

Tiny Tapeout IHP 25a Datasheet

Project Repository

<https://github.com/TinyTapeout/tinytapeout-ihp-25a>

March 31, 2025

Contents

Chip renders	11
Full chip render	11
Top Metal 1/2	12
Logic density view	13
Projects	14
Chip ROM [0]	14
Tiny Tapeout Factory Test 1	16
Verilog ring oscillator V3 2	18
Matmul System [3]	19
4-Bit Toy CPU [4]	20
Hybrid_Adder_8bit [5]	22
Dynamic Threshold Leaky Integrate-and-Fire [6]	24
8-bit Carry Look-Ahead Adder [7]	25
ternary, E1M0, E2M0 decoders [8]	28
RISC-V Mini [9]	29
eksdee [10]	32
8-bit carry-skip [11]	33
Cgates [12]	35
T3 (Tiny Ternary Tapeout) [13]	37
Classic 8-bit era Programmable Sound Generator SN76489 [14]	41
3 Neuron ALIF [15]	49
Giant Ring Oscillator (3853 inverters) [16]	51
STDP Circuit [17]	53
TwoChannelSquareWaveGenerator [18]	55
instrumented_ring_oscillator [19]	57
Linear Timecode (LTC) generator [32]	59
Tiny Shader [34]	61
Sine Synth [36]	66
DRUM [38]	68
Tiny Hash Table [40]	70
Asynchronous FIFO [42]	72
Synchronous FIFO [44]	74
Pulse Width Modulation [46]	75
DaDDS [48]	78
Simple shift Reg [50]	80
2-bit 2x2 Matrix Multiplier [64]	81
8b10b decoder and multiplier [65]	83
VGA Tiny Logo (1 tile) [66]	85
Test Design 1 [67]	86
A simple leaky integrate and fire neuron [68]	88
Decimation Filter for Incremental and Regular Delta-Sigma Modulators [69]	89

Leaky Neuron Network [70]	92
adder-accumulator [71]	94
Neuromorphic Hardware for SNN LSTM [72]	99
ECE 298A 8-Bit CPU Control Block [73]	101
RISCV Processor Design [74]	105
LFSR Encrypter [75]	107
RISCV Processor Design [76]	108
SkyKing Demo [77]	110
tiny cipher 4 bit key [78]	111
Two LIF Neurons with STDP Learning [79]	112
Tutorial: Simple LIF Neuron [80]	115
7-Segment Byte Display [81]	116
RGB Mixer demo [82]	118
Forward Pass Network for Simple ANN [83]	119
Priority-encoder [96]	121
UltraTiny-CPU [98]	122
Priority-Encoded Arbiter [100]	124
ALU in verilog [102]	126
Overengineered Checkers [104]	127
toni_clk_gen [106]	129
spi_pwm [108]	130
BINCounterAndGates [110]	137
tt09-pettit-wokproc-trainer [112]	140
Duffy [114]	145
pulse_add [128]	146
nyan [130]	147
Brailliance [132]	148
Adder with Flow Control [134]	149
i2c peripherals: leading zero count and fnv-1a hash [136]	152
Rotary Encoder WS2812B Control [138]	154
Alarm Clock [140]	156
TSAL_TT [142]	158
Divided Ring Oscillator [144]	160
HACK CPU [146]	163
simon_cipher [161]	167
Wirecube [163]	169
TT08 Pachelbel's Canon demo [165]	170
Neural Net ASIC [166]	171
Sequential Shadows [TT08 demo competition] [167]	172
CYCLIPSONIC [171]	178
TDC with SPI [175]	179
Atari 2600 [178]	180

SPI FPU [179]	181
MAC [192]	184
DPMU [194]	187
7 Segment Decode [196]	190
PS2 Decoder [198]	194
Super Mario Tune on A Piezo Speaker [200]	196
AES Inverse S-box [202]	199
TT08 - experiments with latch-based shift registers [204]	200
Obstacle Detection [206]	202
resfuzzy [208]	204
CEJMU Beers and Adders [210]	206
RGBW Color Processor [225]	208
Stochastic Multiplier, Adder and Self-Multiplier [227]	214
DL float MAC [229]	219
schoolRISCV CPU with Fibonacci program [231]	222
Rounding error [233]	224
SIC-1 8-bit SUBLEQ Single Instruction Computer [234]	226
Sea Battle [235]	229
Comm_IC [237]	231
16 Mic Beamformer [239]	233
PDM Pitch Filter [241]	234
Zoom Zoom [242]	236
PDM Correlator [243]	241
SPI Logic Analyzer with Charlieplexed Display [258]	242
Find The Damn Issue [259]	244
Sequential Shadows Deluxe [TT08 demo competition] [262]	246
DDC [266]	249
mulmul [270]	250
Warp [274]	252
Supermic [275]	254
DPM_Unit [289]	255
Generate VGA output for Color Blindness Test [291]	258
4-bit CLA [293]	259
SkyKing Demo [295]	260
Flame demo [297]	261
Metaballs [299]	263
Simple Stopwatch [301]	264
PWM generator [303]	266
DMTD [305]	268
I2S to PWM [307]	269
Basys 3 Over UART Link [322]	270
ITS-RISCV [326]	272

Zilog Z80 [330]	275
2048 sliding tile puzzle game (VGA) [334]	279
ChatGPT-generated Spiking Neural Network with Delays [335]	281
Space Invaders ASIC [338]	283
Demo by a1k0n [339]	284
Clock Divider [353]	286
TinyFPGA resubmit for TT08 [355]	287
Dummy Counter [357]	288
RGB Mixer [359]	289
32x8 LED Matrix Animation [361]	290
TT09Ball VGA Screensaver [363]	292
Color Bars [365]	294
Hardware UTF Encoder/Decoder [367]	296
Styler [369]	301
VGA Timing Experiments [371]	306
JTAG TAP [385]	309
7-segment with LFSR [387]	311
TT10 HPDL 1414 Uart [389]	314
KCH CD101 Saw Synth [391]	317
tt10_zhouzhouthezhou_adder [393]	319
Asynchronous Locking Unit [395]	320
XOR Cipher [397]	321
Verilog based clock to 7-segment counter [399]	323
TT10_Luke_Clock [401]	324
SSMCI [403]	328
Configurable Logic Block [416]	330
Gamepad Pmod Demo [417]	332
4-bit up/down binary counter [418]	334
6Digit7SegClock [419]	336
Team 17's 8 bit DAC [420]	338
MAC Operation [421]	339
Tiny Registers [422]	340
Xor-Logic [423]	345
Leaky Integrate Fire Neuron [424]	346
Simon Says memory game [425]	348
Tiny Tapeout Group 7 Lab D [426]	351
SPI 7-segment display [427]	353
8-bit-CARRY_SKIP [428]	355
AtomNPU [429]	357
Semana UCU Verilog [430]	361
Enigma - 52-bit Key Length [431]	367
Frequency Encoder and Decoder [432]	374

synth_simple [433]	375
carry skip adder [434]	376
VGA clock [435]	378
Crossyroad [449]	380
zc-sushi-demo [451]	381
kch cd101 [453]	382
SimpleSPIdev [455]	384
RNG_test [457]	385
15bit GCD [459]	386
XY Spacewar [461]	388
16-bit Logarithmic Approximate Floating Point Multiplier [463]	389
TT_spiralPattern [465]	392
ledtest [467]	393
I2C and SPI [480]	394
VGA Screensaver with Tiny Tapeout Logo [481]	395
Perceptron Neuron [482]	397
SPI test [483]	398
Histogramming [484]	399
Huffmann_Coder [485]	402
RLE Video Player [486]	403
Vedic multiplier [487]	406
8-Bit CPU [488]	407
Tiny piano [489]	422
carry_select [490]	424
Asynchronous I2C Registerfile Interface [491]	426
test_friday2 [492]	427
Tappu [493]	431
Perceptron [494]	432
mp_LIF_neuron [495]	433
Hopfield Network with Izhikevich-type RS and FS Neurons [496]	434
digital LIF Neuron [497]	435
Tinysynth [498]	436
Hero on Tape [499]	437
16 Bit Izhikevich Neuron [512]	438
dff_mem [514]	441
Verilog ring oscillator V2 [516]	444
Basic model for Systolic array implementation of LIF [518]	446
Leaky integrate and fire spiking neural network [520]	448
tinydsp-lol [522]	449
Shifter [524]	450
LRC - Longitudinal Redundancy Check generator [526]	451
Workshop demo [528]	452

A Tale of Two NCOs [530]	453
Wokwi Group #7 [544]	455
Wokwi Group #6 [546]	456
Wokwi Group #5 [548]	457
Wokwi Group #4 [550]	458
Wokwi Group #3 [552]	459
Wokwi Group #2 [554]	460
Wokwi Group #1 [556]	461
Will It NAND? [558]	462
sphereinabox hello [560]	463
L display [562]	464
7-Segment Digital Desk Clock [576]	466
Basic Perceptron + ReLU [578]	468
Basic Matrix-Vector Multiplication [580]	469
8 bit MAC Unit [582]	470
Programmable PWM Generator [584]	472
Verilog test project [586]	474
Basic LIF Neuron [588]	475
Integrate-and-Fire Neuron Circuit [590]	477
Michaels Tiny Tapeout ALU [592]	479
8-bit CBILBO [594]	480
Wokwi Group #8 [608]	482
Wokwi Group #9 [610]	483
Wokwi Group #10 [612]	484
Wokwi Group #11 [614]	485
Wokwi Group #12 [616]	486
triggerer [618]	487
Wokwi Group #13 [620]	489
Multiplier Group #1 [622]	490
Multiplier Group #2 [624]	491
Multiplier Group #3 [626]	492
Ternary 128-element Dot Product [640]	493
GUS16 CPU [642]	494
Warp [644]	496
VGA Drop (audio/visual demo) [646]	498
Classic 8-bit era Programmable Sound Generator AY-3-8913 [648]	499
SoCET UART with FIFO buffers [650]	507
Simon's Caterpillar [652]	509
Stochastic Integrator [654]	511
E2M0 x INT8 Systolic Array [656]	513
VGA Nyan Cat [658]	515
Collatz conjecture brute-forcer [673]	517

APA102 to WS2812 Translator [675]	519
pio-ram-emulator example: Julia fractal [677]	521
Tiny Neural Network Accelerator [678]	524
Fuzzy Search Engine [679]	527
VGA Pride [681]	534
donut [683]	538
UART [685]	539
Why not? [687]	541
FSK Modem +HDLC +UART (PoC) [689]	542
Spectrogram extractor, 2 channels [690]	546
Bouncy Capsule [691]	549
TinyTapeout Minimal Branch Predictor [704]	550
Moody-mimosa [706]	553
Classic 8-bit era Programmable Sound Generator AY-3-8913 [708]	558
Orion Iron Ion [TT10 demo competition] [710]	566
My Project [712]	575
simple-viii [714]	576
ttUART [716]	577
Bitty [718]	578
IHP VGA demo [720]	590
UW ASIC - Optimized Dino [722]	592
PID Controller [737]	593
Frequency Counter SSD1306 OLED [739]	595
Tiny 1-bit AM Radio [741]	597
FIREngine [743]	600
znah_vga_ca [745]	602
TRNG [746]	603
CORA-16 [747]	605
T3 (Tiny Ternary Tapeout) CSA [749]	609
Basic Oszilloscope and Signal Generator [751]	613
1bit_am_sdr [752]	616
15 channels emission counter [753]	619
VGA Pong with NES Controllers [754]	622
Tiny RAM DFF 2r1w [755]	624
Sprite Bouncer with Looping Background Options [768]	628
Glyph Mode [769]	629
VGA Scroller [771]	631
DDR throughput and flop aperature test [773]	632
Wildcat RISC-V [774]	634
Calculator [775]	635
Crispy VGA [777]	636
asic design is my passion [779]	638

TinyQV Risc-V SoC [780]	639
Dice [781]	643
4-bit minicomputer ALU [783]	644
RGB Mixer demo5 [785]	645
AlphaOneSoC [786]	646
Asynchronous Multiplier [787]	647
Hamming Code (7,4) [801]	650
Space Detective Maze Explorer [803]	655
Senol Gulgonul tt09 [805]	657
4 bit ALU [807]	658
Elevator Design [809]	660
LED Bitserial Cipher [811]	661
freqSweep [813]	664
Simple PWM Module [815]	669
INTERCAL ALU [817]	670
Universal Binary to Segment Decoder [819]	674
RO [833]	683
CMOS design of 4-bit Signed Adder Subtractor [835]	685
LaRVa CPU [836]	687
Patater Demo Kit Wagging Rainbow on a Chip [837]	689
DemoSiine [839]	694
"SQUARE-1": VGA/audio demo [840]	702
Munch [841]	706
cfib Demoscene Entry [843]	709
VGA donut [844]	711
4-bit ALU [845]	713
Morse Code Keyer [847]	716
VGA Mandelbrot [848]	719
nVious Graphics [849]	722
TinyMandelbrot [850]	724
8-Bit Calculator [851]	726
tiny-tapeout-8bit-GPTPrefixCircuit [865]	727
LIF on a Ring Topology [867]	730
Delta-Sigma ADC Decimation Filter [869]	732
an Ifsr with synaptic neurons (excitatory or inhibitory) [871]	733
Perceptron [873]	735
Matmul System [875]	736
Verilog ring oscillator [877]	737
Delta RNN and Leaky Integrate-and-Fire Nueron Circuit [879]	738
Generador PWM multiproposito con frecuencia y ciclo de trabajo modulable [881]	739
Linear Feedback Shift Register [883]	741

Pinout	742
The Tiny Tapeout Multiplexer	743
Overview	743
Operation	743
Pinout	747
Funding	749
Team	749

Chip renders

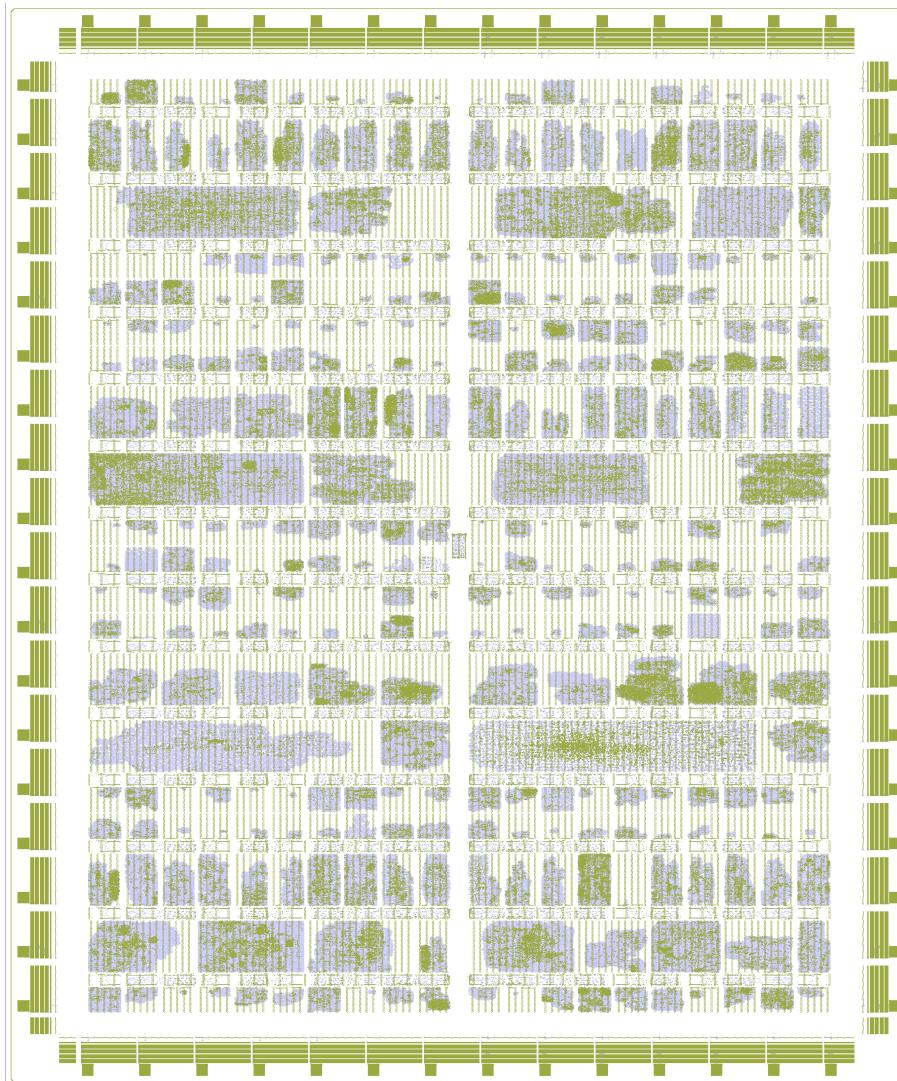
Full chip render



Top Metal 1/2



Logic density view



Projects

Chip ROM [0]

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

How it works

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

The ROM layout The ROM layout is as follows:

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

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

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

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

Here is a complete example of a chip descriptor:

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

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

Reading the ROM There are two ways to address ROM, depending on the value of the `rst_n` pin:

1. When `rst_n` is high: Set the `ui_in` pins to the desired address.
2. When `rst_n` is low: Toggle the `clk` pin to read the ROM contents sequentially, starting from address 0.

In both cases, the ROM data for the selected address will be available on the `uo_out` pins, one byte at a time.

How to test

The first 16 bytes of the ROM are 7-segment encoded and contain the shuttle name and commit hash. You can dump them by holding `rst_n` low and toggling the `clk` pin, and observing the on-board 7-segment display.

Alternatively, you can keep `rst_n` high and set the `ui_in` pins to the desired address using the first four on-board DIP switches, while observing the on-board 7-segment display.

Pinout

#	Input	Output	Bidirectional
0	addr[0]	data[0]	
1	addr1	data1	
2	addr2	data2	
3	addr[3]	data[3]	
4	addr[4]	data[4]	
5	addr[5]	data[5]	
6	addr[6]	data[6]	
7	addr[7]	data[7]	

Tiny Tapeout Factory Test 1

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

How it works

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

It has three modes of operation:

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

The following table summarizes the modes:

<code>rst_n</code>	<code>sel</code>	Mode	<code>uo_out</code> value	<code>ui_o</code> pins
0	X	Input mirror	<code>ui_in</code>	High-Z
1	0	Bidirectional mirror	<code>ui_o_in</code>	High-Z
1	1	Counter	counter	counter

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

How to test

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

Pinout

#	Input	Output	Bidirectional
0	sel / in_a[0]	output[0] / counter[0]	in_b[0] / counter[0]
1	in_a1	output1 / counter1	in_b1 / counter1
2	in_a2	output2 / counter2	in_b2 / counter2
3	in_a[3]	output[3] / counter[3]	in_b[3] / counter[3]
4	in_a[4]	output[4] / counter[4]	in_b[4] / counter[4]
5	in_a[5]	output[5] / counter[5]	in_b[5] / counter[5]
6	in_a[6]	output[6] / counter[6]	in_b[6] / counter[6]
7	in_a[7]	output[7] / counter[7]	in_b[7] / counter[7]

Verilog ring oscillator V3 2

- Author: algofoogle (Anton Maurovic)
- Description: sky130 inv_2 ring oscillator with externally-selectable length
- GitHub repository
- HDL project
- Mux address: 2
- Extra docs
- Clock: 0 Hz

What is this?

See tt09-ring-osc and tt09-ring-osc2 for my other ring oscillator experiments on TT09.

This one has a configurable ring oscillator; the feedback can be tapped at different parts of the chain.

This uses Verilog to instantiate a ring of (an odd number of) sky130_fd_sc_hd__inv_2 cells – **UPDATE:** Actually, since this is targeting IHP instead, there is a polyfill that somebody else wrote to map sky130 cells to generic cells (that OpenLane will then map to IHP cells).

Pinout

#	Input	Output	Bidirectional
0	tap[0]	out[0]	
1	tap1	out1	
2	tap2	out2	
3		out[3]	
4		out[4]	
5		out[5]	
6		out[6]	
7		out[7]	

Matmul System [3]

- Author: Abarajithan
- Description: Matmul System
- GitHub repository
- HDL project
- Mux address: 3
- Extra docs
- Clock: 0 Hz

How it works

This is a simple system that performs matrix-vector multiplication. The matrix $K[R,C]$ and vector $X[R]$ is sent from outside through UART. They are decoded by a UART RX module, and sent into the matrix-vector multiplication core as AXI-Stream. The core performs the multiplication and outputs the result as AXI-Stream. The result is then packed into UART format by the UART TX module and sent outside.

How to test

```
iverilog -g2012 -o compiled src/mvm_uart_system.v src/uart_rx.v src/uart_
```

External hardware

None

Pinout

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

4-Bit Toy CPU [4]

- Author: Stefan Wallentowitz
- Description: This is a simple 4-bit CPU from a popular German textbook
- GitHub repository
- HDL project
- Mux address: 4
- Extra docs
- Clock: 0 Hz

How it works

This is a 4 bit Toy CPU from a popular German textbook (Hoffmann, “Technische Informatik”, <https://www.dirkwhoffmann.de/TI/>). It is extremely simple and not extremely useful but a useful CPU to transition from digital design to microprocessors in a fundamental way.

The CPU is based on a 4 bit accumulator. It has 4 bit instructions with 4 bit operands. The memory is organized in 16 words of each 8 bit. The upper four bit are the instruction, the lower 4 bit the operand. A nop instruction (or any other instruction without operand) can be used for variables.

How to test

The memory is outside the logic and the clock along with some scan logic that reads the internal state for debug and visualization.

Each cycle is driven externally usually as:

- Reset the logic with cycling `usr_rst` 1 -> `usr_clk` 1 -> `usr_clk` 0 -> `usr_rst` 0
- Each execution starts with the fetch phase where `usr_clk` is 0 and the data from `addr` assigned to the bidirectional data
- With the rising edge of `usr_clk` the execution starts. The `we` signal indicates a write cycle, but the controller driving the execution grants access with `mem_grant`, and can then read the data from the pins

The internal 19 bit state can be scanned on either positive or negative clock period with a separate clock. Both clocks are assumed in the kHz range, so timing and domain crossing are no problem. `scan_clk` cycles through the data, `scan_en` indicates the start when high during a positive edge.

External hardware

It requires a testbed to properly drive the pins. There is a microcontroller program to cycle through those states including the handling of the tristate.

Pinout

#	Input	Output	Bidirectional
0	usr_clk	addr[0]	data[0]
1	usr_RST	addr1	data1
2	scan_clk	addr2	data2
3	scan_en	addr[3]	data[3]
4	mem_grant	we	data[4]
5		scan_out	data[5]
6			data[6]
7			data[7]

Hybrid_Adder_8bit [5]

- Author: James Xie, Cameron Bedard
- Description: 8-bit hybrid adder (using CLA and KSA)
- GitHub repository
- HDL project
- Mux address: 5
- Extra docs
- Clock: 0 Hz

How it works

The 8-bit Hybrid Adder combines the gate efficiency of a 4-bit Kogge Stone and the low latency of a 4-bit Carry Look Ahead Adder. The resultant 8-bit Hybrid Adder is faster than the an 8-bit Kogge Stone Adder and more gate efficient than a 8-bit Carry Look Ahead Adder.

How to test

The first number you want to add, use the eight inputs for ui_in for the input number A and the eight inputs for uio_in for the input number B. The output of the two numbers added together will be outputs on the eight outputs on uo_out.

External hardware

The only external hardware needed is applying the 3.3v on the inputs and reading the output.

Pinout

#	Input	Output	Bidirectional
0	a[0]	sum[0]	b[0]
1	a1	sum1	b1
2	a2	sum2	b2
3	a[3]	sum[3]	b[3]
4	a[4]	sum[4]	b[4]
5	a[5]	sum[5]	b[5]
6	a[6]	sum[6]	b[6]

#	Input	Output	Bidirectional
7	a[7]	sum[7]	b[7]

Dynamic Threshold Leaky Integrate-and-Fire [6]

- Author: Kai Linsley
- Description: Leaky Integrate-and-Fire model simulating a spiking biological neuron
- GitHub repository
- HDL project
- Mux address: 6
- Extra docs
- Clock: 1000000 Hz

How it works

This is how my model works. Why does test say I haven't added a how it works section?

How to test

This is how to test the model, you just gotta do that and this and that again. Why is there no how it works section? I don;t particulalry understand.

External hardware

These are all the external hardware requirements, there are actually none because we aren't fancy like that. In fact, I am only writing this out to avoid tests failing.

