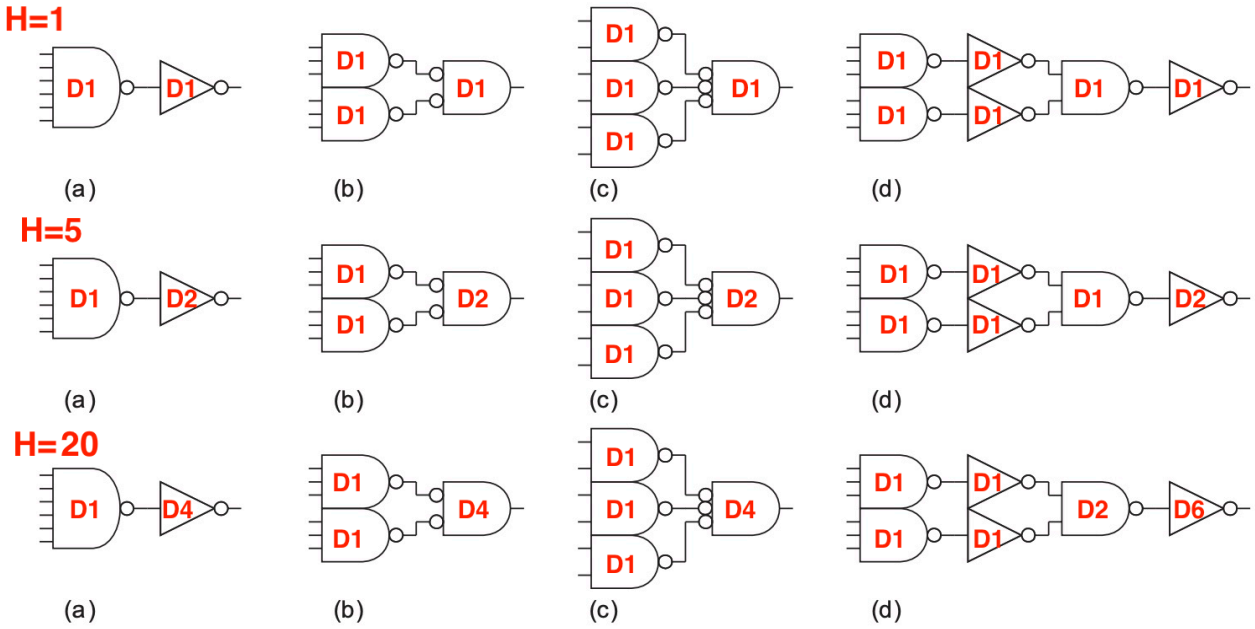
$D = G * H + P$

Variable	Name	Description
H	Electrical Effort	The amount of output capacitance this gate needs to drive relative to the input capacitance of an inverter.
G	Logical Effort	Logical Effort is a rough measure of the gate's complexity. It can be thought of as the amount of input capacitance relative to an inverter with equal drive strength, the amount of drive strength when the input capacitance is the same as an inverter, or the slope of the fanout line. More "complex" gates will have higher logic efforts.
P	Parasitic Delay	Parasitic delay measures the output capacitance of this gate relative to the output capacitance of an inverter of the same strength. More complex gates have more output capacitance.

Variable	Name	Description
D	Stage Delay	The delay of this stage relative to the delay of one inverter.

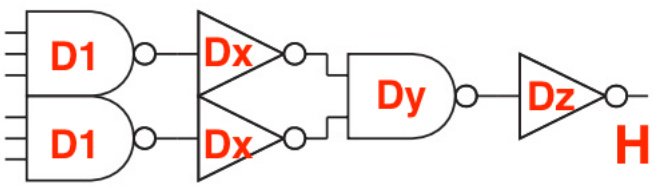
The linear delay model has several possible issues (Weste & Harris covers this), but critically, it ignores wires. It's known to work well enough for 0.25u processes and above, where wire loading is less significant relative to the delay and loading of the transistors themselves. We'll see how well it works at 130nm for Sky130A in the TinyTapeout/OpenLane flow, where placement density is not particularly high.

What I implemented



How I determined the drive values

The key is to notice that when we increase the drive of one cell, we increase the load on the previous cell. This changes the electrical effort of that cell. So, for part D we end up with the following chain



There are equations one can use to optimize this but I just used mathematica because it's faster. Here is how I solved H=20 for part D.

```
NMinimize[{
  (x 5/3 + 3) + (y/x * 1 + 1) + (z/y * 4/3 + 2) + (20/z * 1 + 1),
  (*G and P from Table 4.2 and 4.3 in Weste & Harris *)
  1 ≤ x, x < 8, 1 ≤ y, y < 8, 1 ≤ z, z < 8}, {x, y, z}]

{17.328, {x → 1.54918, y → 3.99994, z → 7.74588}}
```

Then I just used the cell with the nearest drive from the library and checked the result was within one inverter delay of the optimal value.

This procedure is roughly equivalent to what ABC does in the OpenLane flow after mapping. Remember that although ABC can optimize drive, it can't modify structure. As we can see, the choice of structure does have implications on performance, even after the drive is optimized.

Calculated delays with optimized drive

Design	H=1	H=5	H=20
A	10.7	14.8	22.7
B	8.3	12.5	20.0
C	9.7	14.5	23.0
D	12.0	14.8	18.6

These are in units of one inverter delay which is about 70pS in Sky130A.

What is drive?

Drive counts the number of equivalent parallel gates to increase the output current the resulting compound gate. More gates in parallel allows sourcing larger load capacitances with higher dv/dt at the cost of input capacitance, area, and power.

Opting for an integrated cell for the parallel gates is a practical choice. It empowers the layout engineer to implement strategies that effectively minimize wire loading and output capacitance of the overall structure, thereby enhancing the gate's performance.

See the various incarnations of nand2 in Sky130A for an example.

https://diychip.org/sky130/sky130_fd_sc_hd/cells/nand2/

To those with a more analog bend you might think the previous gate is acting like a pre-amplifier, and you're right. It is exactly the same. You might also notice the inverter is basically a class B stage, except it inverts. It is and that's what makes it such a good reference when one considers driving loads.

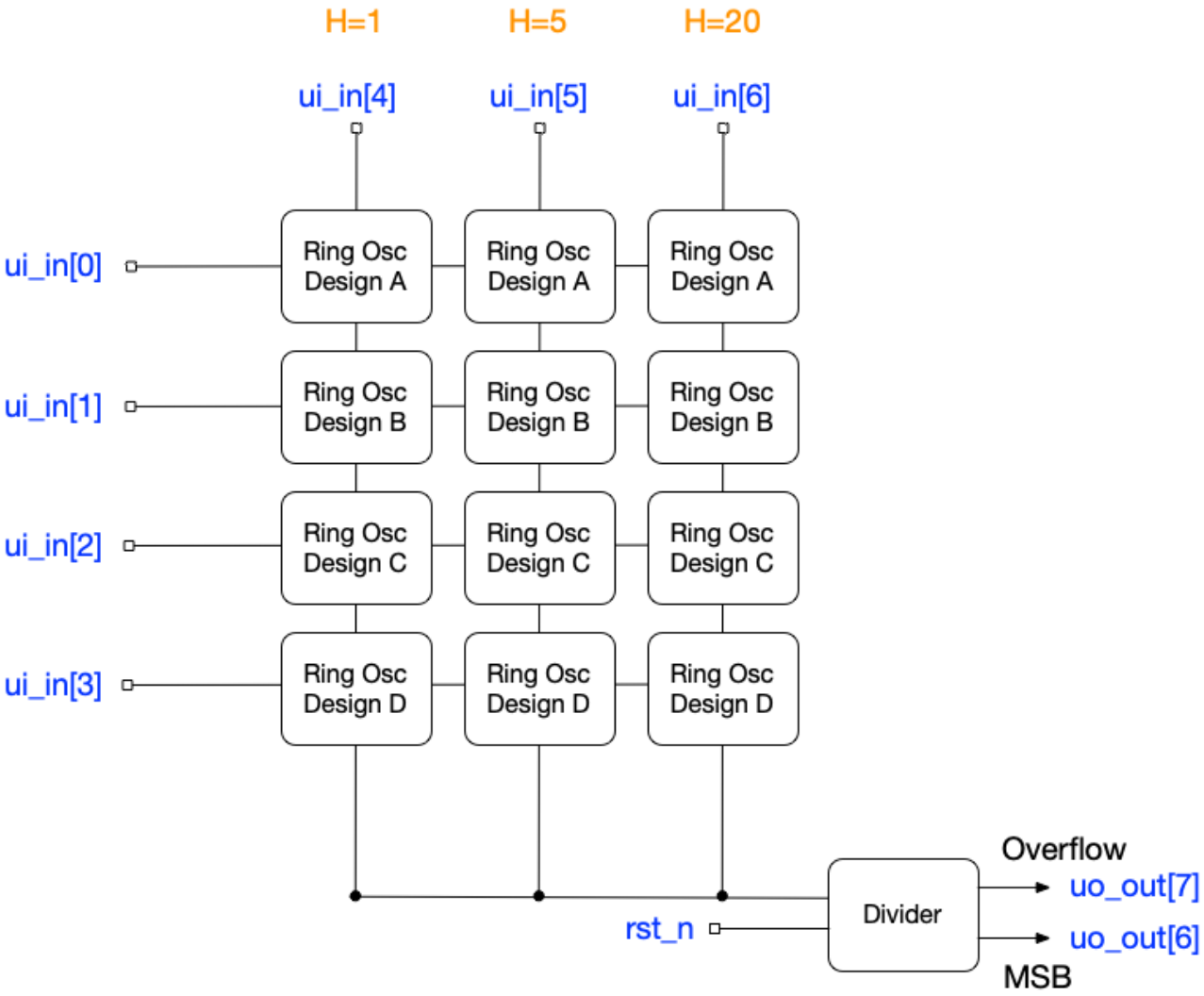
This provides a bit of theory behind the ZtA video

When are 2 logic gates faster than 1?

The video is a bit different than this problem in that it kept the load constant but shrunk the period constraints.

Here we keep the period constraint the same but increase the load. At the end of the day it's the same idea in that we're changing I to get a different dv/dt for a given C.

Test structure



Raw Delay Calculations

```
In[1]:= NMinimize[{
(x 5/3+3)+(y/x*1+1)+(z/y*4/3+2)+(5/z*1+1),(*G and P from Table 4.2 and 4.
1<=x,x<8,1<=y,y<8,1<=z,z<8},{x,y,z}]
Out[1]= {14.303,{x->1.09546,y->2.00004,z->2.73865}}
In[2]:= (*Part A*)
In[3]:= Round[(H/x 1+1+(x (6+2)/3+6))/.{H->{1,5,20},x->{1,2,4}},0.1]
Out[3]= {10.7,14.8,22.7}
In[4]:= (*Part B*)
In[5]:= Round[((H/x 5/3+2)+(x 5/3+3))/.{H->{1,5,20},x->{1,2,4}},0.1]
Out[5]= {8.3,12.5,20.}
In[6]:= (*Part C*)
Round[((H/x 7/3+3)+(x 4/3+3))/.{H->{1,5,20},x->{1,2,4}},0.1]
```

```
Out[6]= {9.7,14.5,23.}
In[7]:= (*Part D*)
In[8]:=
Round[((x 5/3+3)+(y/x*1+1)+(z/y*4/3+2)+(H/z*1+1))/.{H->{1,5,20},x->{1,1,1
Out[8]= {12.,14.8,18.}
```

Hopefully there are no typos or transcription errors.

Pinout

#	Input	Output	Bidirectional
0	sel[0]	b[0]	a[0]
1	sel[1]	b[1]	a[1]
2	sel[2]	b[2]	a[2]
3	sel[3]	b[3]	a[3]
4	h[0]	&(A[5:0])	a[4]
5	h[1]	ntest	a[5]
6	h[2]	count	a[6]
7	ntest	overflow	a[7]

TDM Digital Clock [555]

- Author: Hassan & Huzaifa tariq
- Description: a digital clock that uses time division multiplexing, using 8 outputs to drive six seen segmetns displas
- GitHub repository
- HDL project
- Mux address: 555
- Extra docs
- Clock: 50000000 Hz

How it works

i dont know Explain how your project works it works because it works

How to test

you dont Explain how to use your project you dont

External hardware

none List external hardware used in your project (e.g. PMOD, LED display, etc), if any just switche and leds

Pinout

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

Hardware Trojan Part II [556]

- Author: Jeremy Hong
- Description: Pseudorandom number generator with and without hardware trojan
- GitHub repository
- Wokwi project
- Mux address: 556
- Extra docs
- Clock: 10000 Hz

Credit

My School and Instructor: Wright State University EE-4550/6550 IC Hardware Security and Trust by Dr. Saiyu Ren

My employer: Two Six Technologies

My teammates: Celeste Irwin and Nicholas Nissen

How it works

This pseudorandom number generator (PRNG) is compromised of scan flip-flops (SFF) and XOR gates. There are two PRNGs in this design, a PRNG with and without a hardware trojans

How to test

Test by giving design a clock signal, and then set the PRNG by setting the scanin pins, and then toggle the scan enable pin. To reset turn off all the scanin pins and then leave the scan enable pin on for a few seconds.

External hardware

Pattern generator and logic analyzer recommended.

Pinout

#	Input	Output	Bidirectional
0	Scan Enable	PRNG 1 output Trojan Free	Input, ScanIn 8
1	ScanIn 1	PRNG 2 output trojan inserted	Input, ScanIn 9
2	ScanIn 2		Input, ScanIn 10
3	ScanIn 3		Input, External Trojan Trigger
4	ScanIn 4		Output, single inverter test
5	ScanIn 5		Input, single inverter test
6	ScanIn 6		Input, 8 inverters test
7	ScanIn 7		Output, 8 inverters test

4-Bit ALU [557]

- Author: Daniel Kaminski
- Description: 4-Bit ALU with Cornell ECE2300 op-code instructions.
- GitHub repository
- HDL project
- Mux address: 557
- Extra docs
- Clock: 0 Hz

How it works

This project uses the Cornell University ECE2300 (SP24 taught by Zhiru Zhang) ISA to implement a 4-bit ALU. This ALU uses a top-level controller module to pipe the correct inputs and select the correct outputs (src/alu.v). The logical functions are implemented using the following instructions.

FS	Function	Logical Equivalent
000	ADD	A + B
001	SUB	A - B
010	SRA (Shift Right Arithmetic)	A»>
011	SRL (Shift Right Logical)	A »
100	SLL (Shift Left Logical)	A «
101	AND	A & B
110	OR	A + B
111	INVALID	INVALID

How to test

Input your first value, A to UI[7:4], and B to UI[3:0]. You then put your function select (FS) input into UIO[2:0], and you can read your Y output at UO[7:4], your carry out at UO[3], overflow at UO[2], negative at UO[1], and zero at UO[0].

External hardware

None! Just a way to input your desired values and read the outputs.

Pinout

#	Input	Output	Bidirectional
0	B[0]	Z	FS[0]
1	B[1]	N	FS[1]
2	B[2]	V	FS[2]
3	B[3]	C	
4	A[0]	Y[0]	
5	A[1]	Y[1]	
6	A[2]	Y[2]	
7	A[3]	Y[3]	

8-bit DEM R2R DAC [558]

- Author: Eric Fogleman
- Description: 8-bit segmented mismatch-shaping R2R DAC
- GitHub repository
- HDL project
- Mux address: 558
- Extra docs
- Clock: 10000000 Hz

Operation

This design implements a linear 8-bit DAC suitable for dc and low-frequency inputs. An analog voltage is produced by connecting the encoder's outputs to a modified R-2R ladder on the PCB (see External Hardware). It achieves high-linearity by using segmented mismatch-shaping, so the DAC does not require matched resistors. The encoder provides 1st order mismatch and quantization noise shaping. With a clock frequency of 6.144 MHz and a lowpass filter corner of 24 kHz, the oversampling ratio (OSR) is 256.

Error due to resistor mismatch appears at the output as 1st-order highpass shaped noise. The encoder also reduces the bit-width from 8-bits, and quantization error is also 1st-order highpass shaped. Thus, with passive filtering, a linear, low-noise dc output can be achieved. The theory behind this encoder is described in: A. Fishov, E. Fogleman, E. Siragusa, I. Galton, "Segmented Mismatch-Shaping D/A Conversion", IEEE International Symposium on Circuits and Systems (ISCAS), 2002

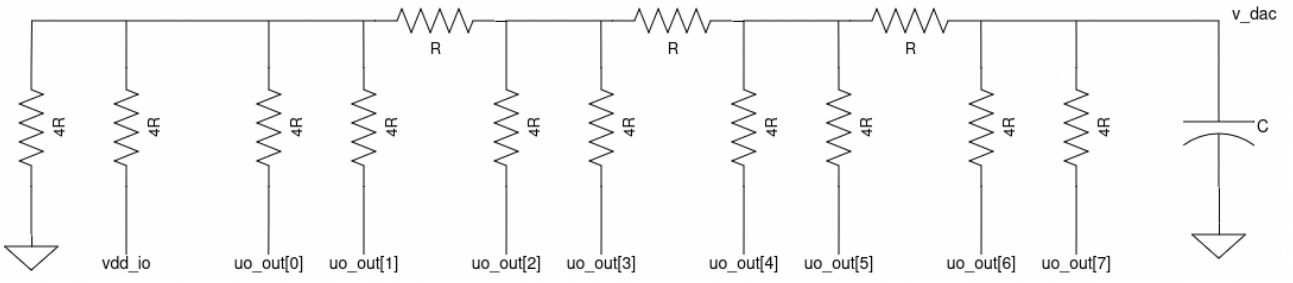
Ideally, this encoder would be buffered through a clean analog supply and retimed to reduce glitches on output transitions. However, reasonable performance should be possible driving the resistor ladder directly from the encoder through the IO supply.

How to test

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

```
out = 8*uo_out[7]+uo_out[6]) + 4*(uo_out[5]+uo_out[4]) + 2*uo_out[3]+uo_o
```

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



The DAC is free-running off the project clock, and inputs appear at the output immediately after passing through a pair of clock sync registers. A simple dc test can be performed using the input DIP switches and the resistor ladder. It is possible to input dynamic waveforms from the microcontroller as well.

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

- 3: 1st order mismatch-shaping with dither
- 2: randomization (flat spectral shaping)
- 1: 1st order shaping, no dither
- 0: static encoding (no linearization)

External hardware

Technically, this is a mismatch shaping DAC encoder. For a high-performance DAC, it is best to use a precision reference voltage and a clean clock source for edge retiming. However, it is possible to connect the encoder directly to a resistor ladder. In this case, the digital IO supply acts as the DAC's reference voltage, and timing skews between the `uo_out` bits may impact performance.

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

The suggested unit R value is 10 kOhm. The equivalent output resistance of the network at **v_out** is 10 kOhm. A 680 pF output capacitor provides a 23 kHz lowpass corner. With this choice of R, the minimum load resistance on each `uo_out` pin is 60 kOhm, and the driver will source a maximum of 55 uA at 3.3 V.

Pinout

#	Input	Output	Bidirectional
0	d_in[0]	d_out_0[0]	en_enc
1	d_in[1]	d_out_0[1]	en_dith
2	d_in[2]	d_out_1[0]	
3	d_in[3]	d_out_1[1]	
4	d_in[4]	d_out_2[0]	
5	d_in[5]	d_out_2[1]	
6	d_in[6]	d_out_3[0]	
7	d_in[8]	d_out_3[1]	ena_and_rst_n

UART Transceiver [559]

- Author: Saad Khan, Saim Iqbal
- Description: 8 bit data frame, with little endian transmit and receive signals
- GitHub repository
- HDL project
- Mux address: 559
- Extra docs
- Clock: 50000000 Hz

How it works

to be added later

How to test

to be added later

External hardware

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

Pinout

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

Universal Motor and Actuator Controller [582]

- Author: Assoc. Prof. Dinçer Gökcen, Ethem Buğra Arslan, Batu Cem Özyurt
- Description: bldc motor controller and autotuner for controller by MNS lab
- GitHub repository
- HDL project
- Mux address: 582
- Extra docs
- Clock: 50000000 Hz

How it works

For some motor driver and BLDC motor combinations with encoders, the controller works universally through its I2C interface via PID control and Nichols-Ziegler auto-tuning algorithm for automated PID constants. I2C Addressing: Slave Address: 0x72 Subaddresses:

How to test

Test when a motor setup is ready by simply communicating through I2C with SCL at about 100kHz. With the adressing above, one can automate PID control or take over (override) to manual settings. Generated PWM and desired motor period is also interfaced through I2C and fully configurable.

External hardware

Motor Driver has to be used in order to convert digital pwm signals to power signals.

BLDC motor with positive and negative inputs & at least 2 encoders must be used to infer speed and direction.

Pullup resistors are needed to communicate through i2c, if not provided.

Pinout

#	Input	Output	Bidirectional
0	encoder_a	motor_positive	sda
1	encoder_b	motor_negative	
2			

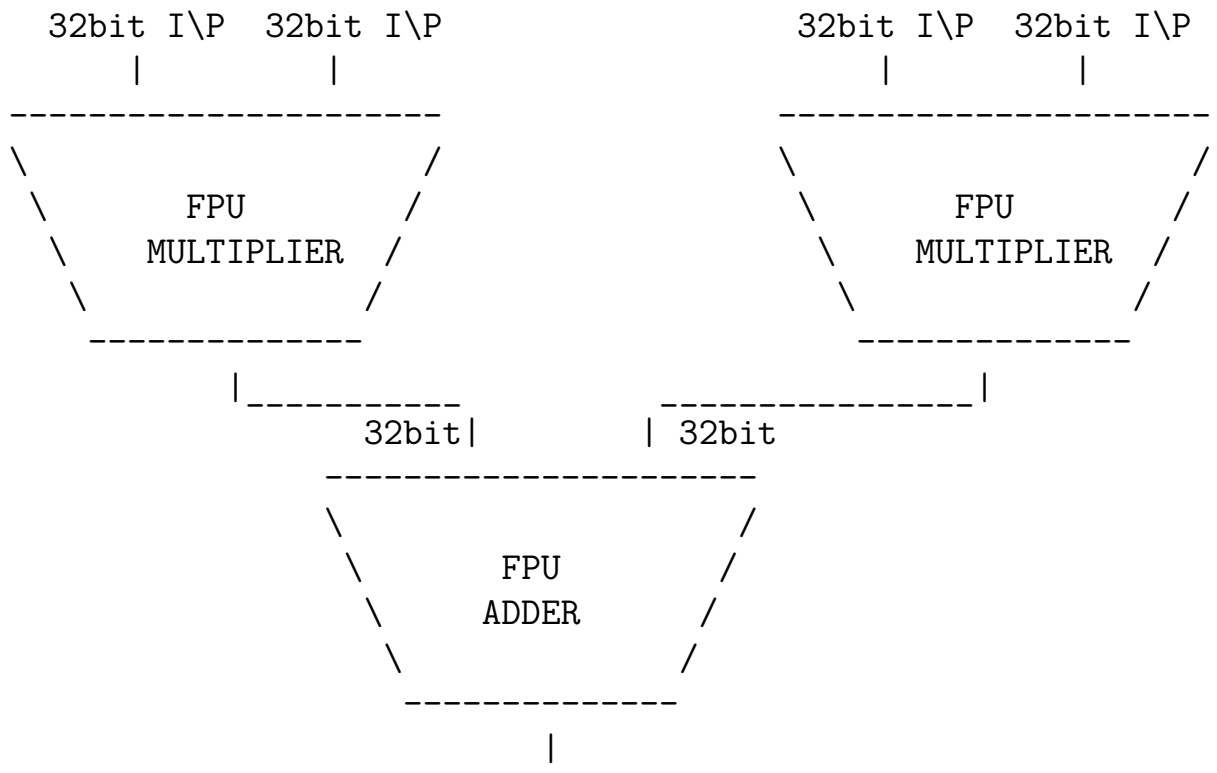
#	Input	Output	Bidirectional
3			
4			
5			
6			
7	scl		

Dgrid_FPU [590]

- Author: Aravind-Prasad-Abhinav-Prakash
- Description: 4 Input FPU for MAC at 40MHZ
- GitHub repository
- HDL project
- Mux address: 590
- Extra docs
- Clock: 40000000 Hz

How it works

The **Dgrid_FPU (Floating Point Unit)** is an integral component of computer hardware engineered to execute floating-point arithmetic operations. It features four 32-bit inputs organized to conduct dual multiplications followed by an addition in series. Specifically, the first pair of 32-bit inputs is multiplied, and simultaneously, the second pair is processed similarly. The results from these multiplications are then fed into a two-input adder, producing a 32-bit final output. This configuration is highly effective in applications that demand robust computing capabilities, such as high-performance computing, digital signal processing, scientific simulations, and graphics processing. The Dgrid_FPU's architecture, which enables the parallel processing of multiple arithmetic operations, significantly boosts performance in these computationally intensive tasks.



32bit O\P

The Dgrid_FPU top module is designed with a configuration that supports 8-bit input and output interfaces, necessitating a systematic process to handle the 128-bit data (comprising four 32-bit inputs) required for operations. The input process involves 16 clock cycles to load the four 32-bit registers sequentially. Once the data is loaded, the computation begins, producing a 32-bit output over the subsequent two clock cycles.

After the computation phase, the 32-bit result is output through the 8-bit interface, which requires an additional four clock cycles to read out the data thoroughly. Additionally, two clock cycles are utilized for data transfer, bringing the total cycle count to 24 for an entire operation sequence from input loading to output retrieval.

A reset operation is required to prepare the module for a new data set, ensuring that the Dgrid_FPU is ready to process subsequent inputs efficiently. It is important to note that both the input and output data conform to the IEEE 754 standard for floating-point numbers, ensuring compatibility and precision in high-stake computational applications.

This Verilog code outlines a Floating Point Unit (FPU) for use in Machine Arithmetic Cores (MACs) within AI accelerators. The FPU facilitates key operations such as adding and multiplying floating-point numbers, which are crucial for executing complex mathematical computations in AI algorithms. It includes modules for managing data input and output, processing up to 128-bit and 32-bit registers, and handling edge cases like infinity and zero. This architecture is especially beneficial for AI applications, allowing parallel processing and enhancing computational efficiency and precision in neural networks. By accelerating operations and ensuring robust data handling, this FPU is instrumental in optimizing AI accelerators, ultimately speeding up learning and inference processes.

How to test

To effectively test the Dgrid_FPU, follow these step-by-step instructions:

- **Reset the Circuit:** Initiate by resetting the circuit to clear any previous data and prepare it for new input.
- **Write Data:** Write a 128-bit bitstream that includes all four inputs, entering the data 8 bits at a time. This step requires 16 clock cycles to complete.
- **Wait for Computation:** Allow the module to process the inputs, which will take up to the 20th clock cycle.
- **Read Output Data:** From the 21st to the 24th clock cycle, read out the 32-bit result in increments of 8 bits to verify the output and ensure the system's functionality.

Example

- I1 = 2.2 (HEX - 400ccccd)
- I2 = 3.3 (HEX - 40533333)
- I2 = 4.4 (HEX - 408ccccd)
- I2 = 5.5 (HEX - 40b00000)
- FINAL OUTPUT = 31.46 (HEX - 41fbae13)

Then the bitstream will be 400ccccd_40533333_408ccccd_40b00000 and start sending it from LHS e.g. - Fist data to be send is 40 and last data is 00 and then observe the output.

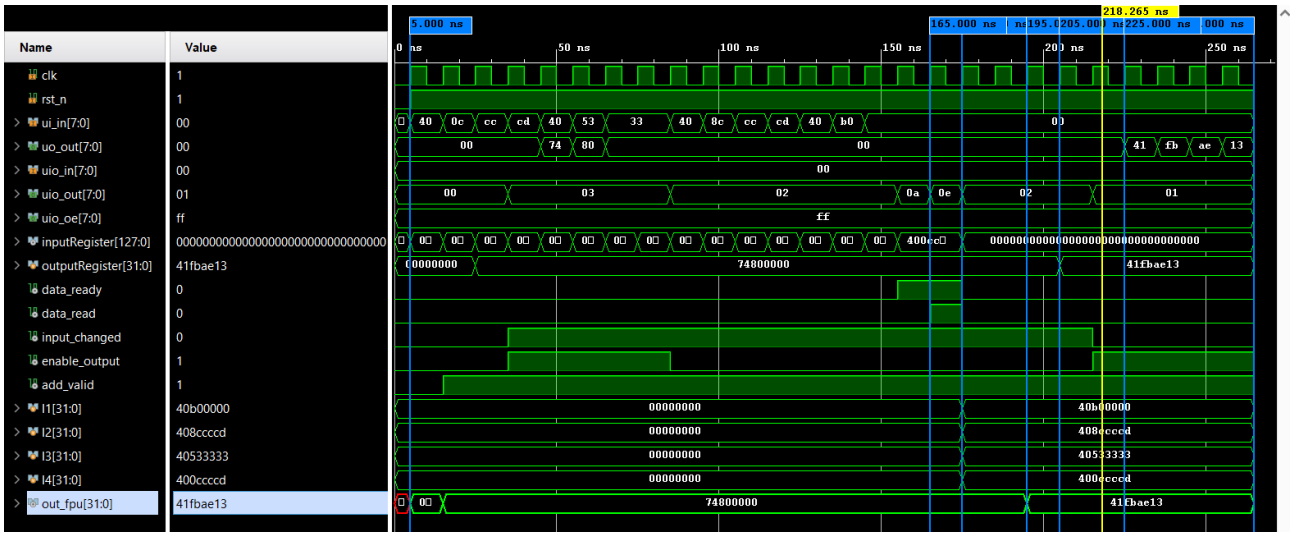


Figure 55: image

Waveform

Pinout

#	Input	Output	Bidirectional
0	Bit 0 Input	Bit 0 Output	Output used as valid Signal
1	Bit 1 Input	Bit 1 Output	Output used as valid Signal
2	Bit 2 Input	Bit 2 Output	Output used as valid Signal
3	Bit 3 Input	Bit 3 Output	Output used as valid Signal
4	Bit 4 Input	Bit 4 Output	0
5	Bit 5 Input	Bit 5 Output	0
6	Bit 6 Input	Bit 6 Output	0

#	Input	Output	Bidirectional
7	Bit 7 Input	Bit 7 Output	0

Parity Generator [608]

- Author: Eric Ulteig
- Description: TT06 Learning Exercise
- GitHub repository
- Wokwi project
- Mux address: 608
- Extra docs
- Clock: 0 Hz

How it works

Data present on the 8 input pins is converted and readable on the output pins.

How to test

Set the input switches and view the output LEDs.

External hardware

No external hardware is used.

Pinout

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

24 H Clock [609]

- Author: UABC Team
- Description: typical 23h-format 4 digits clock. Two digits for hours and the other for minutes.
- GitHub repository
- HDL project
- Mux address: 609
- Extra docs
- Clock: 1000000 Hz

How it works

This is a typical 23h-format 4 digits clock. Two digits for hours and the other for minutes. Digits are in BCD format. It uses a 1 megahertz signal for reference. One push button for setting the hour or minute and another push button for advancing forward the hours or the minutes. In order to display the hour it is needed four 7-segment-BCD decoders.

How to test

A one MegaHertz clock signal must be connected to the clk pin. Reset goes from 1 to 0 to start the clock operation. In order to set the correct hour, a pulse signal is needed in the set pin, then M0 digit should be blinking, a pulse in the P0 pin will change this digit. A new pulse in the set pin will change the process to the M1, and another pulse to the H0 and H1.

External hardware

3 push buttons, 1 MHz signal generator, 4 seven segment decoders.

Pinout

#	Input	Output	Bidirectional
0	rst	M0[0]	H0[0]
1	clk	M0[1]	H0[1]
2	P0	M0[2]	H0[2]
3	set	M0[3]	H0[3]

#	Input	Output	Bidirectional
4		M1[0]	H1[0]
5		M1[1]	H1[1]
6		M1[2]	Dots
7		M1[3]	

Sequence detector using 7-segment [610]

- Author: Atharv Sharma & Lipika Gupta
- Description: Detects sequence '1001' and displays '8.' on 7-segment led display, otherwise displays '-' only
- GitHub repository
- HDL project
- Mux address: 610
- Extra docs
- Clock: 1 Hz

How it works

- In this project, we have designed a sequence detector using finite state machine (FSM)
- It is designed using verilog, and detects sequence '1001'
- The logic is made using cases, and it detects the sequence while covering overlapping cases as well

How to test

- If the sequence is detected, the output register z is set to logic 1 that displays '8.' on 7-segment display
- If the sequence is not detected (the output register is 0), 7-segment display shows '-'
- LEDs can be tested in two ways when ui_in [7:1] is kept 7'b1111111 (status for testing - condition = 7'b1111111):

    1. If first 4 bits of reg seg_test (uio_in [7:4]) are 0 during testing, we can display numbers from 0 to 9 if we vary last 4 bits (uio_in[3:0]) from 0000 to 1001
    2. If first 4 bits of reg seg_test (uio_in [7:4]) are 1 during testing, we can display each led seperately by varying last 4 bits (uio_in[3:0]) from 0000 to 0111

External hardware

- We need to use 8 LEDs for 7-segment LED display output ([7:0] uo_out), so that the output can be displayed and verified accordingly at seg

- In addition to this, we need to use an input source from which we can manipulate input logic onto the input register x (ui_in[0])

Pinout

#	Input	Output	Bidirectional
0	x	seg[0]	seg_test[0]
1	condition[0]	seg[1]	seg_test[1]
2	condition[1]	seg[2]	seg_test[2]
3	condition[2]	seg[3]	seg_test[3]
4	condition[3]	seg[4]	seg_test[4]
5	condition[4]	seg[5]	seg_test[5]
6	condition[5]	seg[6]	seg_test[6]
7	condition[6]	seg[7]	seg_test[7]

CDMA_2024 [611]

- Author: Santiago Robledo Acosta, Jóse Miguel Rocha Pérez
- Description: This is a CDMA circuit for lab testing in order to see the properties of Gold Codes in an Osciloscope
- GitHub repository
- HDL project
- Mux address: 611
- Extra docs
- Clock: 1000000 Hz

How it works

```
This is a very simple circuit, it consists in two LSFR register lineary
generate this pair of m-sequences, we input an initial vaule called see
initial value of 0s. For this design we used two LSFR with 5 D Flip-Flo
With this PN signal and modulus 2 adding the signal we want to tranmit,
an OPAM in order to add noise to the CDMA and feed it back to the desig
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 initiat
```

How to test

```
As we used a hardware description language (Verilog), we created an spe
to 0 in order to generate the adequate signal such as the clock, a test
The stimulus simple will assign a value to set_i and deactivate it to l
each clock cycle. With the test signal_i we the system will generate th

For the reception process, we simply assign the value of CDMA_o to rece
it shows the same time diagram as receptor_o.

As for the LED_o it works as a simple indicator that the inputed seed i

The template will include the verilog file with its testbench.
```

External hardware

OPAM, Testboard, LED

Pinout

#	Input	Output	Bidirectional
0	signal_i		cdma_o
1	seed_i[0]		gold_o
2	seed_i[1]		receptor_o
3	seed_i[2]		led_o
4	seed_i[3]		
5	seed_i[4]		
6	seed_i[5]		
7	load_i		

Simple Stopwatch [612]

- Author: Fabio Ramirez Stern
- Description: A simple stopwatch counting in 100th seconds and outputing it via SPI to a MAX7219 chip controlling an 8 digit 7-segment display.
- GitHub repository
- HDL project
- Mux address: 612
- Extra docs
- Clock: 1000000 Hz

How it works

A clock divider turns 1 MHz into 100 Hz, which drives a stopwatch going from 00:00:00 to 59:59:99. To achieve this, a chain of two types of counting circuit, one per digit gives it's output to an SPI master that encodes the result to be displayed on a 7-segment display with at least 6 digits.

How to test

The start/stop button toggles the clock, the lap time button pauses the display, while the clock keeps running in the background. Pressing it again re-enables the display. The time can be reset with the reset button on input 2, or with the chip/PCB wide reset. The PCB wide reset affects everything, the input pin driven reset does only resets the counters.

External hardware

2-3 buttons, one for start/stop and one for lap times. For the reset, either a third button or the dev board's reset for the whole chip can be used. 1 MAX7219/MAX7221 driven 7-segment display, or something that can interpret the SPI signal according to the MAX's specifications.

Pinout

#	Input	Output	Bidirectional
0	start/stop	SPI MOSI	
1	lap time	SPI CS (active low)	

#	Input	Output	Bidirectional
2	reset (active high)	SPI CLK	
3		stopwatch enabled (counting up)	
4		display enabled (goes low when showing lap time)	
5			
6			
7			

Clock [613]

- Author: Hilburn
- Description: ASIC Desktop Clock
- GitHub repository
- HDL project
- Mux address: 613
- Extra docs
- Clock: 50000000 Hz

How it works

Generates a 1 second pulse that increments a seconds, minutes, and hours timer that then gets decoded and displayed on the 7-segment display

How to test

Connect 7-segment display and adjust imput switches to desired clock frequency

External hardware

PMOD, 7-segment display, breadboard

Pinout

#	Input	Output	Bidirectional
0	Clock 10 MHz	7-seg[0]	
1	Clock 12 MHz	7-seg[1]	
2	Clock 14 MHz	7-seg[2]	
3	Clock 20 MHz	7-seg[3]	
4	Show Minuetes	7-seg[4]	
5	Show Hours	7-seg[5]	
6		7-seg[6]	
7		7-seg[7]	

MULDIV unit (8-bit signed/unsigned) [614]

- Author: Darryl Miles
- Description: Combinational Multiply and Divide Unit (signed and unsigned)
- GitHub repository
- HDL project
- Mux address: 614
- Extra docs
- Clock: 0 Hz

Background

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

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

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

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

How It Works

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

The data is latched at the CLK NEGEDGE and the value provided to the combinational logic MUL/DIV operations (which are seperate logic modules) with the answer becoming immediately available (after propagation and ripple settling time) at the outputs.

The result output is also multiplexed and has an immediate and register mode. The immediate mode provides a direct visibility of the MUL/DIV combintational timing between input and outputs (you need to account for address multiplex of high-low 8bit sides of result). The registered mode capture the result in full so that it is possible to pipeline interleave request and result information to achieve higher throughput.

So one half of the answer is immediately available to read and the other half of the answer can be read by toggling UI_IN[0] (address bit0). Clocking is needed for registered output mode, but not necessarily for immediate mode.

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

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

// FIXME explain IMMediate mode and REGistered mode (to pipeline)

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

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

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

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

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

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

How to test

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

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

There should be sufficient instructions here start you own journey.

External hardware

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

Thoughts to the future (next iteration)

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

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

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

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

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

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

TODO

Fixup the original logicsim schematic labels.

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

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

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

Fill out SpinalHDL unit testing testing.

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

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

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

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

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

Pinout

#	Input	Output	Bidirectional
0	Data0 see docs	Result0 see docs	Addr bit0 HI=1/lo=0 mux of Data and Result (input only)
1	Data1 see docs	Result1 see docs	unused
2	Data2 see docs	Result2 see docs	unused
3	Data3 see docs	Result3 see docs	Result mux registered=1/immediate=0 (input only)
4	Data4 see docs	Result4 see docs	DIV error overflow (output only)

#	Input	Output	Bidirectional
5	Data5 see docs	Result5 see docs	DIV error divide-by-zero (output only)
6	Data6 see docs	Result6 see docs	OPSIGNED mode (input only)
7	Data7 see docs	Result7 see docs	MULDIV mode (input only)

motor a pasos [615]

- Author: Alan Tavira
- Description: Motor a pasos con base de tiempo para control de velocidad, cambio de sentido de giro y de tipo de paso
- GitHub repository
- HDL project
- Mux address: 615
- Extra docs
- Clock: 50000000 Hz

How it works

En este trabajo se realiza un motor a pasos con la implementación de una máquina de estados tipo moore. El cambio de un estado a otro se hace a diferentes velocidades, 1s, 0.5s, 0.25s y 0.125s, utilizando la base de tiempo realizada en el trabajo anterior. Se realizan 3 tipos de pasos para este motor los cuales son el paso completo, el medio paso y el paso doble y cada uno de estos puede ser realizado en el sentido horario o antihorario.

El motor a pasos puede implementarse como una máquina de estados en la cual tendremos 3 entradas y 4 salidas. Las entradas son :

-Selector de velocidad: Viene de una base de tiempo previamente realizada. En el código puede verse como un wire llamado vel. Si está en un valor alto se cambia al siguiente estado mientras que en un valor bajo se mantiene en el mismo estado. Con esta entrada podemos variar la velocidad con la que cambian los estados.

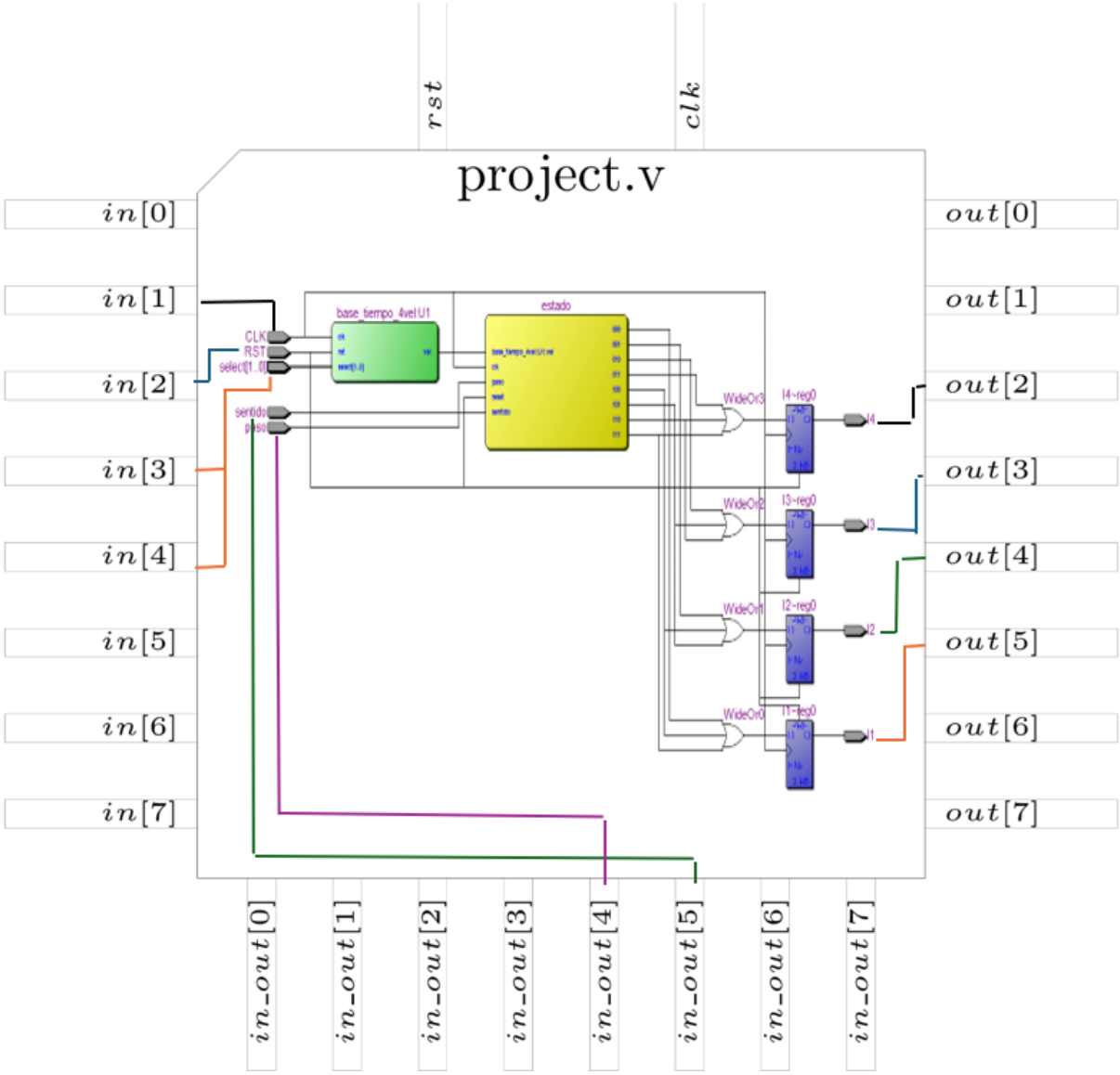
-Sentido: Esta entrada indica si el motor a pasos tendrá el sentido antihorario u horario, en el código esta representada por la entrada sentido.

-Paso: Con esta entrada definimos si el paso será completo, medio o doble. En el código se define como paso.

Las 4 bobinas de salidas se definen como I1, I2, I3 y I4. En la figura podemos ver el diagrama de la maquina de estados que representa al motor a pasos. La entrada H corresponde a la salida de la base de tiempo vel, la salida D al sentido y P al tipo de

paso.

En la siguiente figura se presentan las conexiones de las entradas y salidas de la máquina de estados con las correspondientes al chip del proyecto.



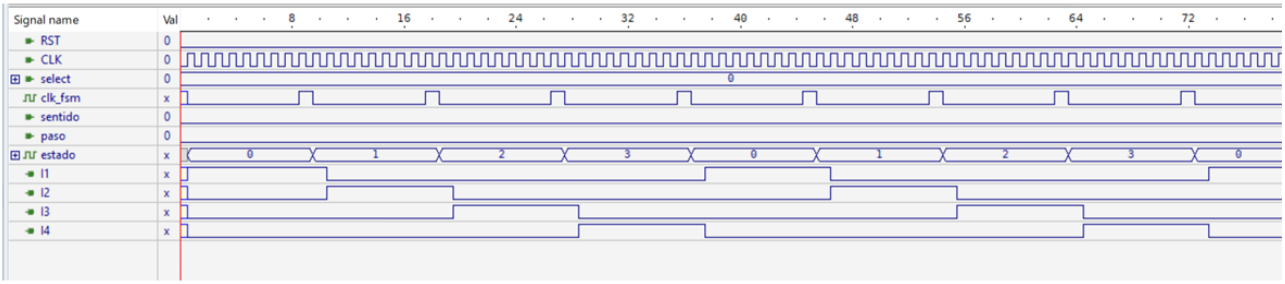
Las entradas RST, select, sentido y paso son entradas que pueden ser controladas con un switch o push button ya que únicamente se requiere de un valor lógico alto o bajo para ellas. La entrada CLK corresponde al reloj, no se conecta al reloj del chip para poder tener más libertad en el valor de la frecuencia de la base de tiempo. Aunque originalmente se utilizó un reloj de 50MHz. Finalmente, las salidas I1, I2, I3 y I4 son las salidas de la máquina de estados. La frecuencia de operación de estas dependen del valor de la entrada select, ya que esta controla la velocidad del pulso de la base de tiempo que a su vez controla la velocidad a la que operara la máquina de estados (50MHz, 25MHz, 12.5MHz o 6.25MHz).

How to test

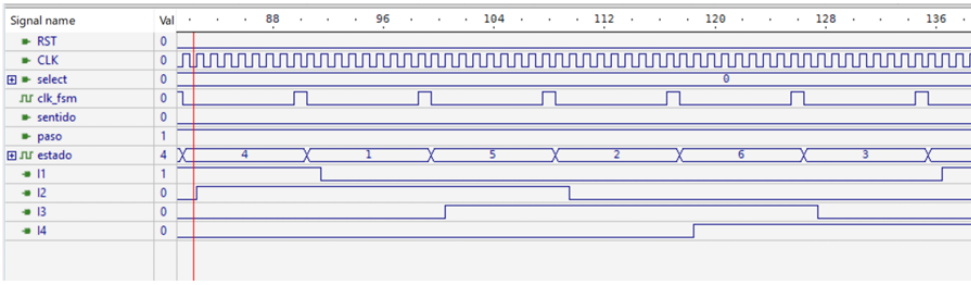
Se utiliza un reloj de 50MHz para la máquina de estados y la base de tiempo. Dependiendo del selector de velocidad el cambio de un estado a otro se dará en 1s (cuando se cuenten todos los ciclos), 0.5s (conteo de la mitad de los ciclos), 0.25s (conteo de 1/4

del total de los ciclos) o 0.125s (1/8 de los ciclos). Si la entrada "paso" esta activa entonces el cambio será de medio paso y cuando este en bajo el paso será completo, si la entrada paso está en bajo y nos encontramos en un estado correspondiente a medio paso (4, 5, 6 o 7) entonces el paso será doble. Si la entrada "sentido" está en bajo entonces el cambio se dará en sentido horario y cuando la entrada este en un valor alto el sentido será antihorario. Por último, la salida de la base de tiempo nos indica si cambiar a otro estado (valor en alto) o permanecer en el mismo estado (valor en bajo).

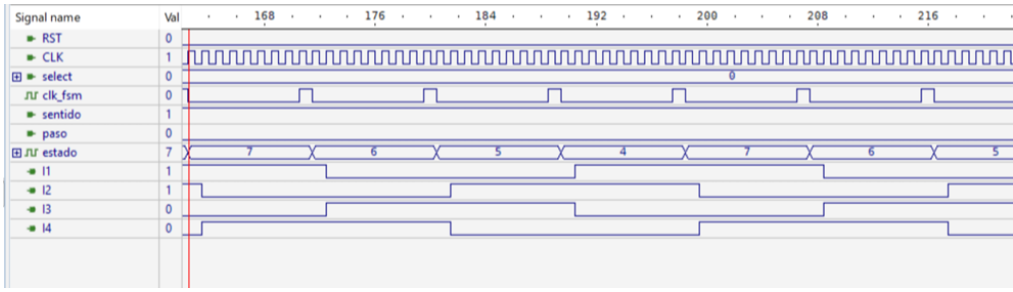
Para la simulación se cambia el parámetro f de la base de tiempo de 50000000 a 8 para facilitar la simulación. La señal de reset se deja en el valor fijo 0, el reloj tiene un periodo de 1ns, el select y el sentido se establecen con un periodo de 324ns y el paso con 162ns. De esta manera podemos ver todas las combinaciones para las entradas sentido y paso con una sola velocidad. En la figura se observan las salidas correspondientes a cada estado con el sentido horario (sentido=0) y el paso completo (paso=0).



los estados cuando el sentido es horario (sentido=0) y se tienen medios pasos (paso=1) se tienen en la siguiente imagen

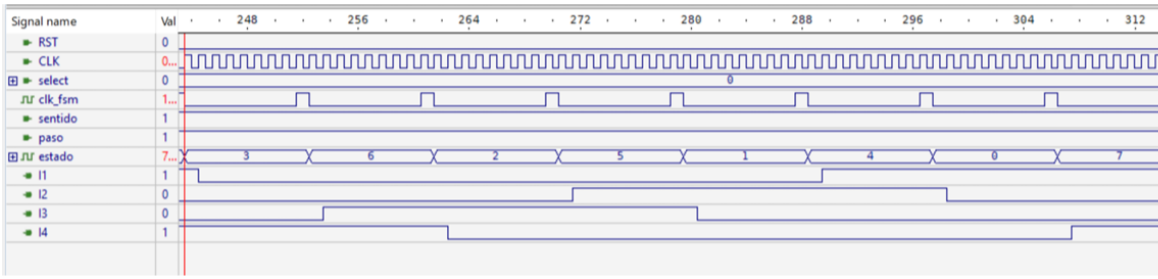


La imagen de abajo muestra la combinación de entradas correspondientes a un sentido antihorario (sentido=1) y ya que el paso es completo entre los estados 4 al 7 se consid-



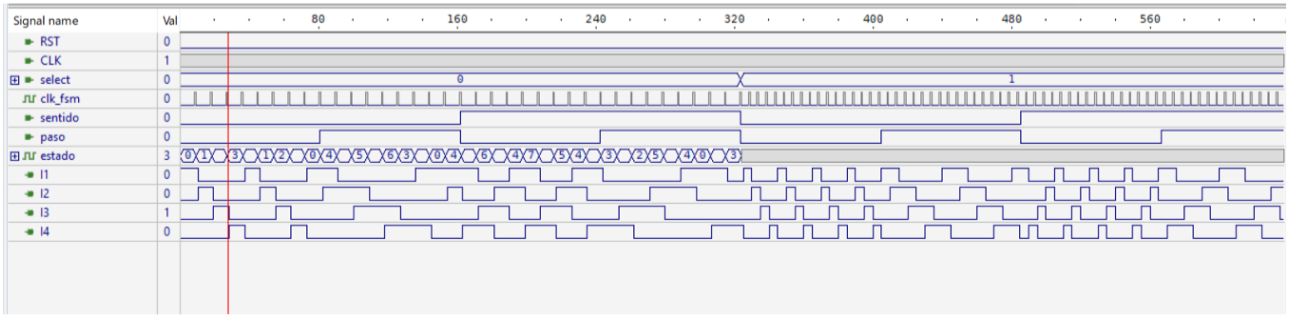
era un paso doble (paso=0).

En la próxima imagen se regresa al medio paso (paso=1) pero ahora en sentido antiho-

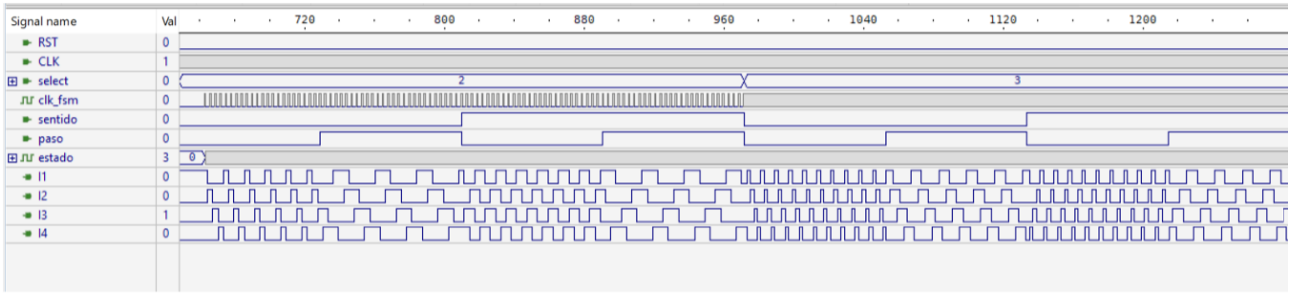


rario (sentido=1).

Finalmente, tenemos los diferentes valores que puede tomar el selector de velocidad (1s, 0.5s, 0.25s y 0.125s). Para cada caso las salidas son las mismas pero con una frecuencia más alta en comparación a la anterior.



Velocidad de paso de 1s y 0.5s



Velocidad de paso de 0.25s y 0.125s.

External hardware

FPGA Cyclone II EP2C35F672C6ES

Pinout

#	Input	Output	Bidirectional
0			
1	CLK		
2	RST	I4	

#	Input	Output	Bidirectional
3	select[0]	I3	
4	select[1]	I2	paso
5		I1	sentido
6			
7			

MULDIV unit (8-bit signed/unsigned) with sky130 HA/FA cells [616]

- Author: Darryl Miles
- Description: Combinational Multiply and Divide Unit (signed and unsigned)
- GitHub repository
- HDL project
- Mux address: 616
- Extra docs
- Clock: 0 Hz

Background

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

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

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

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

How It Works

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

The data is latched at the CLK NEGEDGE and the value provided to the combinational logic MUL/DIV operations (which are seperate logic modules) with the answer becoming immediately available (after propagation and ripple settling time) at the outputs.

The result output is also multiplexed and has an immediate and register mode. The immediate mode provides a direct visibility of the MUL/DIV combintational timing between input and outputs (you need to account for address multiplex of high-low 8bit sides of result). The registered mode capture the result in full so that it is possible to pipeline interleave request and result information to achieve higher throughput.

So one half of the answer is immediately available to read and the other half of the answer can be read by toggling UI_IN[0] (address bit0). Clocking is needed for registered output mode, but not necessarily for immediate mode.

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

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

// FIXME explain IMMediate mode and REGistered mode (to pipeline)

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

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

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

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

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

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

How to test

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

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

There should be sufficient instructions here start you own journey.

External hardware

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

Thoughts to the future (next iteration)

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

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

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

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

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

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

TODO

Fixup the original logicsim schematic labels.

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

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

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

Fill out SpinalHDL unit testing testing.

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

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

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

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

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

Pinout

#	Input	Output	Bidirectional
0	Data0 see docs	Result0 see docs	Addr bit0 HI=1/lo=0 mux of Data and Result (input only)
1	Data1 see docs	Result1 see docs	unused
2	Data2 see docs	Result2 see docs	unused
3	Data3 see docs	Result3 see docs	Result mux registered=1/immediate=0 (input only)
4	Data4 see docs	Result4 see docs	DIV error overflow (output only)

#	Input	Output	Bidirectional
5	Data5 see docs	Result5 see docs	DIV error divide-by-zero (output only)
6	Data6 see docs	Result6 see docs	OPSIGNED mode (input only)
7	Data7 see docs	Result7 see docs	MULDIV mode (input only)

mult_2b [617]

- Author: Juan Manuel Lpez Pasten
- Description: Multiplixador de 2 bits utilizando compyertas logicas
- GitHub repository
- HDL project
- Mux address: 617
- Extra docs
- Clock: 0 Hz

How it works

EL proyecto es un multiplicador sencillo de 2 bits para cada entrada a,b. Se realizó describiendo el circuito con compiertas lógicas utilizando multiplicaciones con compuertas AND y medios sumadores con la combinación de compuertas AND y XOR. Este circuito es completamente combinacional y es una aplicación práctica sencilla y didáctica para obtener un circuito físico final a partir de la descripción en verilog.

A1	A0	B1	B0	P3	P2	P1	P0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

How to test

Las entradas del circuito de 2 bits, a y b, deben conectarse a interruptores como lo pueden ser DIP switch, con sus respectivas resistencias. Las salida out de 4 bits se

puede conectar a leds, de igual manera con sus respectivas resistencias para evitar dañar algún componente.

External hardware

El hardware externo utilizado es:

-DIP switch 4 posiciones. -4 IEDS.

Pinout

#	Input	Output	Bidirectional
0	a (bit 0)	out (bit 0)	not used
1	a (bit 1)	out (bit 1)	not used
2	b (bit 0)	out (bit 2)	not used
3	b (bit 1)	out (bit 3)	not used
4	not used	not used	not used
5	not used	not used	not used
6	not used	not used	not used
7	not used	not used	not used

NCL LFSR [618]

- Author: Tommy Thorn
- Description: A trivial little example to try out self-timed logic
- GitHub repository
- HDL project
- Mux address: 618
- Extra docs
- Clock: 0 Hz

How it works

Logic is dual redundantly encoded so we can distingush data (DATAx) and no data (NULL). … to be filled in.

How to test

You can't really test without a scope and a way to drive inputs

External hardware

Scope

Pinout

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

Decodificador binario a display 7 segmentos hexadecimal [619]

- Author: Victor Manuel Cante Saloma
- Description: Muestra un número binario de 4 bits en un diplay de 7 segmentos (ánodo común) en hexadecimal
- GitHub repository
- HDL project
- Mux address: 619
- Extra docs
- Clock: 0 Hz

How it works

The operation is quite simple; when entering a 4-bit binary number, this number is shown at the output on a 7-segment common anode display in hexadecimal. The input "h" is a 4-bit vector, and the output "S" is a 7-bit vector. For the output "S", the most significant bit corresponds to segment "a", and so on, until the least significant bit, which corresponds to segment "g", as shown in figure 1. Since the display is anode common, to indicate that a segment is on, it is indicated with a "0".

In the simulation shown in Figure 2, we can see that given a binary number that we introduce at the input, an output combination corresponds to the value to be shown on the 7-segment display in hexadecimal form, that is, given The binary number at the input corresponds to a 7-bit binary number, which is actually a pattern to light each segment of the 7-segment display, which obviously corresponds to the input number to be displayed.

According to Figure 3, the connections of the proposed circuit to those of the project in general are detailed below.

1. For the input, which is a 4-bit vector "h", the overall project pins connected to the proposed circuit are as follows:

in[0]: "h[0]" //Bit 0

in[1]: "h[1]" //Bit 1

in[2]: "h[2]" //Bit 2

in[3]: "h[3]" //Bit 3

in[4]: "no use"

in[5]: "no use"

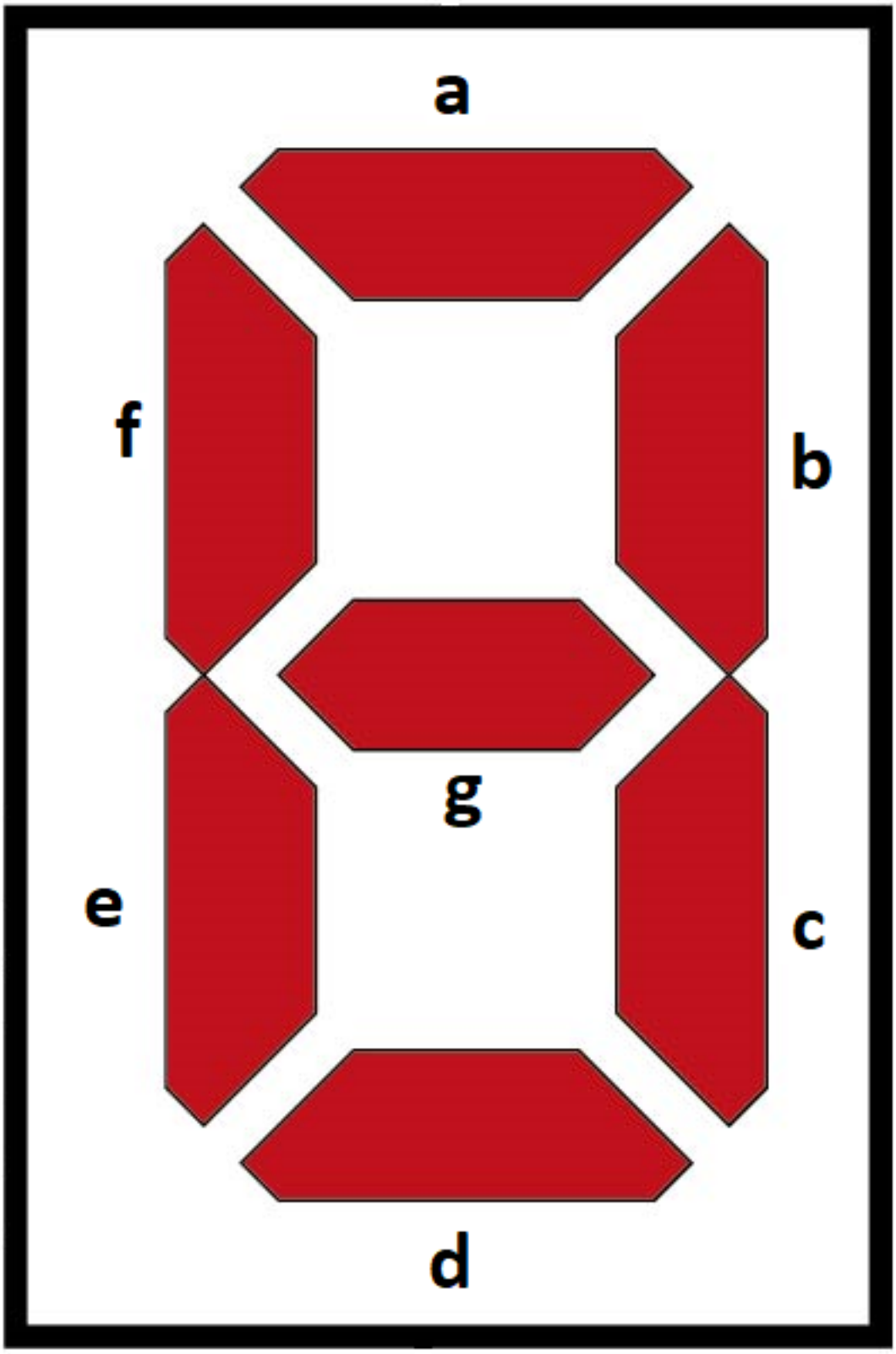


Figure 56: display

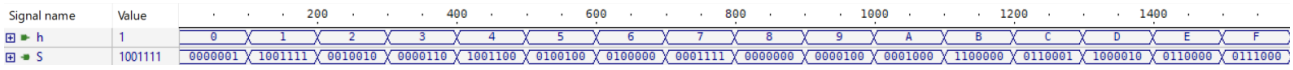


Figure 57: Simu

in[6]: "no use"

in[7]: "no use"

2. For the output, which is a 7-bit vector "S", the overall project pins connected to the proposed circuit are as follows:

out[0]: "S[0]" //Segmento g

out[1]: "S[1]" //Segmento f

out[2]: "S[2]" //Segmento e

out[3]: "S[3]" //Segmento d

out[4]: "S[4]" //Segmento c

out[5]: "S[5]" //Segmento b

out[6]: "S[6]" //Segmento a

out[7]: "no use"

The signals, both input and output, are logic highs and lows, that is, usually 5 volts to define a logic "1", and 0 volts for a logic "0". Let us remember that in the case of the output, an inverse logic is applied to the output since it is a common anode display, but in essence they are logical "1" and "0".

How to test

To check the operation, a 4-position dip switch is connected to the input, connected to a suitable power supply for the system, with its respective precautions (resistances), according to the number that you want to show on the display, for which appropriately connect each switch to the corresponding bit it represents. For the output, it is convenient to connect a 7-segment display (common anode) to corroborate its operation, according to the pins that correspond to each segment, mentioned in the previous section.

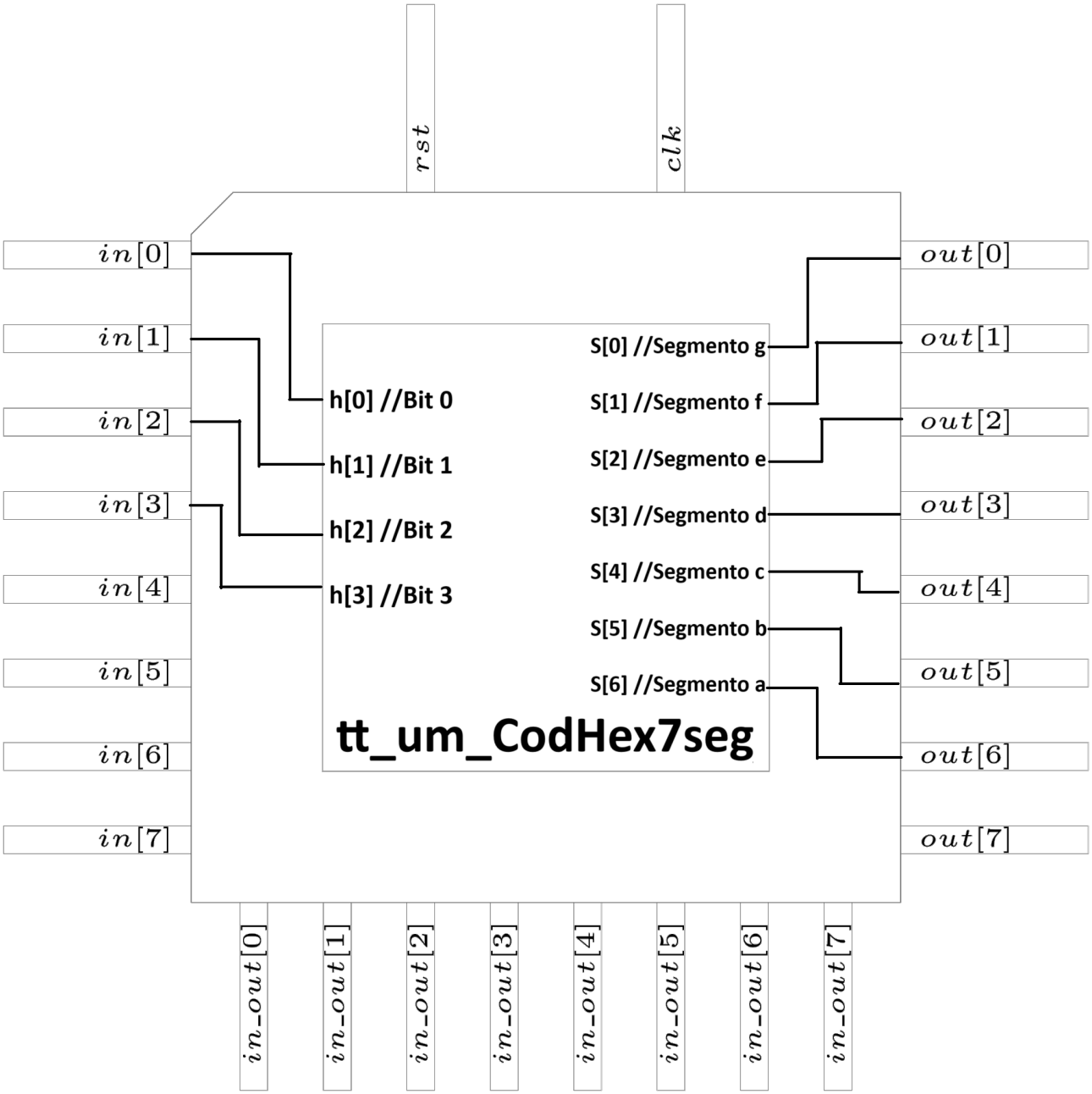


Figure 58: latin2

External hardware

A 4-position DIP Switch for the input, which will serve to form the 4-bit binary number, along with its proper power supply, and a 7-segment display (common anode), to visualize its operation, connected with due precautions to avoid damage. Added to all this is a breadboard to place these components.