Pinout

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

8-bit Carry Look-Ahead Adder [7]

- Author: Seongwan Jeon and Michael Zeng
- Description: Fast 8-bit adder
- GitHub repository
- HDL project
- Mux address: 7
- Extra docs
- Clock: 0 Hz

How it works

A carry-lookahead adder (CLA) is a type of adder designed for fast speeds. First, it calculates the propagate and generate signals. The propagate signal determines if a carry bit can propagate through to the next bit, and the generate signal bit determines if there is a carry bit. As the name implies, a carry-lookahead adder works by generating a carry bit for every bit in the sum. This works by determining every possible way a carry bit can be generated by combining the generate and propagate signal from previous bits. The equations for the propagate, generate, sum, and carry bit are shown below:

$$P_i = A_i \oplus B_i$$

$$G_i = A_i \cdot B_i$$

$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i \cdot C_i$$

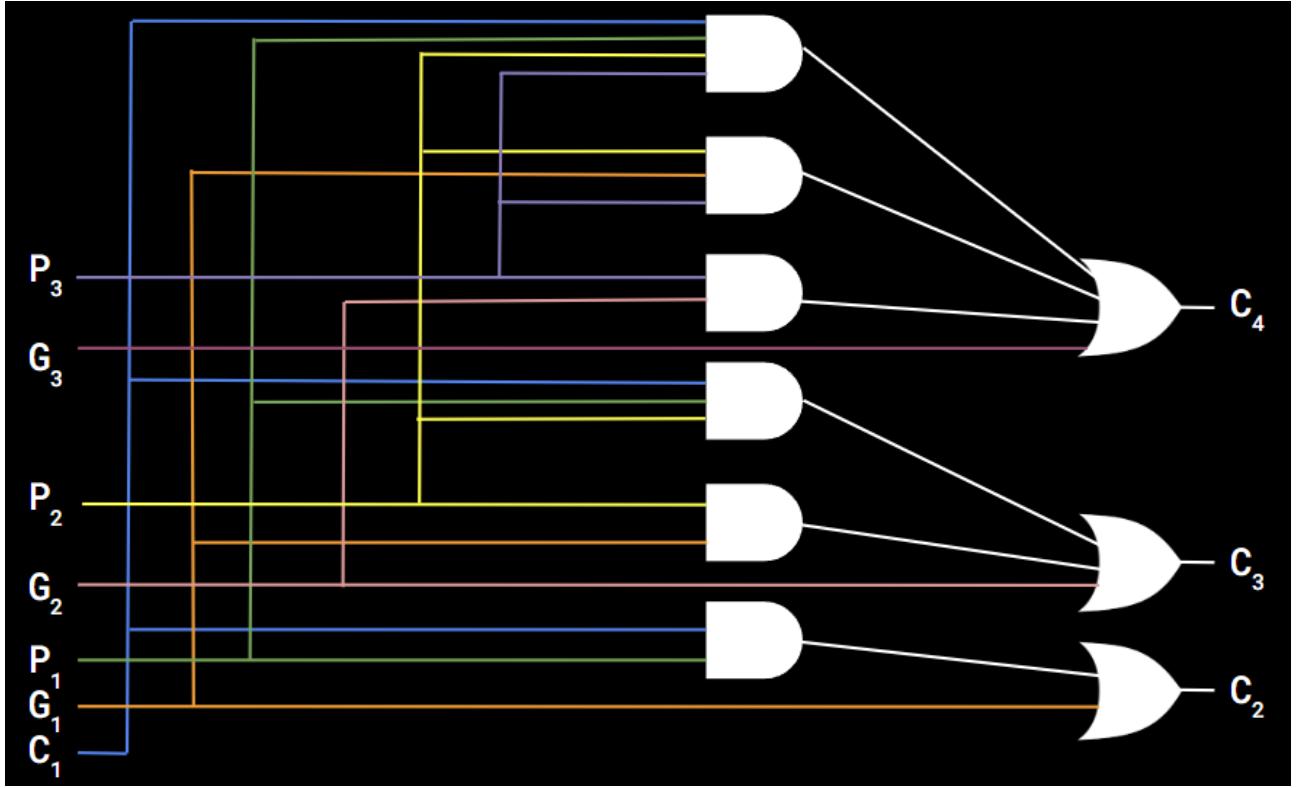
The calculations for the propagate, generate, and sum signals are trivial, but the calculation for the carry bit is dependent on its value in the previous bit, which makes it more complicated to solve. For example, all of the carry bits in a 4-bit CLA adder can be seen in the equation and diagram below:

$$C_1 = G_0 + P_0 \cdot C_{in}$$

$$C_2 = G_1 + P_1 \cdot C_1 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_{in}$$

$$C_3 = G_2 + P_2 \cdot C_2 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_{in}$$

$$C_4 = G_3 + P_3 \cdot C_3 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_{in}$$



By calculating the carry bits by using combinatorial logic, a CLA is able to calculate all of the carry bits of the sum without relying on sequential operations, unlike a ripple carry adder. The main time complexity of the ripple carry adder is based on the implementation of the last (and largest) AND gate of the most significant carry bit in the combinatorial equation. This AND gate has $n+1$ inputs, where n is the bits of the input. The implementation of multiple input AND gates in hardware consists of multiple smaller input AND gates organized in a tree structure, which inherently has a logarithmic time complexity. This logic extends to the CLA which possesses a logarithmic time complexity, and it makes CLAs viewed as one of the fastest implementations of digital adders due to its combinatorial nature. CLAs that calculate large bit-widths can also be designed by using multiple CLAs with smaller bit-widths in parallel to calculate intermediate values. This implementation using a tree structure of adders allows CLAs to also possess a modular design which can be scaled up to handle large bit-widths. However, this tree-like design is an implementation that other parallel prefix adders such as the Kogge-Stone adder utilize to a greater effect. Although CLAs are praised for their speed, it comes at the cost of a large area, as the components needed to calculate the carry bits for larger bit-widths become exponentially larger.

The CLA in this project is an 8-bit adder that does not utilize the implementation using smaller CLAs; rather, it is a fully combinatorial circuit to calculate all 8 bits of the carry signal.

How to test

`ui_in[7:0]` is addend 1, and `uo_in[7:0]` is addend 2. `ui_out[7:0]` is sum.

The adder was tested using all possible pairs of integers from 0 to 255 as inputs, which resulted in 25536 test cases total. For example, the adder would use `0x25` and `0xD7` as inputs, add them up to `0xFC`, and the result would be checked to make sure it was the correct output. Carry out was not checked as there is no output pin for a carry out on the board.

External hardware

No external hardware needed.

Pinout

#	Input	Output	Bidirectional
0	<code>a[0]</code>	<code>sum[0]</code>	<code>b[0]</code>
1	<code>a1</code>	<code>sum1</code>	<code>b1</code>
2	<code>a2</code>	<code>sum2</code>	<code>b2</code>
3	<code>a[3]</code>	<code>sum[3]</code>	<code>b[3]</code>
4	<code>a[4]</code>	<code>sum[4]</code>	<code>b[4]</code>
5	<code>a[5]</code>	<code>sum[5]</code>	<code>b[5]</code>
6	<code>a[6]</code>	<code>sum[6]</code>	<code>b[6]</code>
7	<code>a[7]</code>	<code>sum[7]</code>	<code>b[7]</code>

ternary, E1M0, E2M0 decoders [8]

- Author: ReJ aka Renaldas Zioma
- Description: Ternary, Quinary and Septenary 1.6 .. 2.6 bits/param packed weights
- GitHub repository
- HDL project
- Mux address: 8
- Extra docs
- Clock: 0 Hz

How it works

Unpacks Ternary, Quinary and Septenary 1.6 .. 2.6 bits/param packed weights

How to test

Provide packed weights on INPUT pmods

External hardware

Use Pico

Pinout

#	Input	Output	Bidirectional
0	packed weights LSB	unpacked	(out) unpacked
1	packed weights	unpacked	(out) unpacked
2	packed weights	unpacked	(out) unpacked
3	packed weights	unpacked	(out) unpacked
4	packed weights	unpacked	(out) unpacked
5	packed weights	unpacked	(out) unpacked
6	packed weights	unpacked	(out) dummy
7	packed weights MSB	unpacked	(in) Ternary / Septenary

RISC-V Mini [9]

- Author: RickGao
- Description: RISC-V Mini 8 Bit
- GitHub repository
- HDL project
- Mux address: 9
- Extra docs
- Clock: 100000 Hz

How it works

This project aims to design and implement a compact 8-bit RISC-V processor core optimized for Tiny Tapeout, a fabrication platform for small-scale educational IC projects. The processor employs a customized, compressed RISC-V instruction set (RVC) to reduce instruction width to 16 bits, leading to a more compact design suited to Tiny Tapeout's area and resource constraints. Developed in Verilog, this processor will handle computational, load/store and control-flow operations efficiently and undergo verification through simulation and testing.

Processor Components The processor comprises the following core components, optimized to meet Tiny Tapeout's area requirements:

1. Control Unit Generates control signals for instruction execution based on opcode and other instruction fields.
2. Register File Contains 8 general-purpose, 8-bit-wide registers. Register x0 will always return zero when read, adhering to RISC-V convention.
3. Arithmetic Logic Unit (ALU) Performs basic arithmetic (addition, subtraction) and logical (AND, OR, XOR, SLT) operations as specified by the decode stage. Supports custom compressed RISC-V instructions.
4. Datapath Single-cycle execution, optimized for minimal hardware complexity, reducing the processor's area and power consumption.

How to test

Simply set the input to the instruction and clock once to receive the output.

R-Type, I-Type, and L-Type instructions will output 0.

The S-Type instruction will output the value of the register.

The B-Type instruction will output 1 if the branch is taken and 0 if it is not taken.

Instructions List

R-Type

Name | funct3 [15:13] | funct2 [12:11] | rs2 [10:8] | rs1 [7:5] | rd [4:2] | Opcode(00)
AND | 000 | 00 | XXX | XXX | XXX | Opcode(00)
OR | 001 | 00 | XXX | XXX | XXX | Opcode(00)
ADD | 010 | 00 | XXX | XXX | XXX | Opcode(00)
SUB | 011 | 00 | XXX | XXX | XXX | Opcode(00)
XOR | 001 | 01 | XXX | XXX | XXX | Opcode(00)
SLT | 111 | 00 | XXX | XXX | XXX | Opcode(00)

I-Type

Name | funct3 [15:13] | Imm [12:8] (5-bit unsigned) | rs1 [7:5] | rd [4:2] | Opcode(01)
SLL | 100 | XXXXX | XXX | XXX | Opcode(01)
SRL | 101 | XXXXX | XXX | XXX | Opcode(01)
SRA | 110 | XXXXX | XXX | XXX | Opcode(01)
ADDI | 010 | XXXXX | XXX | XXX | Opcode(01)
SUBI | 011 | XXXXX | XXX | XXX | Opcode(01)

L-Type

Load | Imm [15:8] (8-bit signed) | 000 | rd [4:2] | Opcode(10)

S-Type

Store | 00000 | 000 | rs1 [7:5] | 000 | Opcode(11)

B-Type

Name | funct3 [15:13] | funct2 [12:11] | rs2 [10:8] | rs1 [7:5] | 000 | Opcode(11)
BEQ | 011 | 00 | XXX | XXX | 000 | Opcode(11)
BNE | 011 | 10 | XXX | XXX | 000 | Opcode(11)
BLT | 111 | 00 | XXX | XXX | 000 | Opcode(11)

External hardware

No External Hardware

Pinout

#	Input	Output	Bidirectional
0	instruction[0]	result[0]	instruction[8]
1	instruction1	result1	instruction[9]
2	instruction2	result2	instruction[10]
3	instruction[3]	result[3]	instruction[11]
4	instruction[4]	result[4]	instruction[12]
5	instruction[5]	result[5]	instruction[13]
6	instruction[6]	result[6]	instruction[14]
7	instruction[7]	result[7]	instruction[15]

eksdee [10]

- Author: lucy revi
- Description: That's for none of us to know and all of us to find out.
- GitHub repository
- HDL project
- Mux address: 10
- Extra docs
- Clock: 0 Hz

How it works

I honestly don't know yet.

How to test

I honestly don't know yet.

External hardware

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

I honestly don't know yet.

Pinout

#	Input	Output	Bidirectional
0	IN0	OUT0	
1	IN1	OUT1	
2	IN2	OUT2	
3	IN3	OUT3	
4	IN4	OUT4	
5	IN5	OUT5	
6	IN6	OUT6	
7	IN7	OUT7	

8-bit carry-skip [11]

- Author: Dennis_Du
- Description: two 8-bit input adder
- GitHub repository
- HDL project
- Mux address: 11
- Extra docs
- Clock: 0 Hz

How it works

This project implements an 8-bit carry-skip adder using a combination of ripple-carry and skip logic for enhanced performance. The adder is divided into two 4-bit sections. The lower 4 bits compute the initial partial sum and generate a carry-out, which is then either passed directly to the upper 4-bit section or skipped, depending on the carry-propagate signal. This design reduces the delay associated with carry propagation, making it more efficient than a conventional ripple-carry adder. The final 8-bit sum is registered and outputted in sync with the clock signal.

How to test

To test the carry-skip adder:

1. Load the design into your simulation environment.
2. Set the `ui_in` and `uo_in` inputs with the desired 8-bit values for addition.
3. The result of the addition will appear on `uo_out` after each rising edge.
4. Verify that the output matches expected values by comparing `uo_out` with the sum of the inputs.

For more extensive testing, a testbench can be used to automate input combinations and check results across various cases.

External hardware

No external hardware is required for this project.

Pinout

#	Input	Output	Bidirectional
0	a[0]	sum[0]	b[0]
1	a1	sum1	b1
2	a2	sum2	b2
3	a[3]	sum[3]	b[3]
4	a[4]	sum[4]	b[4]
5	a[5]	sum[5]	b[5]
6	a[6]	sum[6]	b[6]
7	a[7]	sum[7]	b[7]

Cgates [12]

- Author: Tommy Thorn
- Description: Testing two different Cgate implementations and rings
- GitHub repository
- HDL project
- Mux address: 12
- Extra docs
- Clock: 0 Hz

How it works

(This is a variant of tt06-ncl-lfsr, but with different C-gate implementations)

Muller's C-gate is a state-holding element with two inputs A and B, and an output Q. Q holds the previous state unless A == B in which case it takes on this value. There are many ways to implement the C-gate. In this design, we try two: building it from a latch and building it out of combinatorial logic. The two inputs ui[0] and ui1 are fed to two C-gates Cl and Cc, build with a latch and combinatorial logic respectively. Their respective outputs are wired to uo[0] and uo1.

We also build four rings from this, with uo2 and uo[3] being the output of a four stage build from Cl and Cc gates respectively. Similar for uo[4]/uo[5] except using 16 stage rings and uo[6]/uo[7] for (TBD) stage rings.

How to test

Set ui[0] and ui1 different values and verify that uo[0]/uo1 only changes when both agree. Observe uo[7:2] and look for transitions.

External hardware

For the basic test the rp2040 on the bringup board should be enough for the ring test, an oscilloscope is [probably] required.

#	Input	Output	Bidirectional
---	-------	--------	---------------

Pinout

#	Input	Output	Bidirectional
0	A	QI	
1	B	Qc	
2		R4I	
3		R4c	
4		R16I	
5		R16c	
6		RTBDI	
7		RTBDC	

T3 (Tiny Ternary Tapeout) [13]

- Author: Arnav Sacheti & Jack Adiletta
- Description: Ternary Matmul Processor
- GitHub repository
- HDL project
- Mux address: 13
- Extra docs
- Clock: 50000000 Hz

Tiny Ternary Tapeout Project Documentation

Inspiration The inspiration for this Tiny Tapeout project comes from the “Scalable MatMul-free Language Modeling” paper, which explores a novel approach to language modeling that bypasses traditional matrix multiplication (MatMul) operations. Standard neural network models, especially those used for language processing, rely heavily on matrix multiplications to handle complex data transformations. However, these operations can be computationally expensive and power-intensive, especially at large scales.

The key insight of this research is to leverage alternative mathematical structures and sparse representations, reducing the need for resource-heavy MatMul operations while still enabling efficient language processing. By reimagining the model architecture to avoid these multiplications, it opens up possibilities for more energy-efficient, scalable models, particularly in hardware-constrained environments like microchips. This Tiny Tapeout project aims to implement and experiment with these principles on a small scale, designing circuitry that emulates the core ideas of this MatMul-free approach. This can pave the way for more efficient and compact language models in embedded systems, potentially transforming real-time, on-device language processing applications.

How it works The `tt_um_tiny_ternary_tapeout.v` module is designed to perform matrix multiplication using a pipelined architecture. Here’s a step-by-step explanation of how it works:

Loading the Weights (`tt_um_load.v`):

The module starts by loading the weights for the matrix. These weights are stored in an internal register array and are used for the matrix multiplication operations.

Matrix Multiplication (`tt_um_mult.v`):

The module performs matrix multiplication by iterating over the columns of the weight matrix and calculating the temporary output values based on the weights and input vectors. For each column, the module multiplies the input vector elements by the corresponding weights and sums the results to produce the output values.

Pipelined Architecture:

The module is pipelined, meaning that it can continuously accept new input vectors while performing computations on the previous inputs. As new inputs are driven into the module, the current computations are completed, and the results are stored in a pipeline register. During the next clock cycle, the outputs are produced as 8-bit integers, allowing for continuous data processing without interruption.

Outputting Results:

After driving all the inputs, the outputs are produced as 8-bit integers. These outputs represent the result of the matrix multiplication operation. By leveraging a pipelined architecture, the `tt_um_mult.v` module ensures efficient and continuous data processing, allowing for high-throughput matrix multiplication operations.

Example: Using a Ternary Array for Efficient Computation In this example, we'll create a 4x2 ternary array and demonstrate how it can be used to process a 1x4 input vector.

Step 1: Define a Ternary Array

A ternary array is one where each element can take on one of three possible values, commonly -1 , 0 , or $+1$. These values simplify calculations because instead of performing complex multiplications, you can use additions, subtractions, or ignore the zero entries altogether.

Let's create a sample 4x2 ternary array:

$$\text{Array} = [+1 \ 0 \ -1 \ +1 \ 0 \ -1 \ +1 \ +1]$$

Step 2: Define the Input Vector

Let's assume we have a 1x4 input vector:

$$\text{Input} = [2 \ -1 \ 3 \ 0]$$

Step 3: Compute the Output without Matrix Multiplication

Instead of performing a matrix multiplication, we'll calculate the output using simpler operations based on the ternary values.

For each column in the ternary array:

- Multiply +1 entries by the corresponding input values.
- Subtract the values for -1 entries.
- Ignore the 0 entries.

Step 4: Calculate Each Column's Output

Let's compute each column separately:

- **Column 1 Calculation:**

- Row 1: ($+1 \times 2 = 2$)
- Row 2: ($-1 \times -1 = +1$)
- Row 3: ($0 \times 3 = 0$)
- Row 4: ($+1 \times 0 = 0$)

$$\text{Sum of Column 1: } (2 + 1 + 0 + 0 = 3)$$

- **Column 2 Calculation:**

- Row 1: ($0 \times 2 = 0$)
- Row 2: ($+1 \times -1 = -1$)
- Row 3: ($-1 \times 3 = -3$)
- Row 4: ($+1 \times 0 = 0$)

$$\text{Sum of Column 2: } (0 - 1 - 3 + 0 = -4)$$

Final Output Vector

Combining the results from each column, we get the final output vector:

$$\text{Output} = [3 \ -4]$$

How to test To test the Matrix Multiplier with an external MCU like a Raspberry Pi, follow these steps:

1. Setup:

- Connect the Raspberry Pi to the Matrix Multiplier hardware using appropriate GPIO pins.
- Ensure that the Raspberry Pi has the necessary libraries installed for GPIO manipulation.

Pinout

#	Input	Output	Bidirectional
0	A1	Q1	B1
1	A2	Q2	B2
2	A3	Q3	B3
3	A4	Q4	B4
4	A5	Q5	B5
5	A6	Q6	B6
6	A7	Q7	B7
7	A8	Q8	B8

Classic 8-bit era Programmable Sound Generator SN76489

[14]

- Author: ReJ aka Renaldas Zioma
- Description: The SN76489 Digital Complex Sound Generator (DCSG) is a programmable sound generator chip from Texas Instruments.
- GitHub repository
- HDL project
- Mux address: 14
- Extra docs
- Clock: 4000000 Hz

How it works

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

The goals of this project

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

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

The future work

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

Chip technical capabilities

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

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

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

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

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

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

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

Historical use of the SN76489

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

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

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

The original pinout of the SN76489AN

```

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

```

Difference from the original hardware

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

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

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

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

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

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

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

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

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

A configurable clock divider was introduced in this implementation.

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

The reverse engineered SN76489

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

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

How to test

Summary of commands to communicate with the chip

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

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

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

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

Note frequency

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

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

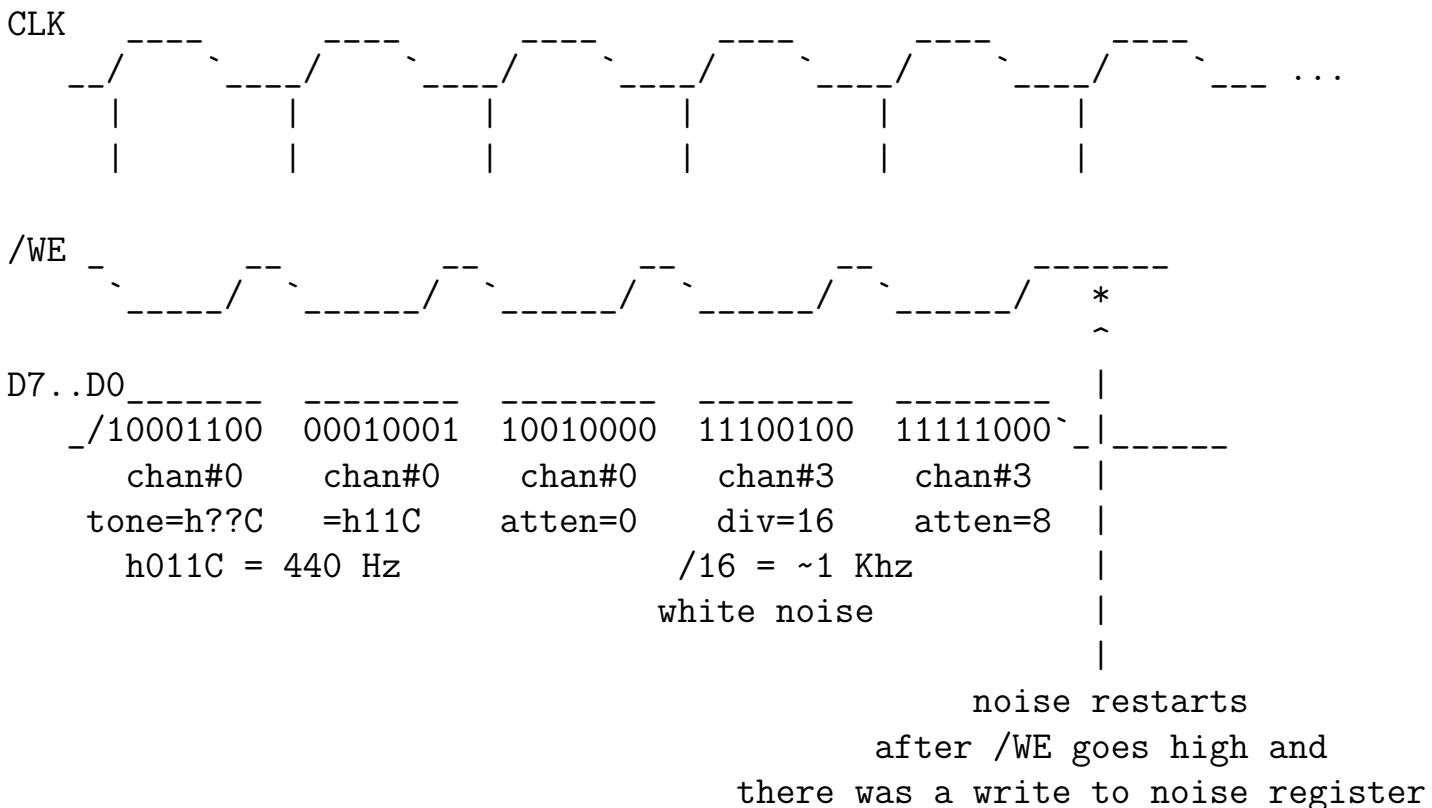
For example 10-bit value that plays 440 Hz note on a chip clocked at 4 MHz would be:

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

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

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

Timing diagram



Configurable clock divider

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

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

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

Some examples of music recorded from the chip simulation

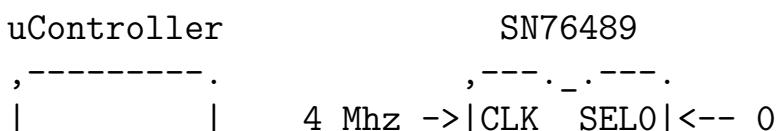
- [https://www.youtube.com/watch?v=ghBGasckpSY](Craze Rider BBC Micro game)
 - [https://www.youtube.com/watch?v=HXLAdA02I-w](MISSION76496 tune for Sega Master System)

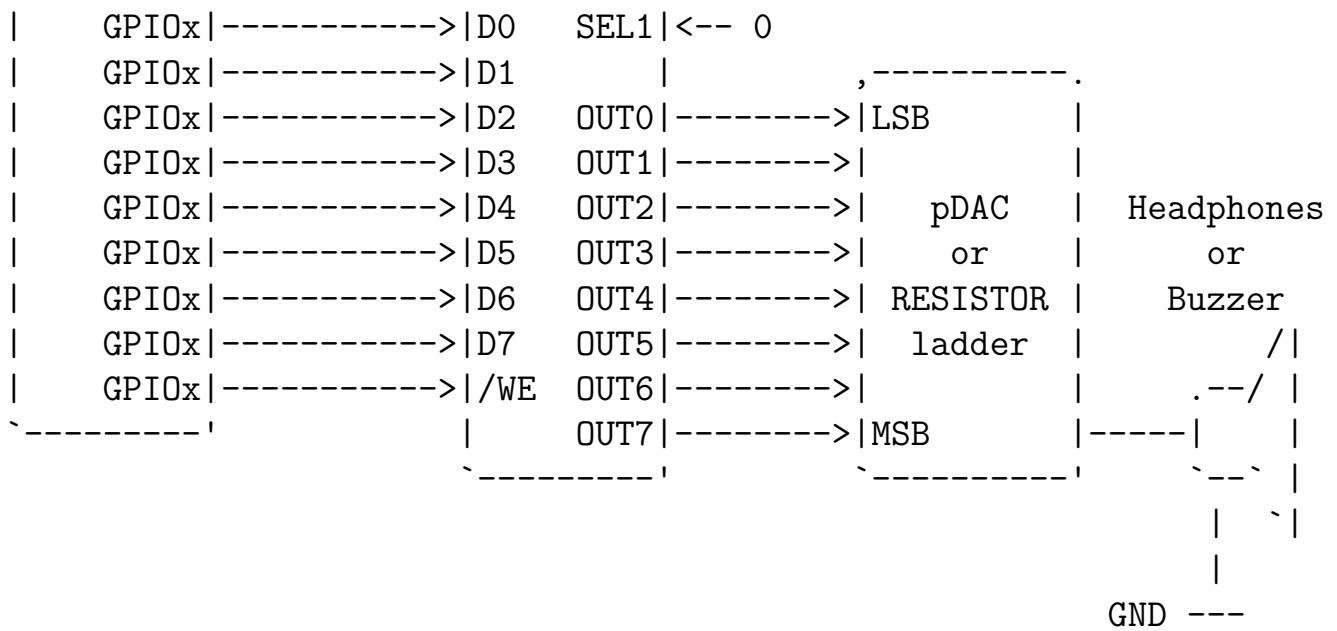
External hardware

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

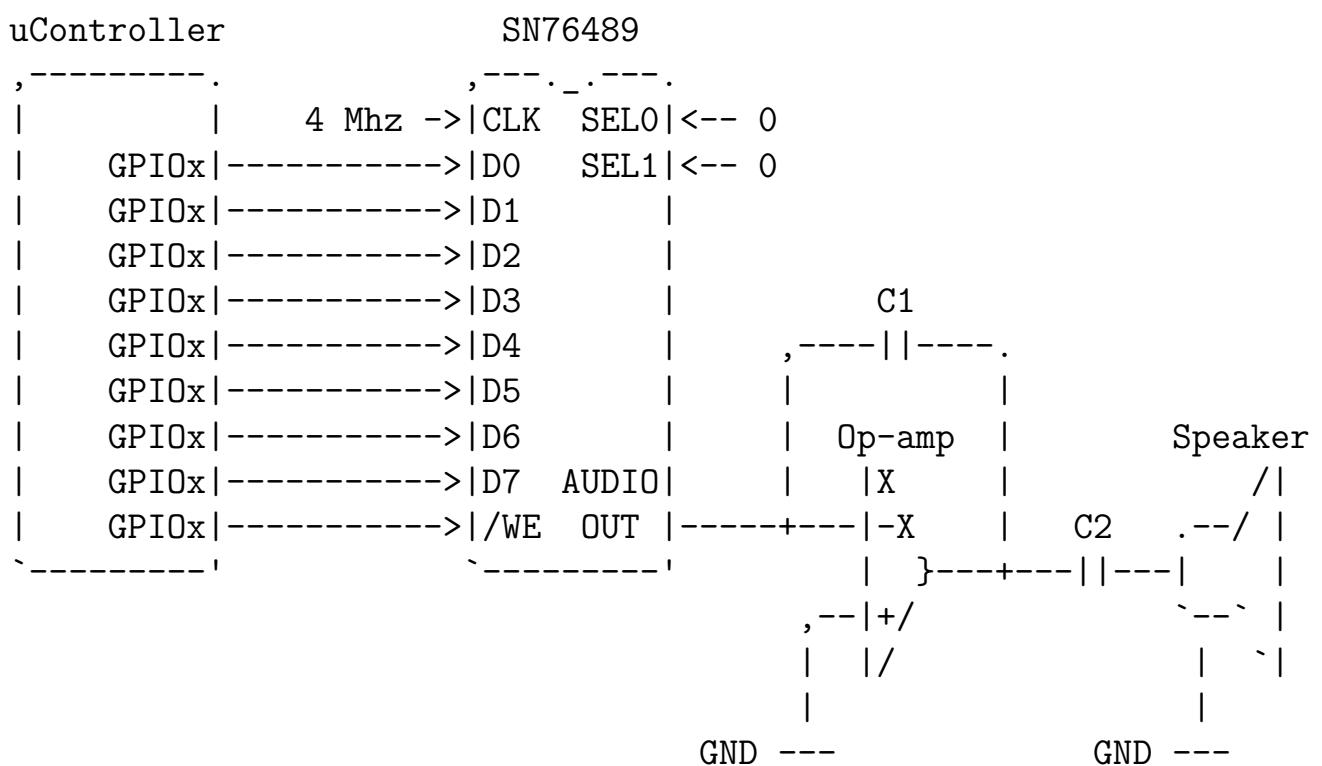
The data bus of the SN76489 chip has to be connected to microcontroller and receive a regular stream of commands. The SN76489 produces audio output and has to be connected to a speaker. There are several ways how the overall schematics can be established.