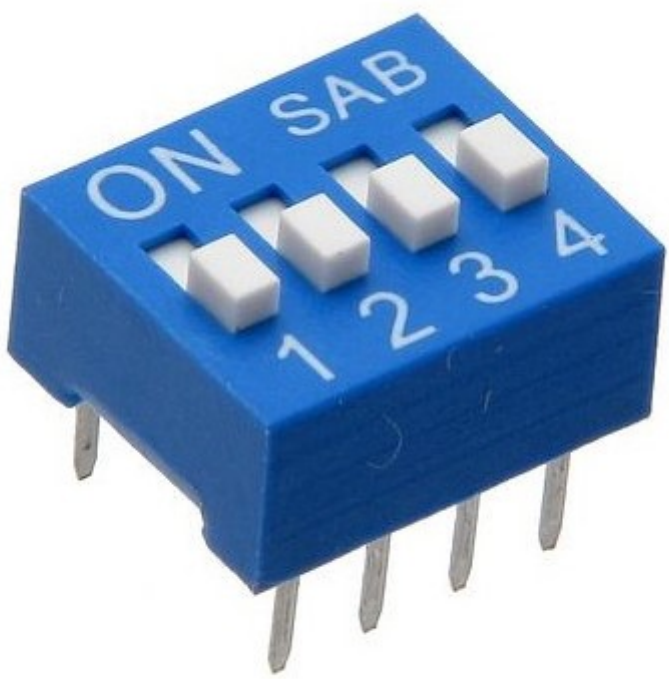


Figure 59: SWI-18-3

Pinout

#	Input	Output	Bidirectional
0	Bit 0	Segmento g	no use
1	Bit 1	Segmento f	no use
2	Bit 2	Segmento e	no use
3	Bit 3	Segmento d	no use
4	no use	Segmento c	no use
5	no use	Segmento b	no use
6	no use	Segmento a	no use

#	Input	Output	Bidirectional
7	no use	no use	no use

Figure 60: AR1112-KPS1203D-Fuente-de-Alimentacion-120V-3A-V8

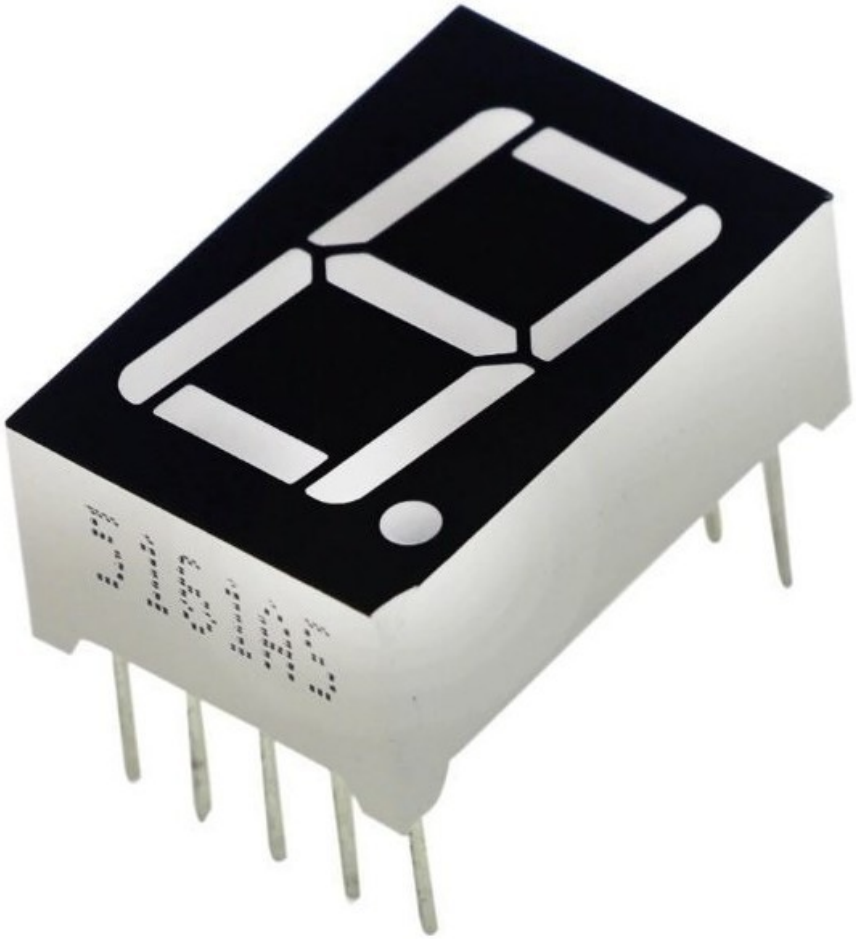


Figure 61: Displaysa

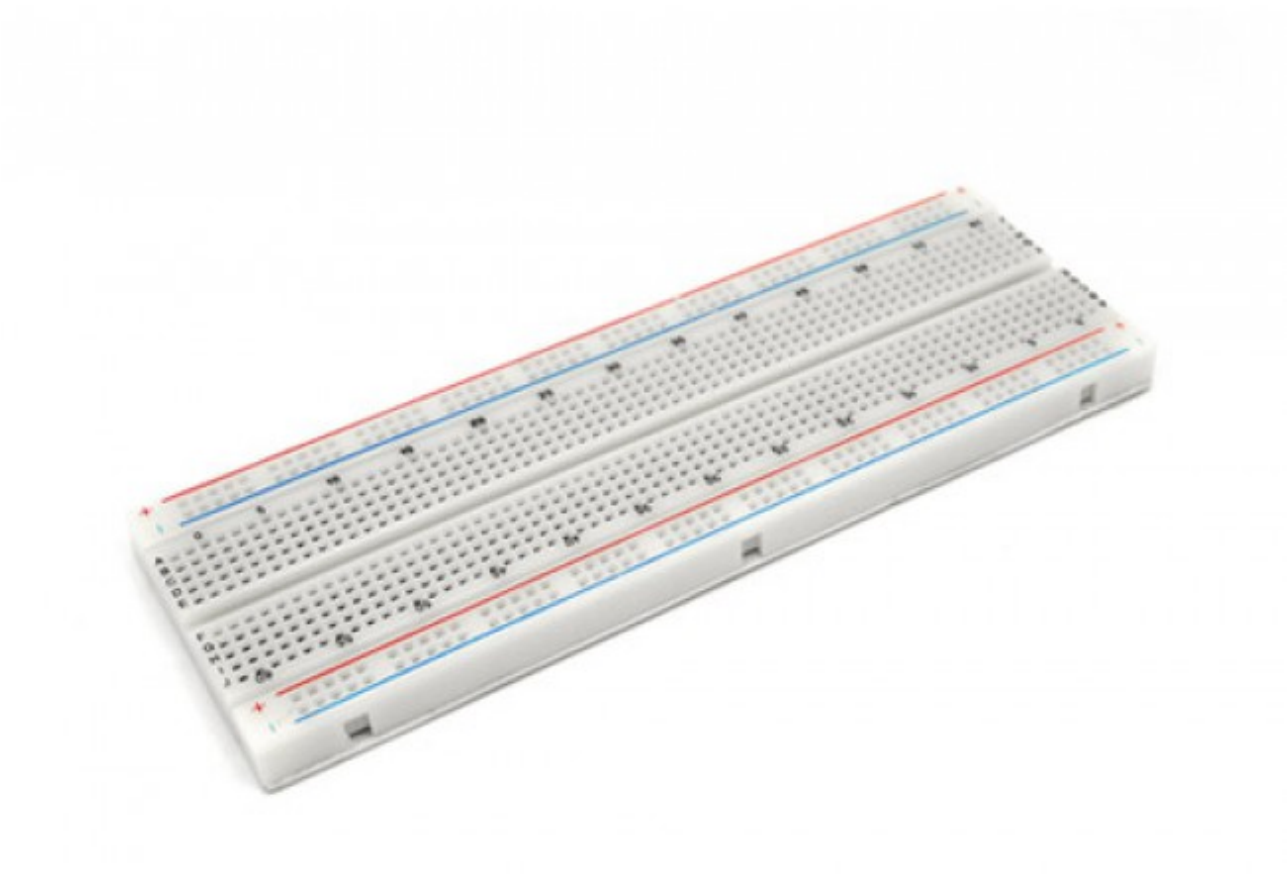


Figure 62: image

Latch RAM (64 bytes) [620]

- Author: Mike Bell
- Description: 64 byte RAM built out of latches
- GitHub repository
- HDL project
- Mux address: 620
- Extra docs
- Clock: 0 Hz

What's the project?

A 64 byte RAM implemented using 512 latches.

Resetting the project does not reset the RAM contents.

How to test

To read a byte from memory:

- Set the `addr` pins to the desired address and set `wr_en` low
- Pulse `clk`
- `data_out` (the output pins) reads the value at the memory location.

To write a byte to memory:

- Set the `addr` pins to the desired address, set `data_in` (the bidirectional pins) to the desired value, and set `wr_en` high
- Pulse `clk`
- The memory location will be written on the next cycle. The `data_out` pins will now read the old value at this address.
- The next cycle can not be a write.

On the cycle immediately after a write the value of `wr_en` and `data_in` will be ignored - the cycle is always a read. If `addr` is left the same then the value read will be the value just written to that location.

How it works

Setting values into latches reliably is a little tricky. There are two important considerations:

- The latch gate must only go high for the latches for the byte that is addressed. The other latch gates must not glitch.
- The data must be stable until the latch gate is definitely low again.

To ensure the restrictions are met, writes take 2 cycles, and only 1 write can be in flight at once, so the cycle after any write is always treated as a read.

The scheme used is described in detail below.

Writing: Ensuring stable inputs to the latches. The write address, `addr_write`, is always set to the same value for 2 clocks when doing a write. When the write is requested `addr_write` and `data_to_write` are captured. `wr_en_next` is set high. If `wr_en_next` was already high the write is ignored, so the inputs to the latches aren't modified when a write is about to happen.

On the next clock, `wr_en_valid` is set to `wr_en_next`. `addr_write` is stable at this time so the `sel_byte` wires, that contain the result of the comparison of the write address with the byte address for each latch, will already be stable at the point `wr_en_valid` goes high.

`wr_en_ok` is a negative edge triggered flop that is set to `!wr_en_valid`. This will therefore go low half a clock after `wr_en_valid` is set high. And because two consecutive writes are not allowed it will always be high when `wr_en_valid` goes high.

The latch gate is set by anding together `wr_en_valid`, `wr_en_ok` and the `sel_byte` for that byte. This means the latch gate for just the selected byte's latches goes high for the first half of the write clock cycle. data_to_write is stable across this time (it can not change until the next clock rising edge), so will be cleanly captured by the latch when the latch gate goes low.

Reading: Mux and tri-state buffer. Reading the latches is straightforward. However, a 64:1 mux for each bit is relatively area intensive, so instead for each bit we have 4 16:1 muxes feeding 4 tri-state buffers.

Only the tri-state buffer corresponding to the selected read address is enabled, and the output is taken from the wire driven by those 4 buffers.

To minimize contention, the tri-state enable pin of the buffers is driven directly from a flop which captures the selected read address directly from the inputs, at the same cycle as the `addr_read` flops are set.

The combined output wire then goes to a final buffer before leaving the module, ensuring the outputs are driven cleanly.

Pinout

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

Serial to Parallel Register [621]

- Author: Ricardo M. Rocha Torres
- Description: This is a simple Serial to Parallel Register
- GitHub repository
- HDL project
- Mux address: 621
- Extra docs
- Clock: 0 Hz

How it works

A erial-parallel register, is a digital circuit used to store data sequentially and then transfer it in parallel. It works in the following way:

Serial Input: Data is input sequentially, bit by bit, through a single input line. These bits move through the register, being temporarily stored in the register's memory cells.

Temporary Storage: As bits are entered, each bit is loaded into a memory cell within the register. This is typically done using a serial shifting mechanism, where each new bit pushes the next bit to the next memory cell, thus shifting all previous bits forward.

Parallel Transfer: When all the bits have been entered into the register serially and temporarily stored, they can be transferred simultaneously or in parallel from the memory cells of the register to a parallel width data bus. This is achieved by loading each bit stored in the memory cells into parallel output lines that are connected to the data bus.Typically, type D latches or flip flops are used, which are controlled by the clock, which determines when a data chain begins or ends.

Control and Synchronization: The operation of the serial-parallel register is controlled by control signals that indicate when to start inputting data, when to stop serial input, when to start parallel transfer, and when to stop transfer. Accurate synchronization is crucial to ensure that data moves correctly through the register and is transferred to the data bus at the right time.

For this project, a 4-bit serial-parallel register was made, which consists of a clock, a reset, the serial input and the parallel output.

The importance of this register is found in a binary search block used in converters such as ADC SAR by its acronym Digital Analog Converter successive approximation register which introduces a series of data into the system in serial form and requires a series in parallel to determine the value to convert.

How to test

The testbench used was proposed, carried out in ACTIVE HDL-Student Version, the stimuli used were.

10ns period clock. Reset, through a formula which at 0 fs is 1 and 0 after 1 ns to clean the data and start with a known value with is 0. Serial input, a 20 ns clock.

The parallel output is updated every 4 clock cycles and displays the result until it updates the next 4 clock cycles with a new result.

External hardware

Wave generator:This controls the system clock externally

Switch, connected in the reset and is used when we perform a conversion, It can also be used with a button or with a wave generator using a square pulse once. The reset switch value must be 0 to allow a value that is different from 0 on the parallel output.

Logic Analyzer. This allows a serial signal to be introduced into the system that varies its values non-periodically to read its conversion in parallel, the same logic analyzer can read the output in parallel. Keysight 1681AD Logic Analyzer in INAOE con be used. Another way is to use an FPGA programmed with serial values and it can obtain the output values in parallel.

Pinout

#	Input	Output	Bidirectional
0	no use	no use	Bit 0
1	no use	no use	Bit 1
2	no use	no use	Bit 2
3	no use	no use	Bit 3
4	no use	no use	no use
5	no use	no use	Serie_in
6	no use	no use	rst
7	no use	no use	clk

Combination Lock [622]

- Author: Eric Cheng
- Description: 4-bit combination lock with a maximum of 3 attempts per lock and a master reset
- GitHub repository
- Wokwi project
- Mux address: 622
- Extra docs
- Clock: 10000 Hz

How it works

Everything is controlled using the CLOCK, RESET, and input pins. The first step after starting the simulation is pressing RESET.

Setting a Passcode To set a passcode, IN0 will need to be set to HIGH for the duration of the setup. Then, create a combination of IN1, IN2, IN3, and IN4. This will be your passcode after setting IN0 back to LOW. The passcode can be reset anytime with IN0. OUT 0~3 represent the current password of the lock.

Unlocking To unlock the combination lock, you will set IN1, IN2, IN3, and IN4 to the previous combination in `Setting a Passcode`. To verify, set IN5 to HIGH. If correct, the LED at OUT4 will go HIGH. The lock will only be in an unlocked state if IN5 is held at HIGH. Returning IN5 back to LOW will lock the combination lock again.

Number of Attempts The user will only have 3 tries to get the right combination before the input pins IN1, IN2, IN3, and IN4 become pin-locked (unusable). Once the lock become unusable, OUT5 will go LOW. A press of the RESET button will turn it back to normal.

How to test

The normal flow of using the design is to first set a password of your liking (assuming you are the admin). Then, the lock would be free to use. If in a case where the user failed three times to unlock the lock, it is up to the admin to reset the pin-lock for continued use.

External hardware

A microcontroller (or other hardware of sorts) that allows only the admin to be able to reset the pin-lock is reccomended. Buttons, switches, or other forms of input are necessary for physical operation of the lock.

Pinout

#	Input	Output	Bidirectional
0	Set	CurrPswd[0]	
1	Pswd[0]	CurrPswd[0]	
2	Pswd[1]	CurrPswd[0]	
3	Pswd[2]	CurrPswd[0]	
4	Pswd[3]	Unlocked	
5	Enter	PinLocked	
6			
7			

PWM [623]

- Author: NoeReyes
- Description: This project involves Pulse Width Modulation, enabling the duty cycle to be adjusted between 10% and 90% using switches.
- GitHub repository
- HDL project
- Mux address: 623
- Extra docs
- Clock: 50000000 Hz

How it works

Pulse Width Modulation (PWM) is a technique used in electronics to control the average voltage applied to a load by rapidly switching a digital signal on and off at varying duty cycles. This method is commonly employed in applications such as motor speed control, LED brightness adjustment, and power regulation. By adjusting the duty cycle of the signal, PWM enables precise control over the output voltage or power, allowing for efficient and flexible manipulation of electrical devices.

My project involves Pulse Width Modulation (PWM), allowing the duty cycle to be adjusted between 10% and 80% using 3 switches with 7 different combinations. Each combination increments the duty cycle by 10%. For example, '000' represents a 10% duty cycle, and '111' represents an 80% duty cycle. The PWM was designed for a frequency of 1KHz.

The above image represents the PWM module that was designed.

[2:0] LOAD	Duty Cicle
000	10%
001	20%
010	30%
011	40%
100	50%
101	60%
110	70%
111	80%

In the previous table, the variation of the duty cycle is shown as a function of the LOAD input combination.

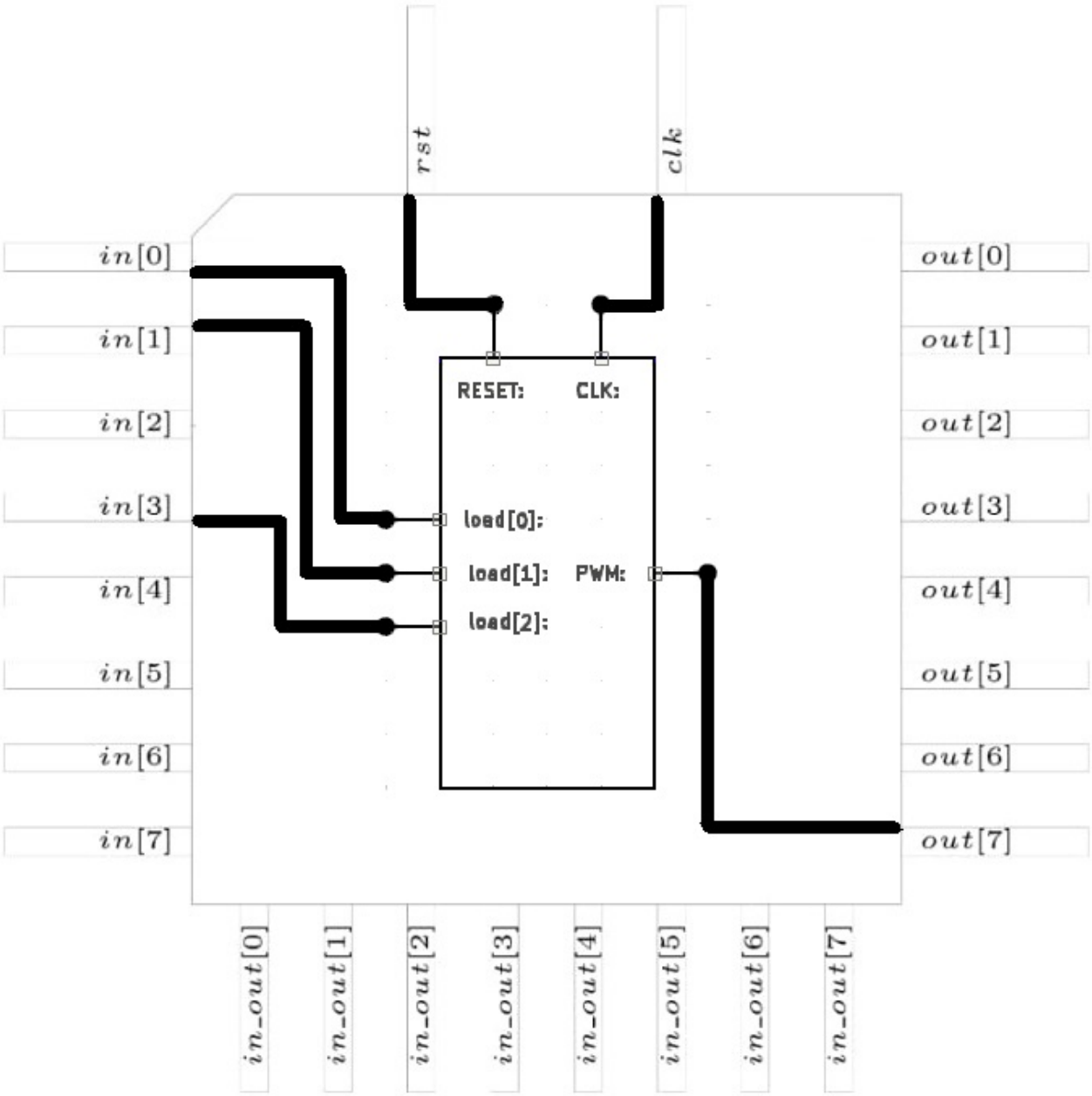


Figure 63: PWM

How to test

The LOAD input should be connected to a switch, CLK is connected to a 50MHz clock, and RESET to a button. To ensure proper operation, press RESET to set the initial conditions. Once this is done, choose the LOAD input combination to set the desired duty cycle.

External hardware

The external hardware includes one LED, a 1k ohm resistor, and three 2-position switches.

Pinout

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

SPELL [642]

- Author: Uri Shaked
- Description: SPELL is a minimal, cryptic, stack-based programming language crafted for The Skull CTF
- GitHub repository
- HDL project
- Mux address: 642
- Extra docs
- Clock: 10000000 Hz

How it works

SPELL is a minimal, stack-based programming language created for The Skull CTF.

The language is defined by the following cryptic piece of Arduino code:

```
void spell() {

                  uint8_t*a,pc=16,sp=0,
                s[32]={0},op;while(!0){op=
              EEPROM.read(pc);switch(+op){case
            ',':delay(s[sp-1]);sp--;break;case'>':
           s[sp-1]>>=1|1;break;case'<':s[sp-1]<<=1;
         break;case'=':pc=s[sp-1]-1;sp--;break;case
        '@':if(s[sp-2]){s[sp-2]--;pc=s[sp-1]-1;sp+=
       1;}sp-=2;break;case'&':s[sp-2]&=s[sp-1];sp-=1;
       break;case'|':s[sp-2]|=s[sp-1];sp-=1;break;case
     '^':s[sp-2]^=s[sp-1];sp--;break;case'+':s[sp-2]+=
   s[sp-1];sp=sp-1;break;case'-':s[sp-2]-=s[sp-1];sp--;
   break;case'2':s[sp]=s[sp-1];sp=sp+1;break;case'?':s[
 sp-1]=EEPROM.          read(s[sp-1]|0        );break;case
  "!!!"[0]:              666,EEPROM              .write(s
   [sp-1]                 ,s[sp-2]               );sp=+
    sp-02;                ;break;                1;case
    "Arr"[                1]:  s[+               sp-1]=
    *(char*)             (s[+   sp-1            ]);break
      ;case'w':*   (char*)(     s[+sp-1])  =s[sp-+2];
        sp-=2;break;case+       'x':s[sp] =s[sp-1
           ];s[sp-1]=s[sp   +    -2];s[sp-2]=s[
            0|sp];break;     ;;    case"zzz"[0
             ]:sleep();"   Arrr  ";break;case
```

```
  255  :return;;  default:s  [sp]
   =+   op;sp+=    1,1   ;}pc=
    +    pc  +      1;      %>
}
```

This design is an hardware implementation of SPELL with the following features:

- 32 bytes of program memory (volatile, simulates EEPROM)
- 32 bytes of stack memory
- 8 bytes of internal RAM
- 8 I/O pins (uio)

Initially, all the program memory is filled with `0xFF`, and the stack and data memory are filled with `0x00`. The program counter and the stack pointer are both set to `0x00`.

To load a program or inspect the internal state, the design provides access to the following registers via a simple serial interface:

Address	Register name	Description
0x00	PC	Program counter
0x01	SP	Stack pointer
0x02	EXEC	Execute-in-place (write-only)
0x03	STACK	Stack access (read the top value, or push a value)

The serial interface is implemented using a shift register, which is controlled by the following signals:

Pin	Type	Description
`reg_sel`	input	Select the register to read/write
`load`	input	Load the selected register with the value from the shift register
`dump`	input	Dump the selected register value to the shift register
`shift_in`	input	Serial data input
`shift_out`	output	Serial data output

When `load` is high, the value from the shift register is loaded into the selected register. When `dump` is high, the value of the selected register is dumped into the shift register, and can be read after two clock cycles by reading `shift_out` (MSB first).

For example, if you want to read the value of the program counter, you would:

1. Set `reg_sel` to 0x00 and set `dump` to 1
2. Wait for two clock cycles for the first bit (MSB) to appear on `shift_out`.
3. Read the remaining bits from `shift_out` on each clock cycle.

To write a value to the program counter, you would:

1. Write the value to the shift register, one bit at a time, starting with the **MSB**.
2. Set `reg_sel` to 0x00 and set `load` to 1.
3. Wait for a single clock cycle for the value to be loaded.

Writing an opcode to the `EXEC` register will execute the opcode in place, without modifying the program counter (unless the opcode is a jump instruction).

The `STACK` register is used to push a value onto the stack or read the top value from the stack (for debugging purposes).

Data memory and I/O registers The data memory space is divided into two regions:

Address range	Description
0x00 - 0x07	General-purpose data storage (data memory)
0x20 - 0x5F	I/O and control registers

Other addresses are unmapped.

The following registers are available in the data memory space:

Address	Name	Description
0x36	PINB	Read the value of the `portb` pins, or toggle the output when written to
0x37	DDRB	Set the direction of the `portb` pins (0 = input, 1 = output)
0x38	PORTB	Write to the `portb` pins

For example, to toggle the value of the `portb[2]` (`uio[2]`) pin, you would write `0x04` to the `PINB` register.

How to test

To test SPELL, you need to load a program into the program memory and execute it. You can load the program by repeatedly executing the following steps for each byte of the program:

1. Write the byte to the top of the stack (using the `STACK` register)
2. Write the address of the byte in the program memory to top of the stack
3. Write the opcode `!` to the `EXEC` register

After loading the program, you can execute it by writing the address of the first byte in the program memory to the `PC` register, and then pulsing the `run` signal.

Test program The following program which will rapidly blink an LED connected to the `uio[0]` pin. The program bytes should be loaded into the program memory starting at address 0:

```
[1, 55, 119, 1, 54, 119, 250, 44, 3, 61]
```

For a more complex test program, see the TT06 SPELL bringup script.

External hardware

None

Pinout

#	Input	Output	Bidirectional
0	run	sleep	gpio[0]
1	step	stop	gpio[1]
2	load	wait_delay	gpio[2]
3	dump	shift_out	gpio[3]
4	shift_in		gpio[4]
5	reg_sel[0]		gpio[5]
6	reg_sel[1]		gpio[6]
7			gpio[7]

RNG3 [654]

- Author: Luca Collini
- Description: NIST RNG TESTS
- GitHub repository
- HDL project
- Mux address: 654
- Extra docs
- Clock: 50000000 Hz

How it works

This design takes a bit every clock cycles and evaluates if the bit source is random. This particular test is the Monobit test from NIST 800.22. The output is given every 65536 cycles. The is_random signal is to be checked only when the valid signal is high.

How to test

Provide Clock and input bit.

External hardware

Non for now. Planning to add soon

How to use

send one bit per clock cycle to the epsilon port. check is_random when valid is high. The design evaluates every 65536 bits.

Pinout

#	Input	Output	Bidirectional
0	epsilon	is_random	
1		valid	
2			
3			

#	Input	Output	Bidirectional
4			
5			
6			
7			

4-Bit Full Adder and Subtractor with Hardware Trojan [672]

- Author: Jeremy Hong
- Description: Externally triggered hardware trojan in a 4-bit full adder and subtractor
- GitHub repository
- Wokwi project
- Mux address: 672
- Extra docs
- Clock: 100000 Hz

How it works

4-Bit Full Adder and Subtractor with hardware trojan inserted in the critical path

How to test

Use DIP Switches, provide external input for last B input bit and hardware trojan trigger signal.

External hardware

Pattern Generator and logic analyzer

Pinout

#	Input	Output	Bidirectional
0	Add/Subtract	7 Segment Display	Input - B3
1	A0	7 Segment Display	Input - Hardware Trojan Trigger
2	B0	7 Segment Display	Output - LFSR 1
3	A1	7 Segment Display	Output - LFSR 2
4	B1	7 Segment Display	Output - LFSR 3
5	A2	7 Segment Display	Input - LFSR 3
6	B2	7 Segment Display	Input - LFSR 4
7	A3	7 Segment Display	Output - LFSR 7

Notre Dame Dorms LED [673]

- Author: Allison Fleming, Daniel Yu, Matthew Morrison
- Description: Solves a puzzle based on the correct selection of Notre Dame dorms
- GitHub repository
- Wokwi project
- Mux address: 673
- Extra docs
- Clock: 0 Hz

How it works

There are eight switches on this DIP Switch, each representing a dorm on campus. The LED can be lit up with a minimum of three switches turned on. These three switches represent the best male dorm (Alumni, Carroll, Keenan, or Dillon), the best female dorm (Flaherty, Welsh Fam, or PE), and the best gender-neutral dorm (Fischer Grad). For example, a valid combination would be Alumni, Flaherty, Fischer Grad.

Switch #

1- Alumni, 2- Carroll, 3- Keenan, 4- Dillon, 5- Flaherty, 6- Welsh Fam, 7- Pasquerilla East, 8- Fischer Grad

How to test

This puzzle is supposed to be a trial and error practice for students to learn about logic gates. Thus, to test it, play around with using different switches to see what lights up the LED.

External hardware

One external LED is needed

Pinout

#	Input	Output	Bidirectional
0			
1			
2			

#	Input	Output	Bidirectional
3			
4			
5			
6			
7			

Tiny ALU [674]

- Author: Adam Majmudar
- Description: A super simple ALU from my GPU design
- GitHub repository
- HDL project
- Mux address: 674
- Extra docs
- Clock: 0 Hz

How it works

I'm designing a minimal GPU, which is a large project and also challenging to connect with the tt06 pins. Rather than try to wrangle with this setup and buy up a bunch of tiles, I figured I would put together something simple so I can get the peace of mind of having a submission!

I'll still be shipping my GPU project, which I'll add a link to here soon - but for tapeout I didn't want to block on that.

How to test

The test bench is quite simple - just testing the addition function of the ALU.

External hardware

No external hardware is needed.

Pinout

#	Input	Output	Bidirectional
0	rs[0]	alu_out[0]	alu_arithmetic_mux[0]
1	rs[1]	alu_out[1]	alu_arithmetic_mux[1]
2	rs[2]	alu_out[2]	
3	rs[3]	alu_out[3]	
4	rt[0]		
5	rt[1]		
6	rt[2]		

#	Input	Output	Bidirectional
7	rt[3]		

clk frequency divider controled by rom [675]

- Author: Gilberto Ramos Valenzuela
- Description: Clock divider control by ROM
- GitHub repository
- HDL project
- Mux address: 675
- Extra docs
- Clock: 50000000 Hz

How it works

This is a clock divider. It works by dividing the frequency by a counter. The parameters for dividing were calculated for 27 frequencies, multiples of 50mHz. These parameters are stored in a ROM, which can be accessed by a 4-bit input, and it can output 28-bit data.

How to test

To function, it needs a 50MHz input. You select the frequency by choosing it from the table shown in the Readme.md. Then, you get the output through an output pin, which can be tested by an oscilloscope and fed into a microcontroller or any circuit requiring a square wave signal to function.

External hardware

Requires a 50MHz oscillator.

Pinout

#	Input	Output	Bidirectional
0	F_select [0]		clk
1	F_select [1]		clk_out
2	F_select [2]		
3	F_select [3]		
4	F_select [4]		
5	reset_n		
6			

#	Input	Output	Bidirectional
7			

SAP-1 Computer [676]

- Author: Jonathan Zhou, Rana Singh, Anika Agarwal
- Description: Simple as Possible computer with multiplier and divider into ASIC
- GitHub repository
- HDL project
- Mux address: 676
- Extra docs
- Clock: 10000000 Hz

How it works

```
(Forked from Brandon Cruz's SAP-1 Design)

Originally, Malvino and Brown presented the SAP-1 architecture in a boo
The design gained massive popularity when it was build as a bread board
The architecture contains various modules, including
- Clock
- Program Counter
- Register A
- Register B
- Adder
- Multiplier
- Divider
- Memory
- Instruction Register
- Bus
- Controller
This design doesn't have inputs, it is dependent only on the clock that
Its operation consists on the communication that that bus provides betw
and the enable signal allows the bus to receive a signal. The bus is 8-
are also 8-bit registers.
The computer can only perform addition, whether it is positive numbers
The signals information is stored within the memory module. There bus o
the instruction execution set gives the SAP-1 a total of six stages fro
The more important module      is the controller. It controlls the asse
The stages 3 to 5 five depend on the instructions of the operation code
```

How to test

```
Design Output Reading Section
The design is engineered to read the output signal generated from the b
of the add and subtract operations executed by the design. Currently, t
is through an oscilloscope. However, a significant enhancement would be
controlling a 3 7-segment display to show the numbers on the 8-bit bus
```

External hardware

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

Oscilloscope

Pinout

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

PWM Controller [677]

- Author: Ziyi Zhao, Yuchen Ma
- Description: A Low-Cost Pulse Width Modulation (PWM) Controller
- GitHub repository
- HDL project
- Mux address: 677
- Extra docs
- Clock: 0 Hz

How it works

The PWM generates a 10 MHz PWM signal whose duty cycle can be adjusted using two buttons. The PWM duty cycle can be increased or decreased in steps, constrained between 10% and 90%.

How to test

Change the inputs ui_in to simulate button presses and check if the PWM duty cycle increases or decreases as expected.

Pinout

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

4 bit RAM [678]

- Author: Alejandro Silva Juarez
- Description: Is a Memory RAM (4 bits)
- GitHub repository
- HDL project
- Mux address: 678
- Extra docs
- Clock: 0 Hz

How it works

It is a 4 bit RAM

How to test

It is tested with 4 inputs, the clock, the write enable input and the 4-bit input data, the output is 4 bits as well.

External hardware

The chip may need a Microcontroller, Raspberry, Arduino or FPGA for data inputs and memory addresses or 8 switches can be placed for data inputs and memory addresses.

Pinout

#	Input	Output	Bidirectional
0	Memory entry address [0]	Memory output data [0]	RAM write enable input
1	Memory entry address [1]	Memory output data [1]	
2	Memory entry address [2]	Memory output data [2]	
3	Memory entry address [3]	Memory output data [3]	
4	Memory input data [0]		
5	Memory input data [1]		
6	Memory input data [2]		
7	Memory input data [3]		

8bit ALU [679]

- Author: David Parent,Chih-Kaun Ho, Edric Ong
- Description: Building a simple 8-bit unsigned Arithmetic logic unit (ALU)
- GitHub repository
- HDL project
- Mux address: 679
- Extra docs
- Clock: 1000 Hz

How it works

How to test

External hardware

ADALM2000. https://www.analog.com/en/resources/evaluation-hardware-and-software/evaluation-boards-kits/adalm2000.html#eb-overview

Pinout

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

ALU with a Gray and Octal decoders [680]

- Author: Luis Antonio Quezada Santos, Santiago Robledo Acosta, José Miguel Rocha Pérez
- Description: This is a simple 3 bit ALU with 4 operations: Substraction, Addition, XOR and AND which its output is conected to an Octal and Gray Decoders.
- GitHub repository
- HDL project
- Mux address: 680
- Extra docs
- Clock: 0 Hz

How it works

This is a pure combinational design. This is a simple ALU (Arithmetic Logic Unit) whose output is connected to two different decorders, an Octal decoder for two 7 segments displays and an Gray decoder for the same two 7 segments displays. The output displays will show the result of the operations between two 3 bit numbers according to Sel_A_in and Sel_M_in.

The Sel_A_in has the following operations according to its value.

- Sel_A_in = 2'b00 , the ALU will be set in substraction. Num_A_in - Num_B_in.
- Sel_A_in = 2'b01 , the ALU will be set in Adition. Num_A_in + Num_B_in.
- Sel_A_in = 2'b10 , the ALU will be set in XOR. Num_A_in ^ Num_B_in.
- Sel_A_in = 2'b11 , the ALU will be set in AND. Num_A_in & Num_B_in.

The Sel_M_in has the following operations.

- Sel_M_in = 1'b0 , The output will be displayed in the octal system as the multiplexer selects the output of the Octal Decoder.
- Sel_M_in = 1'b1 , The output will be displayed in the Gray system as the multiplexer selects the output of the Gray Decoder.

Note: The Gray Decoder has been specially decoded to be shown in a decimal system for the 7 segments displays.

How to test

In order to test this device, you will need to input the numbers to the pin where Num_A_in and Num_B_in are located, this values go from 0 =3'b000 up to 7 = 3'b111. From this point forward modify the correponding bits on the correspondent selectors based on list displayed in How it works and see the result on the 7 segment displays (External).

External hardware

For external hardware you'll need: . An external DC power source. . 14 330 ohms resistors. . 2 7 segments displays common cathode.

Pinout

#	Input	Output	Bidirectional
0	Sel_A_in [0]	Disp_out[6]	
1	Sel_A_in [1]	Disp_out[7]	Disp_out[0]
2	Num_B_in [0]	Disp_out[8]	Disp_out[1]
3	Num_B_in [1]	Disp_out[9]	Disp_out[2]
4	Num_B_in [2]	Disp_out[10]	Disp_out[3]
5	Num_A_in [0]	Disp_out[11]	Disp_out[4]
6	Num_A_in [1]	Disp_out[12]	Disp_out[5]
7	Num_A_in [2]	Disp_out[13]	Sel_M_in

EVEN AND ODD COUNTERS [681]

- Author: Dr.LIPIKA GUPTA, DEVRAJ, JUGRAJ
- Description: WE CAN CHOOSE TO COUNT EVEN AND ODD BCD
- GitHub repository
- Wokwi project
- Mux address: 681
- Extra docs
- Clock: 10000 Hz

How it works

Welcome, This is basically a even and odd BCD counter:- *If input 1 is high, the even counter is on,* If input 2 is high, the odd counter is on.

How to test

We can test it by making input 1 high and connecting 7-segment display to output, if the even count of BCD is shown, the logic circuit is cleadred the test .

External hardware

- Push button
- 7 - segment display

Pinout

#	Input	Output	Bidirectional
0	IN1	OUT0	
1	IN2	OUT1	
2		OUT2	
3		OUT3	
4		OUT4	
5		OUT5	
6		OUT6	
7			

Generador digital trifásico [682]

- Author: Jesús García Guzmán, Jorge Rafael Frantos Morales, Brian Isaías Landa León, Alan Eduardo Peralta Fuentes, Juan José Guzmán Lagunes, Adolfo de Jesús García Méndez, Paola Gabriela Rodríguez Sánchez, José de Jesús Sánchez Hernández
- Description: El proyecto consiste en un microgenerador digital trifásico que tiene una señal de reloj de entrada y produce tres señales de salida con desfases de 120° entre sí, con el objetivo de demostrar su viabilidad y funcionalidad en aplicaciones de ingeniería eléctrica y electrónica.
- GitHub repository
- HDL project
- Mux address: 682
- Extra docs
- Clock: 0 Hz

How it works:

El circuito requiere una entrada de reloj, en este caso llamada "SE", ya que no se usará el reloj del sistema, para manejar un registro de tres bits en configuración de anillo (contador Johnson), con tres salidas digitales desfasadas 120° entre ellas.

La idea de aplicación del circuito se basa en alimentar motores trifásicos que se pudieran implementar en el área biomédica o similares. Por lo que para llegar a alimentar un motor se require procesar de las señales de salida de este circuito generador, la idea es filtrar analógicamente las ondas digitales para obtener tres señales senoidales de amplitudes iguales y desfasadas 120°. Esto como complemento del proyecto, pues en este circuito solo se llega hasta la parte de tres salidas cuadradas desfasadas 120°.

En el enlace siguiente se tiene el circuito en diagrama de bloques, en donde se muestra cómo se tiene el desfase de las tres señales de salida. Para poder usarlo se usó un switch, en el que manualmente encenciéndolo y apagándolo, se simula la función de pulsos de entrada y se observa el desfasamiento entre los 3 leds que se tienen en la salida. https://wokwi.com/projects/392934671873904641

Para comprobar la funcionalidad del código en verilog de nuestro ciircuito, se simuló usando la herramienta de simulación del software Quartus II. En la imagen siguiente se puede apreciar que en las salidas "P0, P1 y P2" existe un desfasamiento de 120° entre sí, teniendo una sola señal de entrada de reloj. La señal de reloj de entrada se designó como "SE". De igual forma el código se probó en una FPGA Altera Cyclone II, para comprobar su funcionamiento y arrojó los resultados esperados en las simulaciones. Siguiendo el procedimiento que se describe en este mismo archivo README, en la sección llamada "How to test".

How to test:

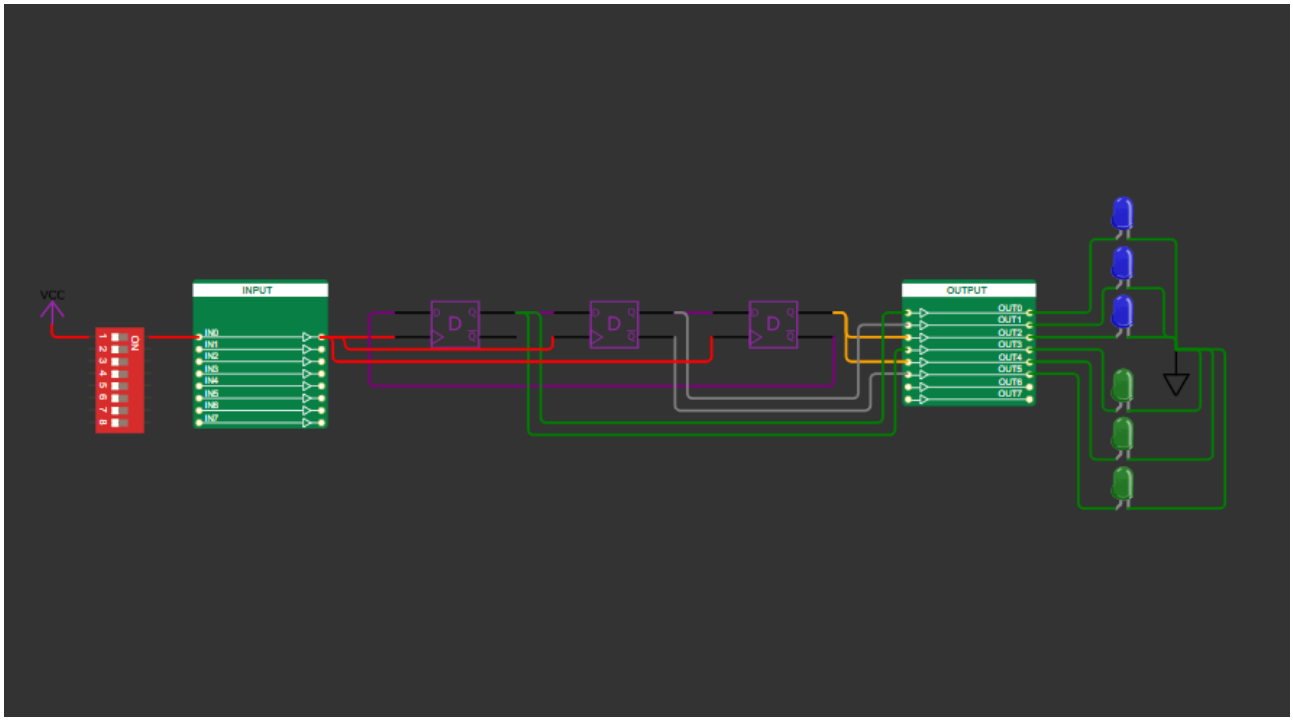


Figure 64: image

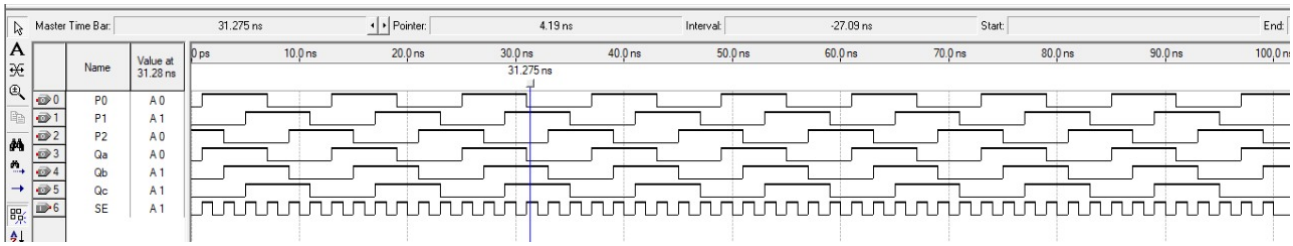


Figure 65: Imagen de WhatsApp 2024-03-31 a las 20 20 17_92d28c8a

Para probar el circuito se requieren un generador de señales digitales y un osciloscopio, preferentemente con al menos 3 canales.

El circuito necesita una señal de reloj de entrada, puesto que no se usó el reloj de la placa, se requiere conectar un generador de señales en la entrada del circuito la cual está nombrada como "SE". Esta señal de entrada se asignó al pin bidireccional "uio[7]"

La señal "Se" que se inyectará al circuito debe ser de pulsos (una señal cuadrada), con un voltaje que suba a 5 Vcd y baje a 0 Vcd, con un ciclo de trabajo del 50%.

La frecuencia que se tenía pensada implementar para la señal de entrada es de 60 Hz, pues la idea original era alimentar un pequeño motor que pudiera servir para usos biomédicos o similares, pero para fines de prueba se puede usar cualquier otra frecuencia, siempre y cuando el osciloscopio que se vaya a conectar a las salidas, sea capaz de leer señales a dicha frecuencia.

Antes de conectar la señal de entrada al circuito, es importante comprobar sus valores en un osciloscopio y un voltímetro, para asegurarse que la señal tiene la forma y magnitud adecuadas, y que el circuito no se dañe por exceso de voltaje.

Las salidas que estarán desfasadas 120° entre sí serán las llamadas P0, P1 y P2. Estas tres salidas se asignaron a los pines bidireccionales "uio[2], uio[1], uio[0]" respectivamente. A estos pines de salida se deberán conectar las entradas de los tres canales del osciloscopio, todo esto habiendo calibrado antes el osciloscopio y verificando que la señal que se le inyectará al circuito es la correcta. También es importante conectar las tierras de los cables del generador de señales digitales y del osciloscopio.

Una vez habiendo hecho todo lo anterior, se debería de poder observar en las salidas, los tres pulsos de igual magnitud, desfasados 120° entre sí.

Si se desean observar las salidas complementarias "Qa, Qb y Qc" estas se asignaron a los pines "uio[5], uio[4], uio[3]" respectivamente.

External hardware:

Generador de señales digitales: Para alimentar la señal de entrada del circuito. La señal debe ser un pulso (una señal cuadrada) el cual vaya de 0 a 5V, con un ciclo de trabajo del 50%. La frecuencia de la señal no debe ser necesariamente una en particular, puede ser cualquiera, mientras que el osciloscopio que se conente a las salidas sea capaz de leerlas.

Osciloscopio: Este debe ser de al menos 3 canales, para poder leer las tres salidas principales del circuito, las cuales estarán desfasadas 120° entre sí.

Para cualquier duda en cuanto a las pruebas del circuito, pueden mandar un correo electrónico a la siguiente dirección deusjrfm@gmail.com o al numero +522283543917

Pinout

#	Input	Output	Bidirectional
0			P2
1			P1
2			P0
3			Qc
4			Qb
5			Qa
6			no used
7			SE

Random number generator [683]

- Author: VineetaVNair & ShilpaPavithran
- Description: Randomly generates bit stream
- GitHub repository
- HDL project
- Mux address: 683
- Extra docs
- Clock: 10000000 Hz

How it works

Based upon user choice, LFSR data is used to randomly generate bit streams

How to test

Vary input seed to change output bits randomly and observe the same when connected to LEDs.

External hardware

Displays and LED can be added

Pinout

#	Input	Output	Bidirectional
0	seed[0]	output_data[0]	
1	seed[1]	output_data[1]	
2	seed[2]	output_data[2]	
3	seed[3]	output_data[3]	
4	mode[0]	output_data[4]	
5	mode[1]	output_data[5]	
6		output_data[6]	
7		output_data[7]	

Stepper [684]

- Author: Miguel de Jesús Salazar Pedraza
- Description: Stepper Control Bipolar Motor
- GitHub repository
- HDL project
- Mux address: 684
- Extra docs
- Clock: 1000000 Hz

How it works

The project consists of a state machine with 4 outputs, which is used to control a bipolar stepper motor.

How to test

The system has an external clock input, a reset input, a control input for selecting the direction of rotation of the motor, an enable input and 4 outputs for the stepper motor coils.

External hardware

The system requires an external clock input to control the speed of the pulses and thus regulate the rotation speed of the motor.

Pinout

#	Input	Output	Bidirectional
0	Unused	Unused	Output Bit 0
1	Unused	Unused	Output Bit 1
2	Unused	Unused	Output Bit 2
3	Unused	Unused	Output Bit 3
4	Unused	Unused	enable
5	Unused	Unused	direction
6	Unused	Unused	Unused
7	Unused	Unused	Unused

TinyTapeout SPI Master [685]

- Author: Samit & Fahim
- Description: SPI Msster
- GitHub repository
- HDL project
- Mux address: 685
- Extra docs
- Clock: 400000 Hz

How it works

If `sel` is high, then a counter is output on the output pins and the bidirectional pins (`data_o = counter_o = counter`). If `sel` is low, the bidirectional pins are mirrored to the output pins (`data_o = data_i`).

How to test

Set `sel` high and observe that the counter is output on the output pins (`data_o`) and the bidirectional pins (`counter_o`).

Set `sel` low and observe that the bidirectional pins are mirrored to the output pins (`data_o = data_i`).

Pinout

#	Input	Output	Bidirectional
0	i2c_data_in	sck_o	
1	i2c_clk_in	mosi_o	
2	miso_i	i2c_data_out	
3		i2c_clk_out	
4		i2c_data_oe	
5	i2c_wb_err_i	i2c_clk_oe	
6	i2c_wb_rty_i		
7			

serie_serie_register [686]

- Author: C.A. Velázquez-Morales
- Description: Registro Serie-Serie, con 4 registros y corrimiento hacia la izquierda o derecha
- GitHub repository
- HDL project
- Mux address: 686
- Extra docs
- Clock: 10000000 Hz

How it works

Se describe un registro Serie-Serie, con una terminal de entrada (data_in), terminales de control de reloj, reset negado y habilitación (clk, rst, ena). Se asigna una terminal para cambiar el tipo de corrimiento del registro (leri): ALTO para la izquierda y BAJO para la derecha. Se designa una salida del dato en el registro (data_out).

How to test

Se coloca Rst en un valor BAJO, y el Ena en un valor ALTO. El flanco del reloj clk irá actualizando el valor de 4 registros internos que realizarán el corrimiento (data_reg) con el valor del dato de entrada (data_in). Del mismo modo, la entrada de dirección (leri) asignará la dirección del corrimiento y el bit del registro interno que se mostrará en la salida (data_out).

Banco de Prueba para Simulación en Active-HDL:

Imagen Simulación:

El diagrama que se presenta a continuación ilustra el RTL del circuito generado por Quartus II, se observa las entradas rst, clk y ena como las entradas básicas del circuito. Además se observa la entrada data_in como la entrada del dato, así como la entrada leri, que indica si el corrimiento del registro se realizará hacia la derecha (right) o hacia la izquierda (left). Del mismo modo se observa que el bit de entrada leri selecciona el bit que se considerará como el bit de salida data_out.

La conexión se propone como la siguiente imagen

```
1  `timescale 10ns/10ns
2  `define clk1 2
3
4  module tt_ss_reg();
5      reg data_in, clk;
6      reg rst, ena, leri;
7      reg data_out;
8
9      serie_serie_register U1 (data_in, clk, rst, ena, leri, data_out);
10
11     always #`clk1 clk = ~clk;
12
13     initial begin
14         rst = 0;
15         clk = 0;
16         ena = 1;
17         leri = 0;
18         data_in = 1;
19         #2
20         rst = 1;
21         #16
22         leri = 0;
23         data_in = 0;
24         #16
25         leri = 1;
26         data_in = 0;
27         #16
28         leri = 1;
29         data_in = 1;
30     end
31 endmodule
```

Figure 66: Captura de pantalla 2024-04-02 162549

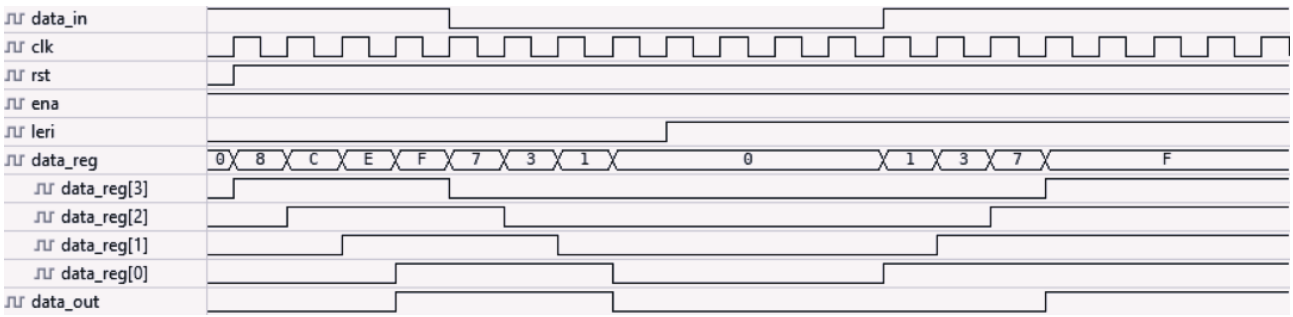


Figure 67: Captura de pantalla 2024-04-02 162240

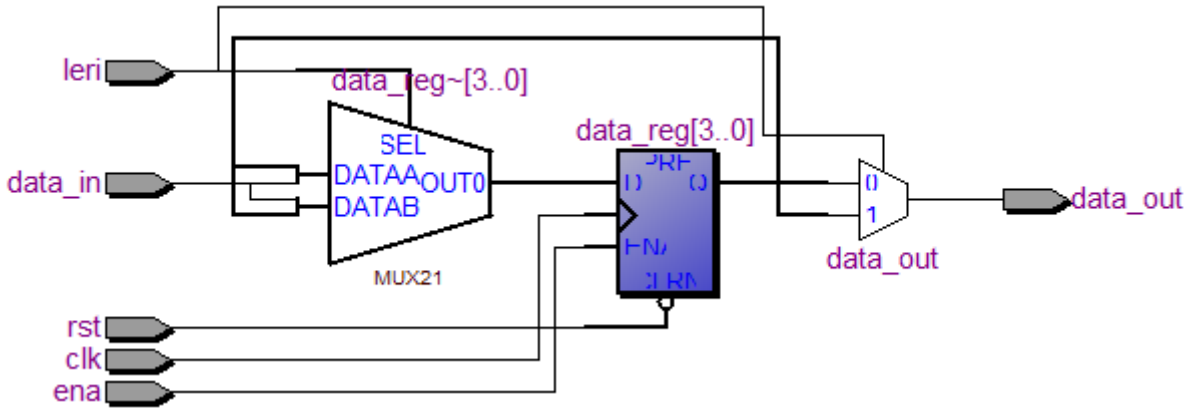


Figure 68: RTL

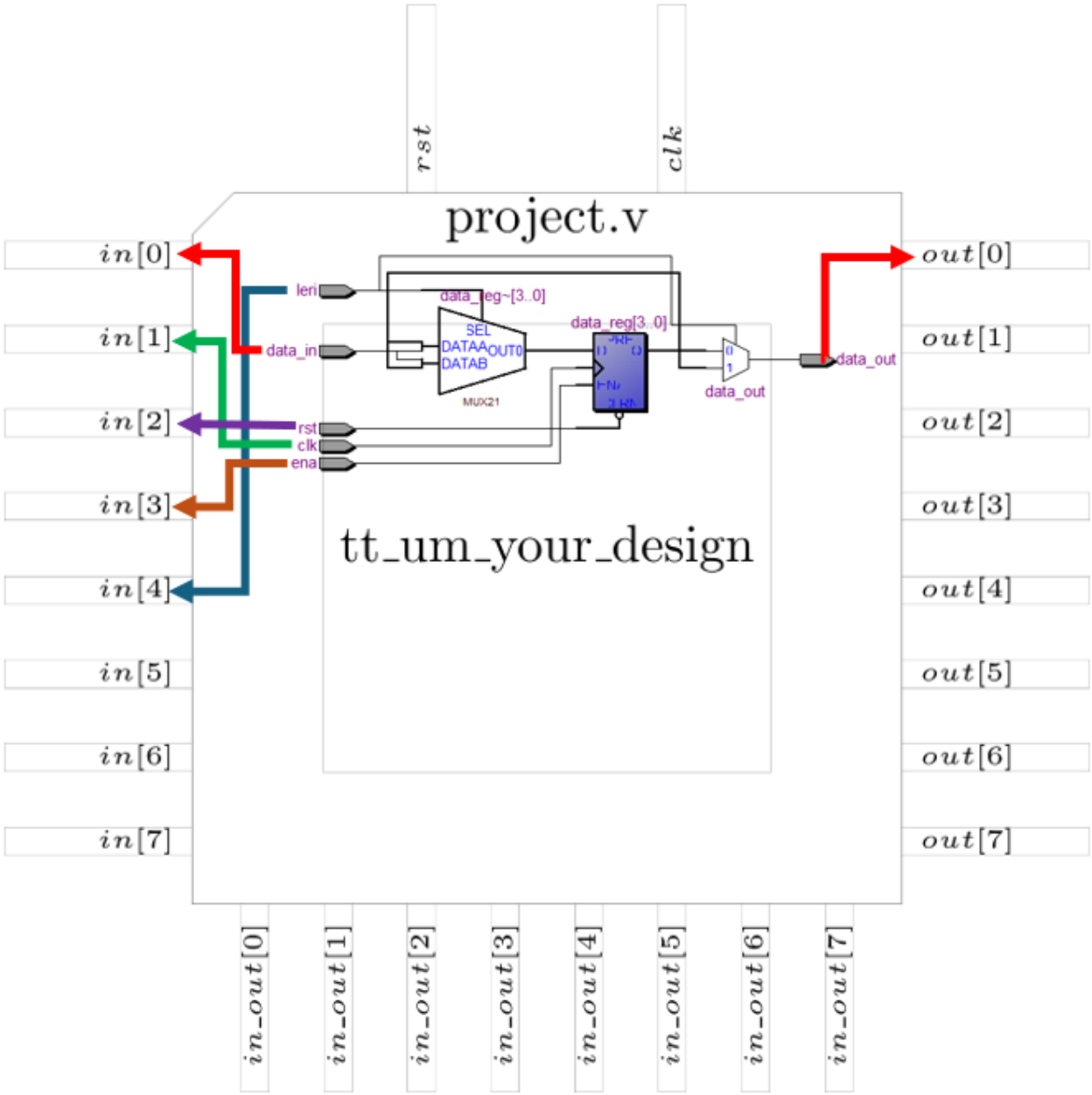


Figure 69: Imagen_Diagrama

External hardware

Se puede utilizar un generador de señales para el reloj (clk) y una base de tiempo para la habilitación (ena), así como botones o interruptores para las entradas y un led para visualizar la salida.

Pinout

#	Input	Output	Bidirectional
0	data_in	Data_Out	No_Used
1	clk	No_Used	No_Used
2	rst	No_Used	No_Used
3	ena	No_Used	No_Used
4	leri	No_Used	No_Used
5	No_Used	No_Used	No_Used
6	No_Used	No_Used	No_Used
7	No_Used	No_Used	No_Used

UACJ-Wallace multiplier [687]

- Author: UACJ Group A
- Description: Wallace multiplier
- GitHub repository
- HDL project
- Mux address: 687
- Extra docs
- Clock: 0 Hz

How it works

This project is center in the implementation of Verilog code for 4 bit Wallace tree multiplier. The design uses half adder and full adder Verilog designs. These modules will be instantiated for the implementation 4 bit Wallace multiplier.

How to test

Under test file, a wallace_tb.v code is located, this code is the testbench

External hardware

You do not need any special external hardware.

Pinout

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

UART-Programmable RISC-V 32I Core [710]

- Author: Ethan Nieman
- Description: RISC-V core implementing reduced set of RV32I ISA; programmable via UART
- GitHub repository
- HDL project
- Mux address: 710
- Extra docs
- Clock: 50000000 Hz

Motivation

This project was developed as a part of the MEST course "ChipCraft: The Art of Chip Design for Non-Experts". As a part of the course, students are walked through the design and implementation of a RISC-V core. At the time that I took this course, students opting to tape out their RISC-V core were limited to a single, hard-wired program in place of a true instruction "memory". This led me to put together a simple UART controller tied to a 64-byte register file that could act as programmable instruction memory. Future students (or anyone experimenting with processor design) can utilize the UART modules in this design to enable programmability of their processor designs.

I decided on UART over other protocols because of its simplicity and my own familiarity with it. USB-serial adapters are easy to find and there are several serial terminals out there. I also decided to implement a very simple auto-baud detection in my design instead of a fixed baud rate. This was done due to my own uncertainty in the clock frequency.

How it works

This project implements a simplified RISC-V core that runs instructions from a 64-byte register file that is programmed by the user via a UART interface.

The RISC-V core adheres to RV32I with the following exceptions:

1. Does not implement FENCE, ECALL, or EBREAK instructions.
2. Only 32-bit loads are implemented. LH, LHU, LB, LBU are all treated as LW.
3. Only 32-bit stores are implemented. SH and SB are treated as SW.
4. Only implements 16 registers (x0 - x15)

Instruction memory and data memory are isolated. Instruction memory and data memory are implemented as 64-byte (16-word) and 16-byte (4-word) register files that are written to and read from via a UART interface.

The UART controller operates in two modes: "PROGRAM" and "DATA READ". Upon reset of the device, the controller enters "PROGRAM MODE". During this time, the user sends a sync packet, followed by the RV32I binary (64-bytes max). Once 64 bytes have been written (unused space can be filled with "ADD x0, x0, x0" instructions), the controller will enter "DATA READ" mode. In this mode, the user can read the contents of data memory by sending a single packet (the contents of this packet do not matter).

For those wanting to implement their own processor design, use the template TL-Verilog file `/src/tlv/uart_template.tlv` and modify line 89 with a URL pointing to your design. See `/src/tlv/cpu_custom.tlv` for an example processor design.

Step-by-step usage:

1. Connect the USBUART Pmod to the demo board via jumpers and a breadboard. The RX pin of the Pmod connects to in2, the TX pin to out2, and PWR/GND should be connected. No other pins of the Pmod are used.
2. (Optional) Connect the BTN Pmod to the demo board. This Pmod should connect only to in4-in7.
3. (Optional) Connect 8LD Pmod to out4-out7 (optional). out7 is high when reset button is pushed, out6 is high when device is in "PROGRAM" mode, out5 is high when zeros are transmitted on RX, out4 is high when zeros are transmitted on TX.
4. Connect the host computer to the USBUART Pmod. A serial terminal will be needed on the laptop. A serial terminal capable of sending binary files is highly recommended.
5. Connect the demo board to power.
6. Press the reset button (or press BTN3 on the BTN Pmod) to reset the device.
7. Configure the serial terminal for 8 data bits, no parity, and one stop bit.
8. Send a program file from `/test/bin` and go to step 12. If the serial terminal does not support this, you can manually program via steps 9-11.
9. Send a single sync packet. A packet of hex value 0x55 is recommended, however the only requirement is that it should be an odd-numbered value (the device measures the width of the start bit in clocks).
10. Send instructions as packets. Start with the least-significant byte of the first instruction, end with the most-significant byte of the last instruction.
11. The device will not switch to "DATA READ" until all instruction memory is written. If less than 16 instructions were written, fill in with no-ops (e.g. "ADD x0, x0, x0").

12. Read data memory by sending a single packet. The content of the packet does not matter.
13. To run a different program, repeat steps 6-12.

How to test (option 1)

To run cocotb tests, run `make` in `/test` directory. The cocotb test will iterate through each program name in `/test/programs.f` and load/execute each program's corresponding binary file in `/test/bin`. Data memory contents are compared to each program's corresponding text file in `/test/solutions` to determine pass/fail.

To add new programs to the test, my current workflow is as follows. I did not get around to figuring out a simple, straightforward way of assembling in python, so the current workflow is admittedly cumbersome.

1. Create assembly program and store in `/test/asm/[program_name].asm`. Keep in mind that there is a max of 16 instructions!
2. Assemble `.asm` file with an online RV32I assembler. Paste output hex into a text file and save it as `/test/hexstr/[program_name].hexstr`.
3. Run `/scripts/hexstr_to_bin.py`. This will convert the hex strings to a binary file and save to `/test/hexstr/[program_name].bin`. It will also write the UART sync word to the beginning of the file (0x55) and backfill unused instructions with ADD x0, x0, x0 instructions.

How to test (option 2)

Use Makerchip IDE (makerchip.com) for testing of RISC-V core only.

1. Go to makerchip.com and load the IDE.
2. Open `/src/tlv/uart_template.tlv`.
3. At line 25, change `set(MAKERCHIP, 0)` to `set(MAKERCHIP, 1)`
4. Lines 59 through 79 list a simple test program. If you want to use a different program, replace these lines with your program. Keep in mind that code under labels needs to be indented with three spaces.
5. Line 122 can be edited to set pass/fail criteria.
6. Use Ctrl + Enter to compile and start simulation. I have found the VIZ window to be most helpful in troubleshooting the CPU. Zoom in on the TinyTapeout chip on the dev board to see a CPU visualization. Advance clock cyles to watch the simulation progress.

External hardware

1. USBUART Pmod (https://digilent.com/reference/pmod/pmodusbuart/start) - Note that jumpers and a breadboard are required to connect as this design does not use the bidirectional I/O yet.
2. BTN Pmod (https://digilent.com/reference/pmod/pmodbtn/start) - This is only necessary if ui_in[7] is needed for resetting the design (the demo board I developed on had a non-functioning reset button). If your reset button works, this isn't needed.
3. 8LD Pmod (https://digilent.com/reference/pmod/pmod8ld/start) - This is optional, use if you want to see the UART TX and RX feedback, as well as what mode the design is in.

Pinout

#	Input	Output	Bidirectional
0			
1			
2	RX	TX	
3			
4		Data on TX	
5		Data on RX	
6		UART Controller in Program Mode	
7	System Reset	System Reset (LED)	

Monobit Test [718]

- Author: Luca Collini
- Description: HLS implementation of NIST 800.22 Monobit Test for RNGs.
- GitHub repository
- HDL project
- Mux address: 718
- Extra docs
- Clock: 50000000 Hz

How it works

This design takes a bit every clock cycles and evaluates if the bit source is random. This particular test is the Monobit test from NIST 800.22. The output is given every 65536 cycles. The is_random signal is to be checked only when the valid signal is high.

How to test

Provide Clock and input bit.

External hardware

Non for now. Planning to add soon

How to use

send one bit per clock cycle to the epsilon port. check is_random when valid is high. The design evaluates every 65536 bits.

Pinout

#	Input	Output	Bidirectional
0	epsilon	is_random	
1		is_random_triosy_lz	
2		valid	
3		valid_triosy_lz	

#	Input	Output	Bidirectional
4		epsilon_triosy_lz	
5			
6			
7			

UACJ-MIE-Booth 4 [736]

- Author: UACJ Group A
- Description: Booth 4 multiplier
- GitHub repository
- HDL project
- Mux address: 736
- Extra docs
- Clock: 0 Hz

How it works

The Booth-4 algorithm is a multiplication algorithm that uses a combination of shifting and addition/subtraction operations to perform signed multiplication of two numbers. It is specifically designed to optimize the multiplication process by reducing the number of required partial products and improving efficiency.

How to test

using test bench, applying phhysicial outputs and see output

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	X[0]	Z[0]	Y[0]
1	X[1]	Z[1]	Y[1]
2	X[2]	Z[2]	Y[2]
3	X[3]	Z[3]	Y[3]
4		Z[4]	
5		Z[5]	
6		Z[6]	
7		Z[7]	

4-Digit Scanning Digital Timer Counter [737]

- Author: Muhammad Shofuwan Anwar
- Description: Automatic timer and manual counter controlled by buttons
- GitHub repository
- HDL project
- Mux address: 737
- Extra docs
- Clock: 1000 Hz

How it works

This Verilog module, named timer_counter, is designed to implement a timer and counter functionality along with a display interface. It utilizes a 1 kHz clock (clk) and generates a 1 Hz signal (clk_1Hz) using a clock divider. The module has inputs (start, stop, mode) for controlling the timer operation and outputs (segment, digit) for displaying the timer's state. It incorporates counters (count0 to count3) which increment from 0 to 9, each with its reset trigger (rst_count0 to rst_count3). The mode of operation can be toggled between manual and automatic modes (mode_reg), with a lock mechanism to prevent bouncing on mode changes (mode_lock). Additionally, it includes debouncing logic for the input buttons (start, stop, mode). The display logic selects the appropriate digit to display (selector) and blinks the display when idle or in manual mode. Finally, it uses a multiplexer (mux) to select the appropriate segment to display based on the current count and a BCD converter (bcd) to convert the count to BCD format for display.

How to test

To test the timer_counter module, you can create a Verilog testbench that instantiates the module and provides stimuli to its inputs (clk, start, stop, mode, rst). You can simulate various scenarios such as starting and stopping the timer manually, toggling between manual and automatic modes, and observing the outputs (segment, digit) to ensure they behave as expected. Additionally, you can simulate edge cases such as reaching the maximum count value, testing debounce logic, and verifying that the display blinks correctly when idle or in manual mode. By analyzing the waveform generated by the simulation, you can verify the functionality of the module and ensure it meets the design requirements.

External hardware

There are additional hardware such as:

- 4 digits seven segment display (https://www.tokopedia.com/cncstorejogja/cnc-7-segment-pin-4-digit-clock-common-anode-red-merah?extParam=src%3Dshop%26whid%
- transistors (https://www.tokopedia.com/cncstorejogja/cnc-bc547-to-92-100ma-npn-amplifier-transistor?extParam=src%3Dshop%26whid%3D325343)
- button (https://www.tokopedia.com/cncstorejogja/cnc-tactile-push-button-6x6x10mm-switch-murah?extParam=src%3Dshop%26whid%3D325343)
- Custom PCBs (I'm still working on it, I'll update later)

Pinout

#	Input	Output	Bidirectional
0	inv	segment[0]	digit[0]
1	mode	segment[1]	digit[1]
2	start	segment[2]	digit[2]
3	stop	segment[3]	digit[3]
4		segment[4]	
5		segment[5]	
6		segment[6]	
7		segment[7]	

FSK Modem +HDLC +UART (PoC) [738]

- Author: Darryl Miles
- Description: FSK Modem w/ HDLC transciever + UART (PoC digital side)
- GitHub repository
- HDL project
- Mux address: 738
- Extra docs
- Clock: 0 Hz

How it works

This is a proof-of-concept design to sketch out the TT_UM digital interface for a later project design that will attempt to incorporate both analogue and digital aspects of the basic skeleton shown in this project.

The design is based on the classic circa 1988 model design used in Amateur Radio Packet systems by G3RUH. The initial specification is looking to achieve data rates of between 4800 and 64000 baud, but the design maybe able to service audio 1200 baud packet radio as well.

The design is 1-data-bit per symbol.

The original TNC (Terminal Node Controller) was a Z80 CPU and 8530 Serial Communications Controller. So inline with this I expect to provide an 8-bit CPU (as a future TT project) as a companion to this so the two items taken together should be able to form a complete communications solution of a capable TNC. This is an area I spent a significant amount of my teenage youth understanding and experimenting with that gave me a good grounding in all the digital electonic, radio and computer/CPU theory/practice that is still in use today.

The original PCB board design used:

- a x16 master TX CLOCK line of the data rate.
- was based on 12v audio interface/opamps, and 74HC TTL logic
- was capable of the range of baud rates with minor modifications, the most used speed in my experience is 9600 baud
- the TX DAC was 4 x 8-bit samples per bit, with the waveform lookup using a 12bit address that can see previous bit information sent
- EPROM were used directly to provide waveforms, these have a number of jumper set modes to allow compensation for non-linear responses at the TX-AUDIO and RX-AUDIO

Due to the need to perform ROM lookups, this is operating in 4 phases sharing 6-bit output from module, and 4-bit input to module. The 4 phases cover a sequence of:

- TX nibble low (6bit address)
- TX nibble high (6bit address)
- RX nibble low (6bit address)
- RX nibble high (6bit address) It is not clear if this arrangement a good choice. There is also a programmable latency on the reply, of zero-cycles or one-cycle, the shifts the expectation of the result.

I also need to validate the DAC 8bit loading scheme prevents any chirping (visibily to DAC of partially loaded data, due to multiplex timing differences) of the data because it is loaded in 2 halves.