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

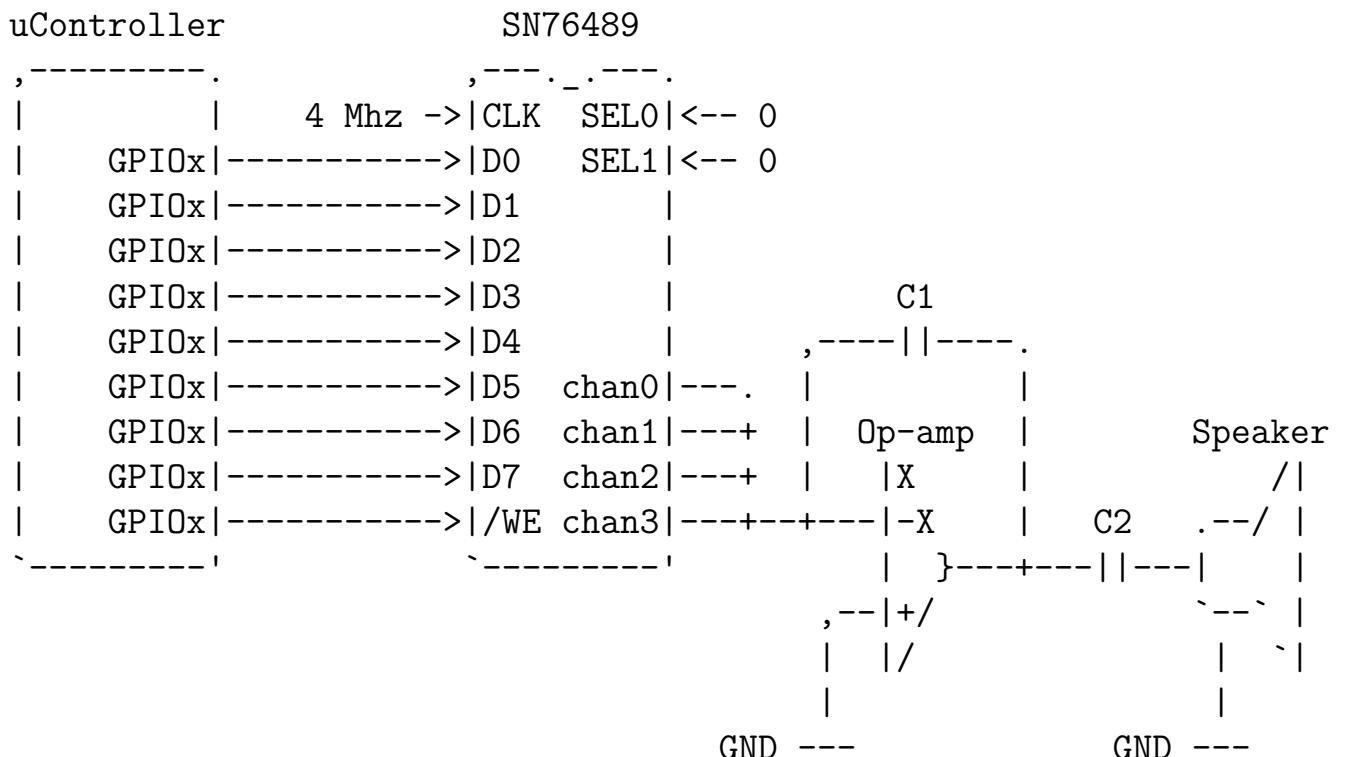




AUDIO OUT through RC filter Another option is to use the Pulse Width Modulated (PWM) AUDIO OUT pin that combines 4 channels with the Resistor-Capacitor based low-pass filter or better the Operation Amplifier (Op-amp) & Capacitor based integrator:



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



Pinout

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

3 Neuron ALIF [15]

- Author: Andrew Smith
- Description: TODO
- GitHub repository
- HDL project
- Mux address: 15
- Extra docs
- Clock: 0 Hz

How it works

3 Adaptive Leaky Integrate and Fire Neurons

1. Receives an 8-bit input signal (`ui_in`) with small offset variations
2. Processes the signal through the LIF model which simulates biological neuron behavior by:
 - Integrating (accumulating) input current over time
 - Applying a leak factor to gradually decrease membrane potential
 - Generating a spike when membrane potential exceeds threshold
 - Adjusting a moving threshold based on periods of past inputs
3. Outputs:
 - Spike signals on `ui_out[7:5]`:
 - `ui_out[7]`: Neuron 1 spike output
 - `ui_out[6]`: Neuron 2 spike output
 - `ui_out[5]`: Neuron 3 spike output
 - Internal state of Neuron 1 on `uo_out[7:0]` for debugging/testing

How to test

1. Basic Functionality Test:
 - Apply a constant input value through `ui_in`
 - Monitor `ui_out[7:5]` to observe spike patterns
 - Check `uo_out` to monitor Neuron 1's internal state
2. Threshold Response Test:

- Gradually increase ui_in value
- Observe spike behavior on uio_out[7:5]
- Verify neurons spike when input exceeds threshold

3. Reset Test:

- Assert rst_n (active low)
- Verify all spike outputs (ui_out[7:5]) go low
- Verify internal state (uo_out) resets to initial value

External hardware

No external hardware required. The design uses only the built-in TinyTapeout inputs and outputs:

- 8 input pins (ui_in[7:0])
- 8 output pins (uo_out[7:0])
- 8 bidirectional pins (ui_out[7:0])
- Clock (clk)
- Reset (rst_n)

Pinout

#	Input	Output	Bidirectional
0	Input current bit [0]	State variable bit [0]	
1	Input current bit 1	State variable bit 1	
2	Input current bit 2	State variable bit 2	
3	Input current bit [3]	State variable bit [3]	
4	Input current bit [4]	State variable bit [4]	
5	Input current bit [5]	State variable bit [5]	
6	Input current bit [6]	State variable bit [6]	
7	Input current bit [7]	State variable bit [7]	Spike bit

Giant Ring Oscillator (3853 inverters) [16]

- Author: Uri Shaked
- Description: Configurable ring oscillator with up to 3853 inverters
- GitHub repository
- HDL project
- Mux address: 16
- Extra docs
- Clock: 0 Hz

How it works

A giant, configurable ring oscillator with up to 3853 stages. To enable the ring oscillator, connect one of the output pins to the first input pin (`ring_in / ui_in[0]`). Each output pin is connected at a different point in the ring oscillator chain, making it possible to create rings of different lengths:

Pin	Chain length
uo[0]	1
uo1	3
uo2	5
uo[3]	7
uo[4]	11
uo[5]	21
uo[6]	51
uo[7]	101
ui0[0]	201
ui01	501
ui02	1001
ui0[3]	2001
ui0[4]	3001
ui0[5]	3853

There is also an option to connect the ring oscillator internally, by driving `internal_loopback` high. This will create a ring oscillator with 3853 stages.

How to test

Connect one of the output pins (e.g. `ui0_out[5]`) to `ring_in` or set `internal_loopback` to 1, and measure the output frequency.

External hardware

A scope / logic analyzer to measure the output frequency and the delay between different points in the inverter chain.

Pinout

#	Input	Output	Bidirectional
0	ring_in	len1	len201
1	internal_loopback	len3	len501
2		len5	len1001
3		len7	len2001
4		len11	len3001
5		len21	len3853
6		len51	
7		len101	

STDP Circuit [17]

- Author: Mariah Regalado
- Description: STDP Circuit using a trace to model exponential behavior
- GitHub repository
- HDL project
- Mux address: 17
- Extra docs
- Clock: 0 Hz

How it works

The point of this circuit is to detect spikes and measure the time interval between them. My code uses `delta_t` to measure the time. If a pres-synaptic spike happens, if no spike was detected before, my `pre_spike_detected` signal is set to 1 and `delta_t` is set to. If there has been a post synaptic spike, and `post_spike_detected` has been triggered, `delta_t` decrements to measure the time difference. `Delta_t` accumulates otherwise.

If `pre_spike_detected` and `post_spike_detected` are both high, both spikes have been detected and the sign of `delta_t` is used to determine if depression or potentiation should occur. I used a trace to model the exponential behavior of STDP. I modified the trace depending on whether it was necessary to depress or potentiate the weight. I also included edge cases to ensure the newly calculated weight doesn't cause overflow.

How to test

I am still working on it.

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	Input Current Bit [0]	State Variable bit[0]	
1	Input Current Bit 1	State Variable bit1	
2	Input Current Bit 2	State Variable bit2	

#	Input	Output	Bidirectional
3	Input Current Bit [3]	State Variable bit[3]	
4	Input Current Bit [4]	State Variable bit[4]	
5	Input Current Bit [5]	State Variable bit[5]	
6	Input Current Bit [6]	State Variable bit[6]	
7	Input Current Bit [7]	State Variable bit[7]	Spike bit

TwoChannelSquareWaveGenerator [18]

- Author: Sam Kho
- Description: Like having two apple2-style speakers
- GitHub repository
- HDL project
- Mux address: 18
- Extra docs
- Clock: 256000 Hz

How it works

Two 8-bit inputs, TA and TB, are used to reload internal countdown timers when they reach zero, at which time, respective outputs OUTA and OUTB are toggled. A 2-bit SUM output is also provided as a convenience ($SUM = OUTA + OUTB$).

How to test

Apply arbitrary 8-bit reload values to TA (ui_in) and TB (uio_in). Probe OUTA and OUTB with oscilloscope or logic analyzer. Time period for outputs is proportional to $(input+1)$; i.e. to get two waves with period T and period 2T, provide values like 3 and 7 (instead of 4 and 8). Also check 2-bit output SUM (should be OUTA + OUTB, possibly delayed by one cycle).

External hardware

External hardware not needed, but intent is to drive speakers (probably bring down voltage level via resistor dividers, then feed into speaker amplifier).

Pinout

#	Input	Output	Bidirectional
0	TA0	OUTA	TB0
1	TA1	OUTB	TB1
2	TA2	SUM0	TB2
3	TA3	SUM1	TB3
4	TA4		TB4
5	TA5		TB5

#	Input	Output	Bidirectional
6	TA6		TB6
7	TA7		TB7

instrumented_ring_oscillator [19]

- Author: Jeremy Mickelsen
- Description: A ring oscillator with a selectable number of stages and initial state.
- GitHub repository
- HDL project
- Mux address: 19
- Extra docs
- Clock: 0 Hz

How it works

Preface: This is probably not a component you want if you want a reliable end device. This is intended to allow studying the decay (or persistence) of high-frequency “modes” which are generally very undesirable.

This project uses ring oscillators with muxes on the inputs to allow setting an initial state or “seed”. This can be configured using a clock (in3) and data (in2) similar to SPI (positive edge clocks the data in). The in0 line is the enable to start the oscillator running, and in1 is a HOLD line that blocks one stage so that the normal long period can be obtained. in7:in4 select the number of stages ($2*n + 5$). In order to have selectable stages without a really big mux (which would have a very different propagation speed than the other stages), two muxes per stage are used, some of them bypassing some of the chain to get the desired number of muxes. This diagram shows the short mux paths as pipes (“|”).

Note that when less than 25 stages are used, all inverters are still driven, but some outputs are not used. Note that the seed state is a FIFO fed in at the little end - it’s always updatable (though its state should not impact operation).

How to test

0. Hook up an analyzer / scope to the output & bidirectional channels. 16 phases are driven out.
1. Select the number of stages (in7:in4).
2. If desired, seed the initial state using in3, in2. It's a
3. Drive enable (in0) high and watch the chaos to see if it stabilizes to the longest frequency, or if high frequency modes persist.
4. The hold (in1) can be briefly driven to get to the longest frequency.

External hardware

A logic analyzer will probably be the most useful tool for this - For FPGA testing, I used a Digilent Digital Discovery (DD) with this projects outputs going to DD channels 0-15, and using DD channels 24-31 to drive the project inputs. A multi-channel oscilloscope might also be interesting to use with this.

Pinout

#	Input	Output	Bidirectional
0	enable	phase[0]	phase[8]
1	hold	phase1	phase[9]
2	bdat	phase2	phase[10]
3	bclk	phase[3]	phase[11]
4	n_stages[0]	phase[4]	phase[12]
5	n_stages1	phase[5]	phase[13]
6	n_stages2	phase[6]	phase[14]
7	n_stages[3]	phase[7]	phase[15]

Linear Timecode (LTC) generator [32]

- Author: Thomas Flummer
- Description: Timecode generator for audio video synchronization
- GitHub repository
- HDL project
- Mux address: 32
- Extra docs
- Clock: 12000000 Hz

How it works

Multiple counters to maintain time and framecount, with serial output of the LTC (80 bit frames, biphase mark code)

How to test

The project should have 12 MHz clock signal applied and after reset, will start out with a 00:00:00:00 timecode and starts to count.

Framerate is controlled by the ui2 and ui[3]

ui[3]	ui2	Framerate	Comment
0	0	24	
0	1	25	
1	0	29.97	Not implemented
1	1	30	

External hardware

This should work with the audio PMOD connected to the bidirectional port, to give levels useable for audio gear.

If testing with logic analyzer or similar, uio[7] can be directly connected. The signal is a digital signal.

Pinout

#	Input	Output	Bidirectional
0			
1			
2	FRAMERATE_0		
3	FRAMERATE_1		
4			
5			
6			
7		LTC_OUT	

Tiny Shader [34]

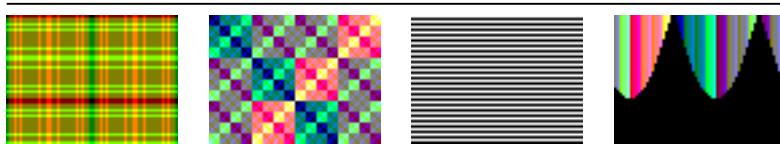
- Author: Leo Moser
- Description: With Tiny Shader you can write a small program to create different images and even animations.
- GitHub repository
- HDL project
- Mux address: 34
- Extra docs
- Clock: 25175000 Hz

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 accessing 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 RA.
SETG RA	G <= RA[1:0]	Set the green channel of the output color to the lower two bits of RA.
SETB RA	B <= RA[1:0]	Set the blue channel of the output color to the lower two bits of RA.

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 pixel.
GETUSER RA	RA <= USER	Set the specified register to the user value, can be set via the command line.

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 than or equal to 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 \leftarrow 0	Clear RA by writing 0.
LDI IMMEDIATE	RA \leftarrow IMMEDIATE	Load an immediate value into RA.

Special

Instruction	Operation	Description
SINE RA	RA \leftarrow SINE[R0[4:0]]	Get the sine value for R0 and write into RA. The sine value is 8 bits.

Boolean

Instruction	Operation	Description
AND RA RB	RA \leftarrow RA & RB	Boolean AND of RA and RB, result written into RA.
OR RA RB	RA \leftarrow RA	RB
NOT RA RB	RA \leftarrow \sim RB	Invert all bits of RB, result written into RA.
XOR RA RB	RA \leftarrow RA \wedge RB	XOR of RA and RB, result written into RA.

Move

Instruction	Operation	Description
MOV RA RB	RA \leftarrow RB	Move value of RB into RA.

Shift

Instruction	Operation	Description
SHIFTL RA RB	RA \leftarrow RA « RB	Shift RA with RB to the left, result written into RA.
SHIFTR RA RB	RA \leftarrow RA » RB	Shift RA with RB to the right, result written into RA.

Pseudo

Instruction	Operation	Description
NOP	RO \leftarrow RO & RO	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	R1	CS
1	debug_i[0]	G1	MOSI
2	debug_i1	B1	MISO
3		vsync	SCK
4		R[0]	next_vertical
5		G[0]	next_frame
6		B[0]	debug_o[0]
7		hsync	debug_o1

Sine Synth [36]

- Author: R. Timothy Edwards
- Description: Keyboard synthesizer with one octave of notes and a sine wave generator
- GitHub repository
- HDL project
- Mux address: 36
- Extra docs
- Clock: 50000000 Hz

How it works

This project implements a (trivially simple) music synthesizer, where “keys” are mapped to the input PMOD bits, and the synthesizer engine generates sine waves for each note. The sine wave generator is similar to the one that Mike Bell created to demonstrate the audio PMOD, but implements a more efficient method using delta steps. The 8-bit output is passed directly to the output PMOD, and simultaneously passed through a PWM generator (the one from Mike Bell’s project) to drive the audio PMOD. I am considering recasting this as an analog project and adding an 8-bit RDAC, but this version is digital only. Output of the synthesizer is monophonic.

How to test

Preferably attach the 8 bits of the input PMOD port to a row of 8 buttons that can be played like a keyboard. Only whole steps are represented, for one octave C to C.

External hardware

Preferably use Mike Bell’s audio PMOD on the bidirectional PMOD port. The project uses bit 7 as a single-bit PWM output that is used according to the instructions for the audio PMOD.

Pinout

#	Input	Output	Bidirectional
0	Key C	8-bit Output, bit 0	
1	Key D	8-bit Output, bit 1	

#	Input	Output	Bidirectional
2	Key E	8-bit Output, bit 2	
3	Key F	8-bit Output, bit 3	
4	Key G	8-bit Output, bit 4	
5	Key A	8-bit Output, bit 5	
6	Key B	8-bit Output, bit 6	
7	Key C	8-bit Output, bit 7	PWM Output

DRUM [38]

- Author: Gökçe Aydos
- Description: an approximate multiplier
- GitHub repository
- HDL project
- Mux address: 38
- Extra docs
- Clock: 0 Hz

How it works

The design consists of a RAM and an approximate multiplier $a * b = r$ based on DRUM: A Dynamic Range Unbiased Multiplier for Approximate Applications by Hashemi et. al.

How to test

$r = a * b$. Write data to a and b. Then read the result/s from the RAM. The product results should differ if the frequency is increased.

Address map:

- 0 to 7 => product result
- 8 => multiplicand 1
- 9 => multiplicand 2

External hardware

Nothing.

Pinout

#	Input	Output	Bidirectional
0	addr[0]	ram_out(0)	ram_in(0) or result(0)
1	addr1	ram_out(1)	ram_in(1) or result(1)
2	addr2	ram_out(2)	ram_in(2) or result(2)
3	addr[3]	ram_out(3)	ram_in(3) or result(3)
4		ram_out(4)	ram_in(4) or result(4)

#	Input	Output	Bidirectional
5	result write enable	ram_out(5)	ram_in(5) or result(5)
6	tristate output enable	ram_out(6)	ram_in(6) or result(6)
7	RAM write enable	ram_out(7)	ram_in(7) or result(7)

Tiny Hash Table [40]

- Author: Sasha Krassovsky
 - Description: Hash table with 8 slots, 4-bit keys, 4-bit values
 - GitHub repository
 - HDL project
 - Mux address: 40
 - Extra docs
 - Clock: 0 Hz

How it works

This is a hash table with 8 slots, 4-bit keys, and 4-bit values. Keys are entered on pins KEY3–KEY0, values are entered on pins VAL3–VAL0. When given a command, and told to execute via the GO line, the hash table hashes the key, and begins linearly probing into the hash table. Once a suitable slot is found, the given command is executed on that slot, or STATUS1–STATUS0 returns a suitable error message. The table takes care to buffer the inputs so that they're not changed during probing.

The commands are: | Command | CMD1 | CMD0 | | | |-----|-----|-----|-----|
CMD_LOOKUP | 0 | 0 | | | |-----|-----|-----|-----|
CMD_INSERT | 0 | 1 | | | |-----|-----|-----|-----|
CMD_DELETE | 1 | 0 | | | |-----|-----|-----|-----|

The status codes are: | Status | STATUS1 | STATUS0 | Description | |
| | | | |
| | | | | STATUS_OK | 0 | 0 | Operation
Succeeded | | STATUS_FULL | 0 | 1 | Insertion failed - hash table full | | STA-
TUS_NOTFOUND | 1 | 0 | Lookup failed - key not found | | STATUS_BUSY | 1 | 1
| Hash table is still probing |

How to test

Choose a key and value to insert, such as 0x4 and 0x2, and set the KEY and VAL lines accordingly (so in this case, 0100 and 0010). Next, set the CMD lines to CMD_INSERT, or 01. Lastly, set the GO line to 1. The STATUS lines should then turn to STATUS_BUSY for a few cycles, empirically 15 cycles is enough for all commands, though the timing varies on the load factor. After it finishes, the STATUS line should return to STATUS_OK, and the OVAL lines should contain the key you inserted (0010)! To run another command on the table, you must set GO to 0 for at least one cycle before triggering another command.

Pinout

#	Input	Output	Bidirectional
0	VAL0	OVAL0	CMD0
1	VAL1	OVAL1	CMD1
2	VAL2	OVAL2	GO
3	VAL3	OVAL3	
4	KEY0		
5	KEY1		
6	KEY2		STATUS0
7	KEY3		STATUS1

Asynchronous FIFO [42]

- Author: RMKGSN
- Description: An Asynchronous FIFO is a type of First-In-First-Out memory buffer that allows data transfer between two clock domains operating at different frequencies
- GitHub repository
- HDL project
- Mux address: 42
- Extra docs
- Clock: 60000000 Hz

How it works

An Asynchronous FIFO is a memory buffer enabling data transfer between two clock domains with different frequencies. It uses separate write and read clocks, along with pointers to track data flow. Full and empty flags prevent overflow and underflow, while synchronization logic ensures safe transfer, avoiding metastability. Gray-coded pointers enhance reliable communication, maintaining data integrity.

How to test

Set the following inputs to control FIFO operation:

- write_enable (ui_in[0]) – Enables writing data into the FIFO.
- read_enable (ui_in1) – Enables reading data from the FIFO.
- reset (ui_in2) – Clears all stored data and resets the FIFO.

Writing to the FIFO:

- Check if the full flag (ui_out[0]) is LOW (FIFO is not full).
- Set write_enable (ui_in[0]) HIGH and provide data to the FIFO.
- On the rising edge of the write clock (w_clk), data is written, and the write pointer advances.
- Release write_enable after writing is complete.

Reading from the FIFO:

- Check if the empty flag (ui_out1) is LOW (FIFO contains data).
- Set read_enable (ui_in1) HIGH to request data.
- On the rising edge of the read clock (r_clk), data is output, and the read pointer advances.
- Release read_enable after reading is complete.

Additional Controls (if using a Debug Interface or Controller):

- Adjust Clock Domains: Modify write and read clock frequencies to test synchronization.
- Monitor Full/Empty Flags: Ensure proper flow control to prevent overflow or underflow.
- Pause/Resume Reads/Writes: Dynamically enable or disable operations based on system requirements.

Pinout

#	Input	Output	Bidirectional
0		full	
1		empty	
2	wr_rq	rdata[0]	
3	rd_rq	rdata1	
4	wdata[0]	rdata2	
5	wdata1	rdata[3]	
6	wdata2		
7	wdata[3]		

Synchronous FIFO [44]

- Author: Monish V.R
- Description: It's a synchronous fifo which has 4 bit of width
- GitHub repository
- HDL project
- Mux address: 44
- Extra docs
- Clock: 0 Hz

How it works

The synchronous FIFO is a first in first out memory. When the data is entered and written it gets stored in the memory. And when read out, it will be removed from the memory.

It's based on debouncing mechanism.

How to test

TBA

External hardware

NIL

Pinout

#	Input	Output	Bidirectional
0		full	
1		empty	
2	wr	dout[0]	
3	rd	dout1	
4	din[0]	dout2	
5	din1	dout[3]	
6	din2		
7	din[3]		

Pulse Width Modulation [46]

- Author: Nithish Reddy KVS
- 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: 46
- Extra docs
- Clock: 50000000 Hz

How it works

This Verilog code implements a Pulse Width Modulation (PWM) generator designed for a 50 MHz input clock. The main functionality revolves around creating a variable duty cycle PWM signal and allowing user control to adjust the duty cycle through two input buttons. The module `tt_um_nithishreddykvs` uses a clock signal to generate a 10 MHz PWM output (`PWM_OUT`), with the duty cycle ranging from 0% to 90% in 10% increments.

To ensure reliable operation, debouncing logic is implemented for the buttons that increase (`ui_in[0]`) and decrease (`ui_in1`) the duty cycle. The debouncing mechanism uses D Flip-Flops (`DFF_PWM`) and a slow clock enable signal generated via a counter. The `slow_clk_enable` ensures that rapid fluctuations caused by button bouncing do not affect the duty cycle adjustment. The duty cycle can be dynamically updated by incrementing or decrementing its value through button presses, and this adjustment directly impacts the PWM signal's ON and OFF durations.

The module ensures that outputs and unused inputs are properly assigned to avoid any synthesis or simulation warnings, and a reset signal (`rst_n`) is included for reinitializing the design.

How to test

1. Simulation Environment:

- Use a Verilog simulator (such as ModelSim, Vivado, or Verilator) to verify the design.
- Apply a 50 MHz clock signal to the `clk` input.

- Provide test signals to ui_in[0] and ui_in1 for increasing and decreasing the duty cycle.
- Observe the PWM_OUT signal to confirm that the duty cycle adjusts correctly in response to button presses.

2.FPGA Testing:

- Synthesize the code for an FPGA board (e.g., Xilinx or Intel FPGA).
- Assign the ui_in signals to physical buttons or switches on the board for user interaction.
- Connect the PWM_OUT signal to an output pin that drives an LED or an oscilloscope for visual verification of the PWM waveform.
- Verify that the duty cycle changes in 10% increments as buttons are pressed and released.