The master clock (CLK pin) due all the above, it is ncessary to run the clock pin at x4 the x16 of the original design.

data rate baud	master clock (CLK)	tx clock	tx sample clock
4,800	307,200	76,800	19,200
9,600	614,400	153,600	38,400
19,200	1,228,800	307,200	76,800
38,400	2,457,600	614,400	153,600
64,000	4,096,000	1,024,000	256,000
76,800	4,915,200	1,228,800	307,200

Table is in Hz or Baud

The master clock (pin CLK) is driven at x64 the synchrnous data rate. The tx clock rate is derrived from this 'CLK divide-by-4'

The UART clocking is also derived from CLK, and each side (uart RX and uart tx) can be individually configured to be 1:1 or 2:1 the synchronous data rate:

- Uart TX x1 = data rate x1
- Uart TX x2 = data rate x2
- Uart RX x1 = data rate x8 (due to majority voter, 8 sample buffer)
- Uart RX x2 = data rate x16 (due to majority voter, 8 sample buffer)

As you can see maybe there is some headroom for faster transmission speeds within a TT project, before needing to increase DAC resolution and explore 4FSK/6FSK/QAM etc...

There are 3 main functional areas with the design:

The type of FSK modem is 2FSK (dual tone) outputting continious wave.

Upper Digital (included here) This incorpotates a full-duplex HDLC frame processor attached to a UART (ttl interface), the UART process encodes the frame in format similar to KISS format used by TNCs, with a few modifications.

Lower Digital (included here) This manages the receiver clock recovery PLL circuit and interface, the original designs used EPROM lookup tables with 12bit address (which has visibility on at least the previous encoded bit) and provides an 8bit data output.

The data outputs are then fed into a respective 8bit DAC

The receiver has a PLL lock detector which is used to provide DCD (Data Carrier Detect) signal. While the hardware design is capable of full-duplex operation it is often used in Amateur Radio situations in a half-duplex situation with a carrier sense channel sharing algorithym.

Lower Analogue (not includes in this PoC design, see next iteration) The parts that are missing from the design:

- 8bit DAC for transmit waveform shaping, using 4 samples per bit
- opamp for transmit audio anti-aliasing (low-pass filter?) circuit to remove harmonic noise from the output audio
- 8bit DAC for receiver clock recovery feedback, using 16 samples per bit.
- opamp for receive audio signal interface, this maybe moved to an external board due to needing to protect the TT IC from over voltage from being attached to usuall 12v equipment or maybe 36v when using some ex-commerial radio trancievers. This may have been a comparator circuit (unsure at this time), fed into a DFF to synchronise the incoming data to the x16 (of datarate) clock recovery timing
- 2 x opamp to provide PLL lock detection (unsure how this works atm), I would guess it can detect when the signal is being centered and has been centered for some number of samples, maybe via slow capacitance charge up when the UP/DOWN line is managing to meet an approximate 50%/50% duty cycle per x16 clock recovery tick.
- 2 x opamp to provide zero-crossing detection, this is used to provide the PLL its feedback mechanism (the UP/DOWN line) to advance or retard the edge alightment.

It is hoped all items can be incorporated into the same design using the analogue GDS facility with TT and connected to the respective lower digital signal.

At this time we bring out the interconnection points (between analogue and lower digital) to the external interface of TT and we provide a configuration mechnism to

be externally or internally driven/internally sourced. This should allow for a significant level of simulation and experimentation by users of the project to understand and explore FSK/PLL theory by picking a testing configuration combintation, being full-duplex it should be able to loop-back at various levels to understand each part better. While also providing those with a Ham Radio license to try out on air communicating with their local users or AMSAT.

Have fun... 73s de G7LED

How to test

When the final design is completed, there should be a number of visible and testable aspects available to observe the working of various functions.

I am not expecting this PoV project to yeild good result due to the limited time spent on it just before submission deadlines for TT06.

Check back with the repo for a testing regime.

External hardware

At this PoC stage, testing with RP2040 and FPGA external boards to validate the electrical interface acrhetecture makes sense and provided the most options.

Pinout

#	Input	Output	Bidirectional
0	Rx Data	UART TX	Rx Clock (bidi)
1	Rx Clock	UART CTS	Up/Down (bidi)
2	UART RTS	UART DCD	TableAddr[0]
3	TableData[0]	Rx Error	TableAddr[1]
4	TableData[1]	Tx Error	TableAddr[2]
5	TableData[2]	Sending	TableAddr[3]
6	TableData[3]		TableAddr[4]
7	UART RX	Tx Clock Stobe	TableAddr[5]

DIP Switch to HEX 7-segment Display [739]

- Author: Lance Mitrex
- Description: Input Binary digit, 1 to 16 with the DIP Switch. The HEX is displayed on the 7-segment display.
- GitHub repository
- HDL project
- Mux address: 739
- Extra docs
- Clock: 0 Hz

How it works

The project was originated during the training class, "ChipCraft: The Art of Chip Design for Non-Experts" presented by eFabless.

The project utilizes techniques associated with Tiny Tapeout and Redwood EDA.

Project Title: Input DIP Switch to HEX 7-Segment Display.

This Project is a 'simple' project for inexperienced FPGA and/or inexperienced Verilog programmers. The project reads the INPUT DIP Switch on the Tiny Tapeout Demo Board and outputs the properly formatted Hex character, to the 7-Segment display which is also located on the Tiny Tapeout Demo Board.

The complete Transaction-Level Verilog code is located in the file, "DIPSwitch_7-segment.v"

The remainder of this README.md file outlines the process utilized for this project.

How to test

Select a HEX character from 1 to 16 (1 to F). Input the HEX character by using the appropriate binary value, using the DIP Switch. The HEX character you selected and input will be displayed on the 7-segment display.

External hardware

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

Pinout

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

tt6-simplez [740]

- Author: Roland Coeurjoly
- Description: Simplez
- GitHub repository
- HDL project
- Mux address: 740
- Extra docs
- Clock: 12000000 Hz

How it works

https://github.com/Obijuan/simplez-fpga

How to test

https://github.com/Obijuan/simplez-fpga

External hardware

https://github.com/Obijuan/simplez-fpga

Pinout

#	Input	Output	Bidirectional
0	RX Serial Input from Keyboard	LED 0	Stop Signal Indicator
1		LED 1	TX Serial Output to Display
2		LED 2	
3		LED 3	
4		LED 4	
5		LED 5	
6		LED 6	
7		LED 7	

PWM_Sinewave_UART [741]

- Author: Luis Gerardo Avila
- Description: This project is a PWM signal generator that creates a sine wave, with frequency variation between 100 Hz and 700 Hz with steps of 100 in 100 Hz, which is manipulated through a UART interface
- GitHub repository
- HDL project
- Mux address: 741
- Extra docs
- Clock: 25000000 Hz

How it works

Through UART communication, a number from 1 to 7 is sent, indicating the frequencies set in the code, which are from 100 Hz to 700 Hz, then through a pin I generate a PWM signal which varies in time to generate a sine wave of the frequency that was requested.

How to test

Only physical tests were carried out with the circuit, a Bluetooth antenna was added to the UART communication port, and a low pass filter was added to the output of the system to improve the signal a little and then it was measured with the oscilloscope and I verify that it delivers the requested frequencies.

External hardware

-Bluetooth HC05. -oscilloscope. -A low pass filter.

Pinout

#	Input	Output	Bidirectional
0	uart_rx	uart_tx	no use
1	no use	no use	no use
2	no use	pwm_outx	no use
3	no use	no use	no use
4	sw_11	no use	no use

#	Input	Output	Bidirectional
5	sw_01	no use	no use
6	rst1	no use	no use
7	no use	no use	no use

drEEm tEEm PPCA [742]

- Author: Calvin Sokk, William Mingham Zhu, Calvin Yeh, Eric Liu
- Description: Projectile Positioning Calculation Accelerator
- GitHub repository
- HDL project
- Mux address: 742
- Extra docs
- Clock: 0 Hz

How it works

This is a simple shoot the target game in a 32 x 32 square which calculates the trajectory of a ball.

A target will randomly be generated in the top two rows of the playing field and your goal is to position your cannon, aim it, and hit the target. The ball that is shot can be bounced off the left and right walls, but once it hits the ceiling, the game is over.

Your cannon will be at the bottom row of the 32 x 32 square. You will be able to move it left or right to any square on this bottom row. You will also have 7 different aiming positions.

Controls:

- ui[0]: "Move Left"
- ui[1]: "Move Right"
- ui[2]: "Aim Left"
- ui[3]: "Aim Right"
- ui[4]: "Shoot"

All shots will be in a linear fashion. For example, if you select the default aim position (which is Left 2 and Up 1), the ball will be shot linearly with a slope that goes left 2 and up 1.

Here are all the available aiming positions:

1. Left 2 Up 1
2. Left 1 Up 1
3. Left 1 Up 2
4. Up 1
5. Right 1 Up 2
6. Right 1 Up 1
7. Right 2 Up 1

The generated target's x and y position will be outputted in the uo[2:6] wires after the game is initialized. In the first cycle after initialization, these five bits will be driven to the x value and for the next cycle, they will be driven to the y value.

After shooting the ball, wait until uo[0] ("Result Valid") is set to high. This indicates that the simulation is over and the ball has hit the top of the screen. Then, uo[1] ("Hit") will tell you whether or not the ball has hit the target.

How to test

To make sure that you hit the target, you will have to draw out the trajectory of the ball in the 32 x 32 grid.

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	Move Left	Result Valid	
1	Move Right	Hit	
2	Aim Left	Select Data	target_x
3	Aim Right	Select Data	target_x
4	Shoot	Select Data	target_x
5		Select Data	target_x
6		Select Data	target_x
7			0 ? target_y = 30 : target_y = 31

TWI Monitor [743]

- Author: Nicklaus Thompson
- Description: A Two Wire Interface (I2C) bus monitor
- GitHub repository
- HDL project
- Mux address: 743
- Extra docs
- Clock: 50000000 Hz

How it works

This project is a Two-Wire Interface (I2C) monitor. The TWI side is essentially a shift register and does not respond like a slave or have an address. The system runs at 50 MHz and uses a UART baud rate of 115200. The system cannot currently capture repeated TWI frames, but captured single frames during testing on an FPGA.

How to test

You can use an Arduino and any TWI-compatible module to generate TWI frames to view. The frames will be converted to three bytes, those being {addr, R/W}, {data}, and {{4{Addr Ack}}, {4{Data Ack}}}. I use Coolterm to view the hex output, you can download it at https://freeware.the-meiers.org/.

External hardware

This project needs an external UART to USB adapter if you want to connect it to your PC.

Pinout

#	Input	Output	Bidirectional
0	SDA_in	TX_out	
1	SCL_in		
2			
3			
4			
5			

#	Input	Output	Bidirectional
6			
7			

Displays CIt [744]

- Author: Cambridge
- Description: To display 'CIt' on 7 segment display
- GitHub repository
- Wokwi project
- Mux address: 744
- Extra docs
- Clock: 1 Hz

How it works

Worwi project using a D flipflop ring to display "CIt" on the 7 segment display.

How to test

Turn on the required inputs ,turn off the unused inputs.

External hardware

None,beside the TT demo board.

Pinout

#	Input	Output	Bidirectional
0	run	led_a	
1	state_init[0]	led_b	
2	state_init[1]	led_c	
3	state_init[2]	led_d	
4	state_init[3]	led_e	
5	unused	led_f	
6	unused	led_g	
7	unused	led_h	

Cambio de giro de motor CD [745]

- Author: Equipo6TESVG
- Description: Cambio de giro de motor CD
- GitHub repository
- Wokwi project
- Mux address: 745
- Extra docs
- Clock: 0 Hz

How it works

Cambio de giro de motor CD.

A schematic of the circuit may be found at:

https://wokwi.com/projects/395344363336889345

The circuit has 10 inputs:

Input	Setting
CLK	Not Used
RST_N	Not Used
01	Input A
02	Not Used
03	Not Used
04	Not Used
05	Input B
06	Not Used
07	Not Used
08	Input C

TENEMOS 3 ENTRADAS CON COMPUERTAS NOT LAS CUALES ESTAN RELACIONADAS CON DOS COMPUERTAS OR,LAS COMPUERTAS NOT LA RELACION QUE ENTRA ES LO CONTRARIO A LO QUE SALE POR EJEMPLO,SI ENTRA 1 SALE 0 Y SE ENTRA 0 NSALE 1 ESTAS ENTRADAS VAN RELACIONADAS CON LAS COMPUERTAS AND PERMITINDO QUE UN VALOR DE ENTRADA NO SEA MODIFICADO POR LA COMPUERTA NOT.LA OTRA ENTRADA DE LA COMPUERTA OR DEPENDE DE LO QUE SALE DE LAS COMPUERTAS NOT PERMITIENDO ASI EL CAMBIO, UNA SALIDA DE LA OR ENTRA AND QUE DEPENDE DE LO ANTERIOR Y OTRAS DE SUS ENTRADAS DEPENDE LE LA PRIMER ENTRADA SI EN LA AND ENTRA 1 Y 1 LA SALIDA ES 1 PERMITIENDO ASI EL

GIRO SI LA COMNIACION DE ENTRADAS CAMBIAN A LA SEGUNDA AND EL CAMBIO SE DA DE RECHA A IZQUIERDA.

Output	Value in
01	A
02	Not Used
03	Not Used
04	Not Used
05	Not Used
06	Not Used
07	Not Used
08	B

EN LA SALIDA A EL VALOR DE LA AND EN 1 EL CUAL HACER QUE EL MOTOR GIRE A LA DERECHA EN LA SALIDA B EL VALOR DE LA SEGUNDA AND EN 1 Y EL DE LA PRIMERA AND REGRESA A 0 ESO PROVOCA EL CAMBIO DE GIRO DE DERECHA A IZQUIERDA.

A python implementation of the 32-bit Fibonacci LFSR can be found at the link below. It may be used for testing the hardware for sequences longer than the initial 100 values.

https://github.com/icarislab/tt06_32bit-fibonacci-prng_cu/main/docs/32-bit-fibonacci-prng_pythong_simulation.py

External hardware

No external hardware is required.

Pinout

#	Input	Output	Bidirectional
0	E0	S0	
1			
2			
3			
4	E4		
5			
6			

#	Input	Output	Bidirectional
7	E7	S7	

Latin_bomba [746]

- Author: Arizaga Silva
- Description: Circuito de control basado en maquina de estados para controlar el llenado de un deposito de agua a traves de una bomba
- GitHub repository
- HDL project
- Mux address: 746
- Extra docs
- Clock: 0 Hz

LATINPRACTICE_2024

Figure 70: Logo

Este proyecto forma parte de la iniciativa LATINPRACTICE_2024 con el cual se pretende que profesores universitarios y alumnos de nivel medio superior y superior, tengan acceso a herramientas de software libre para el diseño de circuitos integrados .

Este proyecto es una máquina de estados sencilla que permite controlar el llenado y vaciado automático de un depósito superior de agua alimentado por una bomba conectada a un depósito inferior de agua.

How it works El circuito consta de una máquina de estados tipo Mealy con tres estados (Espera, llenado y Alarma).

Las entradas del circuito corresponden a sensores que detectan la presencia o ausencia de agua (1 o 0 lógico), es decir son señales digitales. Un sensor para la cisterna (depósito inferior) y dos sensores para el depósito superior.

EL circuito cuenta con dos salidas digitales, la primera para encender y apagar la bomba y la segunda para encender una luz o una alarma que indique que no hay agua en el depósito inferior.

El proyecto utiliza un modelo de máquina de estados finitos con tres estados para controlar el llenado del depósito superior mediante una bomba. Los tres estados son:

1. **Espera (`espera`)**: Este estado indica que el sistema está en espera de alguna acción. En este estado, la bomba está apagada (`bomba_o = 0`) y la alarma está desactivada (`alarma_o = 0`). La transición desde este estado ocurre cuando

se detecta que el depósito superior está vacío y que hay agua en el depósito inferior (`sensores_i = 001`) o cuando se detecta que la cisterna está llena (`sensores_i = 111`) o que no hay agua en el depósito inferior (`sensores_i = xx0`) que lo lleva al estado de alarma.

2. **Llenado (`llenado`)**: En este estado, la bomba está encendida (`bomba_o = 1`) para llenar el depósito. La alarma permanece desactivada (`alarma_o = 0`). La transición desde este estado ocurre cuando se detecta que la cisterna está llena (`sensores_i = 111`), lo que indica que el depósito ha alcanzado su capacidad máxima y regresa al estado de Espera o que no hay agua en el depósito inferior (`sensores_i = xx0`) que lo lleva al estado de alarma.
3. **Alarma (`alarma`)**: Este estado se activa cuando se detecta una condición de alarma, como la falta de agua en el depósito inferior . En este estado, la bomba se apaga (`bomba_o = 0`) y se activa la alarma (`alarma_o = 1`). La transición desde este estado ocurre cuando se detecta que el depósito inferior está lleno (`sensores_i = XX1`), lo que indica que se ha resuelto la condición de alarma.

Cada estado y transición está definido en el código Verilog proporcionado, lo que permite controlar el llenado del depósito mediante la activación y desactivación de la bomba en respuesta a las lecturas de los sensores.

How to test TODO

External hardware La asignación de entradas y salidas del diseño del control de la bomba a las entradas y salidas del proyecto Latinpractice son como se indica a continuación.

ck: Conectado a uio_in[7]. rst_i: Conectado a uio_in[6]. sensores_i: Conectado a uio_in[5:3]. alarma_o: Conectado a uio_out[1]. bomba_o: Conectado a uio_out[0].

Como puede notarse, el proyecto de la bomba, para hacer más legible el código, indica cuando un puerto es de entrada colocando un _i al final del nombre del puerto (rst_i) y cuando un puerto es de salida un _o (bomba_o), excepto en el puerto de reloj.

Las entradas de los sensores pueden ser emuladas con botones o con switches conectados a los puertos bidireccionales uio_in[5:3]. Las salidas pueden emularse utilizando LED's conectados a uio_out[0] y uio_out[1] a traveés de una resistencia limitadora de corriente.

Authors

- [@Arízaga-Silva](https://www.researchgate.net/profile/Juan-Antonio-Arizaga-Silva)
- [@Sanchez-Rincón](https://www.researchgate.net/profile/Ismael_Rincon)
- [@Muñiz-Montero](https://www.researchgate.net/profile/Carlos-Muniz-Montero)

Pinout

#	Input	Output	Bidirectional
0	no used	no used	bomba_o
1	no used	no used	alarma_o
2	no used	no used	no used
3	no used	no used	sensores_i[0]
4	no used	no used	sensores_i[1]
5	no used	no used	sensores_i[2]
6	no used	no used	rst_i
7	no used	no used	ck

Circuito PWM con ciclo de trabajo configurable [747]

- Author: Maria Fernanda Tovany Salvador, Javier Trucios Alonso & Luis David Vazquez Perez
- Description: A partir de tres senales de entrada digitales selecciona el ciclo de trabajo de salida (PWM).
- GitHub repository
- HDL project
- Mux address: 747
- Extra docs
- Clock: 0 Hz

How it works

El PWM configurable de ciclo de trabajo incrementales un dispositivo diseñado para generar señales PWM (Modulación por Ancho de Pulso) con ciclos de trabajo incrementales basados en tres señales de entrada digitales. Este dispositivo permite una fácil configuración del ciclo de trabajo para adaptarse a una variedad de aplicaciones. El ciclo de trabajo será incremental con valor de 12.5% para cada combinación de entrada.

How to test

Usando los puertos de entrada en combinacion de estados logicos y con un uno logico en enable, puede visualizarse en la salida como la señal de ancho de pulso varia de acuerdo a las siguientes combinaciones: 000=12.5% 001=25% 010=37.5% 011=50% 100=62.5% 101=75% 110=87.5% 111=100%

External hardware

Generador de señales para generar la señal de reloj de 10MHz, osciloscopio para poder visualizar el ancho del pulso.

Pinout

#	Input	Output	Bidirectional
0	no use	PWM	no use
1	no use	no use	no use

#	Input	Output	Bidirectional
2	no use	no use	no use
3	speed[0]	no use	no use
4	speed[1]	no use	no use
5	speed[3]	no use	no use
6	enable	no use	no use
7	clock	no use	no use

Bit Control [748]

- Author: K Opong-Mensah
- Description: Bit pattern cycle on every clock cycle.
- GitHub repository
- HDL project
- Mux address: 748
- Extra docs
- Clock: 0 Hz

How it works

This project outputs a different bit pattern on each clock cycle and then resets.

How to test

To test this project, apply a reset signal, cycle the clock several times and compare to the expected outputs.

External hardware

GPIOs can be measured directly to test performance.

Pinout

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

32b Fibonacci Original [749]

- Author: Felipe Serrano
- Description: Fibonacci Serie
- GitHub repository
- Wokwi project
- Mux address: 749
- Extra docs
- Clock: 0 Hz

A schematic of the circuit may be found at:

https://wokwi.com/projects/395618714068432897

Pinout

#	Input	Output	Bidirectional
0		Out0	
1	In1	Out1	
2	In2	Out2	
3		Out3	
4		Out4	
5		Out5	
6		Out6	
7		Out7	

Array Multiplier [750]

- Author: UACJ Group A
- Description: Array Multiplier
- GitHub repository
- HDL project
- Mux address: 750
- Extra docs
- Clock: 0 Hz

How it works

This project is centerd in the implementation of a verilog code for 4 bit Wallace tree multiplier. The design uses half adder and full adder Verilog designs.

How to test

On file Test, there is a testbench call wallace_tb.v Use the code to test the code.

External hardware

You do not need any special external hardware

Pinout

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

Voting thingey [751]

- Author: Zoda + Jade
- Description: Does modified-consensus voting for redundant microcontrollers, as a vaguely functional-safety thing
- GitHub repository
- HDL project
- Mux address: 751
- Extra docs
- Clock: 0 Hz

How it works

Our project used the 8 inputs as voters, expecting 1 as "FAIL" and 0 as "PASS" for votes in the system. As a default, if all systems vote for a pass, the consensus is pass. However, we also give the user the ability to customize the number of voters and the threshold of passing.

For instnace, a user may have 8 voters and fail if 3 votes fail.

This is for a safety system where redundant microcontrollers are used to control some safety critical hardware and we want to avoid single point faults.

How to test

Tested with the unit tests running `make -B` in test subfolder.

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	vote0		
1	vote1		
2	vote2		
3	vote3		
4	vote4		

#	Input	Output	Bidirectional
5	vote5		
6	vote6		
7	vote7		

14 Hour Simple Computer [782]

- Author: Peter Schmidt-Nielsen
- Description: A very simple computer that renders to VGA, designed in the last 14 hours before the submission deadline
- GitHub repository
- HDL project
- Mux address: 782
- Extra docs
- Clock: 50000000 Hz

How it works

Does it even work?

How to test

TODO: Document this. I know, I'm awful, but I'm literally doing this in the last 14 hours of the submission deadline.

External hardware

This project requires an external SPI SRAM, and also an external VGA breakout board! I'm going to make them, and give them out for free.

Pinout

#	Input	Output	Bidirectional
0		vga_r	sram_cs_n
1		vga_g	sram_so
2		vga_b	sram_sio2
3		vga_hs	sram_si
4		vga_vs	sram_sck
5			sram_sio3
6			
7			

Universal gates [812]

- Author: htfab
- Description: A modern take on 74-series chips
- GitHub repository
- HDL project
- Mux address: 812
- Extra docs
- Clock: 0 Hz

How it works

This project is supposed to eventually become a Wokwi port of unigate-gf.

How to test

Run the testbench in the test directory.

External hardware

None

Pinout

#	Input	Output	Bidirectional
0	a	i	u21(a..d)
1	b	j	u31(a..f)
2	c	k	u22(a..f).1
3	d	l	u22(a..f).2
4	e	m	u41(a..j)
5	f	n	nand(e, f)
6	g	o	u21(g..j)
7	h	p	u31(k..p)

UART interface to ADC TLV2556 (VHDL Test) [814]

- Author: Jonas Nilsson
- Description: Simple interface that allows values from a TLV2556 ADC to be read out over a UART
- GitHub repository
- HDL project
- Mux address: 814
- Extra docs
- Clock: 48000000 Hz

How it works

This design is an interface that allows a TI TLV2556 ADC to be controlled via a UART. The ADC settings are static. The UART interface allows prompting a conversion and will receive the measured value as human readable values.

How to test

Requires a 48 MHz clock.

Connect a TLV2556 and a UART to the appropriate pins. Note that the CTS and RTS pins have opposite polarity as they're intended to be connected directly to a CH340 UART-over-USB chip.

Connect with a terminal emulator set to 230400 Baud, 8N1.

Send a single hexadecimal character from 0 to B over the serial port. The digit represents the channel you wish to convert. The design will cause the tlv2556 to do a conversion, read out the measured value and send it back in human readable form over the serial port.

Type CR or LF to enter "continuous mode". The design will loop through all channels, converting each in turn, and print a page worth of measured values to the serial port.

Pinout

#	Input	Output	Bidirectional
0	adc_dout	adc_clk	
1	adc_eoc	adc_cs	
2	UART RxD	adc_din	

#	Input	Output	Bidirectional
3	UART CTS_n (opposite polarity)	UART TxD	
4		UART RTSn (opposite polarity)	
5			
6			
7			

rng Test [842]

- Author: Luca Collini
- Description: HLS implementation of NIST 800.22 Monobit Test for RNGs.
- GitHub repository
- HDL project
- Mux address: 842
- Extra docs
- Clock: 50000000 Hz

How it works

This design takes a bit every clock cycles and evaluates if the bit source is random. This particular test is the Monobit test from NIST 800.22. The output is given every 65536 cycles. The is_random signal is to be checked only when the valid signal is high.

How to test

Provide Clock and input bit.

External hardware

Non for now. Planning to add soon

How to use

send one bit per clock cycle to the epsilon port. check is_random when valid is high. The design evaluates every 65536 bits.

Pinout

#	Input	Output	Bidirectional
0	epsilon	is_random	
1		is_random_triosy_lz	
2		valid	
3		valid_triosy_lz	

#	Input	Output	Bidirectional
4		epsilon_triosy_lz	
5			
6			
7			

Fast Readout Image Sensor Prototype [846]

- Author: Devin Atkin
- Description: This Project Pretends to be an Image Sensor, It's not an Image Sensor
- GitHub repository
- HDL project
- Mux address: 846
- Extra docs
- Clock: 50000 Hz

Figure 71: alt text

How it works

This project simulates an image sensor with the intention of validating a readout method. Light levels are fed in, and then the ouput is used to re-recover those light levels. This is done by using a set of frequency modules to convert the light levels into frequencies, and then a set of frequency counters to measure the frequency of the output. The design is validated through the testbenches both simulating a full 1MP sensor (These are run in the long testbenches action) and a smaller number of pixels in the actual manufactured hardware and shorter testbenches. I cannot unfortunately think of any way to make this design more useful to anyone else, but I wish anyone who tries the best of luck.

How to test

There are testbenches provided in this repository which should verify the functionaltiy of the design. The test directory also includes a top level testbench which can be used to verify the design once it is fabricated.

External hardware

The top level test under the test directory will have a circuit python equivalent written which will allow the design to be tested from the external RP2040 microcontroller. This will allow the design to be tested in the actual hardware as well as in simulation.

Pinout

#	Input	Output	Bidirectional
0	DATA_IN1	DATA_BUS_COL_OUT[0]	DATA_BUS_ROW_IN[0]
1	RCLK_1	DATA_BUS_COL_OUT[1]	DATA_BUS_ROW_IN[1]
2	LOAD_1	DATA_BUS_COL_OUT[2]	DATA_BUS_ROW_IN[2]
3	DATA_IN2	DATA_BUS_COL_OUT[3]	DATA_BUS_ROW_IN[3]
4	RCLK_2	DATA_BUS_COL_OUT[4]	DATA_BUS_ROW_IN[4]
5	LOAD_2	DATA_BUS_COL_OUT[5]	DATA_BUS_ROW_IN[5]
6		DATA_BUS_COL_OUT[6]	DATA_BUS_ROW_IN[6]
7		DATA_BUS_COL_OUT[7]	DATA_BUS_ROW_IN[7]

soundgen [897]

- Author: Yanik Drmla
- Description: plays prelude.wav in endless loop
- GitHub repository
- HDL project
- Mux address: 897
- Extra docs
- Clock: 10000000 Hz

How it works

Plays prelude.wav in real Hardware. This works with a sine stored as lookup-table which is being played with varying frequency.

How to test

To test connect a Piezo to given Pins and listen!

External hardware

piezo

Pinout

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

Simon Says game [899]

- Author: Uri Shaked
- Description: A simple memory game
- GitHub repository
- HDL project
- Mux address: 899
- Extra docs
- Clock: 50000 Hz

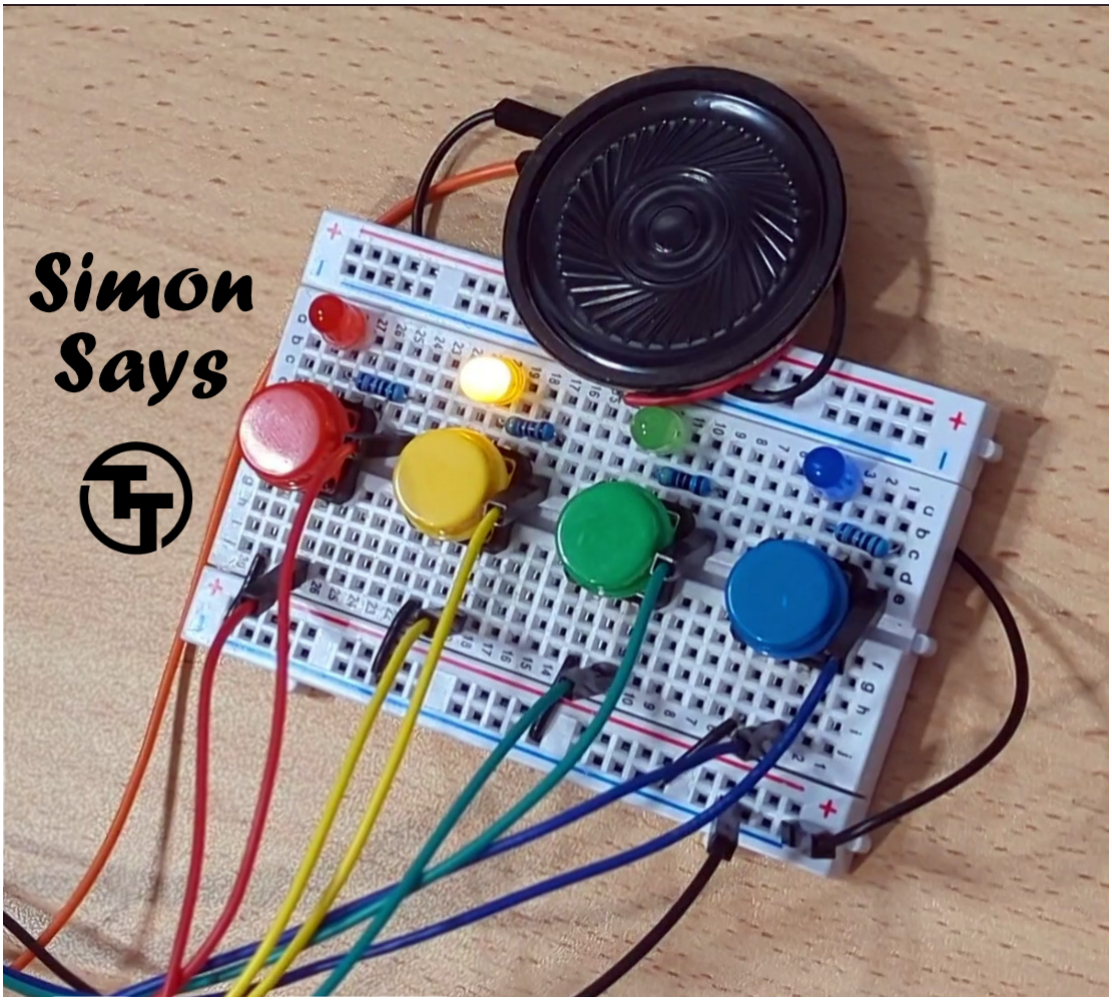


Figure 72: Simon Says Game

How it works

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

In each turn, the game will play the sequence, and then wait for the user to repeat the sequence by pressing the buttons according to the color sequence. If the user repeated the sequence correctly, the game will play a "leveling-up" sound, add a new color at the end of the sequence, and move to the next turn.

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

Check out the online simulation at https://wokwi.com/projects/395431892849488897 (including wiring diagram).

How to test

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

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

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

Note: the game requires 50KHz clock input.

External Hardware

Four push buttons (with pull-down resistors), four LEDs, and optionally a speaker/buzzer and two digit 7-segment display

Pinout

#	Input	Output	Bidirectional
0	btn1	led1	seg_a
1	btn2	led2	seg_b
2	btn3	led3	seg_c
3	btn4	led4	seg_d
4	seginv	speaker	seg_e
5		dig1	seg_f
6		dig2	seg_g
7			

Gate Guesser [901]

- Author: Fabio Ramirez Stern
- Description: A very simple gate guessing game - which I/O is connected to what?
- GitHub repository
- HDL project
- Mux address: 901
- Extra docs
- Clock: 0 Hz

How it works

The input and inout (used as inpputs) pins are connected to 8 different logic gates, which lead to the outputs. Only one logic layer of combinatoric logic. Each input is hooked up to only one gate.

How to test

No clock, enable or reset is used. As this is just one layer of combinatoric logic, you can simply check against a precalculated truth table. To play, flip the inputs and observe the output until you recognise what it must be.

External hardware

Connect 16 switches to the input and inout pins, the 8 outputs are hooked up to one LED each (or other display hardware of your choice).

The solution is:

SPOILER

out0 = in0 and in2

out1 = not in1

out2 = in5 and in7 and inA

out3 = in6 xor inC

out4 = in4 nand in9

out5 = in8 xnor B

out6 = inE nor inF

out7 = in3 or inD

Pinout

#	Input	Output	Bidirectional
0	switch0	gatey0	switch8
1	switch1	gatey1	switch9
2	switch2	gatey2	switchA
3	switch3	gatey3	switchB
4	switch4	gatey4	switchC
5	switch5	gatey5	switchD
6	switch6	gatey6	switchE
7	switch7	gatey7	switchF

sn74169 [903]

- Author: andychip1
- Description: up down counter
- GitHub repository
- HDL project
- Mux address: 903
- Extra docs
- Clock: 1000000 Hz

How it works

Verilog model of the SN74169.

How to test

ui_in[3:0] = A[3:0] 4b parallel load

ui_in[4] = ENPB

ui_in[5] = ENTB

ui_in[6] = LOADB

ui_in[7] = UP/DOWNB

uo_out[3:0] = Q[3:0] 4b output

uo_out[4] = RCOB

uo_out[5] = !ui_in[0] - for debugging

clk = system clock

External hardware

Oscilloscope to observe the outputs.

#	Input	Output	Bidirectional

Pinout

#	Input	Output	Bidirectional
0	A0	Q0	
1	A1	Q1	
2	A2	Q2	
3	A3	Q3	
4	ENPB	RCOB	
5	ENTB		
6	LOADB		
7	UP		

VGA Experiments in Tennis [905]

- Author: Tom Keddie
- Description: Simple Game
- GitHub repository
- HDL project
- Mux address: 905
- Extra docs
- Clock: 25175000 Hz

How it works

VGA game using paddles attached to input.