External hardware

Input Buttons: Two push buttons are required for increasing and decreasing the duty cycle. These buttons are connected to ui_in[0] and ui_in1. Ensure pull-up or pull-down resistors are used to stabilize the input signals when buttons are not pressed.

PWM Output Device: The PWM_OUT signal can be connected to an LED, motor, or any other device that can visually or functionally represent the PWM signal. For testing, an oscilloscope is recommended to observe the PWM waveform and verify the duty cycle.

Clock Source: A 50 MHz clock signal is essential for driving the design. This can be provided by the FPGA's onboard oscillator or an external clock module.

Reset and Power Supply: A reset button should be included to initialize the system via the rst_n signal. The design should also include appropriate voltage levels (e.g., 3.3V or 5V) as per the FPGA board's requirements.

FPGA Board: Use an FPGA development board that supports 50 MHz clock input and has sufficient GPIO pins to connect the input buttons and PWM output. Popular choices include Xilinx Artix-7 or Basys-3.

Pinout

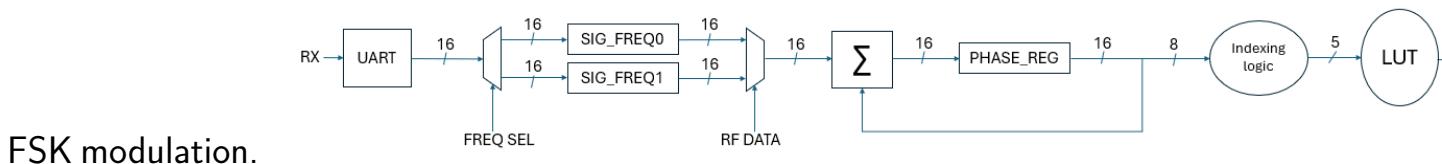
#	Input	Output	Bidirectional
0	clk	PWM_OUT	
1	ui_in[0]		
2	ui_in1		
3			
4			
5			
6			
7			

DaDDS [48]

- Author: Jeremiasz Dados
- Description: 8-bit High-Frequency Direct Digital Synthesizer with OOK and FSK modulation
- GitHub repository
- HDL project
- Mux address: 48
- Extra docs
- Clock: 60000000 Hz

How it works

DaDDS is a high-frequency DDS meant to be used as an RF transmitter using OOK or



FSK modulation.

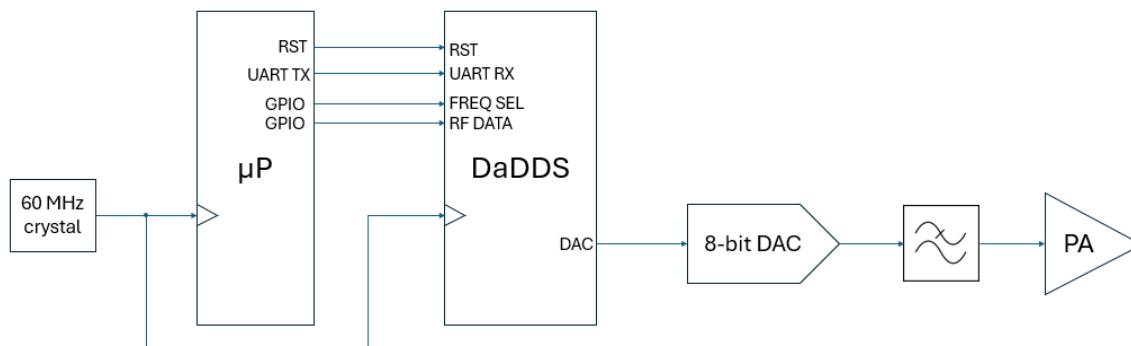
How to test



Baud rate: 115200. System clock: 60 MHz.

- Start bit: always 0.
- Byte sel bit: 1 to program the upper byte of the 16-bit frequency register, 0 to program the lower byte.
- Data bits: number to be programmed into the upper or lower byte of the chosen frequency register.
- Stop bit: always 1.

External hardware



Typical application:

Pinout

#	Input	Output	Bidirectional
0		DAC[0]	
1		DAC1	
2		DAC2	
3	UART RX	DAC[3]	
4	FREQ SEL	DAC[4]	
5	RF DATA	DAC[5]	
6		DAC[6]	
7		DAC[7]	

Simple shift Reg [50]

- Author: test
- Description: Simple shift Reg
- GitHub repository
- HDL project
- Mux address: 50
- Extra docs
- Clock: 1 Hz

How it works

Shift left on CLK and output it on the output and read in new Date when ui[0] is 1

How to test

setup a 8 bit inbut. and set ui[0] high then whathc the output as se it shift.

External hardware

None wher used

Pinout

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

2-bit 2x2 Matrix Multiplier [64]

- Author: Kevin Ma
- Description: multiples two 2-bit 2x2 matrices
- GitHub repository
- HDL project
- Mux address: 64
- Extra docs
- Clock: 1000000 Hz

How it works

Computes matrix multiplication $AB = C$.

Standard input pins are used to input a 2-bit 2x2 matrix A as 8-bit 1x8 matrix. Bidirectional IOs, initialized as inputs, are used to input a 2-bit 2x2 matrix B as 8 bit 1x8 matrix. Standard output pins will show the result of the computation in as a 2-bit 2x2 matrix as 8-bit 1x8 matrix.

Here is the matrix position mapping to input pins. Note each value is 2-bits.

“A” top left: (0,0) -> IN7 | IN6

“A” top right: (0,1) -> IN5 | IN4

“A” bot left: (1,0) -> IN3 | IN2

“A” bot right: (1,1) -> IN1 | IN0

“B” top left: (0,0) -> IO7 | IO6

“B” top right: (0,1) -> IO5 | IO4

“B” bot left: (1,0) -> IO3 | IO2

“B” bot right: (1,1) -> IO1 | IO0

“C” top left: (0,0) -> OUT7 | OUT6

“C” top right: (0,1) -> OUT5 | OUT4

“C” bot left: (1,0) -> OUT3 | OUT2

“C” bot right: (1,1) -> OUT1 | OUT0

The logic will compute the matrix multiplication of AB, and output the result in the 8 output pins (8 bits).

Each pin corresponds to one bit.

How to test

To set a pin to 1, pull up to max voltage of the respective pin. To set a pin to 0, pull down to ground.

Pull the pins respectively to input your A and B matrices based on the mapping in the above section.

The matrix multiplication of AB will be output.

External hardware

No external hardware needed.

Pinout

#	Input	Output	Bidirectional
0	IN0	IO0	OUT8
1	IN1	IO1	OUT9
2	IN2	IO2	OUT10
3	IN3	IO3	OUT11
4	IN4	IO4	OUT12
5	IN5	IO5	OUT13
6	IN6	IO6	OUT14
7	IN7	IO7	OUT15

8b10b decoder and multiplier [65]

- Author: Mike Bell
- Description: 8b10b decoder and multiplier (HD version)
- GitHub repository
- HDL project
- Mux address: 65
- Extra docs
- Clock: 0 Hz

What is it?

This project decodes incoming 8b10b encoded data and optionally multiplies the two decoded bytes.

How it works

After reset, the 8b10b decoders look for the K.28.5 symbol 001111 1010 or 110000 0101. Once this sequence is detected the decoder indicates the stream is valid and then sets its input byte after each data symbol is received.

If a K.28.5 symbol is received when the stream is valid, then the decoder remains in the valid state but does not update its output.

If any symbol other than a data symbol or K.28.5 is received the decoder returns to the reset state until a new K.28.5 symbol is sent.

The remaining inputs allow the decoded data, or the result of multiplying the decoded data to be presented on the outputs.

How to test

Send 8b10b encoded data streams, check the outputs.

While in reset, the inputs are presented on the outputs and bidirs as differential pairs, with `out[0] = in[0]`, `out[1] = ~in[0]`, `out[2] = in[1]`, etc.

If not in reset, the output enables on the bidirectional pins are controlled by `in[7]`.

External hardware

None required

Pinout

#	Input	Output	Bidirectional
0	A 8b10b in	Out 0	Out 8
1	B 8b10b in	Out 1	Out 9
2	Decoder status	Out 2	Out 10
3	Multiply result	Out 3	Out 11
4	Multiply result (update gated)	Out 4	Out 12
5	Decoded values (registered)	Out 5	Out 13
6	Decoded values (unregistered)	Out 6	Out 14
7	Bidir output enable	Out 7	Out 15

VGA Tiny Logo (1 tile) [66]

- Author: Renaldas Zioma
- Description: Large 480x480 pixels Tiny Tapeout logo with bling and dithered colors crammed into 1 tile!
- GitHub repository
- HDL project
- Mux address: 66
- Extra docs
- Clock: 25175000 Hz

How it works

Compressed VGA Logo

How to test

Connect to VGA monitor

External hardware

TinyVGA PMOD, VGA monitor

Pinout

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

Test Design 1 [67]

- Author: Evan Armoogan
- Description: Test design, not sure what it does yet
- GitHub repository
- HDL project
- Mux address: 67
- Extra docs
- Clock: 0 Hz

How it works

This project implements a synchronous 4 bit counter. There are 3 control signals described below.

- Cp: Indicates that the counter value should be incremented on the current clock cycle
- Ep: Outputs the enable signal on the uo_out wire
- Lp: Indicates that the value on the bus should be loaded into the counter.

The counter will enumerate all values between 0 and F (15) before looping back to 0 and starting again. The counter will clear back to 0 whenever the chip is reset.

Signal	TinyTapeout I/O
Cp	ui_in1
Ep	ui_in2
Lp	ui_in[0]
Load Input	ui_in[7:4]
Counter Output	uo_out[3:0]

Note: All control signals (Cp, Ep, and Lp) are active high.

How to test

Connect any probe that allows you to read 4 bits from the hardware to uo_out. Now generate a sequence of operations that tests all of the following operations:

- Enable the output by asserting Ep
- Start counting by asserting Cp
- Pause counting by deasserting Cp

- Disable the output by deasserting Ep. Should see high impedance on the output wire
- Load a new value into the counter while paused
- Load a new value while the counter is incrementing
- Reset the chip and verify the counter is reset to 0

Some example test waveforms are attached:

- test_count: Counts from 0 up to F
- test_load: Counts and loads the value of 5 after 9 clock periods
- test_pause: Counts and pauses for 2 clock periods after 7 clock periods
- test_pause_load: Counts and pauses after 7 clock periods then loads
- test_disable: Disables counter output for 2 cycles after 9 clock periods
- test_loop: Counts from 0 up to F then loops back to 0

External hardware

No external hardware is required to run the counter. It may be helpful to have tools that allow you to easily view the output of the counter.

Pinout

#	Input	Output	Bidirectional
0	in_0	out_0	bidir_0
1	in_1	out_1	bidir_1
2	in_2	out_2	bidir_2
3	in_3	out_3	bidir_3
4	in_4	out_4	bidir_4
5	in_5	out_5	bidir_5
6	in_6	out_6	bidir_6
7	in_7	out_7	bidir_7

A simple leaky integrate and fire neuron [68]

- Author: Heather Knight
- Description: Simulation of lif neuron
- GitHub repository
- HDL project
- Mux address: 68
- Extra docs
- Clock: 0 Hz

How it works

It takes input voltages and treats that as the input current injection to the LIF neuron

How to test

Do something

External hardware

NA

Pinout

#	Input	Output	Bidirectional
0	Input current bit [0]	State variable bit [0]	
1	Input current bit 1	State variable bit 1	
2	Input current bit 2	State variable bit 2	
3	Input current bit [3]	State variable bit [3]	
4	Input current bit [4]	State variable bit [4]	
5	Input current bit [5]	State variable bit [5]	
6	Input current bit [6]	State variable bit [6]	
7	Input current bit [7]	State variable bit [7]	Spike bit

Decimation Filter for Incremental and Regular Delta-Sigma Modulators [69]

- Author: Andrea Murillo Martinez & Jaeden Chang
- Description: Decimation filter that efficiently reduces oversampled data from incremental and regular delta-sigma modulators, while preserving signal accuracy.
- GitHub repository
- HDL project
- Mux address: 69
- Extra docs
- Clock: 50000000 Hz

Overview

The decimation filter efficiently reduces the sample frequency of **Incremental** and **Regular Delta-Sigma Modulators (DSMs)** by a factor of 16. This process minimizes high-frequency noise and downsamples data, supporting effective and accurate signal processing of oversampled ADC outputs.

Specifications

- **Inputs:** 3 total
 - **Input 1 (1 bit):** ADC data input
 - **Input 2 (1 bit):** Decimation mode selection (0 = Incremental DSM, 1 = Regular DSM)
 - **Input 3 (1 bit):** Global reset
- **Output:** 16 bits total
 - **Most Significant 8 bits (MSBs):** Routed to dedicated output pins
 - **Least Significant 8 bits (LSBs):** Routed to general-purpose IO pins
- **Clock Frequency:** 50 MHz (standard operation)

Mode Selection

The decimation mode can be configured based on the DSM type:

- **Incremental DSM:** Set Input 2 to low.
- **Regular DSM:** Set Input 2 to high.

How It Works

1. **Noise Reduction and Downsampling:** The decimation filter reduces high-frequency quantization noise from DSM oversampling, delivering a downsampled output with preserved signal quality.
2. **Adaptive Output Rate:**
 - **Incremental DSM (Input 2 Low):** The output updates after accumulating 16 input samples.
 - **Regular DSM (Input 2 High):** The output updates based on an internal timing controlled by the reset signal.
3. **Output Simplification:** The filter converts a high data rate from the oversampled ADC into a manageable downsampled rate, optimizing data processing.

Operation

The decimation filter requires an initialization pulse on the global reset input upon start-up.

1. Incremental DSM Mode (Input 2 Low):

- Use the ADC's oversampling frequency as the input clock for the filter.
- Set the main reset signal to match the desired decimation rate.
- For example, with a 50 MHz ADC frequency, setting the reset signal to 25 MHz achieves a decimation factor of 2.

2. Regular DSM Mode (Input 2 High):

- The default decimation factor is set to 16.
- For customized decimation factors, follow the configuration steps in Incremental DSM mode.

Testing Procedure

1. Hardware Setup:

- Connect a 1-bit ADC output to Input 1.
- Set Input 2 to low for Incremental DSM or high for Regular DSM.

2. Verification:

- **Incremental DSM:** Set Input 2 low, connect a clock to the reset input, and observe decimated output changes.

- **Regular DSM:** Set Input 2 high, then observe the decimated output, which updates at a rate of 16 samples.

Output Configuration

The decimation filter's 16-bit output is divided as follows:

- **Most Significant 8 Bits (MSBs):** Directed to dedicated output pins.
- **Least Significant 8 Bits (LSBs):** Directed to general-purpose IO pins.

Compatibility

This filter is compatible with 1-bit output ADCs, either **Incremental** or **Regular Delta-Sigma Modulator (DSM)** types.

Pinout

#	Input	Output	Bidirectional
0	X	decimation_output[8]	decimation_output[0]
1	type_dec	decimation_output[9]	decimation_output1
2	global_reset	decimation_output[10]	decimation_output2
3		decimation_output[11]	decimation_output[3]
4		decimation_output[12]	decimation_output[4]
5		decimation_output[13]	decimation_output[5]
6		decimation_output[14]	decimation_output[6]
7		decimation_output[15]	decimation_output[7]

Leaky Neuron Network [70]

- Author: Matthew Randall
- Description: makes a leaky neuron network
- GitHub repository
- HDL project
- Mux address: 70
- Extra docs
- Clock: 0 Hz

How it works

This project is a spiking neural network based on the leaky integrate-and-fire (LIF) neuron model, implemented in Verilog. The design includes three input neurons that each receive a 5-bit input signal representing incoming current. Each neuron accumulates this input over time, and when it reaches a specific threshold, the neuron “spikes,” producing an output signal.

The spike signals from these three input neurons are then combined, with each neuron’s spike weighted according to its contribution, and sent to an output neuron. The output neuron integrates these weighted inputs and produces a spike output when the accumulated value exceeds its threshold. This final spike output represents a decision or response of the network to the inputs, making it suitable for basic pattern recognition or response simulations.

How to test

1. Simulation: Use a Verilog simulator (e.g., ModelSim or Verilator) to test the neuron network. Apply various 5-bit input values to each of the three input neurons and observe when each neuron spikes in response. Check that the output neuron responds as expected to the combined weighted inputs by spiking when the sum of weighted spikes exceeds its threshold.
2. Hardware Testing (if implemented on FPGA): Synthesize the design and program it onto an FPGA. Connect switches or buttons to provide input signals for each neuron. Observe the final spike output on an LED to visualize when the output neuron spikes, or use an oscilloscope to verify spike timings and patterns for more detailed analysis.

External hardware

LEDs are used to display the spike outputs of each neuron, allowing visual feedback of the spiking activity. Switches or buttons provide manual 5-bit inputs to each neuron for testing and simulation on hardware. PMOD or GPIO headers (optional) can be used if testing on an FPGA, allowing GPIO pins for input signals or connections to external displays for monitoring neuron activity.

Pinout

#	Input	Output	Bidirectional
0	input 1	output 1	input/output 1
1	input 2	output2	input/output 2
2	input 3	output3	input/output 3
3	input 4	output4	input/output 4
4	input 5	output5	input/output 5
5	input 6	output6	input/output 6
6	input 7	output7	input/output 7
7	input 8	output8	input/output 8

adder-accumulator [71]

- Author: Damir Gazizullin, Owen Golden
- Description: 8-bit ripple adder and the complementary accumulator register
- GitHub repository
- HDL project
- Mux address: 71
- Extra docs
- Clock: 50000000 Hz

How it works

This repository contains the circuit for a basic 8-bit ripple adder and its complementary accumulator register. The adder assumes 2s complement inputs and thus supports addition and subtraction. It pushes the result to the bus via tri-state buffer. It also includes a zero flag to support conditional operation as well as a carry flag. These flags are synchronized to the rising edge of the clock and are updated when the adder outputs to the bus.

The accumulator register functions to store the output of the adder. It is synchronized to the positive edge of the clock. The accumulator loads and outputs its value from the bus and is connected via tri-state buffer. The accumulator's current value is always available as an output (and usually connected to the Register A input of the ALU) These two modules work in tandem and are a part of a larger project which includes peripheral and control blocks to ultimately create a functioning, basic, 8-bit CPU.

IO Table: Accumulator (A) Register

Name	Verilog	Description	I/O	Width (bits)	Active
clk	clk	Clock Signal	Input	1	Rising edge
bus	bus	Connection to bus	IO	8	NA
load	nLa	Load from Bus	Input	1	0
enable_out	Ea	Output to Bus	Input	1	1
Register A	regA	Accumulator Register	Output	8	NA
reset	rst_n	Reset Signal	Input	1	0

IO Table: ALU (Adder/Subtractor)

Name	Verilog	Description	I/O	Width (bits)	Active
clk	clk	Clock Signal	Input	1	Rising edge
enable_out	Eu	Output to Bus	Input	1	1
Register A	reg_a	Accumulator Register	Input	8	NA
Register B	reg_b	Register B	Input	8	NA
subtract	sub	Perform Subtraction	Input	1	1
bus	bus	Connection to bus	Output	8	NA
Carry Out	CF	Carry-out flag	Output	1	1
Result Zero	ZF	Zero flag	Output	1	1

Tests and Expected Functionality

The waveform in Figure 1 shows the loading and output functionality of the accumulator (RegA). The yellow marker displays the load functionality of the accumulator: On the rising edge of the clock, when nLa is low, the value from the bus is loaded onto the RegA.

The red marker displays the output functionality of the accumulator: On the rising edge of the clock, when Ea becomes high, the value from the accumulator is pushed onto the bus.

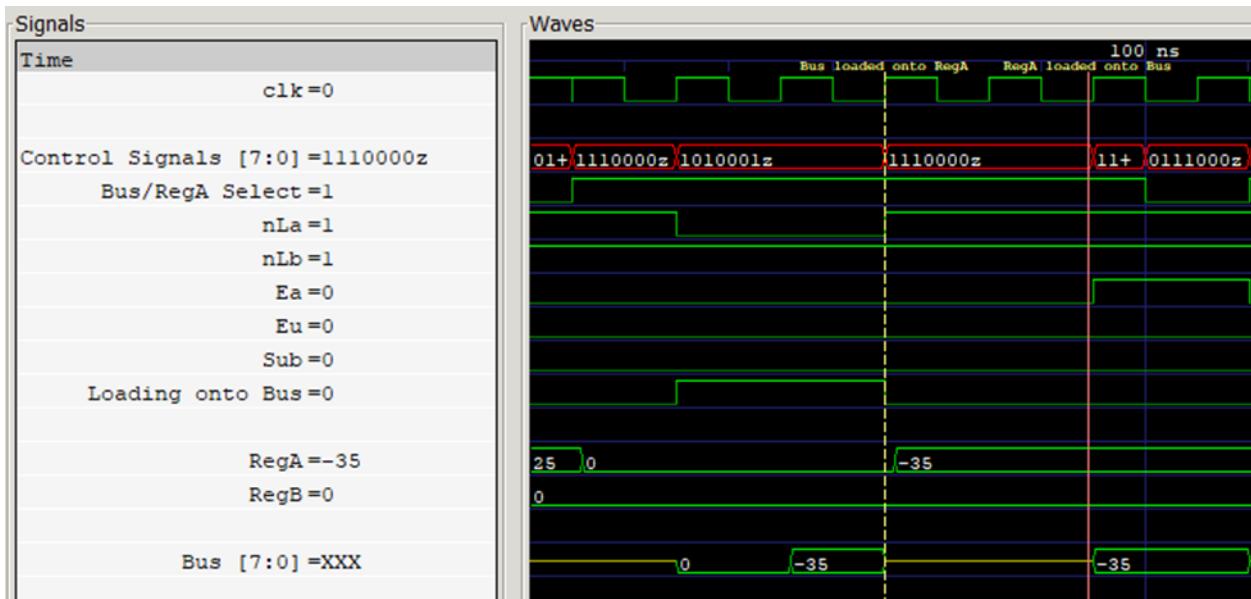


Figure 1: Accumulator Load onto bus and push onto bus

The waveform in Figure 2 demonstrates basic addition done by the adder. Note that at the red marker, Sub is low, thus addition is being performed. The addition is done asynchronously, and the value of Sum goes from 60 ($60 + 0$) to -10 ($60 + -70$). At the yellow marker, Ea is high, and thus the result of the addition is pushed onto the bus. Note that the Sum signal is internal.

Similarly, the waveform in Figure 3 demonstrates basic subtraction by the adder. Note that at the red marker, Sub is high, thus subtraction is being performed. In this case, the rest 9-11 is calculated asynchronously resulting in -2. At the yellow marker, when Eu is set high, the result is pushed onto the bus.

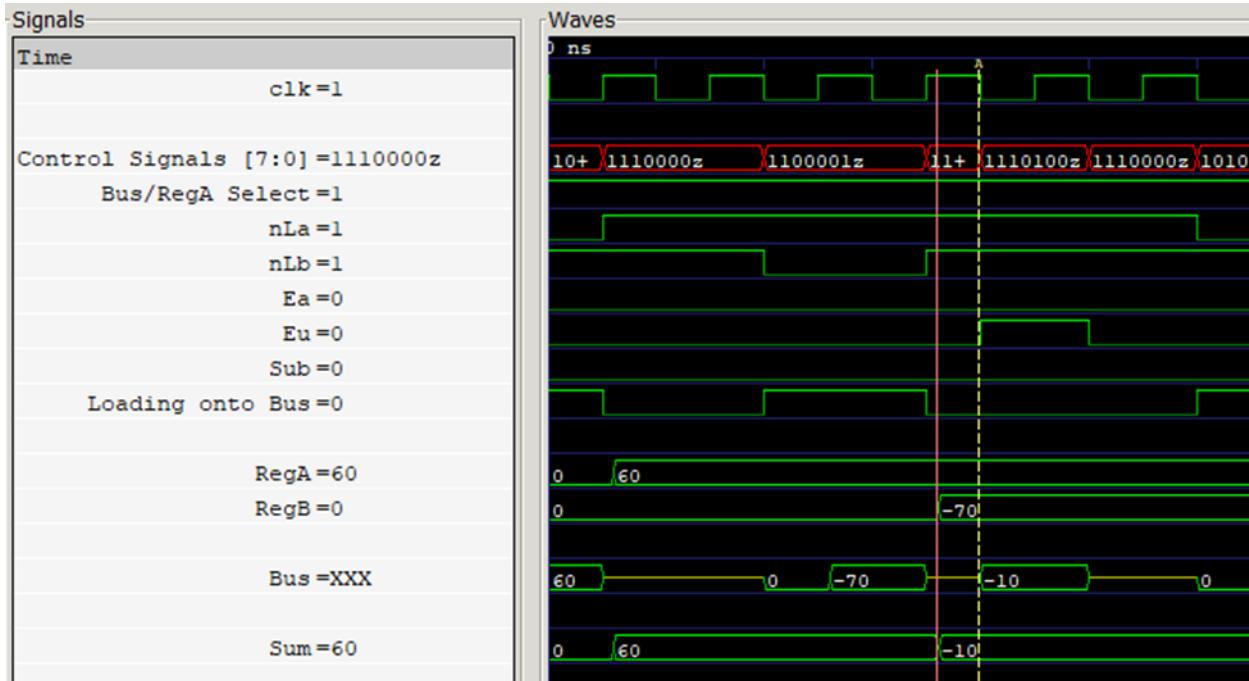


Figure 2: Addition and Output onto Bus

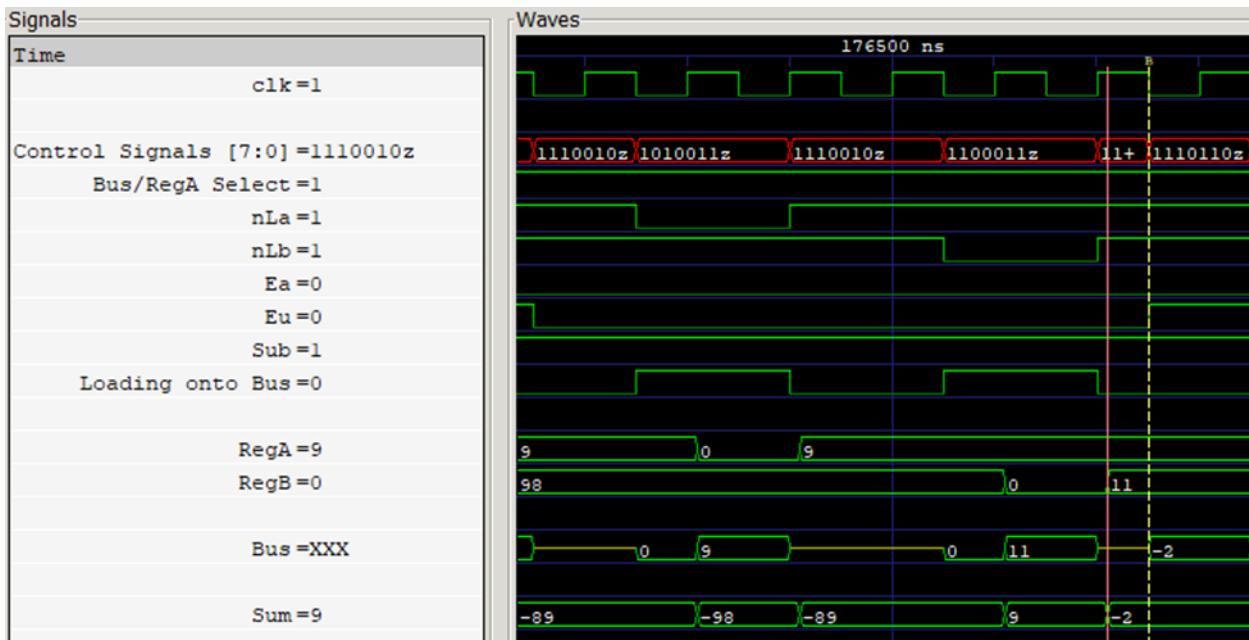


Figure 3: Subtraction and Output onto Bus

The waveform in Figure 4 demonstrates the functionality of ZF (zero flag). As described above, at the red marker, the subtraction 42-42 is performed, resulting in 0. The result is pushed to the bus when Ea is set high. At the rising edge of the clock, when Ea remains high, ZF is also made high, indicating that the result of the operation (in this case, subtraction), was zero.