How to test

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

External Hardware

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

Pinout

#	Input	Output	Bidirectional
0	left paddle up	r1/r0 (mole99/digilent)	g0
1	left paddle down	g1/r1 (mole99/digilent)	g1
2	right paddle up	b1/r2 (mole99/digilent)	g2
3	right paddle down	vsync/r3 (mole99/digilent)	g3
4	score reset	r0/b0 (mole99/digilent)	hsync
5	Speed LSB	g0/b1 (mole99/digilent)	vsync
6	Speed MSB	b0/b2 (mole99/digilent)	tied low
7	pmod sel (high=mole99, low=digilent)	hsync/b3 (mole99/digilent)	tied low

Iron Violet [907]

- Author: John Cope
- Description: It's a little memory game, as a treat.
- GitHub repository
- HDL project
- Mux address: 907
- Extra docs
- Clock: 50000000 Hz

The Team

John Cope, Kasey Hill, Matt Johnson, Jacob Reyna

How it works

Simon't involves the device playing a sequence of button-lamps, and the player needs to repeat the sequence by pressing corresponding buttons.

The game keeps a high score for as long as power is maintained, or until it is reset.

Each button has an associated lamp that lights when being presented to the player, as well as when the player presses the buttons back into the game.

When a new game starts, the device shines a button-light for a half second, and the player has to press the same button-lamp within five seconds. Then the game picks a new button-lamp from any of the four, and plays the first color and the new color for the player. the player must press the buttons in the same order they are presented. This continues until the player presses a wrong button, the player waits too long to press a button, or the game runs out of memory at 32 moves.

When the player enters a correct move in a sequence, the timeout for "forget" death is reset, and, if there is at least one additional color in the sequence, the eng-game timer begins again.

At the end of the game, if the player has set a new high score, the game pulses each lamp in the sequence of red-yellow-green-blue. If the player fails to set a new high score, the game pulses each lamp in the reverse order.

External hardware

Lamps need to be driven from the following outputs: 0 to red, 1 to yellow, 2 to green, and 3 to blue. Four buttons is connected to inputs 0 to 4, and they is physically located to correspond with the lamps.

A fifth button is connected to input 5, and is used to start a game.

Pinout

#	Input	Output	Bidirectional
0	Red Button	Red Lamp	
1	Yellow Button	Yellow Lamp	
2	Green Button	Green Lamp	
3	Blue Button	Blue Lamp	
4	Start Game		
5			
6			
7			

Pong [909]

- Author: Alex Segura
- Description: 2-player pong game
- GitHub repository
- HDL project
- Mux address: 909
- Extra docs
- Clock: 25175000 Hz

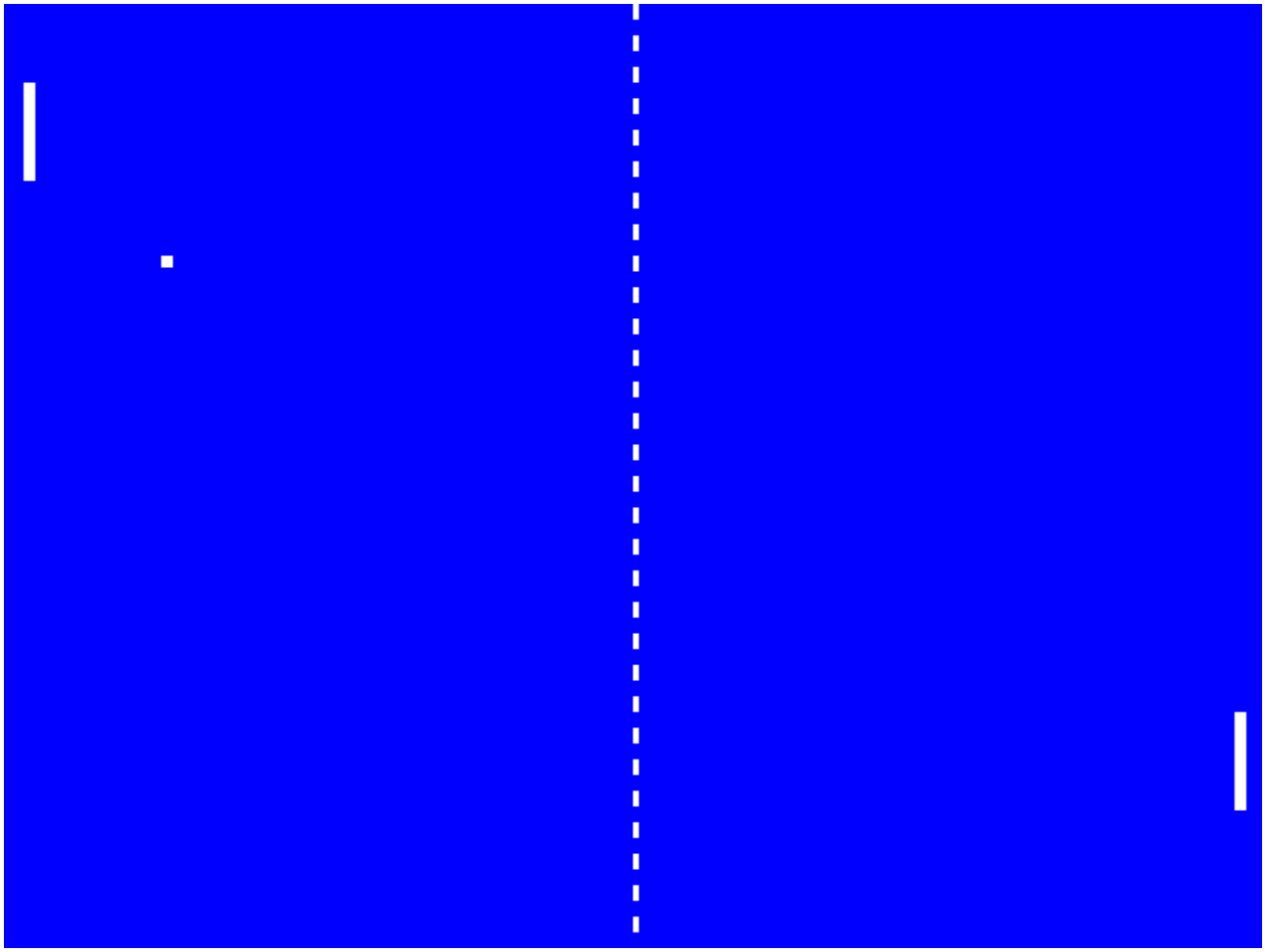


Figure 73: screenshot

How it works

The pixel clock drives a VGA sync generator that scans the screen. Logic in `pong.v` determines whether to draw a pixel.

At each vsync, collisions are detected and the state of the game is updated.

For the paddle controls, the game uses modified versions of the debounce and encoder logic from the course to generate the control signals.

How to test

Two vertical paddles and a ball should render at 640x480 resolution. A vertical "net" should be visible at the middle of the screen. Paddles should respond to the encoders. The ball should bounce from the top and bottom boundaries of the screen and should bounce off the paddles. The game should reset when the ball crosses beyond either paddle.

The verilog code can be run under verilator simulation:

```
cd src
make -B pong
./obj_dir/pong
```

SDL2 is a necessary dependency.

W and S control the left paddle. Up and down arrow keys control the right paddle.

External hardware

- Two rotary encoders, one for each paddle.
- TinyVGA Pmod or similar.

Pinout

#	Input	Output	Bidirectional
0	Paddle 1 encoder A	R0	
1	Paddle 1 encoder B	G0	
2	Paddle 2 encoder A	B0	
3	Paddle 2 encoder B	VSYNC	
4		R1	
5		G1	
6		B1	
7		HSYNC	

KianV uLinux SoC [910]

- Author: Hirosh Dabui
- Description: A RISC-V ASIC that can boot Linux.
- GitHub repository
- HDL project
- Mux address: 910
- Extra docs
- Clock: 0 Hz

How it works

32-bit RISC-V IMA processor, capable of booting Linux. Features 16 MiB of external SPI flash memory, 8 MiB of external PSRAM, a UART peripheral, and a SPI peripheral.

System Memory Map

The system memory map is as follows:

Address	Size	Purpose
0x10000000	0x14	UART Peripheral
0x10500000	0x14	SPI Peripheral
0x11100000	0x04	Reset / HALT control
0x20000000	16 MiB	SPI Flash
0x80000000	8 MiB	PSRAM

The system boots from the SPI flash memory. After reset, the CPU starts executing code from 0x20100000 (corresponding to the offset 0x100000 into the SPI flash memory), where the bootloader is expected to be.

UART Peripheral registers

Address	Name	Description
0x10000000	UART_DATA	Write to transmit, read to receive
0x10000005	UART_LSR	UART line status register
0x10000010	UART_DIV	Clock divider for UART baud rate

SPI Peripheral registers

Address	Name	Description
0x10500000	SPI_CTRL0	SPI Peripheral Control
0x10500004	SPI_DATA0	SPI Data
0x10500010	SPI_DIV	Clock divider for SPI peripheral

CPU control register

Address	Name	Description
0x11100000	CPU_RESET	Write 0x7777 to reset the CPU, 0x5555 to halt the CPU.

How to test

Build the system image as described in the kianRiscV repo and load it into the SPI flash memory:

Flash offset	File name	Description
0x100000	`bootloader.bin`	Bootloader
0x180000	`kianv.dtb`	Device Tree Blob
0x200000	`Image`	Linux kernel + rootfs

The system runs at 30 MHz, with a maximum tested speed of 34.5 MHz.

External hardware

QSPI Pmod - can be purchased from the Tiny Tapeout store.

Pinout

#	Input	Output	Bidirectional
0		spi_cen0	ce0 flash
1		spi_sclk0	sio0
2	spi_sio1_so_miso0	spi_sio0_si_mosi0	sio1
3	uart_rx	led[0]	sck
4		uart_tx	sd2

#	Input	Output	Bidirectional
5		led[1]	sd3
6		led[2]	cs1 psram
7		led[3]	always high

Moosic logic-locked design [911]

- Author: Gabriel Gouvine
- Description: 8-bit counter locked with Moosic logic locking
- GitHub repository
- HDL project
- Mux address: 911
- Extra docs
- Clock: 0 Hz

How it works

This is a simple counter that is incremented every time the first input bit is set.

The trick is that is it locked using logic locking, so that it won't work unless the proper key is set first.

How to test

You need to initialize the key with inputs "11100110" (or 0xe6). The 6 most significant bits (111001) are the key, and the second bit is the key enable. Then you can run the counter: "00000001" will increment it, while "00000000" will keep the same value.

External hardware

This is purely self-contained to demonstrate logic locking.

Pinout

#	Input	Output	Bidirectional
0	DO_INCR	CNT_0	
1	KEY_ENABLE	CNT_1	
2	KEY_0	CNT_2	
3	KEY_1	CNT_3	
4	KEY_2	CNT_4	
5	KEY_3	CNT_5	
6	KEY_4	CNT_6	
7	KEY_5	CNT_7	

ledcontroller [961]

- Author: Mathias Garstenauer
- Description: A WS2812b addressable LED controller configurable via I2C
- GitHub repository
- HDL project
- Mux address: 961
- Extra docs
- Clock: 25000000 Hz

How it works

This tinytapeout projects implements i2c to drive ws2812b individual addressable LEDs. The IC can be addressed with the address 0x4A. The register address corrosponds with the sequential bytes of the leds as follows:

address	data
0x00	green0
0x01	red0
0x02	blue0
0x03	green1
0x04	red1
0x05	blue1
0x06	green2
...	...
0x	green
0x	red
0x	blue

i2c uses external pullups and open collector outputs. In order to implement this the IOs were configured so that the input is handled as input by the verilog module. The out put is set to 0 and the i2c output of the verilog modules toggles between in- and output mode. Therefor listening to the i2c communication should work just fine with the IOs in input mode, where they have a high impedance. If something has to be sent (e.g. acknowledge) the IO is set to output a low value and pulls the i2c line to ground. Should there be any problems with this configuration the SDA and SCL lines are also present on the normal outputs.

How to test

Connect an i2c-master to the chip and write values to the desired registers. The IC supports sequetial write by auti incrementing the register address if more than one byte is sent. The LED should update immediately.

External hardware

Microntroller/computer (e.g. STM32, Arduino, Raspberry Pi, ...), ws2812b LED (strip, matrix, ...), external pullup resistors for i2c

Pinout

#	Input	Output	Bidirectional
0		i2c SCL alternative for PNP open collector	i2c SCL
1		i2c SDA alternative for PNP open collector	i2c SDA
2		ws2812b LED output	
3			
4			
5			
6			
7			

Digitaler Filter [963]

- Author: Nico Rathmayr
- Description: FIR-Filter using two coefficents to filter 8-bit signal
- GitHub repository
- HDL project
- Mux address: 963
- Extra docs
- Clock: 0 Hz

How it works

The goal of this project is to design and implement an efficient digital filter capable of filtering unwanted frequencies from a digital signal.

The FIR filter (Finite Impulse Response) is characterized by its finite impulse response defined by a finite number of coefficients. In this case, only two coefficients are used, simplifying the design process and optimizing the implementation on an FPGA (Field-Programmable Gate Array) or another digital circuit.

The two coefficients are carefully selected to achieve the desired frequency response of the filter. It is essential to consider the requirements of the application, whether it be for audio processing, image processing, or any other signal processing application.

Assignment of digital inputs and outputs:

- ui_in: 8-bit input signal 'x'
- uio_in: 8-bit coefficients 'const_h'
- uo_out: 8-bit output signal 'y'
- uio_out: not used!
- uio_oe: not used!

Originally, the filter programming was planned with four coefficients, but due to chip capacity limitations, this ambition had to be reduced to two coefficients. The code includes comments for additional four-coefficient support, ensuring the possibility of future program expansion.

To read all coefficients with just one 8-bit input and store them in the corresponding register, a shift register has been implemented. Each new clock signal edge allows storing a newly read value at the desired position in the register. Once the maximum number of positions in the register is occupied, the flag is set to low, and the counter is reset. The input signal is also read using a shift register by shifting the value in the register by one position in each step.

Now, for the actual filter operation: the desired coefficients are multiplied with the input signal and summed after each step. It is important to note that the two registers have different sizes and need to be adjusted accordingly. The filtered input signal is output as the output signal from a section of the bit sequence stored in the sum. These steps ensure precise signal processing and demonstrate the program's adaptability for potential future expansions.

How to test

Refer to the Testbench.

External hardware

You do not need any special external hardware.

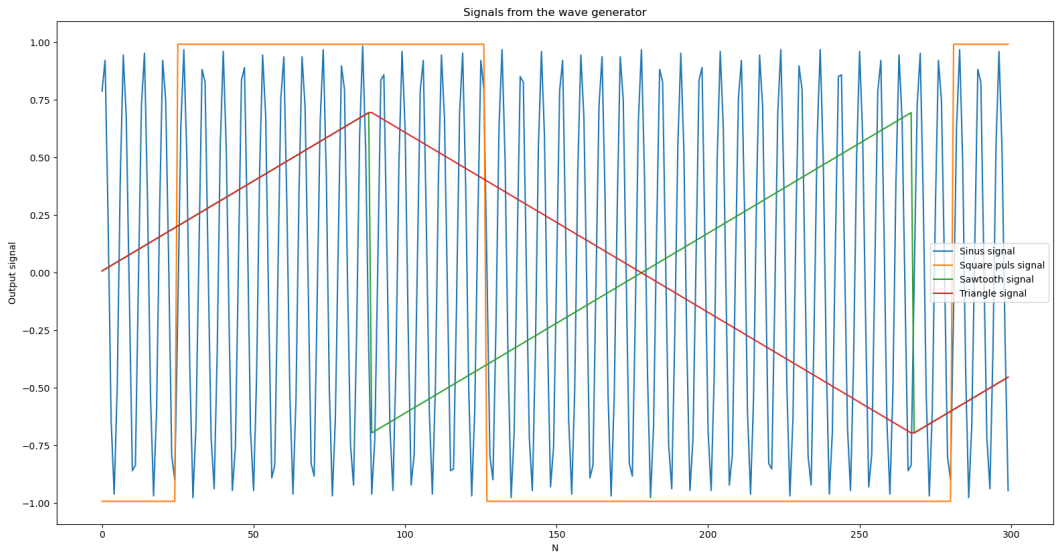
Pinout

#	Input	Output	Bidirectional
0	input signal bit 0	output signal bit 0	input coefficient bit 0
1	input signal bit 1	output signal bit 1	input coefficient bit 1
2	input signal bit 2	output signal bit 2	input coefficient bit 2
3	input signal bit 3	output signal bit 3	input coefficient bit 3
4	input signal bit 4	output signal bit 4	input coefficient bit 4
5	input signal bit 5	output signal bit 5	input coefficient bit 5
6	input signal bit 6	output signal bit 6	input coefficient bit 6
7	input signal bit 7	output signal bit 7	input coefficient bit 7

Wave Generator [965]

- Author: Michael Mayr
- Description: Generates various functions, such as a sine wave, a sawtooth wave, a triangular wave and a squared wave.
- GitHub repository
- HDL project
- Mux address: 965
- Extra docs
- Clock: 0 Hz

How it works



The Wave Generator is a project that deals with the generation of various signals. These signals are a sine function, a triangle function, a sawtooth function and a square pulse function. The desired function, which then provides the output value, can be selected using simple control signals.

Each calculated value is generated as a 2's complement in the fixed point system Q7 and then output accordingly either via the parallel or the serial interface. An SPI interface is used for the serial output, but this can only write and not read. However, this makes it possible to connect a DAC to the wave generator in order to convert the digital signals into analogue signals.

In general, the implemented signals can be influenced by three parameters:

- due to the internal clock frequency f_{clk}

- by the phase parameter
- by the amplitude parameter

These three parameters allow these signals to be flexibly configured in terms of their respective properties. Depending on the type of signal, however, certain restrictions must be taken into account for the parameters. These restrictions are then described in the respective section for the corresponding signal.

In addition, the sampling frequency f_s is linked to the internal clock frequency f_{clk} of the system by a factor of 40. A value therefore requires 40 clock pulses to be calculated and output via the serial interface. This results in the following relationship: $f_s = \frac{f_{clk}}{40}$. The maximum internal clock frequency f_{clk} is 66;MHz and therefore the maximum sampling frequency f_s is 1.65;MHz.

Inputs and outputs The Wave Generator uses all 24 digital pins. The input pins are described in the pinout section. These 8 input pins add up to the bit vector parameter. This is required to set the phase and amplitude, as the respective value is applied to it as a 2's complement in the fixed-point system Q7. The bidirectional pins "set phase" and "set amplitude" are used to adopt this value as the phase value or amplitude value in the chip.

The current value is output in parallel form on the output wave pins and in serial form on the SPI pins. The SPI pins have the same names as defined in the standard, with the exception that the input pin MISO is missing. This is not required due to the pure data generation.

The bit vector wavefrom (see pinout section) is used to select the desired function. The desired function then results depending on the bit pattern:

- 00: Sine wave
- 01: Square puls wave
- 10: Sawtooth wave
- 11: Triangular wave

The value generation can be stopped by the pin enable (see pinout section) with a LOW level and continued with a HIGH level.

Sine wave The sine wave is generated using a Cordic algorithm, as shown in the below figure. This Cordic algorithm is used in the mode rotation and with the coordinate system circular. The following function is implemented in the Cordic algorithm:

$$x_n = x_0 \cos(z_0) - y_0 \sin(z_0)$$

$$y_n = y_0 \cos(z_0) + x_0 \sin(z_0)$$

To generate a sine wave from this formula, y_0 must be set to 0 and x_0 must be loaded with the desired amplitude. The current phase z_0 of the sine wave is calculated in advance by a phase accumulator. This then transfers its current phase value to the Z input of the Cordic algorithm. This allows the Cordic algorithm to generate a sine wave at the Y output, which is then subsequently output.

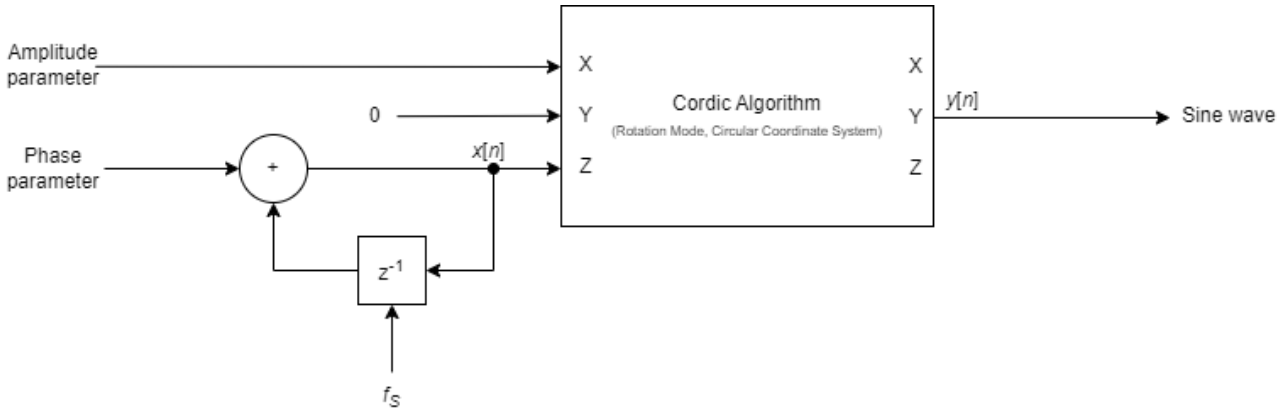
Due to a property of the Cordic algorithm, the amplitude parameter must be scaled by a factor k before loading into the chip. This factor k has a value of 0.6073. This prevents an overflow in the Q7 format and the correct values are calculated by the algorithm.

This results in the following formulae for the parameters:

$$\text{Amplitude parameter} = kA$$

$$\text{Phase parameter} = \frac{2f}{f_s}$$

Where $A \in [-1+2^{-7}, 1-2^{-7}]$ is the desired amplitude and $f \in (0, \frac{f_s}{2}]$ is the desired frequency.



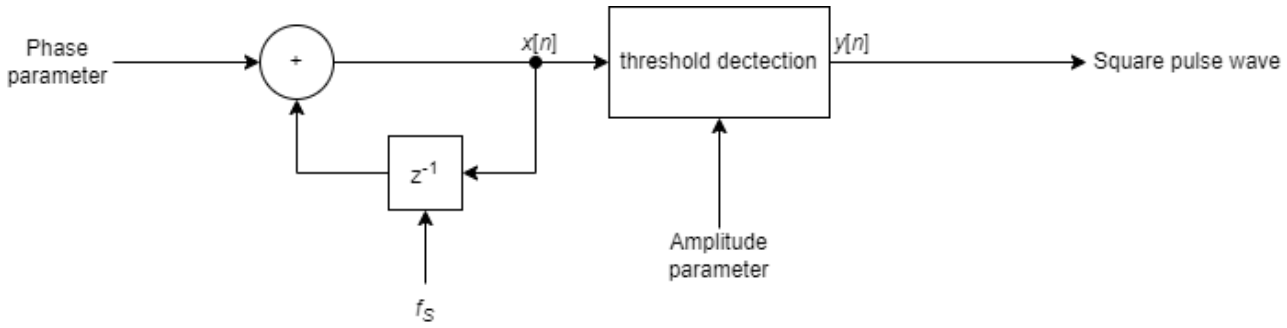
Square pulse wave The square pulse wave is again calculated with a phase accumulator and a threshold dectection unit (see figure below). The phase accumulator generates a sawtooth function $x[n]$, where the difference between one value and its subsequent value is the phase parameter. An exception occurs in the case of an overflow. In this case, the difference is much greater, as the value moves to the negative end of the number format. This generated value is then compared with the current amplitude parameter. In this case, the amplitude parameter is a threshold parameter. The output is then generated according to the following principle: If $x[n]$ is greater than the amplitude parameter, $y[n]$ has the value $1 - 2^{-7}$. Conversely, if $x[n]$ is less than or equal to the amplitude parameter, $y[n]$ becomes $-1 + 2^{-7}$.

The following formulae are required for the parameters:

$$\text{Amplitude parameter} = 1 - 2^{-7} - (2 - 2^{-7})\frac{T_{on}}{T}$$

$$\text{Phase parameter} = \frac{2 - 2^{-7}}{T f_s}$$

Where $T > \frac{1}{f_s}$ is the desired period duration and $T_{on} \in (0, T)$ is the desired pulse width.



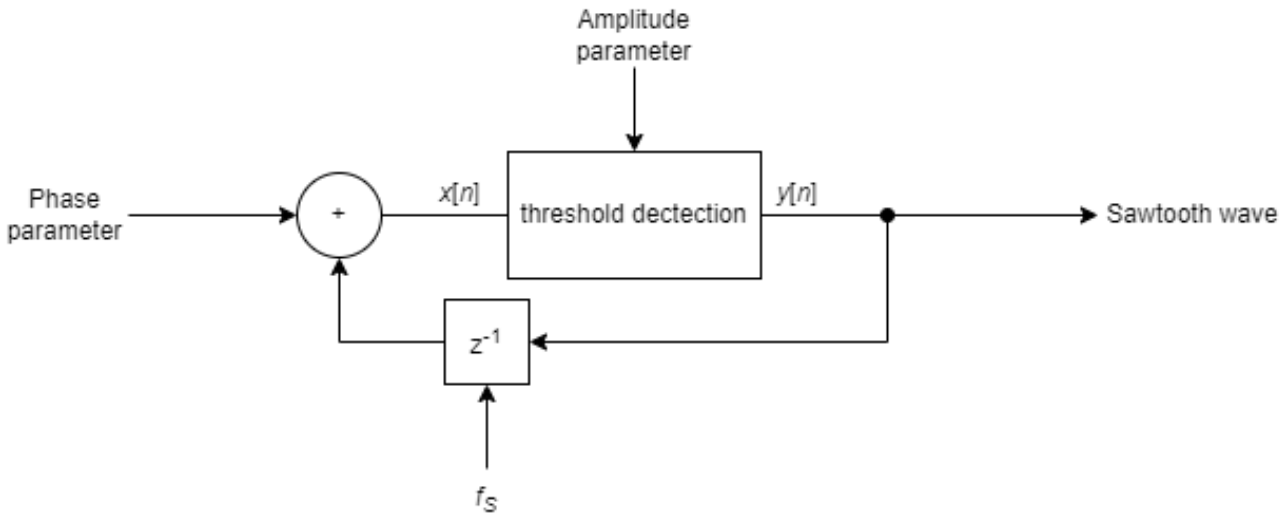
Sawtooth wave The sawtooth wave is basically generated with a phase accumulator (see figure below). The only difference is that $y[n]$ is fed back instead of $x[n]$. This makes it possible to generate a sawtooth function with a specific amplitude value. The threshold dectection unit thus ensures that the function remains in the range from minus amplitude parameter to amplitude parameter. To do this, $x[n]$ is checked and if this value is greater than the amplitude parameter, $y[n]$ is set to the negative value of the amplitude parameter. If this does not occur, $x[n]$ becomes $y[n]$.

The following formulae are required for the parameters:

$$\text{Amplitude parameter} = A$$

$$\text{Phase parameter} = 2A\frac{f}{f_s}$$

Where $A \in [-1 + 2^{-7}, 1 - 2^{-7}]$ is the desired amplitude and $f \in (0, \frac{f_s}{2}]$ is the desired frequency.

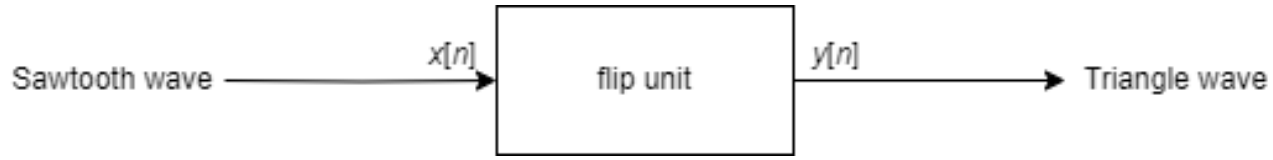


Triangle wave With the triangle wave, the current value is taken from the sawtooth wave and with every second overflow, therefore a change from a positive value to a negative value (see figure below), the output value is multiplied by -1.

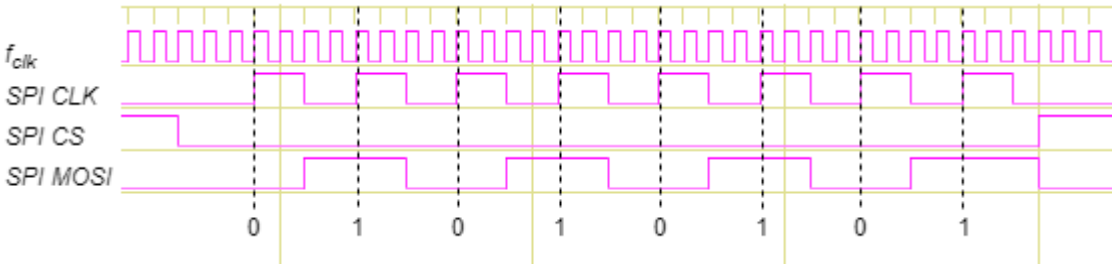
The following formulas are required for the parameters:

$$\text{Amplitude parameter} = A$$

$$\text{Phase parameter} = 4A\frac{f}{f_s}$$



SPI Interface The calculated value is output serially via the SPI interface. The below timing diagram is used for this purpose. SPI is used in this project with CPOL=0 and CPHA=0. The SPI CLK requires 4 cycles of f_{clk} for one cycle. The other properties can be taken from the timing diagram.



How to test

The following procedure is required to generate the desired wave:

1) Initialisation
    - Reset the device.
    - The enable pin should be connected to LOW.
2) Choose the desired wave
    - Set the code for the desired wave on the waveform pins.
3) Calculate the amplitude and the phase parameter with the formulars form the above sections.
4) Set amplitude parameter
    - Load the calculated results on the parameter pins
    - Then send a puls to the set amplitude pin
5) Set phase parameter
    - Load the calculated results on the parameter pins
    - Then send a puls to the set phase pin
6) Enable the output generation
    - The enable pin should be connected to HIGH.
7) Now the chip generates the desired wave and output the values on the parallel and serial interface.

External hardware

Through the SPI interface it is possible to get an analogue signal through a suitable DAC. However, this DAC must fulfill the requirements of the SPI interface.

Pinout

#	Input	Output	Bidirectional
0	parameter bit 0 (LSB)	output wave bit 0	(input) enable pin
1	parameter bit 1	output wave bit 1	(input) waveform bit 0 pin
2	parameter bit 2	output wave bit 2	(input) waveform bit 1 pin
3	parameter bit 3	output wave bit 3	(input) set phase pin

#	Input	Output	Bidirectional
4	parameter bit 4	output wave bit 4	(input) set amplitude pin
5	parameter bit 5	output wave bit 5	(output) spi cs pin
6	parameter bit 6	output wave bit 6	(output) spi mosi pin
7	parameter bit 7 (MSB)	output wave bit 7	(output) spi clk pin

jku-tt06-advanced-counter [967]

- Author: Martin Putz
- Description: Multi-Digit Counter with changeable maximum values and carry over.
- GitHub repository
- HDL project
- Mux address: 967
- Extra docs
- Clock: 1000000 Hz

How it works

The design uses in it's core a series of modified 2-bit-up-down-counters that allow for setting a custom maximum value. Carry information is only provided if this is intended by the operating mode. The mode can is selected from individual inputs by a mode selection module. The output of the counters is serialized and encoded for the 7-segment-display.

External hardware

For each digit used (due to limitations that's only 2) a push button in active high configuration is used. A third push button for the refreshing of limits is also required. Further more, 3 switches are used to determine the operating mode. On the output side, for each digit an 8-bit-shift-register and a 7-segment-display is used, as the output value is serialized. For the shift clock two inverse outputs are provided, in case a shift register with both a load and output input. In that case, the shift_clk is used for loading and not_shift_clk is used for output. The design was made with the register HC595 in mind, but will work with other 8-bit shift registers.

Pinout

#	Input	Output	Bidirectional
0	Button 0 In	Digit 0 Out	Up-Down-Select In
1	Button 1 In	Digit 1 Out	Set-Carry In
2			Set-Max In
3			Refresh-Limits In
4			
5			

#	Input	Output	Bidirectional
6			Shift-Clk Out
7			Not-Shift-Clk Out

PI-Based Fan Controller [969]

- Author: Dominik Brandstetter
- Description: Temperature Based Fan Speed Controller
- GitHub repository
- HDL project
- Mux address: 969
- Extra docs
- Clock: 1000000 Hz

How it works

This project involves reading a 4-bit ADC value through a dedicated interface. Another a 4-bit interface allows the user to set the desired target value. The integrated controller, designed with fixed parameters, regulates the fan speed through a PWM (Pulse Width Modulation) output operating at approximately 25 kHz. The controller maintains a minimum duty cycle of around 20%, and it has the capability to increase this value up to 100%. The output consists of a signed 4-bit controller value along with the corresponding PWM signal. Additionally, the current controller value can be read from the 7-Segment-Display. This configuration ensures precise control and adjustment of the fan speed based on the input parameters provided through the 4-bit interfaces, with the added feature of fixed controller parameters for simplicity and stability.

How to test

After reset, the fan controller should initiate operation, adjusting the fan speed based on the setpoint and ADC value. The PWM output, set at approximately 25 kHz, regulates the fan speed.

External hardware

4-Bit ADC, LED display, Fan

Pinout

#	Input	Output	Bidirectional
0	ADC_BIT_0	segment_a	Controller_SET_BIT_0
1	ADC_BIT_1	segment_b	Controller_SET_BIT_1

#	Input	Output	Bidirectional
2	ADC_BIT_2	segment_c	Controller_SET_BIT_2
3	ADC_BIT_3	segment_d	Controller_SET_BIT_3
4	SET_BIT_0	segment_e	Controller_SIGN_BIT_4
5	SET_BIT_1	segment_f	
6	SET_BIT_2	segment_g	
7	SET_BIT_3	PWM Output	

PS/2 Keyboard to Morse Code Encoder [971]

- Author: Daniel Baumgartner
- Description: PS/2 Keyboard to Morse Code Encoder
- GitHub repository
- HDL project
- Mux address: 971
- Extra docs
- Clock: 10000000 Hz

How it works

This project implements a PS/2 keyboard to Morse code encoder. For this, the output of a PS/2 keyboard is evaluated. If a key is pressed on the keyboard, the input is stored in a 12-bit temporary buffer. All alphanumeric characters are supported, except for umlauts and numbers on the number pad. The output of the design depends on the selected mode. Mode 1 is activated with the F1 key and is the default mode after a reset. In this mode, the contents of the buffer are output when the enter key is pressed. Mode 2 can be activated with the F4 key. If this mode is active, the content of the buffer is output as soon as the space bar is pressed. The Morse code output consists of dots (dits) and dashes (dahs). The timings for dits, dahs, symbol spacing and spaces have been selected so that a Morse signal with approx. 15 WPM (words per minute) is generated. The design has four outputs. Dit and dah each have their own output. A further output is a combination of dit and dah. This output is active when a dit or dah is being output. The last output is intended for connecting a buzzer or a small speaker. This output emits a 600Hz square wave signal when either a dit or a dah is output.

This project is written in Verilog. The design includes three separate modules. One module for decoding the PS/2 data from the keyboard that evaluates the data sent by the keyboard. Another module generates the Morse code output based on the keyboard input, and one additional module generates the 600Hz square wave signal. Across the modules, multiple finite state machines (FSM) are used. The exact implementation can be found on GitHub. For further information about the PS/2 protocol, take a look at PS/2 Wikipedia.

How to Test the Design

Required Hardware To test the design, a PS/2 keyboard is needed. It is important to use a logic converter (5V to 3.3V) for the data and clock line, as the Tiny Tapeout chip only supports 3.3V! Connect the data line to `ui_in[1]` and the clock line to

`ui_in[0]`. Additionally, two pull-up resistors (5kΩ) against 5V must be connected to the two lines. Don't forget to supply the keyboard with 5V and GND. If you don't have a PS/2 keyboard, you can also use a USB keyboard. Some, but not all, USB keyboards still support the PS/2 protocol. In this case, D+ is clock and D- is data (don't forget the pull-up resistors and the level shifter).