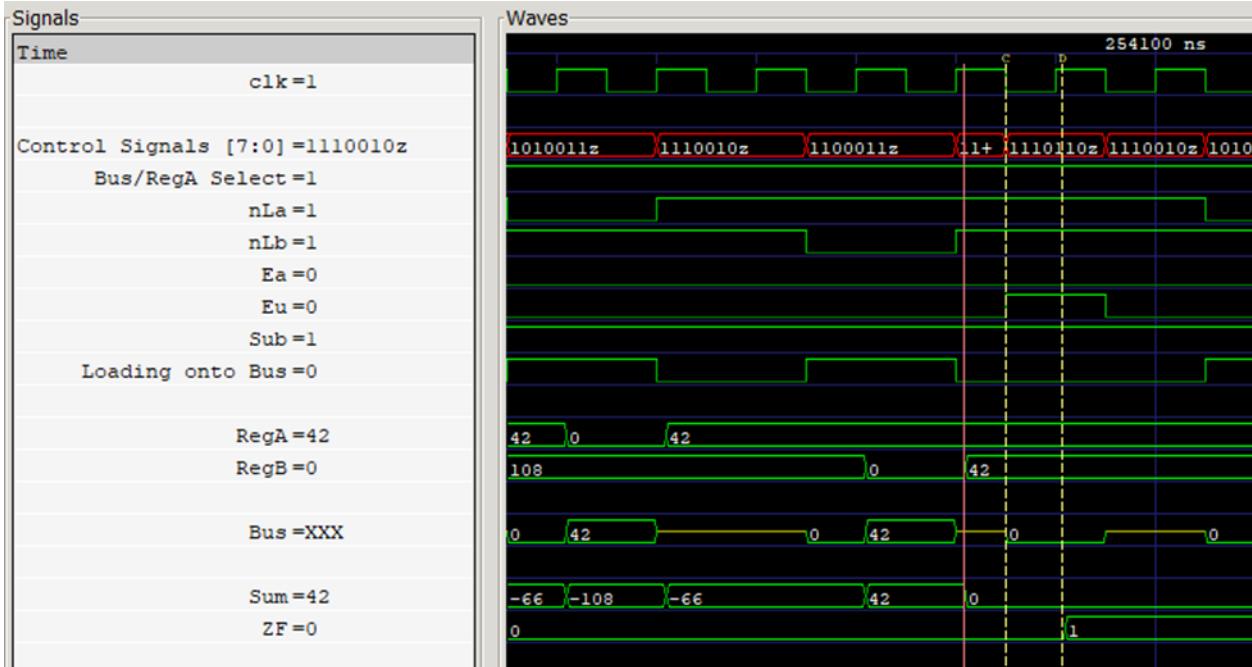


Figure 4: Zero Flag Functionality of Adder

Description of Testbenches

These modules have been tested under six Testbenches. For the purposes of the tests, all random numbers are between 0 and 255. The tests are briefly detailed below:

Adder Tests:

adder_test_addition_range: This test computes the addition of 50 random pairs of numbers and checks to see if the addition was correct.

adder_test_subtraction_range: This test computes the subtraction of 50 random pairs of numbers and checks to see if the subtraction was correct.

adder_test_addsub_range: This test computes either addition or subtraction (randomly determined before each operation) of 50 random pairs of numbers and checks to see if the result is correct.

Accumulator Tests:

accumulator_test_randint: This test loads a random number from the bus onto the accumulator, and checks whether the values on the bus and in the accumulator match.

accumulator_test_randint_out: This test loads a random number from the bus onto the accumulator and checks whether the values on the bus and in the accumulator match. It then outputs the value of the accumulator onto the bus and checks whether the values on the bus and in the accumulator match as expected.

accumulator_test_shuffled_range: This test performs the accumulator_test_randint_out test consequently with 25 randomly chosen non-repeating values

Pinout

#	Input	Output	Bidirectional
0	bus[0] if ~(E_a Eu)	bus[0]/regA[0], bus_regA_sel = 1/0	
1	bus1 if ~(E_a Eu)	bus1/regA1, bus_regA_sel = 1/0	
2	bus2 if ~(E_a Eu)	bus2/regA2, bus_regA_sel = 1/0	
3	bus[3] if ~(E_a Eu)	bus[3]/regA[3], bus_regA_sel = 1/0	
4	bus[4] if ~(E_a Eu)	bus[4]/regA[4], bus_regA_sel = 1/0	
5	bus[5] if ~(E_a Eu)	bus[5]/regA[5], bus_regA_sel = 1/0	
6	bus[6] if ~(E_a Eu)	bus[6]/regA[6], bus_regA_sel = 1/0	
7	bus[7] if ~(E_a Eu)	bus[7]/regA[7], bus_regA_sel = 1/0	

Neuromorphic Hardware for SNN LSTM [72]

- Author: Hunter Schweiger
- Description: efficient neuromorphic hardware for running a SNN LSTM unit
- GitHub repository
- HDL project
- Mux address: 72
- Extra docs
- Clock: 50000000 Hz

Neuromorphic Hardware for SNN LSTM

How it works This LSNN (Leaky Spike Neural Network) implementation features:

- 12-bit membrane potential with configurable decay (DECAY_FACTOR = 1/4)
- Adaptive threshold mechanism with learning rate control
- 3-cycle refractory period after spike generation
- 7-bit spike counter for monitoring activity
- Base threshold of 100 units with dynamic adaptation

The design operates through several key mechanisms:

1. Membrane Dynamics:

- Integrates 8-bit input current
- Applies leaky decay of 1/4 per cycle
- Resets to 0 after spike

2. Adaptation Mechanism:

- Learning-enabled threshold adjustment (controlled by uio_in[0])
- Adaptation increases with each spike
- Gradual decay when not spiking

3. Output Monitoring:

- uo_out[7]: Refractory state indicator
- uo_out[6:0]: Current membrane potential
- uio_out[7]: Spike output
- uio_out[6:0]: Spike counter

How to test

Testing procedure:

1. Reset ($\text{rst_n} = 0$):
 - Verify all state variables reset to 0
 - Threshold should reset to base value (100)
2. Basic Operation:
 - Apply input current through $\text{ui_in}[7:0]$
 - Monitor membrane potential on $\text{uo_out}[6:0]$
 - Observe spike generation on $\text{uo_out}[7]$
 - Check refractory period indicator on $\text{uo_out}[7]$
3. Learning Mode:
 - Set $\text{ui_in}[0]$ to enable learning
 - Verify threshold adaptation after spikes
 - Monitor spike frequency changes
4. Performance Verification:
 - Track spike count through $\text{uo_out}[6:0]$
 - Verify proper threshold adjustment
 - Test different input current levels

External Hardware None required - all testing can be done through digital I/O

Pinout

#	Input	Output	Bidirectional
0	Input current bit [0]	State variable bit [0]	
1	Input current bit 1	State variable bit 1	
2	Input current bit 2	State variable bit 2	
3	Input current bit [3]	State variable bit [3]	
4	Input current bit [4]	State variable bit [4]	
5	Input current bit [5]	State variable bit [5]	
6	Input current bit [6]	State variable bit [6]	
7	Input current bit [7]	State variable bit [7]	Spike bit [7]

ECE 298A 8-Bit CPU Control Block [73]

- Author: Siddharth Nema & Gerry Chen
- Description: Generates the control signals required for other CPU sub blocks
- GitHub repository
- HDL project
- Mux address: 73
- Extra docs
- Clock: 50000000 Hz

How it works

This project implements the control block of an 8-bit CPU design building off the SAP-1.

The control block is implemented using a 6 stage sequential counter for sequencing micro-instructions, and a LUT for corresponding op-code to operation(s).

Supported Instructions

Mnemonic	Opcode	Function
HLT	0x0	Stop processing
NOP	0x1	No operation
ADD {address}	0x2	Add B register to A register, leaving result in A
SUB {address}	0x3	Subtract B register from A register, leaving result in A
LDA {address}	0x4	Put RAM data at {address} into A register
OUT	0x5	Put A register data into Output register and display
STA {address}	0x6	Store A register data in RAM at {address}
JMP {address}	0x7	Change PC to {address}

Instruction Notes

- All instructions consist of an opcode (most significant 4 bits), and an address (least significant 4 bits, where applicable)

Control Signal Descriptions

Control Signal	Array	Component	Function
CP	14	PC	Increments the PC by 1
EP	13	PC	Enable signal for PC to drive the bus
LP	12	PC	Tells PC to load value from the bus
nLma	11	MAR	Tells MAR when to load address from the bus
nLmd	10	MAR	Tells MAR when to load memory from the bus
nCE	9	RAM	Enable signal for RAM to drive the bus
nLr	8	RAM	Tells RAM when to load memory from the MAR
nLi	7	IR	Tells IR when to load instruction from the bus
nEi	6	IR	Enable signal for IR to drive the bus
nLa	5	A Reg	Tells A register to load data from the bus
Ea	4	A Reg	Enable signal for A register to drive the bus
Su	3	ALU	Activate subtractor instead of adder
Eu	2	ALU	Enable signal for Adder/Subtractor to drive the bus
nLb	1	B Reg	Tells B register to load data from the bus
nLo	0	Output Reg	Tells Output register to load data from the bus

Sequencing Details

- The control sequencer is negative edge triggered, so that control signals can be steady for the next positive clock edge, where the actions are executed.
- In each clock cycle, there can only be one source of data for the bus, however any number components can read from the bus.
- Before each run, a CLR signal is sent to the PC and the IR.

Instruction Micro-Operations

Stage	HLT	NOP	STA	JMP
T0	Ep, nLma	Ep, nLma	Ep, nLma	Ep, nLma
T1	Cp	Cp	Cp	Cp
T2	nCE, nLi	nCE, nLi	nCE, nLi	nCE, nLi
T3	**	-	nEi, nLma	nEi, Lp
T4		-	Ea, nLmd	-
T5		-	nLr	-

Stage	LDA	ADD	SUB	OUT
T0	Ep, nLma	Ep, nLma	Ep, nLma	Ep, nLma

Stage	LDA	ADD	SUB	OUT
T1	Cp	Cp	Cp	Cp
T2	nCE, nLi	nCE, nLi	nCE, nLi	nCE, nLi
T3	nEi, nLma	nEi, nLma	nEi, nLma	Ea, nLo
T4	nCE, nLa	nCE, nLb	nCE, nLb	-
T5	-	Eu, nLa	Su, Eu, nLa	-

Instruction Micro-Operations Notes

- First three micro-operations are common to all instructions.
- NOP instruction executes only the first three micro-operations.
- HLT instruction transitions to a holding stage after T3, preventing the system for continuing

IO Table

Name	Description	I/O	Width	Trigger
clk	Clock signal	I	1	Edge Transition
rst_n	Set stage to 0	I	1	Active Low
ui_in[3:0]	Opcode	I	4	NA
uo_out[7]	If 1, the system is halted	O	1	Active High
uo_out[6:0]	control_signals[14:8]	O	7	NA
ui_out[7:0]	control_signals[7:0]	O	8	NA
ui_oe[7:0]	All Bidirectional pins are outputs	O	8	NA
ui_in[7:0]	Unused	I	8	NA
ena	Unused	I	1	Active High

IO Table Notes

- See Control Signal Descriptions for the list of output control signals, and their correspondance in the control_signal vector

How to test

The control block can be tested by:

- Providing an opcode through the ui_in[3:0] input pins.

- Monitoring the `uo_out[7:0]` and `uio_out[7:0]` output pins for the control signals and halt status
- For a given opcode, follow its Instruction Micro-Operation table to validate the control signal sequences
- Consider using a logic analyzer to generate a waveform and analyze the stages, or slow down the clock to manually observe the control signals at various times

Pinout

#	Input	Output	Bidirectional
0	<code>opcode[0]</code>	<code>SIG_RAM_LOAD_N</code>	<code>SIG_OUT_LOAD_N</code>
1	<code>opcode1</code>	<code>SIG_RAM_EN_N</code>	<code>SIG_REGB_LOAD_N</code>
2	<code>opcode2</code>	<code>SIG_MAR_MEM_LOAD_N</code>	<code>SIG_REGB_EN</code>
3	<code>opcode[3]</code>	<code>SIG_MAR_ADDR_LOAD_N</code>	<code>SIG_ADDER_SUB</code>
4		<code>SIG_PC_LOAD</code>	<code>SIG_REGA_EN</code>
5		<code>SIG_PC_EN</code>	<code>SIG_REGA_LOAD_N</code>
6		<code>SIG_PC_INC</code>	<code>SIG_IR_EN_N</code>
7		<code>halted</code>	<code>SIG_IR_LOAD_N</code>

RISCV Processor Design [74]

- Author: Nishanth Kotla
- Description: RISCV Processor Design
- GitHub repository
- HDL project
- Mux address: 74
- Extra docs
- Clock: 64000000 Hz

Project Datasheet: RISCV Processor

Overview The tt_um_Nishanth_RISCV module is a simple, basic processor (or computational unit) designed in Verilog. It operates on a small subset of instructions similar to a RISC-V architecture, with the ability to decode instructions, perform arithmetic or logical operations, and interact with registers and external I/O. This module serves as a building block for a more complex processor design.

How it Works This simple processor module works by fetching instructions, decoding them into different fields, performing operations using the ALU and register file, and finally generating the result. The design is flexible enough to allow for expansion, such as adding memory operations, additional instructions, or more complex control logic, which would be necessary for a complete processor design.

####Summary of How the Processor Works
Fetch the instruction: The instruction is provided as two 8-bit inputs (ui_in and uio_in), forming a 16-bit instruction.
Decode the instruction: The instruction is split into opcode, register addresses (rs1, rs2, rd), function codes (funct3, funct2), and an immediate value (imm).
Register Read: The specified registers (rs1, rs2) are read from the register file.
ALU Operation: The ALU performs the operation based on the decoded instruction (using operands from registers or the immediate value).
Write-back to Register File: The result of the ALU operation (or immediate value) is written back to the register file if the instruction allows it.
Generate the Output: The result is placed on uo_out, and depending on the opcode, might come from the register file or ALU.

How to Test By writing a testbench with cocotb and applying various test cases, we can verify the functionality of your “tt_um_KoushikCSN_RISCV” processor ensuring that all parts of the processor (instruction decoding, ALU, register file, etc.) are tested under different scenarios by varying the Input and IO ports.

Pinout

#	Input	Output	Bidirectional
0	instruction[0]	result[0]	instruction[8]
1	instruction1	result1	instruction[9]
2	instruction2	result2	instruction[10]
3	instruction[3]	result[3]	instruction[11]
4	instruction[4]	result[4]	instruction[12]
5	instruction[5]	result[5]	instruction[13]
6	instruction[6]	result[6]	instruction[14]
7	instruction[7]	result[7]	instruction[15]

LFSR Encrypter [75]

- Author: Mitchell Tansey
- Description: Simple LFSR data encrypter. Takes data in and xor's it with an lfsr output to encrypt data.
- GitHub repository
- HDL project
- Mux address: 75
- Extra docs
- Clock: 0 Hz

How it works

Takes in data in, and xor's it with a random number generated from a LFSR.

How to test

In order to test functionality of this physically, you can take the LFSR value from the bidirectional I/O and XOR it with the encryption. This will decrypt the output which you can check to see if it was the same as the input. As for my testbench, I manually calculated the LFSR value for certain clock cycles and checked the expected encrypted value.

External hardware

N/A

Pinout

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

RISCV Processor Design [76]

- Author: KOUSHIK CSN
- Description: RISCV Processor Design
- GitHub repository
- HDL project
- Mux address: 76
- Extra docs
- Clock: 64000000 Hz

Project Datasheet: RISCV Processor Design

Overview The tt_um_KoushikCSN_RISCV module is a simple, basic processor (or computational unit) designed in Verilog. It operates on a small subset of instructions similar to a RISC-V architecture, with the ability to decode instructions, perform arithmetic or logical operations, and interact with registers and external I/O. This module serves as a building block for a more complex processor design.

How it Works This simple processor module works by fetching instructions, decoding them into different fields, performing operations using the ALU and register file, and finally generating the result. The design is flexible enough to allow for expansion, such as adding memory operations, additional instructions, or more complex control logic, which would be necessary for a complete processor design.

####Summary of How the Processor Works
Fetch the instruction: The instruction is provided as two 8-bit inputs (ui_in and uio_in), forming a 16-bit instruction.
Decode the instruction: The instruction is split into opcode, register addresses (rs1, rs2, rd), function codes (funct3, funct2), and an immediate value (imm).
Register Read: The specified registers (rs1, rs2) are read from the register file.
ALU Operation: The ALU performs the operation based on the decoded instruction (using operands from registers or the immediate value).
Write-back to Register File: The result of the ALU operation (or immediate value) is written back to the register file if the instruction allows it.
Generate the Output: The result is placed on uo_out, and depending on the opcode, might come from the register file or ALU.

How to Test By writing a testbench with cocotb and applying various test cases, we can verify the functionality of your “tt_um_KoushikCSN_RISCV” processor ensuring that all parts of the processor (instruction decoding, ALU, register file, etc.) are tested under different scenarios by varying the Input and IO ports.

Pinout

#	Input	Output	Bidirectional
0	instruction[0]	result[0]	instruction[8]
1	instruction1	result1	instruction[9]
2	instruction2	result2	instruction[10]
3	instruction[3]	result[3]	instruction[11]
4	instruction[4]	result[4]	instruction[12]
5	instruction[5]	result[5]	instruction[13]
6	instruction[6]	result[6]	instruction[14]
7	instruction[7]	result[7]	instruction[15]

SkyKing Demo [77]

- Author: Nicklaus Thompson
- Description: Types some text over an image of a plane flying into the sunset
- GitHub repository
- HDL project
- Mux address: 77
- Extra docs
- Clock: 25200000 Hz

How it works

If you're seeing this, I couldn't the Clock Domain Crossing project running. This is just a TT08 demo again.

How to test

Runs automaticaly.

External hardware

VGA PMOD on UO.

Pinout

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

tiny cipher 4 bit key [78]

- Author: sriram nimmala
- Description: a tiny encryption core that encrypts based on input key
- GitHub repository
- HDL project
- Mux address: 78
- Extra docs
- Clock: 5000000 Hz

How it works

A simple encryption core with a 4 bit input 4 bit key and a 4 bit output

How to test

you can send randomized inputs of 4 bit length for the input and key and get a 4 bit output

External hardware

no external hardware but memory to send test data

Pinout

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

Two LIF Neurons with STDP Learning [79]

- Author: Sebastian Hernandez
- Description: A compact spiking neural network implementation featuring:
 - Two Leaky Integrate-and-Fire (LIF) neurons connected via plastic synapse
 - Spike-timing-dependent plasticity (STDP) for dynamic weight adjustment
 - 8-bit fixed-point arithmetic for state and weight representation
 - Real-time monitoring of spikes and synaptic weight
- GitHub repository
- HDL project
- Mux address: 79
- Extra docs
- Clock: 50000000 Hz

How it works

This design implements a simple spiking neural network using two Leaky Integrate-and-Fire (LIF) neurons connected by a spike-timing-dependent plasticity (STDP) synapse. The system consists of:

Two LIF Neurons:

Basic integrate-and-fire dynamics with leaky integration 8-bit resolution for state and current Configurable threshold (default: 150) Slower decay rate (state \gg 2) for better temporal integration First neuron receives direct current input Second neuron receives weighted input from first neuron

STDP Synapse:

Connects the two neurons with plastic weight Initial weight: 100 Potentiation: +20 when pre-spike precedes post-spike Depression: -10 when post-spike precedes pre-spike Timing window: 10 clock cycles Weight bounded between 0 and 255

Implementation Features:

Simple fixed-point arithmetic Synchronous design with clock and reset Bounded calculations to prevent overflow Modular design with separate neuron and STDP modules

How to test

The design can be tested in several ways:

Basic Functionality:

Apply current through ui_in[7:0] Monitor second neuron's state on uo_out[7:0] Observe spikes on uio_out[7:6] View synapse weight on uio_out[5:0]

Spike Generation Test:

```
verilogCopy// Example test sequence ui_in = 8'h60; // Apply strong current #100;  
// Wait for first neuron to spike ui_in = 8'h00; // Remove current #100; // Observe  
reset and decay
```

STDP Learning:

Generate regular spikes in first neuron with steady current Observe weight changes on uio_out[5:0] Monitor second neuron's response on uo_out[7:0]

External hardware

No external hardware is required for basic operation. For analysis, consider:

Logic Analyzer:

Monitor spike timing Track synaptic weight changes Verify state transitions

Signal Generator (optional):

Generate precise current injection patterns Test different input frequencies Analyze neuron response characteristics

Target Performance

The design aims to achieve:

State Resolution: 8-bit (0-255) Threshold: 150 (configurable) Weight Range: 0-255
STDP Window: 10 clock cycles Decay Rate: state » 2 (75% retention per cycle)

Resource Usage

The implementation utilizes:

Minimal combinational logic for state updates Three 8-bit registers per neuron (state, threshold) 8-bit register for synaptic weight Two 4-bit counters for STDP timing Basic arithmetic operations (addition, multiplication, shift)

Future Improvements

Possible enhancements: 1. Multiple neurons with configurable connectivity 2. Variable thresholds and decay rates 3. More sophisticated STDP rules 4. Inhibitory connections 5. Configurable timing windows 6. Additional input/output neurons 7. Parameter runtime configurability 8. More complex neural dynamics (e.g., adaptive thresholds)

Pinout

#	Input	Output	Bidirectional
0	Input current bit 0 (LSB)	Neuron 2 state bit 0 (LSB)	Synapse weight bit 0 (LSB)
1	Input current bit 1	Neuron 2 state bit 1	Synapse weight bit 1
2	Input current bit 2	Neuron 2 state bit 2	Synapse weight bit 2
3	Input current bit 3	Neuron 2 state bit 3	Synapse weight bit 3
4	Input current bit 4	Neuron 2 state bit 4	Synapse weight bit 4
5	Input current bit 5	Neuron 2 state bit 5	Synapse weight bit 5
6	Input current bit 6	Neuron 2 state bit 6	Neuron 2 spike output
7	Input current bit 7 (MSB)	Neuron 2 state bit 7 (MSB)	Neuron 1 spike output

Tutorial: Simple LIF Neuron [80]

- Author: Zack Bethel
- Description: It simulates a LIF neuron
- GitHub repository
- HDL project
- Mux address: 80
- Extra docs
- Clock: 0 Hz

How it works

it takes input voltages and treats that as the input current injection to LIF neuron

How to test

do something

External hardware

NA

Pinout

#	Input	Output	Bidirectional
0	Input current bit[0]	State variable bit[0]	
1	Input current bit1	State variable bit1	
2	Input current bit2	State variable bit2	
3	Input current bit[3]	State variable bit[3]	
4	Input current bit[4]	State variable bit[4]	
5	Input current bit[5]	State variable bit[5]	
6	Input current bit[6]	State variable bit[6]	
7	Input current bit[7]	State variable bit[7]	Spike Bit

7-Segment Byte Display [81]

- Author: Mike Goelzer
- Description: Drives a single hex digit 7-segment display based on the value of a 1-byte input
- GitHub repository
- HDL project
- Mux address: 81
- Extra docs
- Clock: 0 Hz

How it works

A two digit 7-segment display shows a hex representation of the 8-bit value provided on `ui[7:0]`. Byte `ui[7:0]` is latched when the write enable signal on `ui[0]` is high at a rising clock edge. The display is driven continuously by `uo[6:0]` with `uo[7]` controlling which digit is being driven (0=left digit, 1=right digit).

How to test

Connect the 7-segment display to the `uo[6:0]` outputs (segment 'a' is `uo[0]`, ..., segment 'g' is `uo[6]`). Connect the `uo[7]` signal to a switch to control which digit is being driven.

Connect wires to the `ui[7:0]` and `ui[0]` inputs. Ground all of `ui[7:0]` and set `ui[0]` low and verify that the display is 00. Pull `ui[0]` high and briefly pull `ui[0]` high and the display value should change to 01.

Pull `ui[0]` low again and displayed value should not change; now also pull `ui[0]` high and the display should return to 00.

External hardware

Use this two digit 7-segment display (or this one) to test the project.

#	Input	Output
---	-------	--------

Pinout

#	Input	Output
0	Byte to display on 7-segment display (rightmost / low order bit)	7-segment display (segment)
1	Byte to display on 7-segment display (next bit)	7-segment display (segme)
2	Byte to display on 7-segment display (next bit)	7-segment display (segme)
3	Byte to display on 7-segment display (next bit)	7-segment display (segme)
4	Byte to display on 7-segment display (next bit)	7-segment display (segme)
5	Byte to display on 7-segment display (next bit)	7-segment display (segme)
6	Byte to display on 7-segment display (next bit)	7-segment display (segme)
7	Byte to display on 7-segment display (leftmost / high order bit)	7-segment display (segme)

RGB Mixer demo [82]

- Author: Matt Venn
- Description: Zero to ASIC demo project
- GitHub repository
- HDL project
- Mux address: 82
- 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

Forward Pass Network for Simple ANN [83]

- Author: Arian Heidari
- Description: ANN that takes in a 4-bit value, and completes a forward pass.
- GitHub repository
- HDL project
- Mux address: 83
- Extra docs
- Clock: 50000000 Hz

How it works

The circuit takes in a 4-bit number, with each bit of the input representing an input neuron. It then completes the forward pass for the network, while also calculating the loss function (MSE). Network consists of 4 input neurons, 8 hidden neurons, and 1 output neuron.

How to test

To physically test the circuit, input a 4 bit-number into `ui_in[3:0]`. Use `ui_in[7]` to start the forward pass. The final output calculation can be seen through the output pins `{uo_out[1:0], uo_out[7:0]}`. The current state can be seen through the output pins `uo_out[7:5]`.

To simulate the circuit, change the input value of `ui_un` on line 30 of “`test.py`”. Using the `.vcd` file, analyze the output of the circuit using any waveform viewer.

External hardware

Use switches to connect to `ui_in[3:0]` (allowing for you to input a value). Connect a switch/button to `ui_in[7]` (allowing you to begin the forward pass).

Pinout

#	Input	Output	Bidirectional
0	Input bit [0]	Output Calculation [0]	
1	Input bit 1	Output Calculation 1	
2	Input bit 2	Output Calculation 2	
3	Input bit [3]	Output Calculation [3]	

#	Input	Output	Bidirectional
4		Output Calculation [4]	
5		Output Calculation [5]	
6		Output Calculation [6]	
7		Output Calculation [7]	

Priority-encoder [96]

- Author: Ole Henrik Moller
- Description: 8-bit priority encoder to decimal 7-segment code
- GitHub repository
- HDL project
- Mux address: 96
- Extra docs
- Clock: 0 Hz

How it works

Inputs and prioritizes 8-bit binary number and converts and outputs the decimal result in 7-segment format. If no bits are set (number equals zero) then a decimal point rather than a digit is displayed.

How to test

Have fun playing with switches and observe 7-segment display.

External hardware

None

Pinout

#	Input	Output	Bidirectional
0	data[0]	segments[0]	
1	data1	segments1	
2	data2	segments2	
3	data[3]	segments[3]	
4	data[4]	segments[4]	
5	data[5]	segments[5]	
6	data[6]	segments[6]	
7	data[7]	no_data	

UltraTiny-CPU [98]

- Author: Roméo Estezet
- Description: Simple 8-Bit CPU
- GitHub repository
- HDL project
- Mux address: 98
- Extra docs
- Clock: 0 Hz

How it works

The **UltraTiny-CPU** is a minimal 8-bit CPU with a small instruction set (ALU, data flow, and control flow). It has:

- **Accumulator (ACC)** as the primary register
- **Register B** as a secondary register
- **Program Counter (PC)** to fetch instructions from an internal 16-byte memory
- **Instruction Register (IR)** to decode the current operation

The CPU features a “load mode” that writes data or instructions into the memory and a “run mode” that fetches and executes those instructions:

1. Load Mode:

- Activated when $ui[7] == 1$.
- The address to write is placed on $ui[3:0]$.
- The data/instruction byte is supplied on $uo[7:0]$ and written into memory.

2. Run Mode:

- Activated when $ui[7] == 0$ and $ena == 1$.
- The CPU fetches the instruction from memory at PC, decodes it, performs the operation (arithmetic, logic, load/store, or branch), and increments or modifies PC accordingly.
- The result of arithmetic/logical operations is stored in the accumulator (ACC), and its value is driven onto $uo[7:0]$.

How to test

1. Enter Load Mode ($ui[7] = 1$):

- Provide a memory address (0 to 15) on $ui[3:0]$.
- Provide an 8-bit instruction/data on $uio[7:0]$.
- Toggle clk to store that byte into internal memory.
- Repeat for as many instructions/data bytes as needed.

2. Run the Program:

- Switch to Run Mode ($ui[7] = 0$).
- Ensure $ena = 1$ (the CPU is enabled).
- The CPU will begin fetching instructions starting at address 0.
- Observe the accumulator outputs on $uo[7:0]$ to see results of execution.

If your design environment simulates clocking and signals, you can watch the memory load process and the CPU fetch/execute cycle in a waveform viewer or on actual hardware.

External hardware

No external hardware is strictly required. The UltraTiny-CPU operates solely with its on-chip 16-byte memory and the provided I/O pins. You can optionally attach an external logic analyzer or an LED display to the accumulator outputs ($uo[7:0]$) if you want a visual indication of the CPU state. Otherwise, all interfacing can be done directly via the pin signals.

Pinout

#	Input	Output	Bidirectional
0	Memory load address bit 0 (when $ui[7]=1$)	Accumulator output bit 0	Data bus bit 0 for
1	Memory load address bit 1 (when $ui[7]=1$)	Accumulator output bit 1	Data bus bit 1 for
2	Memory load address bit 2 (when $ui[7]=1$)	Accumulator output bit 2	Data bus bit 2 for
3	Memory load address bit 3 (when $ui[7]=1$)	Accumulator output bit 3	Data bus bit 3 for
4	Unused (available for other custom inputs)	Accumulator output bit 4	Data bus bit 4 for
5	Unused (available for other custom inputs)	Accumulator output bit 5	Data bus bit 5 for
6	Unused (available for other custom inputs)	Accumulator output bit 6	Data bus bit 6 for
7	Load mode enable (1=program load, 0=run)	Accumulator output bit 7	Data bus bit 7 for

Priority-Encoded Arbiter [100]

- Author: Aditya Patra
- Description: Cycles through states based on enable state signals
- GitHub repository
- HDL project
- Mux address: 100
- Extra docs
- Clock: 10000000 Hz

How it works

This project is a priority-encoded state machine with 4 states. It has 3 enable signals - one corresponding to each of states 1, 2, and 3. In each clock cycle, the project checks whether each enable signal is on in order of priority(states 1 to 3). If an enable signal is found to be on, a counter is used to keep track of how long the signal remains on. If the signal is on for 100000 consecutive clock cycles, the corresponding state is enabled for 100000000 clock cycles.

How to test

To test the project, feed it enable signals from ui_in. ui_in[0] is the enable signal for state 1, ui_in1 is the enable signal for state 2, and ui_in2 is the enable signal for state 3. The output state enable signals are sent to the following ports: uo_out[0] is state 1 enabled, uo_out1 is state 2 enabled, and uo_out2 is state 3 enabled.

External hardware

No external hardware is necessary for the core functionality of this project, however, the reason I created it is to create a proximity warning system. To create this, you need 3 LIDAR sensors attached to the input ports and 3 buzzers attached to the output ports

Pinout

#	Input	Output	Bidirectional
0	sensor1	speaker1	
1	sensor2	speaker2	

#	Input	Output	Bidirectional
2	sensor3	speaker3	
3			
4			
5			
6			
7			

ALU in verilog [102]

- Author: Carl Emil Vinten
- Description: Simple ALU in verilog
- GitHub repository
- HDL project
- Mux address: 102
- Extra docs
- Clock: 0 Hz

How it works

Uses a push button to switch between addition subtraction, multiplication and division.

How to test

I don't know.

External hardware

switches for input, LEDS for output

Pinout

#	Input	Output	Bidirectional
0	in_2	out_0	inout_0
1	in_0	out_1	inout_1
2	in_1	out_2	inout_2
3	in_3	out_3	inout_3
4	in_4	out_4	inout_4
5	in_5	out_5	inout_5
6	in_6	out_6	inout_6
7	in_7	out_7	inout_7

Overengineered Checkers [104]

- Author: htfab
- Description: Recreation of the Checkers demo with each layer generated by a different Python-based HDL
- GitHub repository
- HDL project
- Mux address: 104
- Extra docs
- Clock: 25175000 Hz

How it works

Reproduces the classic Checkers demo by Renaldas Zioma from the VGA Playground, but each layer is generated by a separate module written in a different Python-based HDL:

- Layer A uses Amaranth
- Layer B uses MyHDL
- Layer C uses PyCDE
- Layer D uses PyRTL
- Layer E uses PyMTL3

How to test

Connect to a screen using the TinyVGA Pmod. Sit back and enjoy. Optionally toggle the `ui_in` inputs to change the colors.

External hardware

TinyVGA Pmod

#	Input	Output	Bidirectional
---	-------	--------	---------------

Pinout

#	Input	Output	Bidirectional
0	main red	R1	
1	main green	G1	
2	main blue	B1	
3	sub red	VSync	
4	sub green	R0	
5	sub blue	G0	
6	foreground	B0	
7	background	HSync	

toni_clk_gen [106]

- Author: Antoni Ruiz
- Description: Multiple clock generation
- GitHub repository
- HDL project
- Mux address: 106
- Extra docs
- Clock: 50000000 Hz

How it works

Generate multiple clocks dividng the frequency

How to test

DOnt know yet

External hardware

A clock generator

Pinout

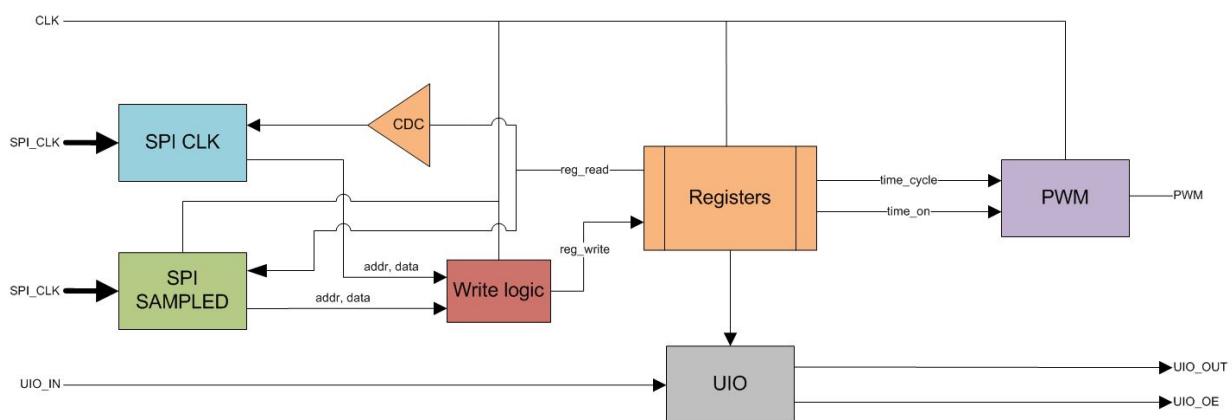
#	Input	Output	Bidirectional
0		clk_div1	
1		clk_div2	
2			
3			
4			
5			
6			
7			

spi_pwm [108]

- 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: 108
- Extra docs
- Clock: 0 Hz

How it works

This design is an SPI controlled PWM generator and 8-pin IO controller. IOs can be configured 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.



The design contains 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 occur 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:

$$\text{cycle_ticks} = T / T_{\text{clk}} = 1 \text{ ms} / (1 / 50 \text{ MHz}) = 50 \text{ MHz} * 1 \text{ ms} = 50000 \text{ ticks}$$

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

$$\text{on_ticks} = \text{cycle_ticks} * \text{duty_cycle} = 50000 * 0.33 = 16500 \text{ ticks}$$

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_in2	in	CS signal of SPI_CLK
ui_in1	in	MOSI signal of SPI_CLK
ui_in[0]	in	SCLK signal of SPI_CLK
uo_out[7:3]	out	Always 0
uo_out2	out	PWM output
uo_out1	out	MISO signal of SPI_SAMPLED
uo_out[0]	out	MISO signal of SPI_CLK
uiio_in[7:0]	in	Input signals of IOs
uiio_out[7:0]	out	Output signals of IOs
uiio_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 accessed 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 mode is CPOL = 0 and CPHA = 1. Signal changes on rising edges and is capture on falling edges.

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

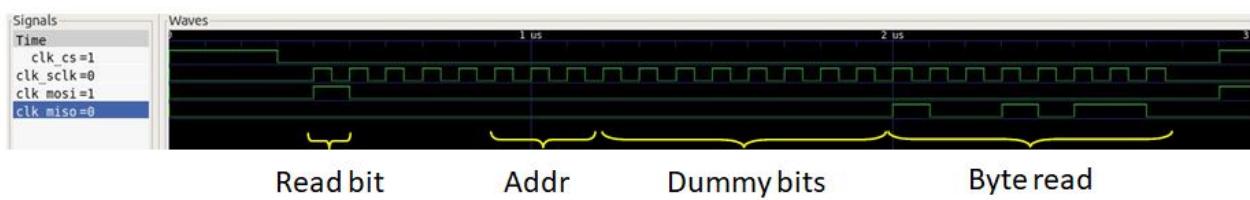
To write a register, 16 bits must be written.

- 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



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



SPI SAMPLED This interface is sampled with the system clk. As this interface needs to be sampled twice in order to avoid errors due to CDC, the frequency for the SPI_CLK must be equal or less than sys_freq/6. If this is not met, reads would be erroneous

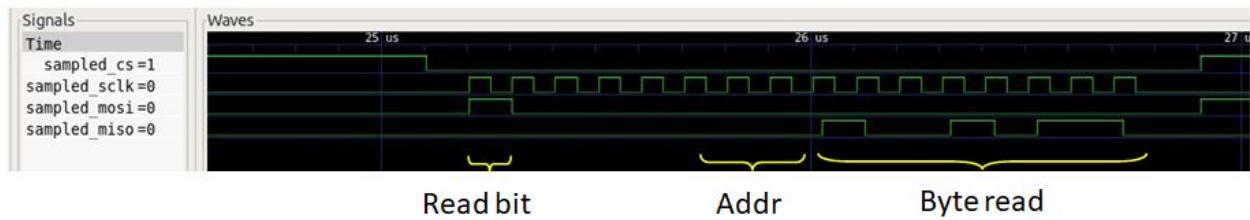
To write a register, 16 bits must be written.

- 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



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



How to test

Different tests to check all functionalities:

- SPI Reads: Read the ID register (0x00) and the byte received should be 0x96. Use both SPI_CLK and SPI_SAMPLED interface.



- SPI Writes: you can write a register different than ID register, and then read it back and check you read the value previously written. Use both SPI_CLK and SPI_SAMPLED interface.



- PWM output: Configure a desired pwm cycle in the corresponding registers TICKS_ON_LSB/MSB and TICKS_CYCLES_LSB/MSB, and activate the PWM output in PWM_CONTROL register. Check PWM output.



- External PWM on/off: Set high value on ui_in[6] and check PWM output.



- Bidir ios: Configure direction of ios with IO_DIR, and set values for the outputs in IO_VALUE, then read IO_VALUE and check correctness.



- Spare in/out: Set ui_io[7] to high and check bit 7 of PWM_CTRL is high when this design is selected.



External hardware

Some devices to perform SPI transactions

#	Input	Output	Bidirectional
---	-------	--------	---------------

Pinout

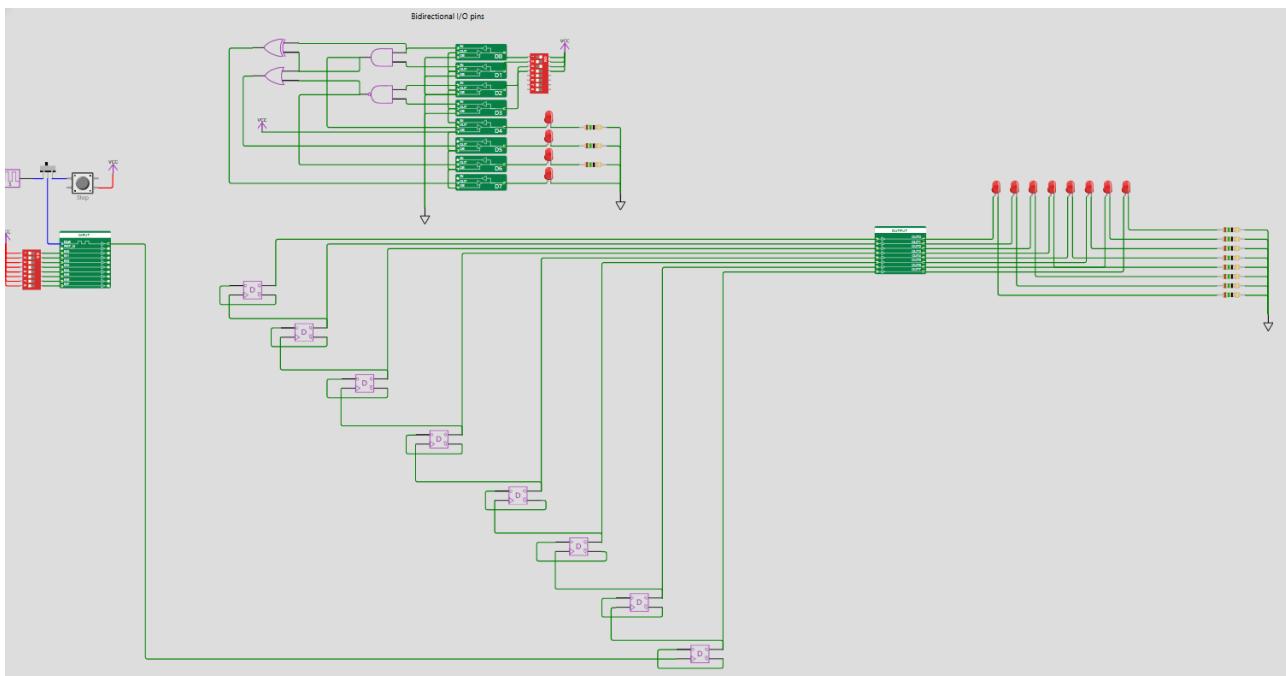
#	Input	Output	Bidirectional
0	clk_sclk	clk_miso	IO0
1	clk_mosi	sampled_miso	IO1
2	clk_cs	pwm	IO2
3	sampled_sclk		IO3
4	sampled_mosi		IO4
5	sampled_cs		IO5
6	start pwm ext		IO6
7	input and'ed with ena		IO7

BINCounterAndGates [110]

- Author: conrad franke
- Description: Binary counter with io gates
- GitHub repository
- Wokwi project
- Mux address: 110
- Extra docs
- Clock: 1 Hz

How it works

This project is fairly straightforward as it is my first TT run and I know time is of the essence ... of which in this case there is a lot of but much waiting. This circuit is a binary counter using D flip flops. There is also a gate example on the GPIO pins that are included with TT09. The way to test this is to hook up a 1HZ oscillator (you could go faster but I would recommend a 1HZ freq) to the clk pin and to provide power to the processor then watch the output as the LEDs start counting up. These will go to 15 (hex F) and then roll back to 0 (so you could even say it goes to 30... a stretch). The figure below has the circuit I created.



The I/O pins can be controlled with pushbuttons or DIP switches such as the ones that are in the schematic/circuit editor.

How to test

Flip through gates for representation of logic elements. For the binary counter attach a 1HZ oscillator and watch the LEDS start to go. To manually crawl through the binary counter, flip the oscillator circuit switch to connect to the pushbutton then step through manually with the button.

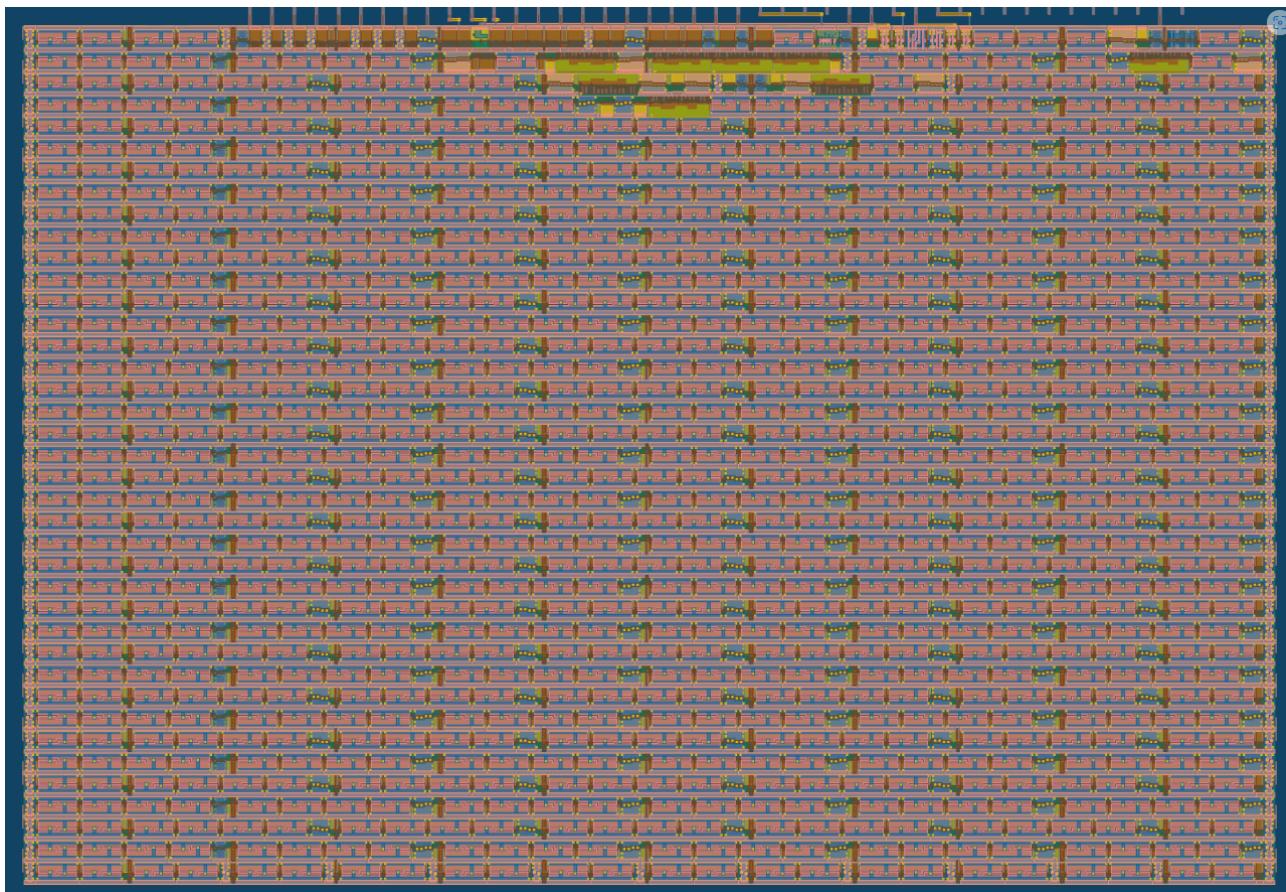
External hardware

555 Timer configured for 1HZ oscillation, A dip switch (2 SPQT would be nice TBF for the Input pins but a 8 pos SPST switch will do), 12 LED's, 12 resistors, the oscillator switch, and the step pushbutton.

Thank you tiny tapeout for this opportunity. It has been very cool building this and I look forward to making more TT IC's in the future.

Update

Build was good. Here is an image of the 2d model.



Pinout

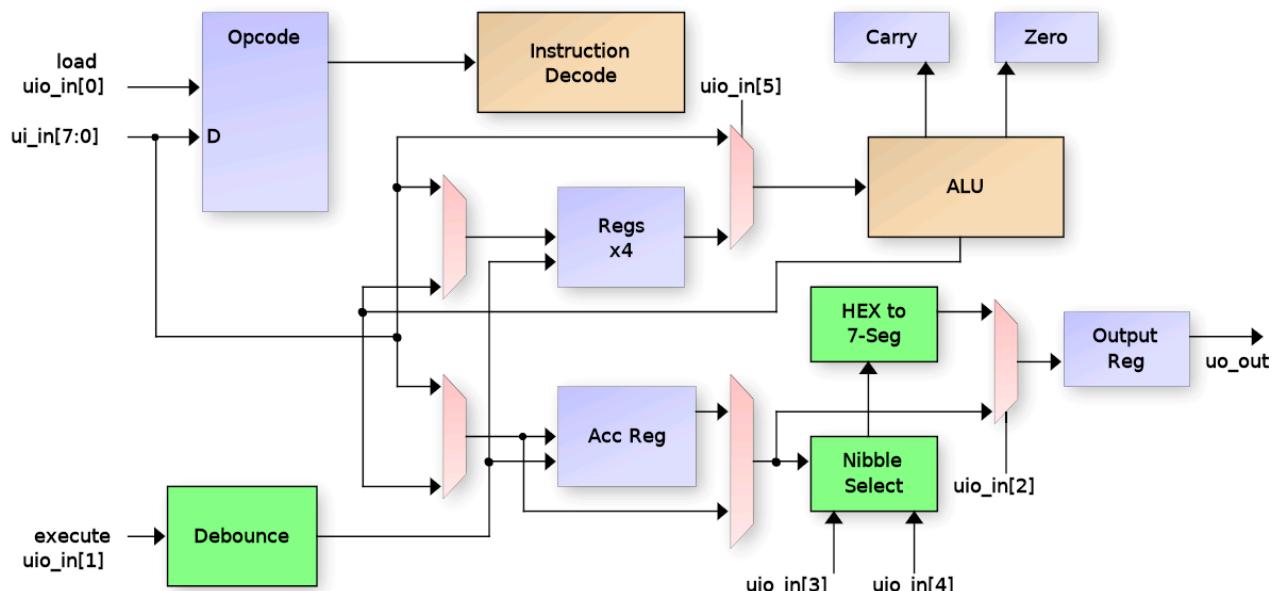
#	Input	Output	Bidirectional
0	IN0	OUT0	D0
1	IN1	OUT1	D1
2	IN2	OUT2	D2
3	IN3	OUT3	D3
4	IN4	OUT4	D4
5	IN5	OUT5	D5
6	IN6	OUT6	D6
7	IN7	OUT7	D7

tt09-pettit-wokproc-trainer [112]

- Author: Ken Pettit
- Description: An 8-bit CPU trainer
- GitHub repository
- Wokwi project
- Mux address: 112
- Extra docs
- Clock: 10000 Hz

What is WocProc Trainer?

WocProc Trainer is a partial CPU implementation coded entirely in Wokwi! While it is not a fully functional CPU capable of fetching instructions and running code, it does provide the ALU, registers and opcode decode for performing CPU operations when you “feed” it instructions. Turning it into a full CPU would require addition of a Program Counter (PC), execution state machine, flow control opcodes (jump, call, return, conditional branches, etc.), and an interface for fetching opcodes.



How it works

It works by “feeding” it opcodes and data via the `ui[7:0]` and `ui[0]` input pins and then executing them by toggling the `ui01` input pin. Some instructions require additional “Immediate Data” to be supplied via the `ui[7:0]` input pins prior to toggling the `ui01` “execute” input.

The WokProc has an 8-bit accumulator and 4 8-bit working registers and can perform ADD, SUBTRACT and the standard logical functions AND, OR, XOR and NOT, as well as shift left/right operations. It also keeps track of CARRY and ZERO bits to reflect the results of operations.

How to test

1. Provide a 10KHz clock then issue `rst_n` pulse.
2. Select the desired output mode for viewing results. For this testing, set `ui_o_in[5:2]` all LOW.

`ui_o_in2`: Selects 7-Segment (LOW) or binary (HIGH) output format `ui_o_in[4]`: Selects auto nibble / digit display (LOW) or manual (HIGH) `ui_o_in[3]`: Manual digit select when `ui_o_in[4]` is HIGH. `ui_o_in[5]`: Selects value to output (LOW = ACC reg, HIGH = new ACC load value)

3. Monitor the results using the 7-Seg display and `ui_o_out[7:6]` bits:

`ui_o_out[6]`: Indicates if CARRY bit is set

`ui_o_out[7]`: Indicates if ZERO bit is set

4. Perform an addition. After reset, the opcode register contains opcode 0x00, which is $A = A + IMM$. So supply a binary input value on `ui_in[7:0]` and toggle the EXECUTE input (`ui_o_in1`) HIGH then LOW. The 7-Seg display should display the HEX value of the sum.
5. The first addition just looked like a ‘load’ since Acc was zero from the reset. Add the value a second time (or supply a different value on `ui_in[7:0]`) and toggle the EXECUTE input again. The 7-Segment display should show the result of the addition.
6. Load register r0 from the A register. First enter the opcode (7'b1100_0000 from the opcode table) and then toggle the LOAD (`ui_in[0]`) input HIGH then LOW to load `ui_in[7:]` to the opcode register. Now toggle the EXECUTE (`ui_o_in1`) input HIGH then LOW. Register r0 should now contain the value from A.
7. Test if register r0 was loaded. First clear the Acc register (load opcode 7'b0111_0000 and EXECUTE it). The 7-Seg should show “00.”. Now load and execute the opcode to load register r0 to Acc (opcode 7'b0110_0000). The 7-Seg should show the result of the summation that was stored in r0.
8. Perform a NOT operation on the A register by loading and executing opcode 7'0111_0001. The 7-Seg should show the compliment value of what was in A.