After everything is connected, perform a reset and start typing on the keyboard. The input should be stored in a buffer (max. 14 characters). With F1 and F4, you can switch between two modes. Mode 1 (F1) is the default mode. In this mode, the characters stored in the buffer are output when enter is pressed on the keyboard. Mode 2 (F4) outputs the buffer as soon as the space bar is pressed. It is worth noting that no new characters can be entered during output. Segment A of the seven-segment display lights up when a dit is output, segment D lights up when a dah is output. Segment G lights up when a dit or a dah is output. A buzzer can be connected to output `uo_out[7]` (segment DP) which emits the Morse code as a tone (600 Hz). Before submission to TT06 the design was tested on a Spartan 3AN Starter Kit, so it should work (hopefully).

PS: You may be surprised when you press the F6 key :)

Pinout

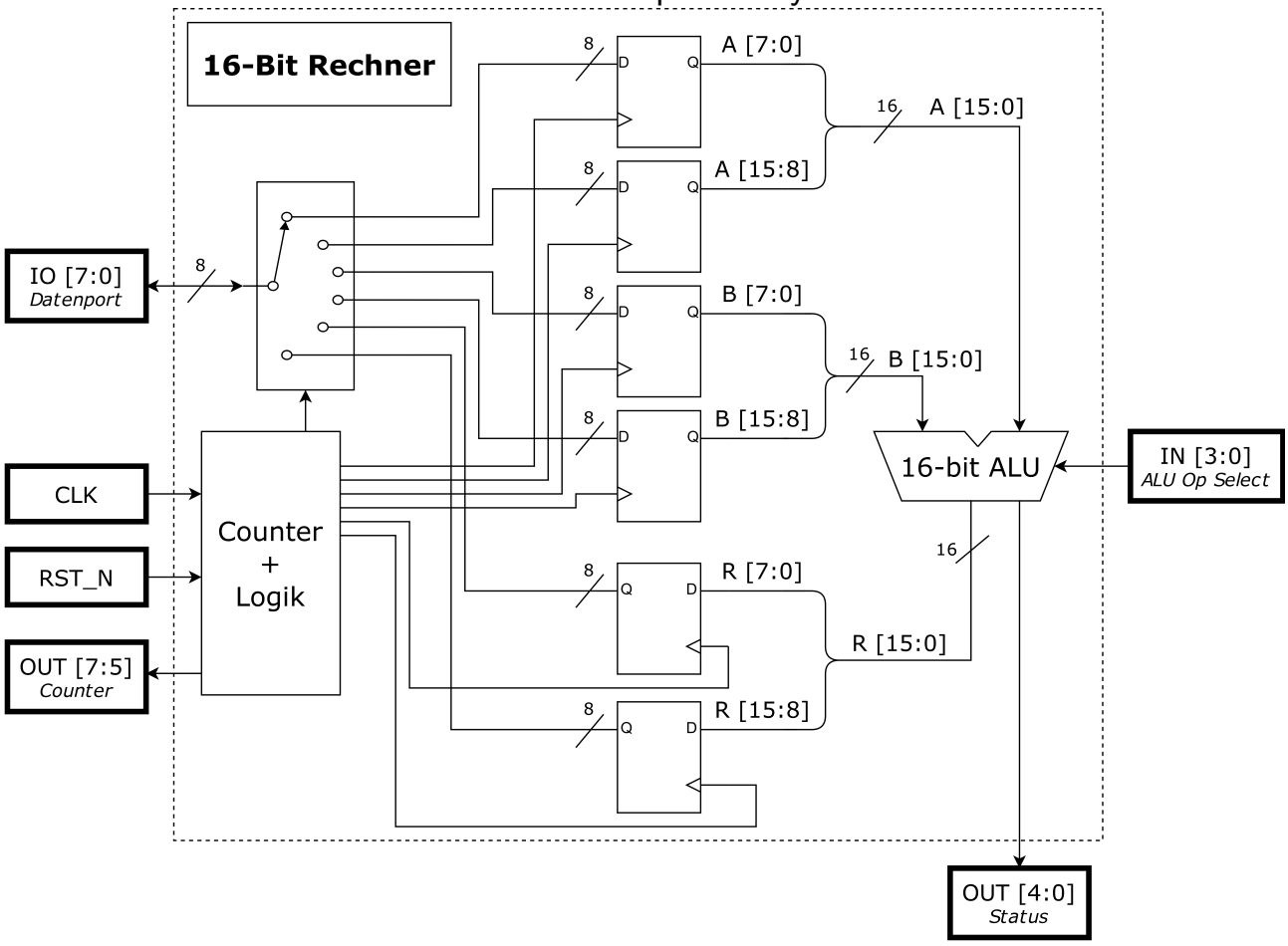
#	Input	Output	Bidirectional
0	PS/2 Clock	Dit Output	
1	PS/2 Data		
2			
3		Dah Output	
4			
5			
6		Morse Code Output	
7		Morse Code Output (Buzzer)	

16-bit calculator [973]

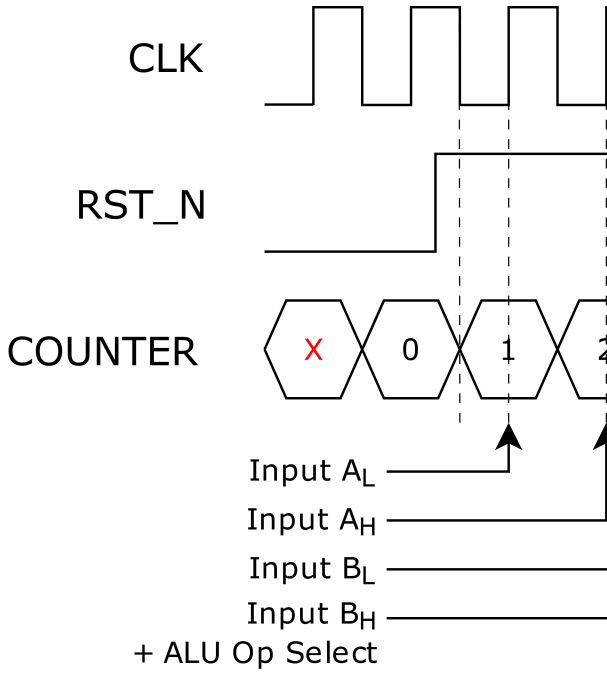
- Author: Benedikt Muehlbachler
- Description: calculator using 16-bit ALU with 8-bit IO-data port reading/writing data
- GitHub repository
- HDL project
- Mux address: 973
- Extra docs
- Clock: 0 Hz

How it works

The 16-bit calculator looks in a simplified symbolic schematic as follows:



You have an IO-Port (8-bit) to load data to registers for the operands A and B for the calculation operation as well as to output the result of the alu operation. The IN[3:0] are used for the alu operation selection (there are 12 different operations possible). The CLK is the clock and RST_N is the reset pin. There are also OUT[4:0] which shows the status of the alu operation as well as the OUT[7:5] to see at which step the whole operation is.



To better clarify how it works, there is a timing diagram:

- As long as the RST_N pin is low, the counter is reset and nothing happens.
- If RST_N is HIGH then the operation starts.
- At a negative CLK edge the counter increments and gets 1. At the following positive CLK edge whatever is on the IO-Port gets loaded into the low-byte of operand A (at the next negative CLK edge the counter increments.
- At Counter=2 and POS EDGE CLK: IO-Port data gets loaded into High-Byte of A.
- At Counter=3 and POS EDGE CLK: IO-Port data gets loaded into Low-Byte of B.
- At Counter=4 and POS EDGE CLK: IO-Port data gets loaded into High-Byte of B. Additionally, the ALU Operation gets selected.
- At Counter=5 and POS EDGE CLK: The result of the ALU operation is on the IO-Port (Low-Byte of result), and the Status of the ALU-Operation is updated at the Status output.
- At Counter=6 and POS EDGE CLK: The high-Byte of the ALU operation is on the IO-Port.
- At the following NEG EDGE CLK, the Counter will restart from zero (the same happens if RST_N gets low during the operation.

The following alu operations are possible: | ALU Op Select | Operation | Name of Operation | |————|————|————————| | 0 | R=0 | Null Operation | | 1 | R=~A | Inverse of A | | 2 | R=A«1 | Shift left A | | 3 | R=A»1 | Shift right A | | 4 | R=rot_l(A) | Rotate left A | | 5 | R=rot_r(A) | Rotate right A | | 6 | R=A+1 | Increment A | | 7 | R=A-1 | Decrement A | | 8 | R=A and B | Bitwise A and B | | 9 | R=A or B | Bitwise A or B | | 10 | R=A xor B | Bitwise A xor B | | 11 | R=A+B | Addition of A and B | | 12 | R=A-B | Subtraction of A and B |

The status out register is as follows: | Status Register | Flag | Description | |——————|——————————|————————————————————-| | Bit 0 | Wrong Operation Flag (WF) | Set when ALU Op Select is 13,14 or 15 (there is no operation). | | Bit 1 | Zero Flag (ZF) | Set when result is zero. | | Bit 2 | Sign Flag (SF) | Set when the highest bit (bit 15) is 1. | | Bit 3 | Carry Flag (CF) | Set for unsigned notation when there is a carry. | | Bit 4 | Overflow Flag (OF) | Set for signed notation when there is an overflow. |

How to test

-

External hardware

You do not need any special external hardware.

Pinout

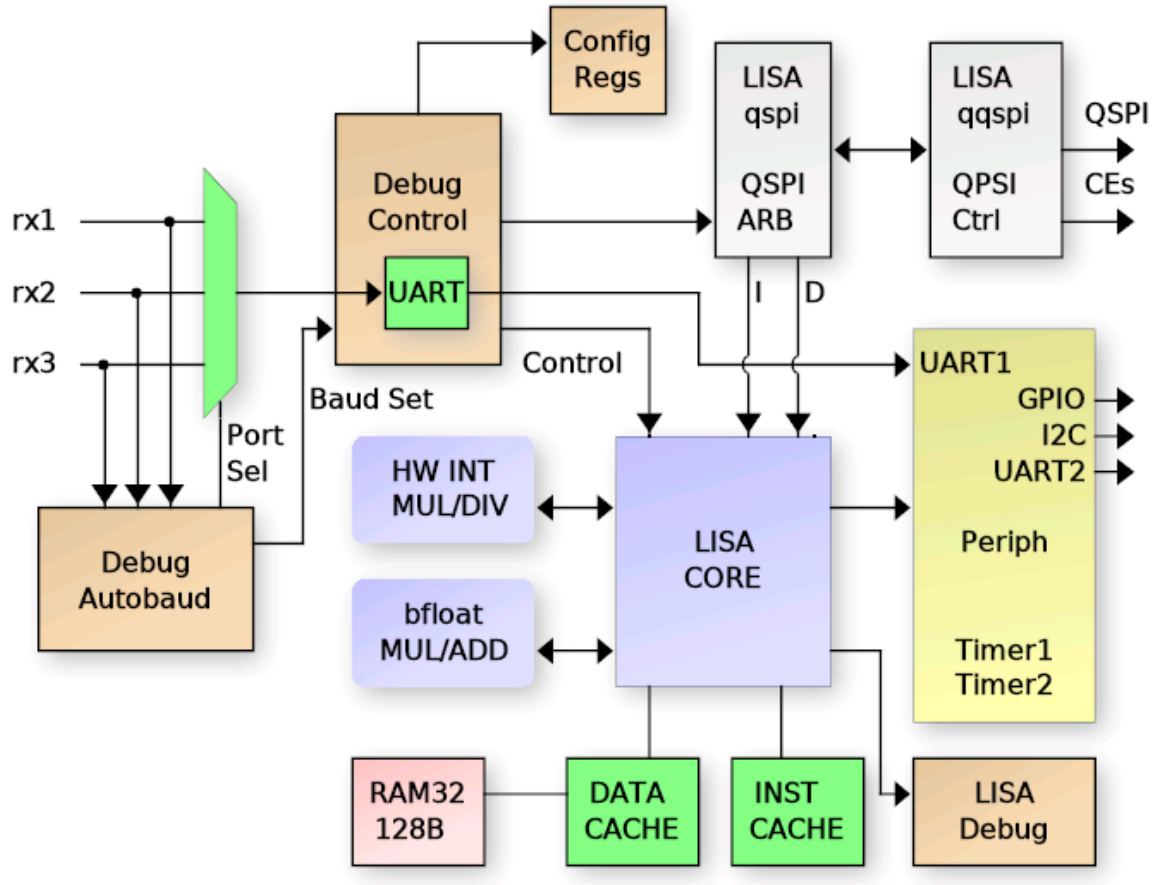
#	Input	Output	Bidirectional
0	alu operation select bit 0	status bit 0 (wrong operation flag)	data port bit 0
1	alu operation select bit 1	status bit 1 (zero flag)	data port bit 1
2	alu operation select bit 2	status bit 2 (sign flag)	data port bit 2
3	alu operation select bit 3	status bit 3 (carry flag)	data port bit 3
4		status bit 4 (overflow flag)	data port bit 4
5		counter bit 0	data port bit 5
6		counter bit 1	data port bit 6
7		counter bit 2	data port bit 7

LISA 8-Bit Microcontroller [974]

- Author: Ken Pettit
- Description: 8-Bit Microcontroller SOC with 128 bytes DFFRAM module
- GitHub repository
- HDL project
- Mux address: 974
- Extra docs
- Clock: 22000000 Hz

LISA Overview?

LISA is a Microcontroller built around a custom 8-Bit Little ISA (LISA) microprocessor core. It includes several standard peripherals that would be found on commercial microcontrollers including timers, GPIO, UARTs and I2C. The following is a block diagram of the LISA Microcontroller:



LISA Microcontroller Block Diagram

- The LISA Core has a minimal set of register that allow it to run C programs:
  - Program Counter + Return Address Resister

  - Stack Pointer and Index Register (Indexed DATA RAM access)
  - 8-bit Accumulator + 16-bit BF16 Accumulator and 4 BF16 registers

Deailed list of the features

- Harvard architecture LISA Core (16-bit instruction, 15-bit address space)
- Debug interface
  - UART controlled
  - Auto detects port from one of 3 interfaces
  - Auto detects the baud rate
  - Interfaces with SPI / QSPI SRAM or FLASH
  - Can erase / program the (Q)SPI FLASH
  - Read/write LISA core registers and peripherals
  - Set LISA breakpoints, halt, resume, single step, etc.
  - SPI/QSPI programmability (single/quad, port location, CE selects)
- (Q)SPI Arbiter with 3 access channels
  - Debug interface for direct memory access
  - Instruction fetch
  - Data fetch
  - Quad or Single SPI. Hereafter called QSPI, but supports either.
- Onboard 128 Byte RAM for DATA / DATA CACHE
- Data bus CACHE controller with 8 16-byte CACHE lines
- Instruction CACHE with a single 4-instruction CACHE line
- Two 16-bit programmable timers (with pre-divide)
- Debug UART available to LISA core also
- Dedicated UART2 that is not shared with the debug interface
- 8-bit Input port (PORTA)
- 8-bit Output port (PORTB)
- 4-bit BIDIR port (PORTC)
- I2C Master controller
- Hardware 8x8 integer multiplier
- Hardware 16/8 or 16/16 integer divider
- Hardware Brain Float 16 (BF16) Multiply/Add/Negate/Int16-to-BF16
- Programmable I/O mux for maximum flexibility of I/O usage.

It uses a 32x32 1RW DFFRAM macro to implement a 128 bytes (1 kilobit) RAM module. The 128 Byte ram can be used either as a DATA cache for the processor data bus, giving a 32K Byte address range, or the CACHE controller can be disabled, connecting the Lisa processor core to the RAM directly, limiting the data space to

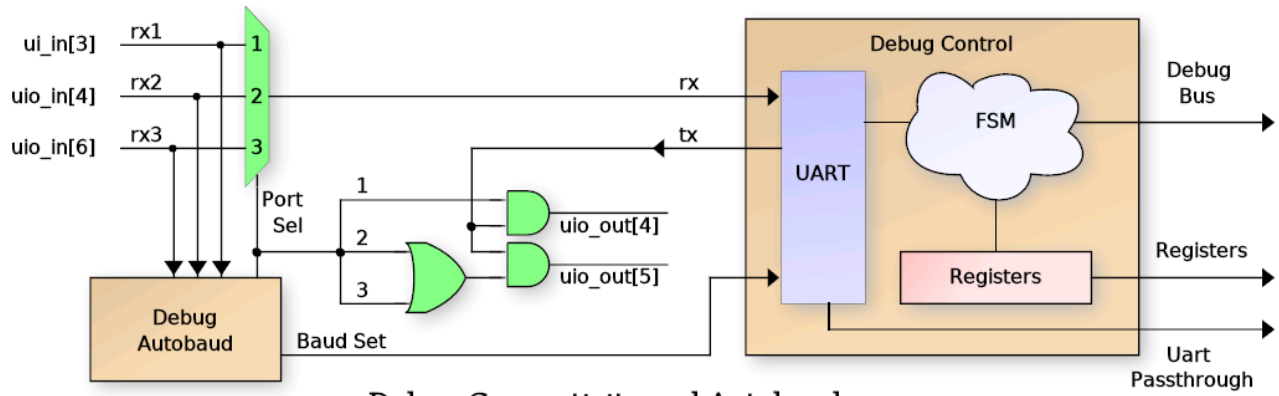
128 bytes. Inclusion of the DFFRAM is thanks to Uri Shaked (Discord urish) and his DFFRAM example.

Reseting the project **does not** reset the RAM contents.

Connectivity

All communication with the microcontroller is done through a UART connected to the Debug Controller. The UART I/O pins are auto-detected by the debug_autobaud module from the following choices (RX/TX):

```
ui_in[3]  / ui_out[4]     RP2040 UART interface
uio_in[4] / uio_out[5]    LISA PMOD board (I am developing)
uio_in[6] / uio_out[5]    Standard UART PMOD
```



Debug Connectivity and Autobaud

The RX/TX pair port is auto-detected after reset by the autobaud circuit, and the UART baud rate can either be configured manually or auto detected by the autobaud module. After reset, the ui_in[7] pin is sampled to determine the baud rate selection mode. If this input pin is HIGH, then autobaud is disabled and ui_in[6:0] is sampled as the UART baud divider and written to the Baud Rate Generator (BRG). The value of this divider should be: clk_freq / baud_rate / 8 - 1. Due to last minute additions of complex floating point operations, and only 2 hours left on the count-down clock, the timing was relaxed to 20MHz input clock max. So for a 20MHz clock and 115200 baud, the b_div[6:0] value would be 42 (for instance).

If the ui_in[7] pin is sampled LOW, then the autobaud module will monitor all three potential RX input pins for LINEFEED (ASCII 0x0A) code to detect baud rate and set the b_div value automatially. It monitors bit transistions and searches for three successive bits with the same bit period. Since ASCII code 0x0A contains a "0 1 0 1 0" bit sequence, the baud rate can be detected easily.

Regardless if the baud rate is set manually or using autobaud, the input port selection will be detect automatically by the autobaud. In the case of manual buad rate selection, it simply looks for the first transition on any of the three RX pins. For autobaud, it select the RX line with three successive eqivalent bit periods.

Debug Interface Details The Debug interface uses a fixed, verilog coded Finite State Machine (FSM) that supports a set of commands over the UART to interface with the microcontroller. These commands are simple ASCII format such that low-level testing can be performed using any standard terminal software (such as minicom, tio. Putty, etc.). The 'r' and 'w' commands must be terminated using a NEWLINE (0x0A) with an optional CR (0x0D). Responses from the debug interface are always terminated with a LINFEED plus CR sequence (0x0A, 0x0D). The commands are as follows (responsce LF/CR ommited):

Command	Description
v	Report Debugger version. Should return: lisav1.2
wAAVVVV	Write 16-bit HEX value 'VVVV' to register at 8-bit HEX address 'AA'.
rAA	Read 16-bit register value from 8-bit HEX address 'AA'.
t	Reset the LISA core.
l	Grant LISA the UART. Further data will be ignored by the debugger.
+++	Revoke LISA UART. NOTE: a 0.5s guard time before/after is required.

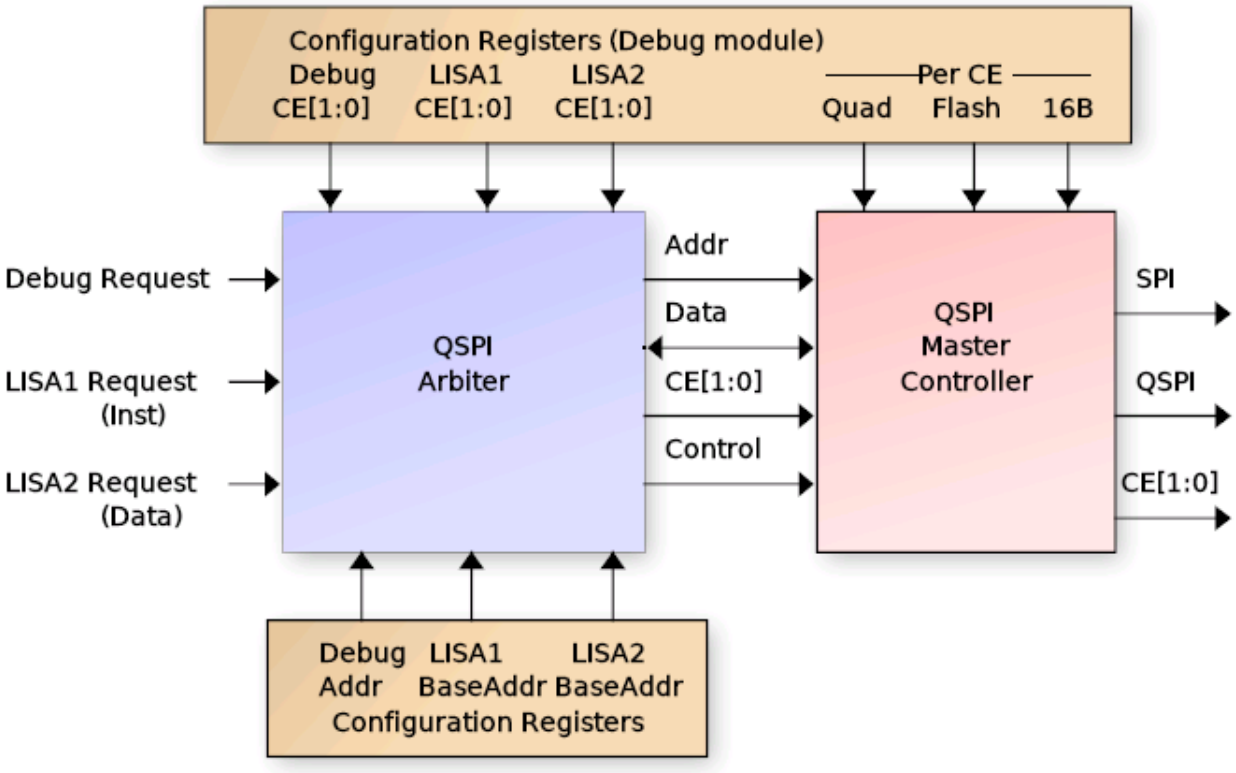
NOTE: All HEX values must be a-f and not A-F. Uppercase is not supported.

Debug Configuration and Control Registers The following table describes the configuration and LISA debug register addresses available via the debug 'r' and 'w' commands. The individual register details will be described in the sections to follow.

ADDR	Description	ADDR	Description
0x00	LISA Core Run Control	0x12	LISA1 QSPI base address
0x01	LISA Accumulator / FLAGS	0x13	LISA2 QSPI base address
0x02	LISA Program Counter (PC)	0x14	LISA1 QSPI CE select
0x03	LISA Stack Pointer (SP)	0x15	LISA2 QSPI CE select
0x04	LISA Return Address (RA)	0x16	Debug QSPI CE select
0x05	LISA Index Register (IX)	0x17	QSPI Mode (QUAD, flash, 16b)
0x06	LISA Data bus	0x18	QSPI Dummy read cycles
0x07	LISA Data bus address	0x19	QSPI Write CMD value
0x08	LISA Breakpoint 1	0x1a	The '+++' guard time count
0x09	LISA Breakpoint 2	0x1b	Mux bits for uo_out

ADDR	Description	ADDR	Description
0x0a	LISA Breakpoint 3	0x1c	Mux bits for uio
0x0b	LISA Breakpoint 4	0x1d	CACHE control
0x0c	LISA Breakpoint 5	0x1e	QSPI edge / SCLK speed
0x0d	LISA Breakpoint 6	0x20	Debug QSPI Read / Write
0x0f	LISA Current Opcode Value	0x21	Debug QSPI custom command
0x10	Debug QSPI Address (LSB16)	0x22	Debug read SPI status reg
0x11	Debug QSPI Address (MSB8)		

LISA Processor Interface Details The LISA Core requires external memory for all Instructions and Data (well, sort of for data, the data CACHE can be disabled then it just uses internal DFFRAM). To accomodate external memory, the design uses a QSPI controller that is configurable as either single SPI or QUAD SPI, Flash or SRAM access, 16-Bit or 24-Bit addressing, and selectable Chip Enable for each type of access. To achieve this, a QSPI arbiter is used to allow multiple accessors as shown in the following diagram:



(Q)SPI Controller Interface Diagram

The arbiter is controlled via configuration registers (accessible by the Debug controller) that specify the operating mode per CE, and CE selection bits for each of the three interfaces:

- Debug Interface
- LISA1 (Instruction fetch)
- LISA2 (Data read/write)

The arbiter gives priority to the Debug accesses and processes LISA1 and LISA2 requests using a round-robbin approach. Each requestor provides a 24-bit address along with 16-bit data read/write. For the Debug interface, the address comes from the configuration registers directly. For LISA1, the address is the Program Counter (PC) + LISA1 Base and for LISA2, it is the Data Bus address + LISA2 Base. The LISA1 and LISA2 base addresses are programmed by the Debug controller and set the upper 16-bits in the 24-bit address range. The PC and Data address provide the lower 16 bis (8-bits overlapped that are 'OR'ed together). The BASE addresses allow use of a single external QSPI SRAM for both instruction and data without needing to worry about data collisions.

When the arbiter chooses a requestor, it passes its programmed CE selection to the QSPI controller. The QSPI controller then uses the programmed QUAD, MODE, FLASH and 16B settings for the chosen CE to process the request. This allows LISA1 (Instruction) to either execute from the same SRAM as LISA2 (Data) or to execute from a separate CE (such as FLASH with permanent data storage).

Additionally the Debug interface has special access registers in the 0x20 - 0x22 range that allow special QSPI accesses such as FLASH erase and program, SRAM programming, FLASH status read, etc. In fact the Debug controller can send any arbitrary command to a target device, using access that either provide an associated address (such as erase sector) or no address. The proceedure for this is:

1. Program Debug register 0x19 with the special 8-bit command to be sent
2. Set the 9-th bit (reg19[8]) to 1 if a 16/24 bit address needs to be sent)
3. Perform a read / write operation to debug address 0x21 to perform the action.

Simple QSPI data reads/write are accomplished via the Debug interface by setting the desired address in Debug config register 0x10 and 0x11, then performing read or write to address 0x20 to perform the request. Reading from Debug config register 0x22 will perform a special mode read of QSPI register 0x05 (the FLASH status register).

Data access to the QSPI arbiter come from the Data CACHE interface (described later), enabling a 32K address space for data. However the design has a CACHE disable mode that directs all Data accesses directly to the internal 128 Byte RAM, thus eliminating the need for external SRAM (and limiting the data bus to 128 bytes).

Programming the QSPI Controller Before the LISA microcontroller can be used in any meaningful manner, a SPI / QSPI SRAM (and optionally a NOR FLASH) must be connected to the Tiny Tapeout PCB. Alternately, the RP2040 controller on the board can be configured to emulate a single SPI (the details for configuring this are

outside the scope of this documentation ... search the Tiny Tapeout website for details.). For the CE signals, there are two operating modes, fixed CE output and Mux Mode 3 "latched" CE mode. Both will be described here. The other standard SPI signals are routed to dedicated pins as follows:

Pin	SPI	QSPI	Notes
uio[0]	CE0	CE0	
uio[1]	MOSI	DQ0	Also MOSI prior to QUAD mode DQ0
uio[2]	MISO	DQ1	Also MISO prior to QUAD mode DQ1
uio[3]	SCLK	SCLK	
uio[4]	CE1	CE1	Must be enabled via uio MUX bits
uio[6]	-	DQ2	Must be enabled via uio MUX bits
uio[7]	-	DQ3	Must be enabled via uio MUX bits

For Special Mux Mode 3 (Debug register 0x1C uio_mux[7:6] = 2'h3), the pinout is mostly the same except the CE signals are not constant. Instead they are "latched" into an external 7475 type latch. This mode is to support a PMOD board connected to the uio PMOD which supports a QSPI Flash chip, a QSPI SRAM chip, and either Debug UART or I2C. For all of that functionality, nine pins would be required for continuous CE0/CE1, however only eight are available. So the external PMOD uses uio[0] as a CE "latch" signal and the CE0/CE1 signals are provided on uio[1]/uio[2] during the latch event. This requires a series resistor as indicated to allow CE updates if the FLASH/SRAM is driving DQ0/DQ1. The pinout then becomes:

Pin	SPI/QSPI	Notes
uio[0]	ce_latch	ce_latch HIGH at beginning of cycle
uio[1]	ce0_latch/MOSI/DQ0	Connection to FLASH/SRAM via series resistor
uio[2]	ce1_latch/MISO/DQ1	Connection to FLASH/SRAM via series resistor
uio[3]	SCLK	
uio[6]	-/DQ2	Must be enabled via uio MUX bits
uio[7]	-/DQ3	Must be enabled via uio MUX bits

This leaves uio[4]/uio[5] available for use as either UART or I2C.

Once the SPI/QSPI SRAM and optional FLASH have been chosen and connected, the Debug configuration registers must be programmed to indicate the nature of the external device(s). This is accompilished using Debug registers 0x12 - 0x19 and 0x1C. To programming the proper mode, follow these steps:

1. Program the LISA1, LISA2 and Debug CE Select registers (0x14, 0x15, 0x16) indicating which CE to use.

- 0x14, 0x15, 0x16: {6'h0, ce1_en, ce0_en} Active HIGH

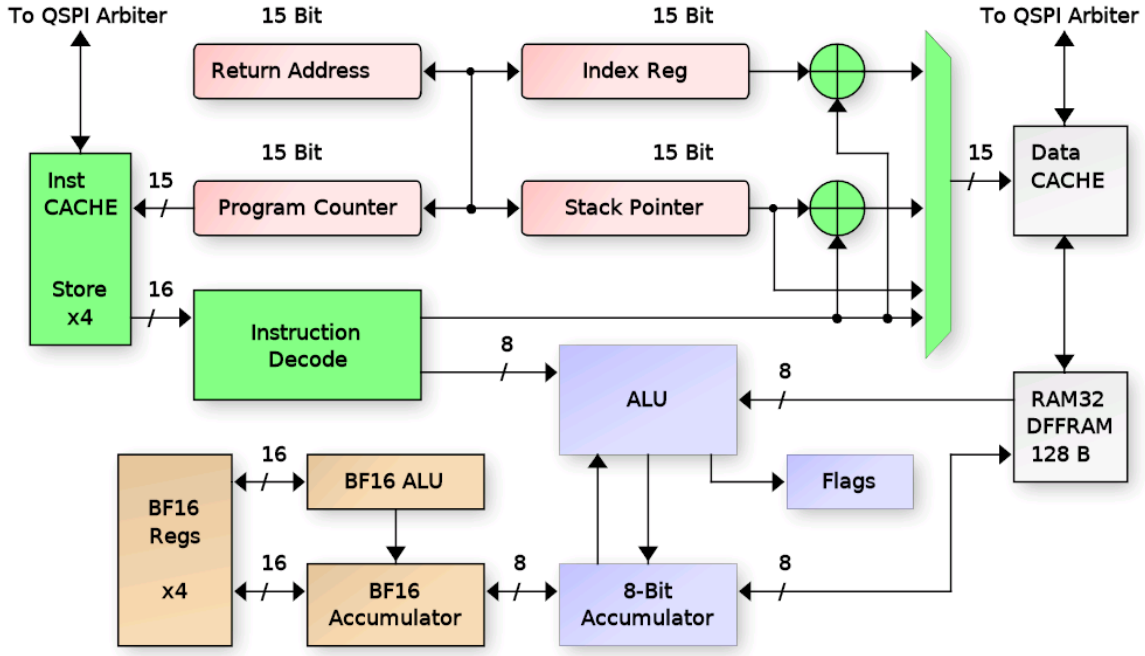
2. Program the LISA1 and LISA2 base addresses if they use the same SRAM:
   - 0x12: {LISA1_BASE, 8'h0} | {8'h0, PC}
   - 0x13: {LISA2_BASE, 8'h0} | {8'h0, DATA_ADDR}
3. Program the mode for each Chip Enable (bits active HIGH)
   - 0x17: {10'h0, is_16b[1:0], is_flash[1:0], is_quad[1:0]}
4. For Quad SPI, Special Mux Mode 3, or CE1, program the uio_mux mode:
   - 0x1C:
     - [7:6] = 2'h2: Normal QSPI DQ2 select
     - [7:6] = 2'h3: Special Mux Mode 3 (Latched CE)
     - [5:4] = 2'h2: Normal QSPI DQ3 select
     - [5:4] = 2'h3: Special Mux Mode 3
     - [1:0] = 2'h2: CE1 select on uio[4]
5. For RP2040, you might need to slow down the SPI clock / delay between successive CE activations:
   - 0x1E:
     - [3:0] spi_clk_div: Number of clocks SCLK HIGH and LOW
     - [10:4] ce_delay: Number clocks between CE activations
     - [12:11] spi_mode: Per-CE FALLING SCLK edge data update
6. Set the number of DUMMY ready required for each CE:
   - 0x18: {8'h0, dummy1[3:0], dummy0[3:0]
7. For QSPI FLASH, set the QSPI Write opcode (it is different for various Flashes):
   - 0x19: {8'h0, quad_write_cmd}

NOTE: For register 0x1E (SPI Clock Div and CE Delay), there is only a single register, meaning this register value applies to both CE outputs. Delaying the clock of one CE will delay both, and adding delay between CE activations does not keep track of which CE was activated. So if two CE outputs are used and a CE delay is programmed, it will enforce that delay even if a different CE is used. This setting is really in place for use when the RP2040 emulation is being used in a single CE SRAM mode only (i.e. you have no external PMOD with a real SRAM / FLASH chip. In the case of real chips on a PMOD, SCLK and CE delays (most likely) are not needed. The Tech Page on the Tiny Tapeout regarding RP2040 SPI SRAM emulation indicates a delay between CE activations is likely needed, so this setting is provided in case it is needed.

Architecture Details

Below is a simplified block diagram of the LISA processor core. It uses an 8-bit accumulator for most of its operations with the 2nd argument predominately coming from either immediate data in the instruction word or from a memory location addressed by either the Stack Pointer (SP) or Index Register (IX).

There are also instructions that work on the 15-bit registers PC, SP, IX and RA (Return Address). As well as floating point operations. These will be covered in the sections to follow.



Simplified LISA Processor Block Diagram

Addressing Modes Like most processors, LISA has a few different addressing modes to get data in and out of the core. These include the following:

Mode	Data	Description
Register	Rx[n -: 8]	Transfers between registers (ix, ra, facc, etc.).
Direct	inst[n:0]	N-bit data stored in the instruction word.
NextOp	(inst+1)[14:0]	Data stored in the NEXT instruction word.
Indirect	mem[inst[n:0]]	Address of the data is in the instruction word.
Periph	periph[inst[n:0]]	Accesses to the peripheral bus.
Indexed	mem[sp/ix+inst[n:0]]	The SP or IX register is added to a fixed offset.
Stack	mem[sp]	Stack pointer points to the data (push/pop).