9. Try additional operations from the opcode table by loading and executing them.
 For any opcode that uses IMM data, uio_in[7:0] inputs must be changed to the immediate data AFTER loading the opcode but BEFORE executing it.

Opcodes supported:

Opcode	Operation	Description
0000_0000	$A \leftarrow A + \text{IMM}$	Add A + immediate data
0000_1000	$A \leftarrow A + \text{IMM} + \text{Carry}$	Add with carry A + immediate
0001_0000	$A \leftarrow A - \text{IMM}$	Subtract immediate from A
0001_1000	$A \leftarrow A - \text{IMM} - \text{Borrow}$	Subtract with borrow immediate
0010_00rr	$A \leftarrow A + R[1:0]$	Add register rr to A
0010_10rr	$A \leftarrow A + R[1:0] + \text{Carry}$	Add with carry register rr
0011_00rr	$A \leftarrow A - R[1:0]$	Subtract register rr from A
0011_10rr	$A \leftarrow A - R[1:0] - \text{Borrow}$	Subtract with borrow register rr
0100_0000	$A \leftarrow \text{IMM}$	Load A with immediate data
0110_00rr	$A \leftarrow R[1:0]$	Load A from register rr
0110_01rr	$A \leftarrow A \wedge R[1:0]$	XOR A with register rr
0110_10rr	$A \leftarrow A \text{ OR } R[1:0]$	OR A with register rr
0110_11rr	$A \leftarrow A \& R[1:0]$	AND A with register rr
0111_0000	$A \leftarrow \text{Zero}$	Clear A
0111_0001	$A \leftarrow \text{!}A$	Invert (1's compliment) A'
0111_01rr	$A \leftarrow \text{!}R[1:0]$	Load A from rr compliment
0111_1000	$Cy \leftarrow 0$	Clear the carry flag
0111_1001	$Cy \leftarrow \text{!}Cy$	Compliment the carry flag
0111_1010	$\{A, Cy\} \leftarrow \{Cy, A\}$	Shift right A through Carry
0111_1011	$\{Cy, A\} \leftarrow \{A, Cy\}$	Shift left A through Carry
0111_1100	$A \leftarrow \{0, A[7:1]\}$	Shift right A
0111_1101	$A \leftarrow \{A[6:0], 0\}$	Shift left A
0111_1110	$A \leftarrow \{A[7], A[7:1]\}$	Signed shift right A
1000_00rr	$R[1:0] \leftarrow A + \text{IMM}$	Load register rr with sum
1001_00rr	$R[1:0] \leftarrow A - \text{IMM}$	Load register rr with difference
1010_00rr	$R[1:0] \leftarrow A + R[1:0]$	Load register rr with sum
1011_00rr	$R[1:0] \leftarrow A - R[1:0]$	Load register rr with difference
1100_00rr	$R[1:0] \leftarrow \text{IMM}$	Load immediate data to rr
1101_00rr	$R[1:0] \leftarrow A$	Load A to rr
1110_RRrr	$R[1:0] \leftarrow R[3:2]$	Copy register RR to rr

Selecting the Output

The `uo_out` port is used to display the state of the WocProc trainer. It can display either 7-Segment LED encoded register data or direct binary data.

<code>ui0_in2</code>	<code>uo_out</code> Format
LOW	7-Segment
HIGH	Binary

The data output to `uo_out` (either 7-Segment or binary) is selected via the `ui0_in[5]` input:

<code>ui0_in[5]</code>	<code>uo_out</code> Data
LOW	Acc Register
HIGH	ALU result (value loaded upon EXECUTE)

For 7-Segment output format, a single digit LED display is used to show both the upper and lower nibble of the selected output data. When the LOWER nibble is being displayed, the 7-Segment Decimal Point (DP) will be illuminated and when the UPPER nibble is displayed, it will be turned off, such as:

F1.

The nibble display can be configured using `ui0_in[4:3]` as follows:

<code>ui0_in[4:3]</code>	Displayed Nibble
2'b0x	Auto toggle (counter tuned for 10KHz clock)
2'b10	Lower nibble (plus DP)
2'b11	Upper nibble

Hardware needed:

Dip switches and 7-Segment LED.

Pinout

#	Input	Output	Bidirectional
0	op/imm[0]	seg_a	load_opcode
1	op/imm1	seg_b	execute_opcode
2	op/imm2	seg_c	sevenSeg_binary
3	op/imm[3]	seg_d	digit_select
4	op/imm[4]	seg_e	manual_digit
5	op/imm[5]	seg_f	digit_a_reg
6	op/imm[6]	seg_g	carry_out
7	op/imm[7]	seg_dp	zero_out

Duffy [114]

- Author: Jonathan Duffy
- Description: trying out an oscillator or delay line
- GitHub repository
- Wokwi project
- Mux address: 114
- Extra docs
- Clock: 0 Hz

How it works

This is pretty much just a string of inverters to try to make a delay line or ring oscillator. Also there's an xor gate on the bidir pins, maybe test as a mixer?

How to test

Basic DC logic on the first couple pins, couldn't describe any way other than the logic itself $OUT1 = IN3 ? (IN0 \& IN1) : IN2$ $OUT0$ and $OUT2$ are both $!OUT1$ and the rest of the OUTs should be the same as $OUT1$ $D2 = D0 \wedge D1$

External hardware

Nothing specific, switches or digital in to the input

Pinout

#	Input	Output	Bidirectional
0	IN0	OUT0	D0
1	IN1	OUT1	D1
2	IN2	OUT2	D2
3	IN3	OUT3	D3
4	IN4	OUT4	
5	IN5	OUT5	
6	IN6	OUT6	
7	IN7	OUT7	

pulse_add [128]

- Author: Jonny Edwards
- Description: a temporal add in digital
- GitHub repository
- HDL project
- Mux address: 128
- Extra docs
- Clock: 0 Hz

How it works

This is a simple circuit to calculate:

- a simple add but temporal ...
- it's one out due to enable

How to test

All tested through cocotb

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	in[0]	out[0]	
1	in1	out1	
2	in2	out2	
3	in[3]	out[3]	
4	in[4]	out[4]	
5	in[5]	out[5]	
6	in[6]	out[6]	
7	in[7]	out[7]	

nyan [130]

- Author: Peter Nørlund
- Description: Nyan Cat
- GitHub repository
- HDL project
- Mux address: 130
- Extra docs
- Clock: 25000000 Hz

How it works

The Nyan Cat animation and music on infinite repeat

How to test

Connect the TinyVGA PMOD to the Out PMOD and Mike' Audio PMOD to Bidir PMOD.

External hardware

- TinyVGA PMOD
- Mike's audio PMOD

Pinout

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

Brailliance [132]

- Author: Akshat B, Evana T, John L, Rittrija M, Riley Gu
- Description: ASCII-to-Braille Converter
- GitHub repository
- HDL project
- Mux address: 132
- Extra docs
- Clock: 5000000 Hz

How it works

Input ASCII signals to 8-bit braille outputs (First two bits are zeroed for redundancy)

How to test

Connect to FPGA and use breadboards + LEDs/Push Pull Solenoid Actuators for product demonstration

External hardware

- Breadboard
- Jumper Wires
- LEDs / Push Pull Solenoid Actuators
- Resistors

Pinout

#	Input	Output	Bidirectional
0	clk	reader1_out[0]	
1	reset	reader1_out1	
2	next	reader1_out2	
3		reader1_out[3]	
4		reader1_out[4]	
5		reader1_out[5]	
6		reader1_out[6]	
7		reader1_out[7]	

Adder with Flow Control [134]

- Author: Yuri Panchul
- Description: An adder with a separate flow control for each argument and the result
- GitHub repository
- HDL project
- Mux address: 134
- Extra docs
- Clock: 3 Hz

How it works

adder_with_flow_control design contains an adder with a separate flow control for each argument and the result.

The design is an answer to an RTL job interview question described by Yuri Panchul in an article (in Russian) on Habr website. The design is used as a part of systemverilog-homework and basics-graphics-music GitHub repositories. These repos are maintained by engineers and educators associated with the Verilog Meetup community.

In this solution to the interview question, the flow control is implemented using one of the following pipeline register/buffer modules. The choice is specified inside the *adder_with_flow_control.sv* module using the define macro *FLOW_CONTROL_BUFFER*.

- fcb_1_single_allows_back_to_back
- fcb_2_single_half_perf_no_comb_path
- fcb_3_single_for_pipes_with_global_stall
- fcb_4_wrapped_2_deep_fifo
- fcb_5_double_buffer_from_dally_harting

More details about the modules:

fcb_1_single_allows_back_to_back This module is a general-purpose flow-controlled register which allows full back-to-back performance but has a combinational path between down_rdy and up_rdy which can introduce timing problems in deep pipelines.

fcb_2_single_half_perf_no_comb_path This flow-controlled register has no combinational path at all, but cannot sustain a back-to-back stream of data. However, it requires fewer gates than *fcb_4_wrapped_2_deep_fifo* or *fcb_5_double_buffer_from_dally_harting*.

fcb_3_single_for_pipes_with_global_stall This flow-controlled register is suitable if the designer wants to always stall the whole pipeline at once, without parts of it making progress.

fcb_4_wrapped_2_deep_fifo The most high-bandwidth flow-controlled buffer among those that have no combinational path between down_rdy and up_rdy. However this solution occupies the largest area.

fcb_5_double_buffer_from_dally_harting This pipeline buffer is Yuri Panchul's edition of the code derived from *Digital Design: A Systems Approach by William James Dally and R. Curtis Harting. 2012*. It has high bandwidth and no combinational path between down_rdy and up_rdy, but not the highest possible bandwidth for this *adder_with_flow_control* design.

How to test

A self-checking testbench for the design is located in a directory *test_extra* that contains:

- *clean.bash* - a script to delete temporary files produced by *simulate.bash*.
- *simulate.bash* - a script that simulates the design together with a testbench using Icarus Verilog and produces a log *log.txt*.
- *tb.sv* - a self-checking testbench that generates a log, a status **PASS** or **FAIL**, and performance data.

After the manufacturing, the design can be manually tested in the same way it is tested in the testbench. It is important to cover the following scenarios:

- Back-to-back data.
- Argument starvation (either *A* or *B*).
- Backpressure.

External hardware

Buttons and LEDs

Pinout

#	Input	Output	Bidirectional
0	a_vld	a_rdy	a_data[0]
1	b_vld	b_rdy	a_data1
2	sum_rdy	sum_vld	a_data2
3		sum_data[0]	a_data[3]
4		sum_data1	b_data[0]
5		sum_data2	b_data1
6		sum_data[3]	b_data2
7		sum_data[4]	b_data[3]

i2c peripherals: leading zero count and fnv-1a hash [136]

- Author: Steve Jenson <stevej@gmail.com>
- Description: An implementation of HyperLogLog in Verilog for sky130
- GitHub repository
- HDL project
- Mux address: 136
- Extra docs
- Clock: 0 Hz

How it works

Fnv-1a 32-bit peripheral: send bytes via write requests, get the hash via a read request. Every read request resets the hash.

LZC: send up to 32 bits with write request, read back the number of leading zeroes with a read request.

ZeroOne: Sends the byte 0101_0101

OneZero: Sends the byte 1010_1010

How to test

Fnv-1a: Send a known set of bytes, get a known hash back.

LZC: Send 32 zeros, get the number 32 back. Send 32 1s, get 0 back

ZeroOne/OneZero: make a read request.

External hardware

i2c master device with test code. Arduino test code provided.

#	Input	Output	Bidirectional
---	-------	--------	---------------

Pinout

#	Input	Output	Bidirectional
0			(INT)
1			(RESET)
2			SCL
3			SDA
4			
5			
6			
7			

Rotary Encoder WS2812B Control [138]

- Author: Fabio Ramirez Stern
- Description: A rotary encoder controls 12 WS2812B LEDs on a ring PCB.
- GitHub repository
- HDL project
- Mux address: 138
- Extra docs
- Clock: 40000000 Hz

How it works

The rotary encoder adds/subtracts from a variable that determines which LED to turn on. Periodically, the chip sends out a signal for 12 LEDs out via `uo0`, according to the WS2812B protocol. The button connected to `in2` inverts the LEDs, whether that happens gets also indicated through `out1`. Further, the register value of the variable will be put out via `out2` to `uo5`. The colour can be activated as follows: `in3` for green, `in4` for red and `in5` for blue. Intensity is set with the remaining two bits, `in6` and `in7`.

How to test

Connect the rotary encoder outputs to `in0` and `in1`. If your rotary encoder also has a built in push button, connect that to `in2`, or use another switch with a pull down resistor. The LEDs should be wired in series. The first LED's DIN input needs to be connected to the `out0` of the chip.

Give the project a reset after power up and then rotate the encoder back and forth to see the light moving.

External hardware

1. A rotary encoder.
2. Any arrangement of 12 WS2812B like controlled LEDs. More or less will also work, you will just not get the full range, or some LEDs will stay off.

The seller called what I bought this: LED Ring 5V RGB WS2812B 12-Bit 37mm

Pinout

#	Input	Output	Bidirectional
0	rotary encoder: CLK	DOUT	
1	rotary encoder: DT	inverted	
2	rotary encoder: SW	count0	
3	green	count1	
4	red	count2	
5	blue	count3	
6	intensity1		
7	intensity2		

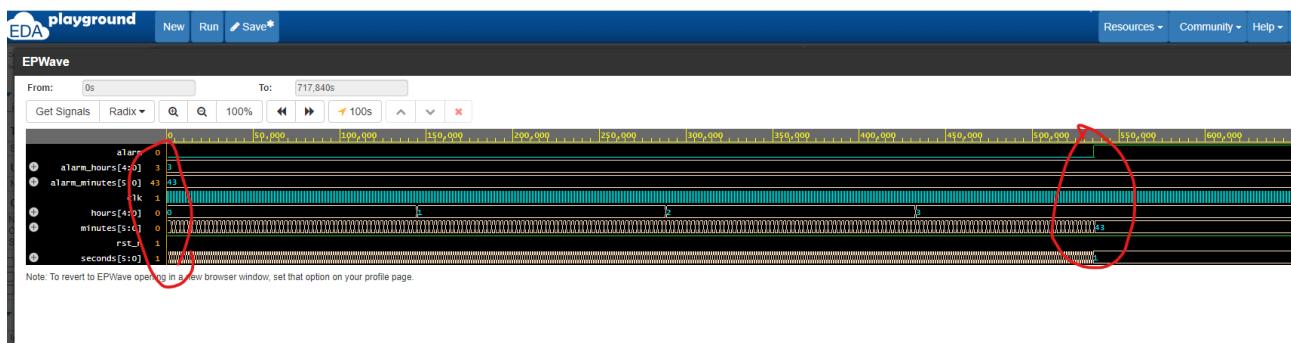
Alarm Clock [140]

- Author: Kapilan Karunakaran
- Description: A simple alarm clock
- GitHub repository
- HDL project
- Mux address: 140
- Extra docs
- Clock: 0 Hz

How it works

This project was sponsored by The MITRE Corporation and MIT/LL Beaverworks Summer Institute <https://beaverworks.ll.mit.edu/CMS/bw/bwsi>. This is a simple alarm clock. There are two inputs `alarm_hours` and `alarm_minutes`. These two inputs are to be set to values less than 23 and 59 respectively. The output pin “alarm” is expected to be asserted when the internal counters hours and minutes hit the expected `alarm_hours` and `alarm_minutes` respectively. Alarm output will be stuck at 1 until reset. Time at which Alarm is triggered are also sent out as output for comparison. Due to limited number of Inputs and outputs available, some of the inout pins are used as well. output enable pins are used to mask the inputs and outputs appropriately.

How to test



Use below testbench to input specific values to inputs and observe alarm asserted and output hours and minutes match with inputs. module tb();

```
reg clk, rst_n; reg [5:0] alarm_minutes; reg [4:0] alarm_hours;
tt_um_kapilan_alarm      dut(.ui_in({alarm_minutes[2:0],alarm_hours[4:0]}),
.uo_out(), .uio_in({5'b0,alarm_minutes[5:3]}), .uio_out(), .uio_oe(), .ena(1'b1),
.clk(clk), .rst_n(rst_n) );
```

```

initial begin rst_n = 1'b0; #100; rst_n = 1'b1; alarm_hours = 5'd3; alarm_minutes
= 6'd43; end initial begin clk = 1'b0; forever begin #20; clk = ~clk; end end initial
begin $dumpfile("alarm_dump.vcd"); $dumpvars(0,tb); #1000000; $finish; end
endmodule

```

External hardware

Pinout

#	Input	Output	Bidirectional
0	alarm_hours[0]	hours[0]	alarm_minutes[3]
1	alarm_hours1	hours1	alarm_minutes[4]
2	alarm_hours2	hours2	alarm_minutes[5]
3	alarm_hours[3]	hours[3]	
4	alarm_hours[4]	hours[4]	minutes[3]
5	alarm_minutes[0]	minutes[0]	minutes[4]
6	alarm_minutes1	minutes1	minutes[5]
7	alarm_minutes2	minutes2	

TSAL_TT [142]

- Author: Ephren Manning
- Description: FSAE EV Tractive System Active Light
- GitHub repository
- HDL project
- Mux address: 142
- Extra docs
- Clock: 8000000 Hz

How it works

This design is meant to fulfill FSAE Rule EV.5.9 on a student built electric vehicle. Rules are shown below. The digital design held on the TinyTapeout chip will take the digital value of an analog signal from an Analog Devices AD7476A 12-bit ADC chip. The value of the signal will be compared against a decided value representing that the tractive system is at 60V. Should the converted value be less than the decided value, a digital line driving a green LED will be driven high. Should the value be greater, a separate digital line driving a red LED will flash at a rate of 4 hertz.

As of the 2024 Rules Ver. 1, operation is described as follows:

EV.5.9 Tractive System Active Light - TSAL

EV.5.9.1 The vehicle must include a Tractive Systems Active Light (TSAL) that must:

- a. Illuminate when the GLV System is energized to indicate the status of the Tractive System
- b. Be directly controlled by the voltage present in the Tractive System using hard wired electronics. Software control is not permitted.
- c. Not perform any other functions.

EV.5.9.2 The TSAL may be composed of multiple lights inside a single housing

EV.5.9.3 When the voltage outside the Accumulator Container(s) exceeds T.9.1.1, the TSAL must:

- a. Be Color: Red
- b. Flash with a frequency between 2 Hz and 5 Hz

EV.5.9.4 When the voltage outside the Accumulator Container(s) is below T.9.1.1, the TSAL must:

- a. Be Color: Green
- b. Stay continuously illuminated

How to test

When testing, the digital line driving the green LED should be driven high only in the case that converted analog value is less than the comparison value. When the converted value is greater than or equal to the comparison value, the red LED should blink at a rate of 4 hertz. This requires that simulations be ran for upwards of a second to confirm LED blink speed.

External hardware

A PMOD AD1 from Digilent was used to test this project. The input/outputs on the TinyTapeout Demo board were configured so that this PMOD could be used on the top *(confirm) bidirectional port. Should a custom board be made to support functionality, the Analog Devices AD7476A or compatible 12-bit ADC converter will need to be used.

Pinout

#	Input	Output	Bidirectional
0	Comparison Value Bit 0	Green Led	Chip Select
1	Comparison Value Bit 1	Red Led	Serial Data
2	Comparison Value Bit 2		
3	Comparison Value Bit 3		Serial Clock
4	Comparison Value Bit 4		
5	Comparison Value Bit 5		
6	Comparison Value Bit 6		
7	Comparison Value Bit 7		

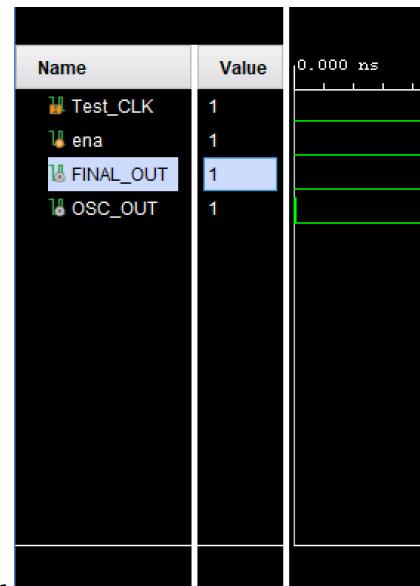
Divided Ring Oscillator [144]

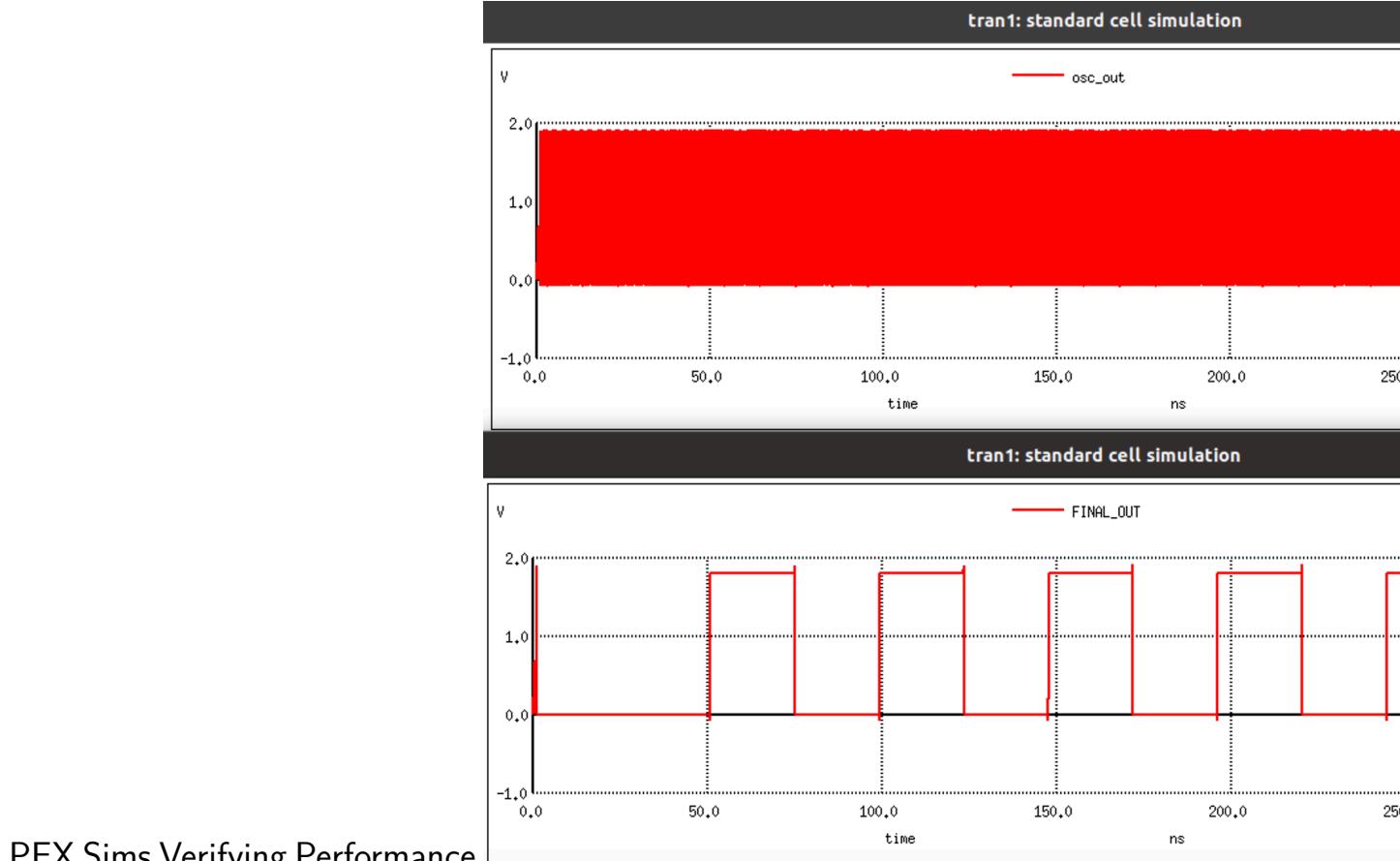
- Author: Ignatius Bezzam, Dhandeep Challagundla, Jarnail Sanghera, Russell Kim
- Description: Ring Oscillator
- GitHub repository
- HDL project
- Mux address: 144
- Extra docs
- Clock: 10000000 Hz

How it works

A ring oscillator working in the GHz range is divided to give an observable output frequency in the 20 MHz range.

Top-Level Complete Mixed-Signal Functionality Verification in Verilog





PEX Sims Verifying Performance

How to test

A supply current of 1-2 mA when enable is high indicates that the ring oscillator is functional. The final output can be observed in the 20 MHz range. Test/debug mode verifies the divider functionality at low frequency. The ring oscillator can be disabled by on-chip signals (ena = low).

External hardware

Oscilloscope (100 MHz), power supply, function generator (10 MHz, digital).