The Control Registers To run meaninful programs, the Program Counter (PC) and Stack Pointer (SP) must be set to useful values for accessing program instructions and data. The PC is automatically reset to zero by rst_n, so that one is pretty much automatic. All programs start at address zero (plus any base address programmed by the Debug Controller). But as far as the LISA core is concerned, it knows nothing of base addresses and believes it is starting at address zero.

Next is to program the SP to point to a useful location in memory. The Stack is a place where C programs store their local variable values and also where we store the Return Address (RA) if we need to call nested routines, etc. The stack grows down, meaning it starts at a high RAM address and decrements as things are added to the stack. Therefore the SP should be programmed with an address in upper RAM. LISA supports different Data bus modes through it's CACHE controller, including CACHE disable where it can only access 128 bytes. But for this example, let's assume we have a full range of 32K SRAM available. The LISA ISA doesn't have an opcode for loading the SP directly. Instead it can load the IX register directly with a 15-bit value using NextOp addressing, and it supports "xchg" opcodes to exchange the IX register with either the SP or RA. So to load the SP, we would write:

```
Example:
  ldx     0x7FFF      // Load IX with value in next opcode
  xchg_sp             // Exchange IX with SP
```

The IX register can be programmed as needed to access other data within the Data Bus address range. This register is useful especially for accessing structures via a C pointer. The IX then becomes the value of the pointer to RAM, and Indexed addressing mode allows fixed offsets from that pointer (i.e. structure elements) to be accessed for read/write.

Loading the PC indirectly can be done using the "jmp ix" opcode which does the operation pc $<=$ ix. Loading ix from the pc directly is not supported, though this can be accomplished using a function call and opcodes to save RA (sra) and pop ix:

```
 Example:
   get_pc:
     sra          // Push RA to the stack (Save RA)
     pop_ix       // Pop IX from the stack
     ret          // Return. Upon return, IX is the same as PC
```

Conditional Flow Processing Program flow is controlled using flags (zero, carry, sign), arithemetic mode (amode) and condition flags (cond) to determine when program branches should occur. Specific opcode update the flags and condition registers based on results of the operation (AND, OR, IF, etc.). Then conditional branches are made using bz, bnz and if (and variants ifte "if-then-else" and iftt "if-then-then"). Also available are rc "Return if Carry" and rz "Return if Zero", though these are less useful in C programs as typically a routine uses local variables and the stack must be restored prior to return, mandating a branch to the function epilog to restore the stack and often the return address. Below is a list of the opcodes used for conditional program processing:

Legend for operations below:

- acc_val = inst[7:0]
- pc_jmp = inst[14:0]
- pc_rel = pc + sign_extend(inst[10:0])

Opcode	Operation	Encoding	Description
jal	pc <= pc_jmp ra <= pc	0aaa_aaaa_aaaa_aaaa	Jump And Link (call).
ret	pc <= ra	1000_1010_0xxx_xxxx	Return
reti	pc <= ra acc <= acc_val	1000_11xx_iiii_iiii	Return Immediate.
br	pc <= pc_rel	1011_0rrr_rrrr_rrrr	Branch Always
bz	pc <= pc_rel if zero=1	1011_1rrr_rrrr_rrrr	Branch if Zero.
bnz	pc <= pc_rel if zero=0	1010_1rrr_rrrr_rrrr	Branch if Not Zero.
rc	pc <= ra if carry=1	1000_1011_0xxx_xxxx	Return if Carry
rz	pc <= ra if zero=1	1000_1011_1xxx_xxxx	Return if Zero
call_ix	pc <= ix ra <= pc	1000_1010_100x_xxxx	Call indirect via IX
jump_ix	pc <= ix	1000_1010_101x_xxxx	Jump indirect via IX
if	cond <= ??	1010_0010_0000_0ccc	If. See below.
iftt	cond <= ??	1010_0010_0000_1ccc	If then-then. See below.
ifte	cond <= ??	1010_0010_0001_0ccc	If then-else. See below.

The IF Opcode The "if" opcode and it's variants "if-then-then" and "if-then-else" control program flow in a slightly different manner than the others. Instead of affecting

the value of the PC directly, they set the two condition bits "cond[1:0]" to indicate which (if any) of the two following opcodes should be executed. the cond[0] bit represents the next instruction and cond[1] represents the instruction following that. All three "if" forms take an argument that checks the current value of the FLAGS to set the condition bits. The argument is encoded as the lower three bits of the instruction word ard operate as shown in the following table:

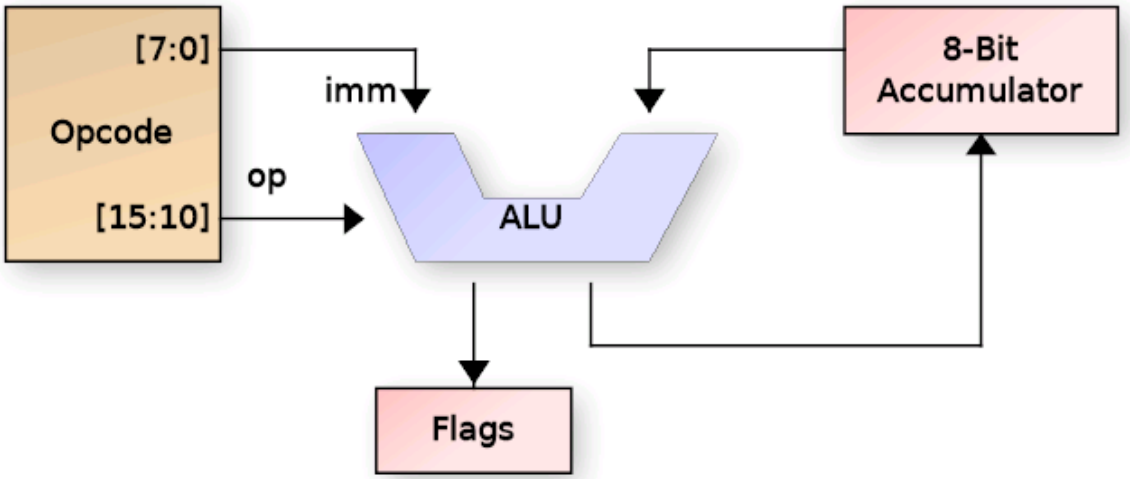
Condition	Test	Encoding	Description
EQ	zflag=1	3'h0	Execute if Equal
NE	zflag=0	3'h1	Execute if Not Equal
NC	cflag=0	3'h2	Execute if Not Carry
C	cflag=1	3'h3	Execute if Carry
GT	~cSigned & ~zflag	3'h4	Execute if Greater Than
LT	cSigned & ~zflag	3'h5	Execute if Less Than
GTE	~cSigned	zflag	3'h6
LTE	cSigned	zflag	3'h7

The "if" opcode will set cond[0] based on the condition above and the cond[1] bit to HIGH. It only affects the single instruction following the "if" opcode. The "iftt" opcode will set both cond[0] and cond[1] to the same value based on the condition above. It means "if true, execute the next two opcodes". And the "ifte" opcode will set cond[0] based on the condition above and cond[1] to the OPPOSITE value, meaning it will execute either the following instruction OR the one after that (then-else).

```
Example:
  ldi      0x41        // Load A immediate with ASCII 'A'
  cpi      0x42        // Compare A immediate with ASCII 'B'
  ifte     eq          // Test if the compare was "Equal"
  jal      L_equal     // Jump if equal
  jal      L_different // Jump if different
```

The above code will load the "jal L_equal" opcode but will not execute it since the compare was Not Equal. Then it will execute the "jal L_different" opcode. Note that if the compare were "ifte ne", it would call the L_equal function and then upon return would not execute the "L_different" opcode. This is because the cond[1] code is saved with the Return Address (RA) during the call and restored upon return. This means the FALSE cond[1] code would prevent the 2nd opcode from executing. As an opcode gets executed, the cond[1] value is shifted into the cond[0] location, and the cond[1] is loaded with 1'b1.

Direct Operations To do any useful work, the LISA core must be able to load and operate on data. This is done through the accumulator using the various addressing modes. The diagram below details the Direct addressing mode where data is stored directly in the opcode / instruction word:

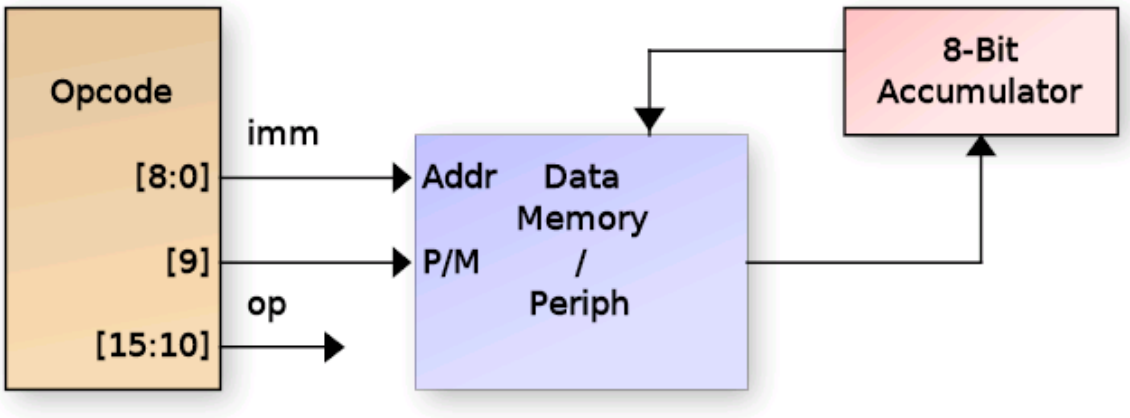


Accumulator Direct Operations Diagram

The instructions that use direct addressing are:

Opcode	Operation	Encoding	Description
adc	A <= A + imm + C	1001_00xx_iiii_iiii	ADD immediate with Carry
ads	SP <= SP + imm	1001_01ii_iiii_iiii	ADD SP + signed immediate
adx	IX <= IX + imm	1001_10ii_iiii_iiii	ADD IX + signed immediate
andi	A <= A & imm	1000_01xx_iiii_iiii	AND immediate with A
cpi	Z,C <= A >= imm	1010_01xx_iiii_iiii	Compare A >= immediate
cpi	Z,C <= A >= imm	1010_01xx_iiii_iiii	Compare A >= immediate

Accumulator Indirect Operations The Accumulator Indirect operations use immediate data in the instruction word to index indirectly into Data memory. That memory address is then used to load, store or both load and store (swap) data with the accumulator.



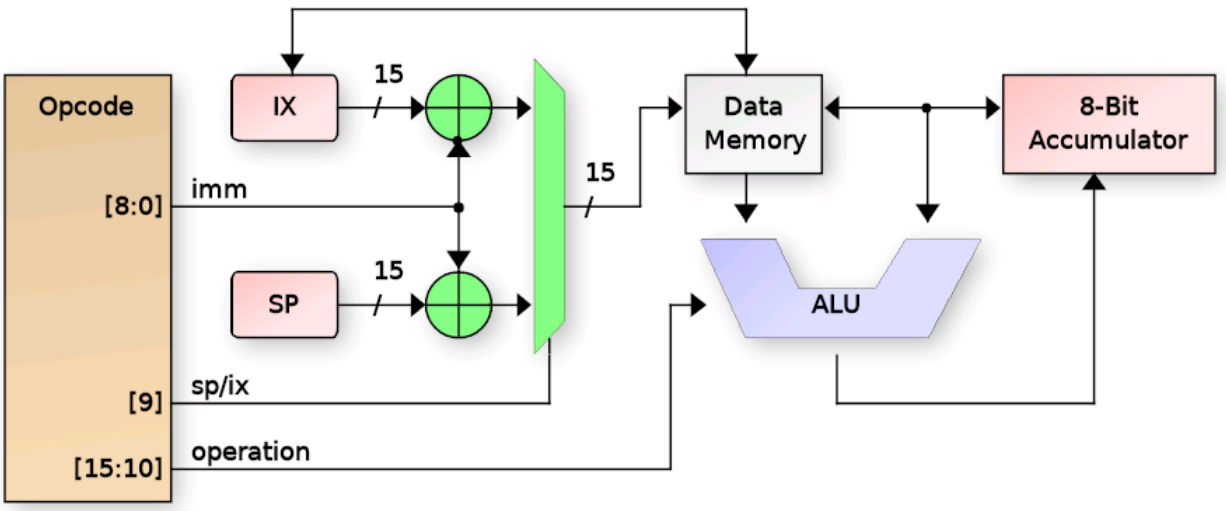
Accumulator Indirect Operations Diagram

Opcode	Operation	Encoding	Description
lda	A <= M[imm]	1111_01pi_iiii_iiii	Load A from Memory/Peripheral
sta	M[imm] <= A	1111_11pi_iiii_iiii	Store A to Memory/Peripheral
swapi	A <= M[imm] M[imm] <= A	1101_11pi_iiii_iiii	Swap Memory/Peripheral with A

- p = Select Peripheral (1'b1) or RAM (1'b0)
- iiii = Immediate data

Indexed Operations Indexed operations use either the IX or SP register plus a fixed offset from the immediate field of the opcode. The selection to use IX vs SP is also from the opcode[9] bit. The immediate field is not sign extended, so only positive direction indexing is supported. This was selected because this mode is typically used

to access either local variables (when using SP) or C struct members (when using IX), and in both cases, negative index offsets aren't very useful. The following is a diagram of indexed addressing:



Indirect Addressing Diagram

Opcode	Operation	Encoding	Description
add	A <= A+ M[ind]	1100_00si_iiii_iiii	ADD index memory to A
and	A <= A & M[ind]	1101_00si_iiii_iiii	AND A with index memory
cmp	A >= M[ind]?	1110_10si_iiii_iiii	Compare A with index memory
dcx	M[ind] -= 1	1001_11si_iiii_iiii	Decrement the value at index memory
inx	M[ind] += 1	1110_01si_iiii_iiii	Increment the value at index memory
ldax	A <= M[ind]	1111_00si_iiii_iiii	Load A from index memory
ldxx	IX <= M[SP+imm]	1100_110i_iiii_iiii	Load IX from memory at SP+imm
mul	A <= A*M[ind]L	1100_10si_iiii_iiii	Multiply index memory * A, keep LSB
mulu	A <= A*M[ind]H	1000_01si_iiii_iiii	Multiply index memory * A, keep MSB
or	A <= A	M[ind]	1101_10si_iiii_iiii
stax	M[ind] <= A	1111_10si_iiii_iiii	Store A to index memory
stxx	M[SP+imm] <= IX	1100_111i_iiii_iiii	Save IX to memory at SP+imm
sub	A <= A-M[ind]	1100_10si_iiii_iiii	SUBtract index memory from A
swap	A <= M[ind]	1110_11si_iiii_iiii	Swap A with index memory
	M[ind] <= A		
xor	A <= A ^ M[ind]	1110_00si_iiii_iiii	XOR A with index memory

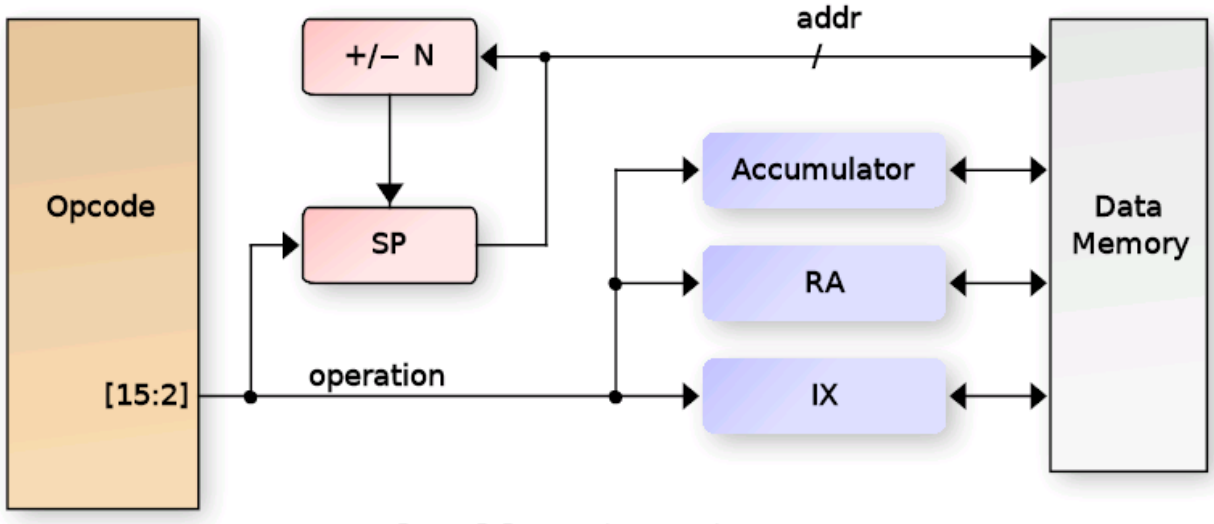
Legend for table above:

- ind = IX or SP + immediate
- s = Select IX (zero) or SP (one)
- iiii = Immediate data

The Zero and Carry flags are updated for most of the above operations. The Carry flag is only updated for math operations where a Carry / Borrow could occur.

Carry	Zero
adc	add and
add	or xor
sub	cmp sub
cmp	dcx inx
dcx	swap ldax
inx	mul mulu

Stack Operations Stack operations use the current value of the SP register to PUSH and POP items to the stack in opcode. As items are PUSHed to the stack, the SP is decremented after each byte, and as they are POPed, the SP is incremented prior to reading from RAM.



Stack Addressing Diagram

Opcode	Operation	Encoding	Description
lra	RA <= M[SP+1] SP += 2	1010_0001_0110_01xx	Load {cond,RA} from stack

Opcode	Operation	Encoding	Description
sra	M[SP] <= RA SP -= 2	1010_0001_0110_00xx	Save {cond,RA} to stack
push_ix	M[SP] <= IX SP -= 2	1010_0001_0110_10xx	Save IX to stack
pop_ix	IX <= M[SP+1] SP += 2	1010_0001_0110_11xx	Load IX from stack
push_a	M[SP] <= A SP -= 1	1010_0000_100x_xxxx	Save A to stack
pop_a	A <= M[SP+1] SP += 1	1010_0000_110x_xxxx	Load A from stack

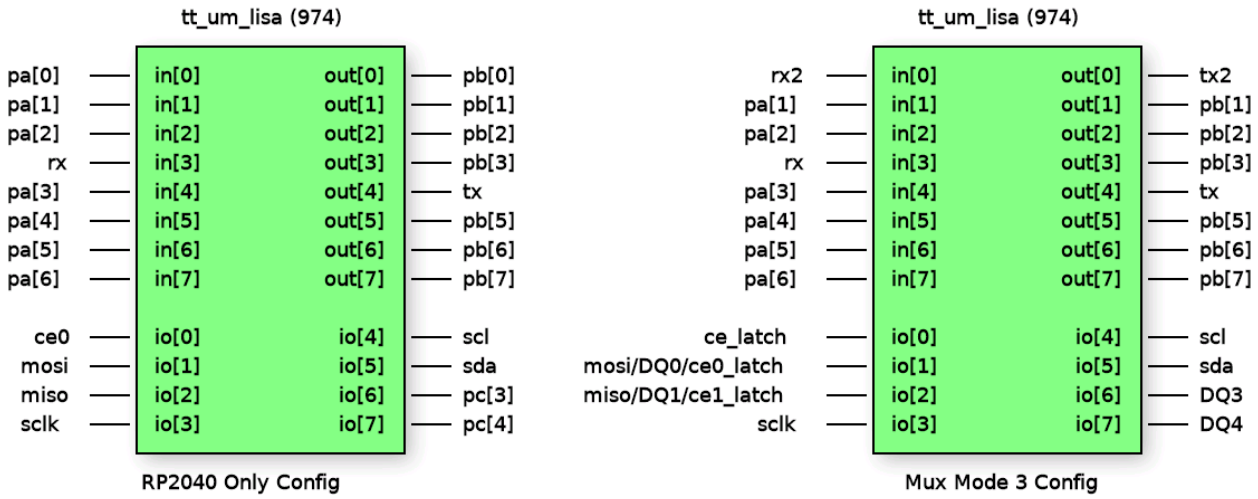
How to test

You will need to download and compile the C-based assembler, linker and C compiler I wrote (will make available) Also need to download the Python based debugger.

- Assembler is fully functional
  - Includes limited libraries for crt0, signed int compare, math, etc.
  - Libraries are still a work in progress
- Linker is fully functional
- C compiler is somewhat functional (no float support at the moment) but has *many* bugs in the generated code and is still a work in progress.
- Python debugger can erase/program the FLASH, program SPI SRAM, start/stop the LISA core, read SRAM and registers.

Legend for Pinout

- pa: LISA GPIO PortA Input
- pb: LISA GPIO PortB Output
- b_div: Debug UART baud divisor sampled at reset
- b_set: Debug UART baud divisor enable (HIGH) sampled at reset
- baud_clk: 16x Baud Rate clock used for Debug UART baud rate generator
- ce_latch: Latch enable for Special Mux Mode 3 as describe above
- ce0_latch: CE0 output during Special Mux Mode 3
- ce1_latch: CE1 output during Special Mux Mode 3
- DQ1/2/3/4: QUAD SPI bidirection data I/O
- pc_io: LISA GPIO Port C I/O (direction controllable by LISA)



Pinout

#	Input	Output	Bidirectional
0	pa_in[0]/baud_div[0]	pb_out[0]	ce0/ce_latch
1	pa_in[1]/baud_div[1]	pb_out[1]	mosi/dq1/ce0
2	pa_in[2]/baud_div[2]	pb_out[2]	miso/dq2/ce1
3	pa_in[3]/baud_div[3]/rx	pb_out[3]	sclk
4	pa_in[4]/baud_div[4]	pb_out[4]/tx	rx /pc_io[0]/scl/sda
5	pa_in[5]/baud_div[5]	pb_out[5]	tx /pc_io[1]/sda/scl
6	pa_in[6]/baud_div[6]	pb_out[6]	scl /pc_io[2]/dq2/rx
7	pa_in[7]/baud_set	pb_out[7]/baud_clk	sda/pc_io[3]/dq3

Temperature Sensor NG [975]

- Author: Harald Pretl
- Description: Temperature sensor synthesized from standard cells
- GitHub repository
- HDL project
- Mux address: 975
- Extra docs
- Clock: 10000000 Hz

How it works

There will be a better explanation in the future. In short, it measures the on-chip temperature, and puts out the result.

How to test

Simply turn it on, and see the result. IO usage documented in the `info.yml`.

For direct outside control (bypassing the internal measurement state machine), use the following settings:

- Set `uio_in[3:0] = 0b1111` (enable debug mode 15).
- Set the DAC by using `ui_in[5:0]` (6b direct control of `tempsens_dat[5:0]`).
- `ui_in[6]` is connected to `tempsens_enable`.
- `ui_in[7]` is connected to `tempsens_prechrgn`.
- The output of the temperature sensor `tempsens_tempdelay` is connected to `uio_out[4]`.
- Use the `clk` input to synchronize the temperature output of falling edge.

External hardware

Requires a logic analyzer or similar to inspect the digital outputs.

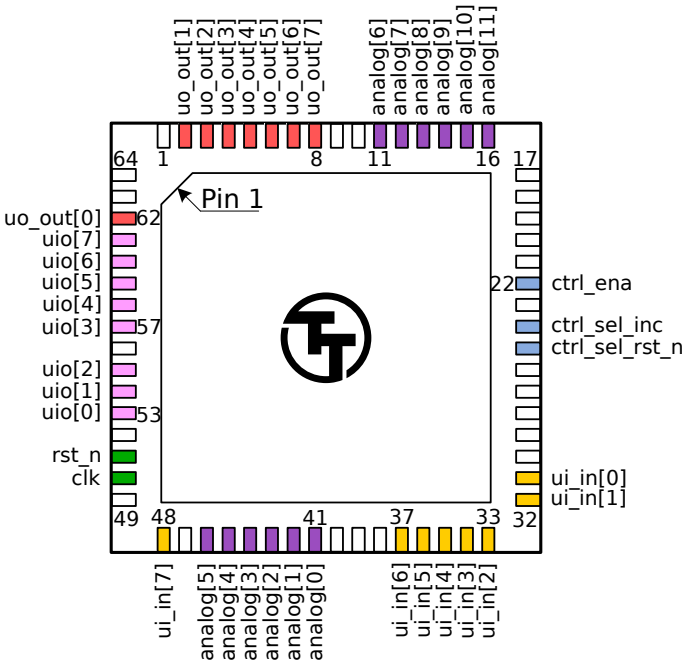
#	Input	Output	Bidirectional

Pinout

#	Input	Output	Bidirectional
0	DAC code [0]	out[0] or out[8] or out[16]	debug sel [0]
1	DAC code [1]	out[1] or out[9] or out[17]	debug sel [1]
2	DAC code [2]	out[2] or out[10] or out[18]	debug sel [2]
3	DAC code [3]	out[3] or out[11] or out[19]	debug sel [3]
4	DAC code [4]	out[4] or out[12]	debug out [0]
5	DAC code [5]	out[5] or out[13]	debug out [1]
6	output selection [0]	out[6] or out[14]	debug out [2]
7	output selection [1]	out[7] or out[15]	debug out [3]

Pinout

The chip is packaged in a 64-pin QFN package. The pinout is shown below.



Bottom View

Figure 74: Pinout

Note: you will receive the chip mounted on a breakout board. The pinout is provided for advanced users, as most users will not need to solder the chip directly.

The Tiny Tapeout Multiplexer

Overview

The Tiny Tapeout Multiplexer distributes a single set of user IOs to multiple user designs. It is the backbone of the Tiny Tapeout chip.

It has the following features:

- 10 dedicated inputs
- 8 dedicated outputs
- 8 bidirectional IOs
- Supports up to 512 user designs (32 mux units, each with up to 16 designs)
- Designs can have different sizes. The basic unit is a called a tile, and each design can occupy up to 16 tiles.

Operation

The multiplexer consists of three main units:

1. The controller - used to set the address of the active design
2. The spine - a bus that connects the controller with all the mux units
3. Mux units - connect the spine to individual user designs

The Controller

The mux controller has 3 inputs lines:

Input	Description
`ena`	Sent as-is (buffered) to the downstream mux units
`sel_rst_n`	Resets the internal address counter to 0 (active low)
`sel_inc`	Increments the internal address counter by 1

It outputs the address of the currently selected design on the `si_sel` port of the spine (see below).

For instance, to select the design at address 12, you need to pulse `sel_rst_n` low, and then pulse `sel_inc` 12 times:

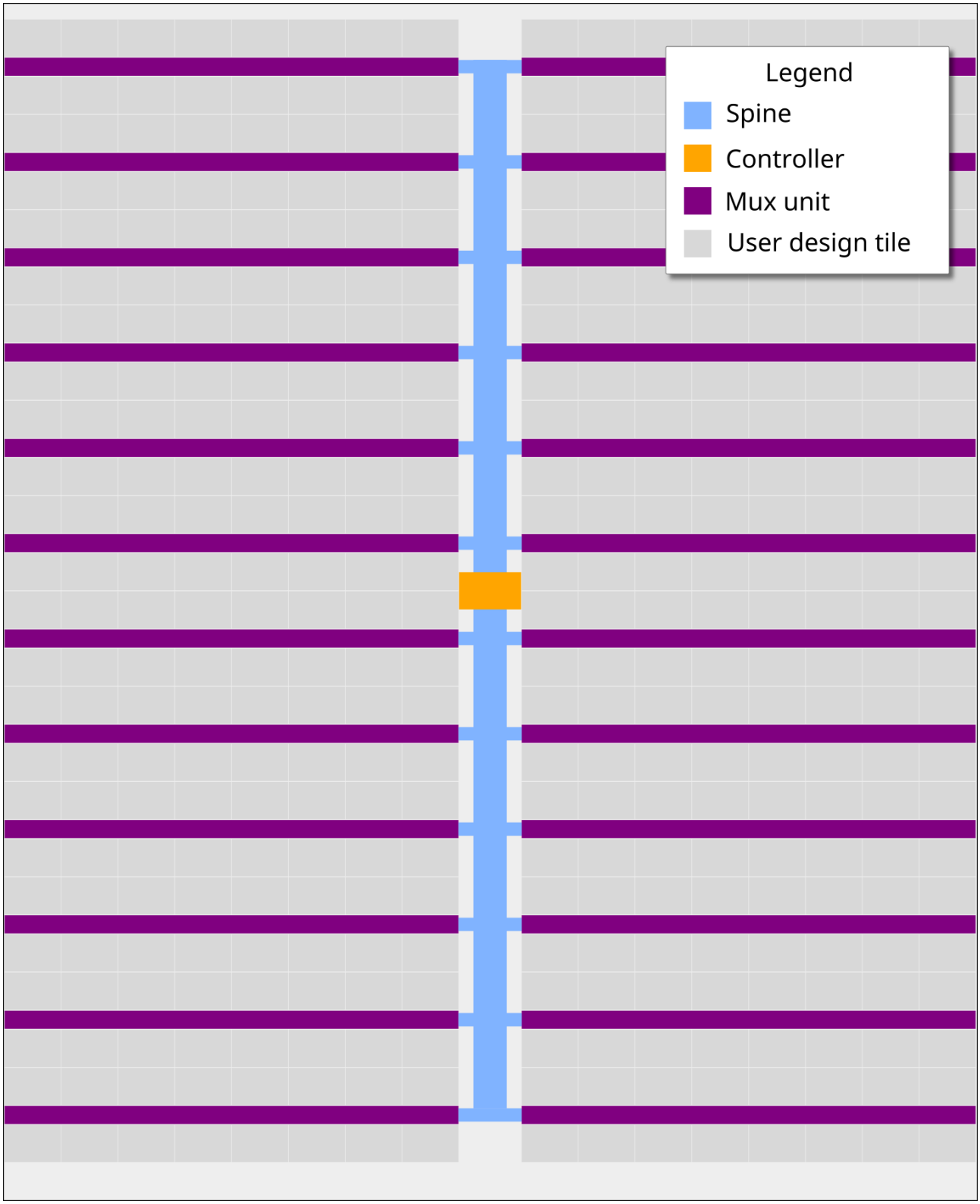


Figure 75: Mux Diagram

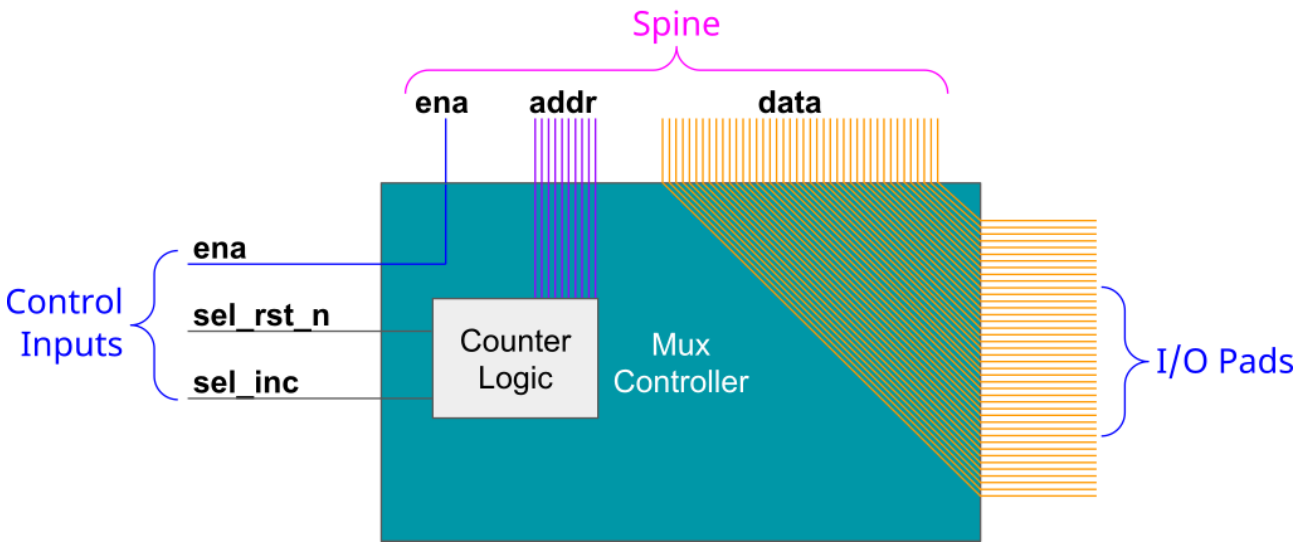


Figure 76: Mux Controller Diagram

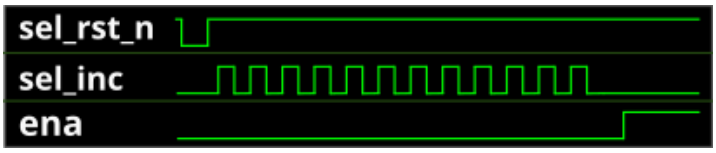


Figure 77: Mux signals for activating the design at address 12

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	QFN64 pin
0	Input	ui_in[0]	31
1	Input	ui_in[1]	32
2	Input	ui_in[2]	33
3	Input	ui_in[3]	34
4	Input	ui_in[4]	35

mprj_io pin	Function	Signal	QFN64 pin
5	Input	ui_in[5]	36
6	Input	ui_in[6]	37
7	Analog	analog[0]	41
8	Analog	analog[1]	42
9	Analog	analog[2]	43
10	Analog	analog[3]	44
11	Analog	analog[4]	45
12	Analog	analog[5]	46
13	Input	ui_in[7]	48
14	Input	clk †	50
15	Input	rst_n †	51
16	Bidirectional	uio[0]	53
17	Bidirectional	uio[1]	54
18	Bidirectional	uio[2]	55
19	Bidirectional	uio[3]	57
20	Bidirectional	uio[4]	58
21	Bidirectional	uio[5]	59
22	Bidirectional	uio[6]	60
23	Bidirectional	uio[7]	61
24	Output	uo_out[0]	62
25	Output	uo_out[1]	2
26	Output	uo_out[2]	3
27	Output	uo_out[3]	4
28	Output	uo_out[4]	5
29	Output	uo_out[5]	6
30	Output	uo_out[6]	7
31	Output	uo_out[7]	8
32	Analog	analog[6]	11
33	Analog	analog[7]	12
34	Analog	analog[8]	13
35	Analog	analog[9]	14
36	Analog	analog[10]	15
37	Analog	analog[11]	16
38	Mux Control	ctrl_ena	22
39	Mux Control	ctrl_sel_inc	24
40	Mux Control	ctrl_sel_rst_n	25
41	Reserved	(none)	26
42	Reserved	(none)	27
43	Reserved	(none)	28

† Internally, there's no difference between `clk`, `rst_n`, and `ui_in` pins. They are all just bits in the `pad_ui_in` bus. However, we use different names to make it easier to understand the purpose of each signal.

Sponsored by

Team

Tiny Tapeout would not be possible without a lot of people helping. We would especially like to thank:

- Uri Shaked for wokwi development and lots more
- Patrick Deegan for PCBs, software, documentation and lots more
- Sylvain Munaut for help with scan chain improvements
- Mike Thompson and Mitch Bailey for verification expertise
- Tim Edwards and Harald Pretl for ASIC expertise
- Jix for formal verification support
- Proppy for help with GitHub actions
- Maximo Balestrini for all the amazing renders and the interactive GDS viewer
- James Rosenthal for coming up with digital design examples
- All the people who took part in TinyTapeout 01 and volunteered time to improve docs and test the flow
- The team at YosysHQ and all the other open source EDA tool makers
- Jeff and the Efabless Team for running the shuttles and providing OpenLane and sponsorship
- Tim Ansell and Google for supporting the open source silicon movement
- Zero to ASIC course community for all your support
- Jeremy Birch for help with STA