Pinout

#	Input	Output	Bidirectional
0	tst_clk	final_out	n1
1		osc_out	n3
2		ena	
3		clk	

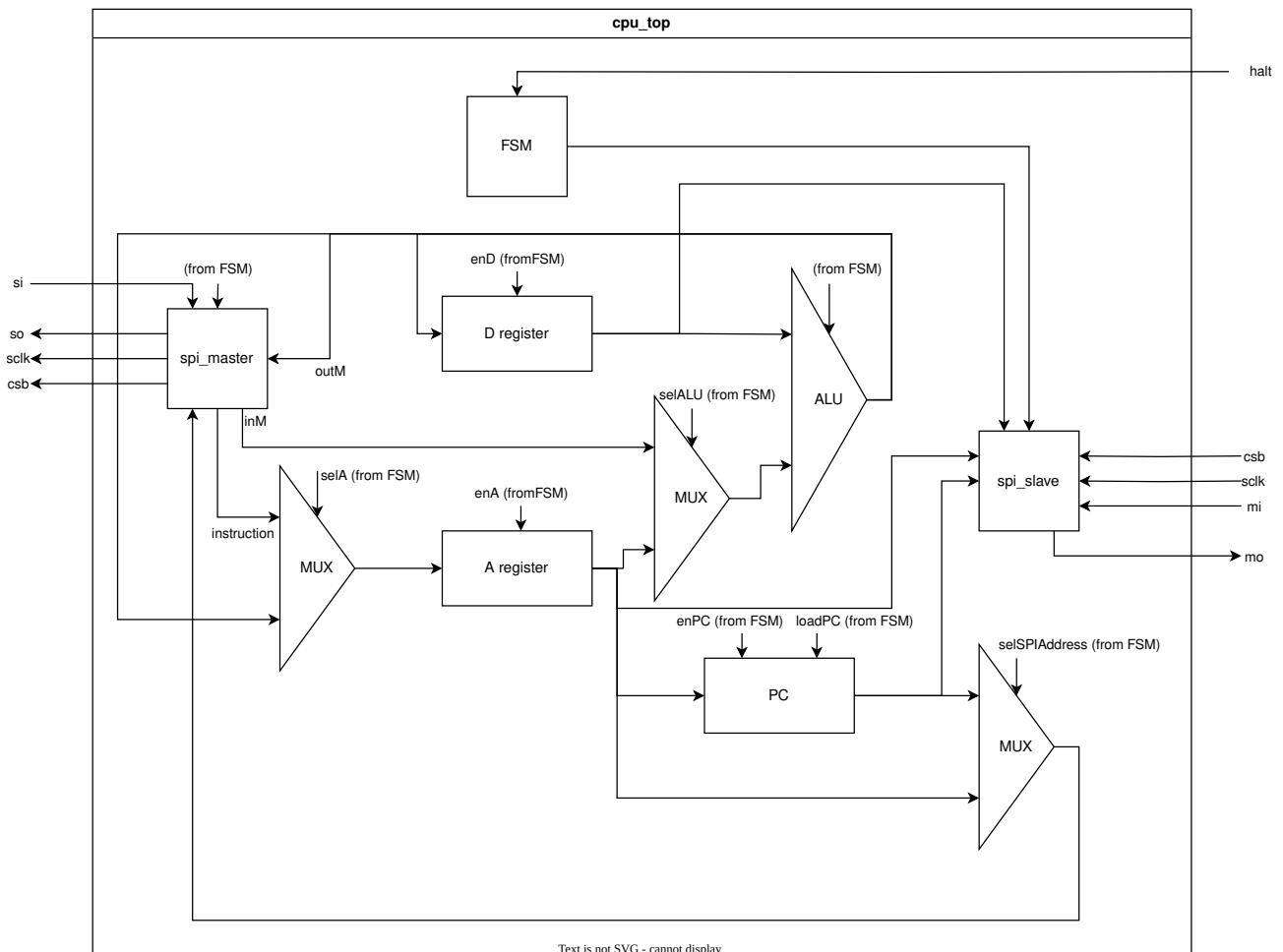
#	Input	Output	Bidirectional
4		rst_n	
5		n2_buf	
6		n4_buf	
7			

HACK CPU [146]

- Author: Dantong LUO, Nour MHANNA, Charbel SAAD
- Description: A 16-bit CPU based on the HACK architecture
- GitHub repository
- HDL project
- Mux address: 146
- Extra docs
- Clock: 12500000 Hz

How it works

The device we developed is a 16-bit CPU based on the HACK architecture. The figure below shows the detailed architecture.

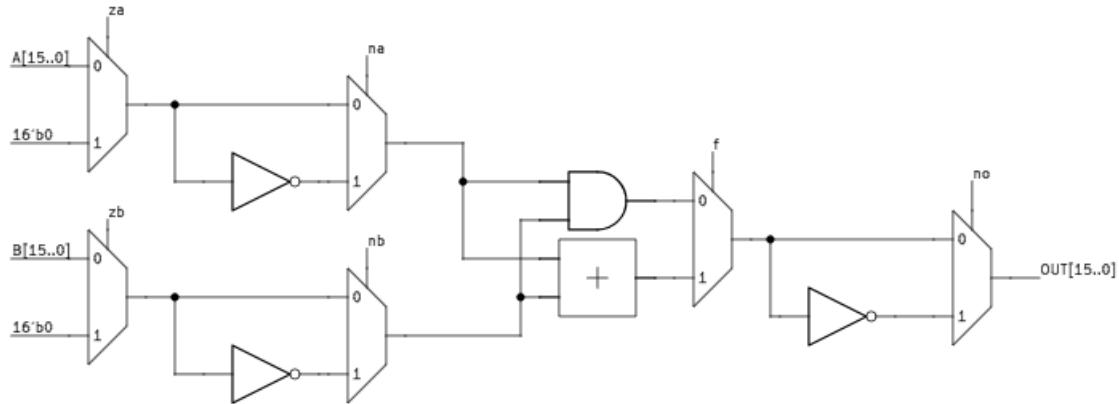


As we can see, it contains three main registers, an ALU, and two SPI modules. Each register has a unique function.

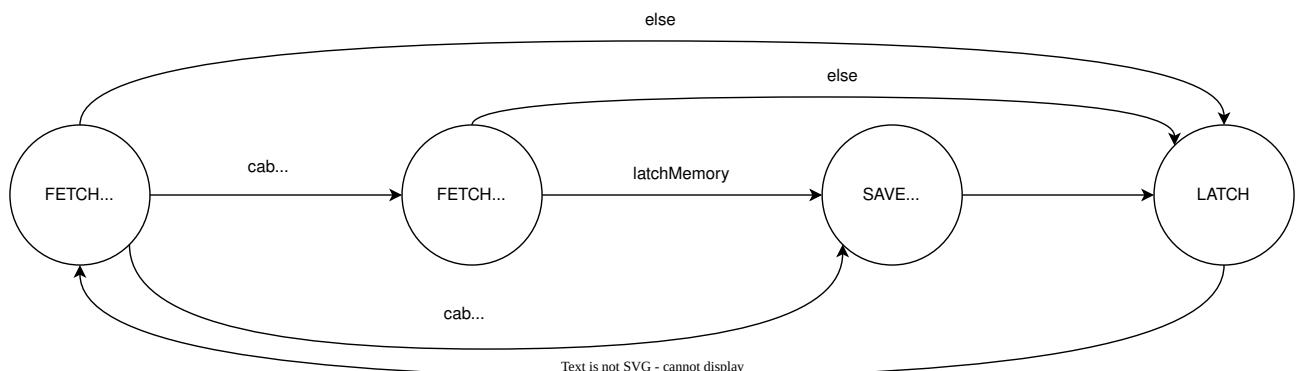
- D register stands as an accumulator.

- A register plays two roles. It first serves as an address register and also as a direct access register.
- PC is the program counter.

The ALU takes two different operands and is driven by 6 control signals, resulting in 18 different operations possible. Control signals can turn an operand to zero, logically reverse it, etc. The figure below shows how it is built.



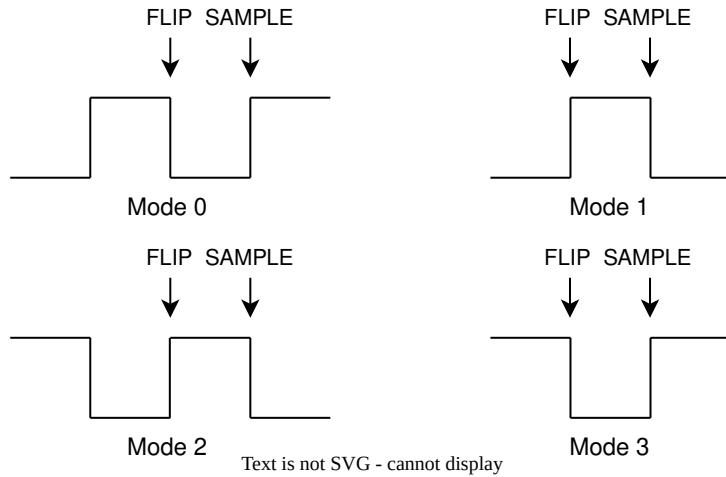
Since we don't have enough space on the chip, we can't include the memory. Moreover, we cannot fetch the 16-bit long instruction and data memory values at the same time because we only have 24 I/O pins. This is why we had to think of another approach. The idea we came up with is to fetch or save a 16-bit word once at a time and use a serial protocol for the transfer. We can see below the state diagram of the CPU.



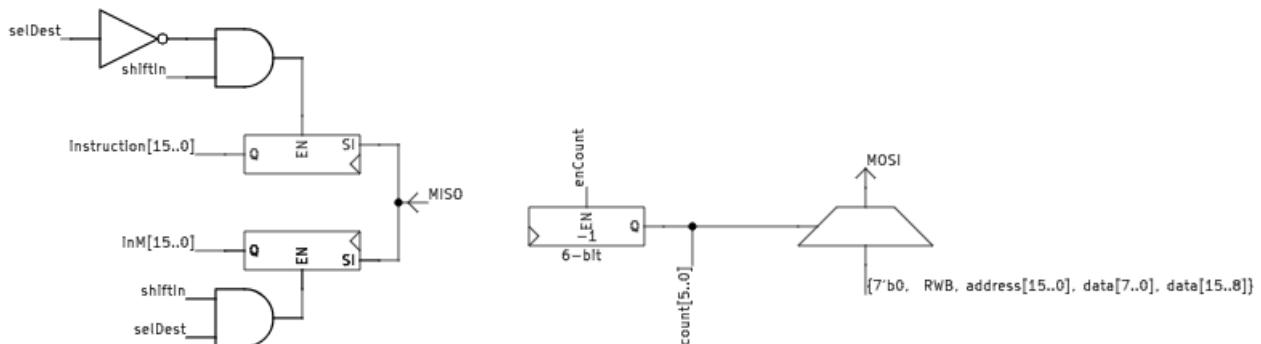
For the serial communication protocol, we chose SPI since it is one of the simplest to implement. We have to take into account 4 signals:

- MOSI: The signal containing the data transferred from the CPU to the memory.
- MISO: The signal transferring the data from the memory to the CPU.
- CSB: The signal that tells the memory that the CPU needs it.
- SCLK: The clock signal that cadences the transfer.

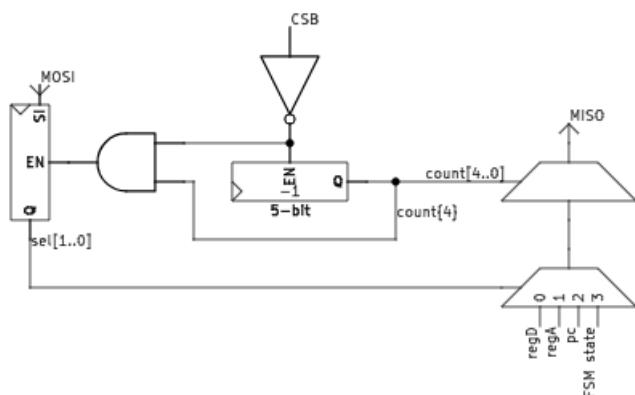
As shown in the figure below, SPI comes in 4 different modes. We are only going to work with modes 0 and 3 (Flip at the negative edge of the clock then sample on the positive edge).



The two following figures contain the logic circuits handling the transfer signals. As we can see, the MISO signal is latched in two different shift registers: one for the instruction, the other for the memory data. The MOSI signal is generated via a 40 to 1 multiplexer driven by a counter. The SPI module is monitored by its own FSM.



The final module also processes SPI signals but is used for debugging purposes. This time the SCLK and CSB are driven by the debugging device and the MISO and MOSI signals are inverted. The figure below shows how the module is built.



To communicate, the debugging device sends two bits of data, and depending on these bits, the CPU will output a specific value:

- 0: register D
- 1: register A
- 2: Program Counter
- 3: FSM state

How to test

The chip needs to be connected to an SPI RAM. We focused the design around the 23XX512, an SPI RAM developed by Microchip Inc. Just add the binary code to the RAM and provide an adequate 12.5 MHz clock signal. For debugging, a microcontroller with an SPI interface can do the job.

Since the chip is going to be soldered to a debug board with an RP2040 on it, we can use the code provided by MichaelBell to emulate the RAM ([Github repository](#)). The RP2040 can also be used as a debugger.

External hardware

- 65 KB SPI RAM.
- Microcontroller with SPI interface.

Pinout

#	Input	Output	Bidirectional
0	external halt signal (to use when debugging)		GPIO21 - RAM CS
1			GPIO22 - RAM MOSI
2			GPIO23 - RAM MISO
3			GPIO24 - RAM SCK
4			DEBUG CS
5			DEBUG MOSI
6			DEBUG MISO
7			DEBUG SCK

simon_cipher [161]

- Author: Simon Cipher
- Description: Bitserial implementation of Simon-128
- GitHub repository
- HDL project
- Mux address: 161
- Extra docs
- Clock: 0 Hz

How it works

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

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

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

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

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

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

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

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

LIMITATIONS

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

How to test

Study the testbench for example test vectors.

External hardware

No external hardware is needed for this project.

Pinout

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

Wirecube [163]

- Author: Leo Moser
- Description: A demo for the Tiny Tapeout demoscene competition - see for yourself!
- GitHub repository
- HDL project
- Mux address: 163
- Extra docs
- Clock: 50350000 Hz

How it works

The documentation will be updated after the competition has concluded.

How to test

Connect a Tiny VGA to the output Pmod port, set the clock frequency to two times 25.175 MHz = 50.350 MHz, make sure ui_in is set to 0x00 and enjoy the show!

External hardware

- Tiny VGA Pmod

Pinout

#	Input	Output	Bidirectional
0	toggle background bit 0	R1	
1	toggle background bit 1	G1	
2	toggle background bit 2	B1	
3	toggle cube bit 0	VS	
4	toggle cube bit 1	R0	
5	toggle cube bit 2	G0	
6	toggle speed bit 0	B0	
7	toggle speed bit 1	HS	

TT08 Pachelbel's Canon demo [165]

- Author: Mike Bell
- Description: Tiny Tapeout visuals with the classic Canon soundtrack
- GitHub repository
- HDL project
- Mux address: 165
- Extra docs
- Clock: 36000000 Hz

How it works

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

How to test

Set the inputs to 0, clock at 36MHz.

External hardware

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

Pinout

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

Neural Net ASIC [166]

- Author: Neural Navigators: Linyang Lee, Harsha Ganta, Stephanie Shen, William Li, Kiana Dai
- Description: MNIST Handwriting prediction on a neural network
- GitHub repository
- HDL project
- Mux address: 166
- Extra docs
- Clock: 10000000 Hz

How it works

Neural network

How to test

Test

Pinout

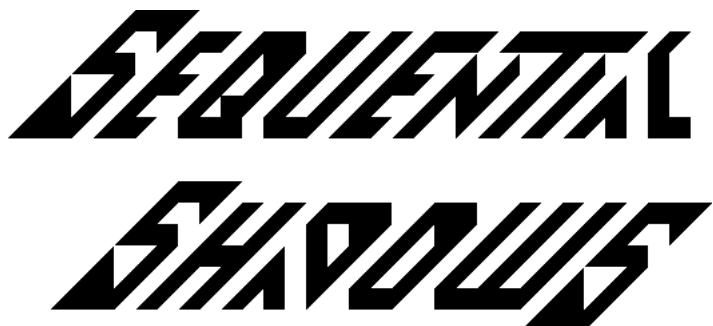
#	Input	Output	Bidirectional
0	ui[0]	uo[0]	uio[0]
1	ui1	uo1	uio1
2	ui2	uo2	uio2
3	ui[3]	uo[3]	uio[3]
4	ui[4]	uo[4]	uio[4]
5	ui[5]	uo[5]	uio[5]
6	ui[6]	uo[6]	uio[6]
7	ui[7]	uo[7]	uio[7]

Sequential Shadows [TT08 demo competition] [167]

- Author: Toivo Henningsson
- Description: My contribution to the TT08 demo competition
- GitHub repository
- HDL project
- Mux address: 167
- Extra docs
- Clock: 50400000 Hz

Intro

Curly / Medieval presents



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

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

How it works

The demo code contains a few different parts:

- Ray caster
- Synthesizer
- Music sequencer
- Logo
- Combined timing generator for raster scan and synthesizer
- Dithering
- Top level sequencer
- Audio visualizer

The code was first written without the audio visualizer and top level sequencer. At this point, there was music, but the demo was always showing the same moving landscape as in the intro (without fade-in) with the static logo on top. Also, there was not very much space left.

To add more contents, I went through the code looking for narrow control signals that might do interesting things when changed, and experimented on FPGA with changing them to see if I could get any interesting results. Examples:

- Sine plasma: Disable 3D part of ray caster
- Logo animation: Change address calculation into logo bitmap
- Jagged landscape: Change when bits are inverted in sine table lookup to modify part of sine function

The final steps were to choose which of these effects to use and to tweak the demo until I ran out of area and time.

Ray caster The ray caster is used to generate the landscapes. The height map is procedurally generated as the sum of 3 sine waves; there was no space to store a full height map. A sine table is used since the sine calculation needs to be fast. Summing 3 sine waves means that each height can be evaluated in 3 cycles, or 1.5 VGA pixels.

The calculated ground height is accumulated and stored in a register. The next ground height can start to be calculated directly, but has to wait to update the register until the previous height is no longer needed. There is also a mode to feed the sum of the 3 sine waves through the sine table to produce the final ground height, requiring 4 cycles per ground height evaluation.

Each sine term has its own phase and phase increment registers. Each phase increment is set based on an angle that is increased for each scan line to look in different directions. The angle is fed through the sine table (and the result scaled) to get the phase increment. The initial phases and the initial angles for the phase increments are updated each frame to animate the landscape.

The ray caster keeps track of the current ray height z , starting at eye level, and current z increment dz , starting at 511 (pointing down as much as possible). If z is above ground, the ray steps forward using dz , and the landscape steps forward to calculate a new height. If z is below ground, the ray steps up by decreasing dz by one, and decreasing z by the distance t the ray has travelled so far. This steps up to the ray given by the new dz value.

The ray caster has to produce output pixels in time with the VGA timing, starting from the left side of each scan line and producing a new pixel every two cycles. The x coordinate where a ground hit should be displayed corresponds to the downward angle

of the ray, and is given by $511-dz$. If the ray caster is about to run ahead of the display (x) coordinate, it waits for the display coordinate to catch up. If the ray caster is running behind the display coordinate, as often happens after running over the top of a hill in the landscape, a shadow (black pixel) is displayed while the ray tries to catch up.

As dz decreases along the scan line, a longer distance along the ray is needed to find each new ground hit. To be able to keep up with the display coordinate, the step length when moving along the ray is successively doubled after a given number of steps. This works out ok visually since details appear smaller at greater distances, so the increased step lengths don't lose as much detail as they would if they were used from the start.

Synthesizer The synthesizer is based on a small ALU, with one accumulator register and 7 numbered registers, each 11 bits wide. A program of 100 ALU operations is looped, producing a new sample value between 0 and 99 for each loop. The program is used to calculate sawtooth, triangle, and square waves, and sum them to create the output sample. For the chords, 6 sawtooth waves are calculated based on the same oscillator value (and the global counter) and added together.

All ALU operations update the accumulator. The accumulator value can then be written to a numbered register. The numbered registers are implemented with latches, and the accumulator value should be held constant while updating one to make sure that the correct value is written. Fortunately, the numbered registers don't need to be updated that often. The numbered registers are:

- chord phase
- drum phase
- bass phase
- lead phase
- B: temporary register
- output accumulator
- output (written during the last cycle in the loop, never read by the ALU)

The output from the previous sample is compared to the current loop position to create a PWM signal to output as the sound signal.

The phase values for the channels are updated in a similar way to the synth in <https://github.com/toivoh/tt06-retro-console>, with bit reversed phase compared to mantissa to get a sawooth wave, and octave divider.

Wave forms used:

- chords: detuned sawtooth
- drum: triangle (with descending frequency)

- bass: triangle
- lead: sawtooth or square, sometimes detuned

Detuning is created by calculating and adding the same waveform twice, but adding the global counter to the phase in one of the cases, suitably shifted.

The chords use different multipliers on the chord phase:

- major chord: 8, 10, 12
- minor chord: 10, 12, 15
- sus2 chord: 8, 9, 12

doubling some of the multipliers to create chord inversions. Each multiplication is calculated as the sum of two shifts. The chord phase is multiplied by each multiplier in turn, creating a sawtooth waveform that is added to the output.

Each ALU instruction has a tag field. A nonzero tag signifies conditional execution for different effects: raise the bass drum one octave, change the lead waveform into a square wave, etc...

Logo The logo stores two bits per 16x16 pixel square, one for each triangle half. Which one to look up is calculated from the current screen coordinates, and an offset for the logo animation effect.

Top level sequencer As much as possible is derived from the global counter. This includes the top level sequencer, which is basically a big case statement that sets different control signals depending on the current frame. Some of the control signals feed into the music sequencer to change the music (alternate melody and bass line, change lead between sawtooth and square wave, raise the bass one octave, ...).

Audio visualizer The audio is produced in sync with the VGA signal, 8 samples per scan line, so the audio visualizer mostly needs to look at the current audio output (0 or 1) after PWM comparison to decide the current pixel value. The synthesizer's ALU program was updated to invert every other sample value, and the audio output is also inverted for these samples. This creates the mirroring effect in the visualizer (and also makes the PWM output almost phase correct).

The music was transposed so that the root note is roughly a power of two times 60 Hz. This avoids most audio channels feeding flicker into the audio visualizer. The drums were cut a bit short when the visualizer is on, since their descending frequency can't avoid creating flicker. The bass line was raised one octave when the visualizer is on, and the amplitude is halved, which also reduces flicker substantially.

How to test

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

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

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

- `v_bass_off`: Setting this high reduces flashing when the audio visualizer is on by turning off the bass.
- `v_drums_off`: Setting this high reduces flashing when the audio visualizer is on by turning off the drums.
- `v_bass_low`: Setting this high keeps the bass at its default octave even when the audio visualizer is on, which increases flashing.
- `pause`: While this is high, the demo is paused and the sound is turned off. Can probably be used to start the demo paused.
- `step_frame`: While this is high, the demo advances one frame per cycle. Used for testing.

External hardware

This project needs

- a TinyVGA VGA Pmod.
- Mike's audio Pmod.

Pinout

#	Input	Output	Bidirectional
0	<code>v_bass_off</code>	R1	
1	<code>v_drums_off</code>	G1	
2	<code>v_bass_low</code>	B1	
3	<code>pause</code>	vsync	

#	Input	Output	Bidirectional
4		R0	
5		G0	
6		B0	
7	step_frame	hsync	audio_out

CYCLIPSONIC [171]

- Author: IITBBS_HEART
- Description: A dual-mode washing machine controller featuring a LiPSi microprocessor and a standalone controller, selectable via a pin configuration.
- GitHub repository
- HDL project
- Mux address: 171
- Extra docs
- Clock: 20000000 Hz

How it works

Not meant for submission

How to test

TBD

External hardware

TBD

Pinout

#	Input	Output	Bidirectional
0	ui[0]	uo[0]	
1	ui1	uo1	
2	ui2	uo2	
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]	

TDC with SPI [175]

- Author: Tautvydas Brukstus
- Description: TDC with SPI design
- GitHub repository
- HDL project
- Mux address: 175
- Extra docs
- Clock: 50000000 Hz

How it works

SPI test design based from https://github.com/calonso88/tt07_alu_74181

How to test

See https://github.com/calonso88/tt07_alu_74181

External hardware

Nothing required

Pinout

#	Input	Output	Bidirectional
0	cpol	busy	
1	cpha		
2	smapling_clk		
3	start_signal		spi_miso
4	stop_signal		spi_cs_n
5			spi_clk
6			spi_mosi
7			

Atari 2600 [178]

- Author: Renaldas Zioma
- Description: Replica of Atari 2600
- GitHub repository
- HDL project
- Mux address: 178
- Extra docs
- Clock: 25175000 Hz

How it works

Replica of a classic Atari 2600 (SoC) System On a Chip

How to test

Plug and play!

External hardware

Tiny (mole99) VGA PMOD, Tiny Audio PMOD, VGA display.

Pinout

#	Input	Output	Bidirectional
0	UP / Difficulty Switch P1	R1	QSPI CS
1	DOWN / Difficulty Switch P2	G1	QSPI SD0
2	LEFT / Monochrome Switch	B1	QSPI SD1
3	RIGHT	VSync	QSPI SCK
4	FIRE / Gamepad LATCH	R0	QSPI SD2
5	SELECT / Gamepad CLK	G0	QSPI SD3
6	Switches / Gamepad DATA	B0	
7	START	HSync	Audio (PWM)

SPI FPU [179]

- Author: Sebastian Pfeiler
- Description: A 32bit floating point adder accessible over spi
- GitHub repository
- HDL project
- Mux address: 179
- Extra docs
- Clock: 50000000 Hz

How it works

This is a floating point unit accesible over SPI.

```
SCLK = ui[0]
NCS  = ui[1]
COPI = ui[2]
CIPO = uo[0]
```

Constraint frequency(SCLK) < 4*frequency(clk)

It features 4 internal floating point registers, writeable and readable over SPI.

Every SPI transaction starts with a command byte followed by the arguments to the command. If the command outputs data, enough bytes need to be sent for the entire data to be received.

```
+-----+-----+--- . . . -+
|Command|Arguments|... . . . |
+-----+-----+--- . . . -+
```

WRITE_TO_REG Writes a floating point value (serialized on COPI) into an internal floating point register.

```
+-----+-----+-----+-----+
IN: | 0x00 | 0x0r | b[0] | b[1] | b[2] | b[3] |
     +-----+-----+-----+-----+
OUT: | und | und | und | und | und | und |
     +-----+-----+-----+-----+
```

r is one of 0,1,2,3 corresponding to the 4 internal registers. b[0] is the lowest byte of the floating point value. b[3] is the highest byte.

PERFORM_ADD Performs the computation register[in_c] = fadd(register[in_a], register[in_b]) This computation takes a couple of cycles (6 I think right now). These cycles are in reference to clk **not** to SCLK. Nevertheless add another dummy byte after the command to ensure that the add is being performed.

```
+-----+  
IN: |0x00|in_a|in_b|in_c|0x00|  
+-----+  
OUT: | und| und| und| und| und|  
+-----+
```

in_a,in_b,in_c can be one of 0x00,0x01,0x02,0x03

READ_FROM_REG Reads a float from the internal register and serialises it on CIPO.

```
+-----+  
IN: |0x00|0x0r|xxxx|xxxx|xxxx|xxxx|  
+-----+  
OUT: | und| und|b[0]|b[1]|b[2]|b[3]|  
+-----+
```

r is one of 0,1,2,3 b[0] is the lowest byte of the float b[3] is the highest byte of the float

How to test

It is best to test with the provided arduino testbench in the repository https://github.com/Qwendu/tt_float_adder in the directory src/testbenches/arduino

I have not managed to get it to work with the provided SPI controllers in arduino. Maybe this is because my spi_rx has some bugs.

External hardware

For integration test:

- Arduino

Known Bugs

- 1 Denormalized numbers do not always add correctly.
- 2 When testing on an fpga it was observed that it sometimes worked flawlessly and othertimes the output was always 0. to what extent that was a fault of the testsetup or the tester or the actual code has yet to be determined.

Pinout

#	Input	Output	Bidirectional
0	SPI_CLK	SPI_OUT	Unused
1	SPI_NCS	Unused	Unused
2	SPI_IN	Unused	Unused
3	Unused	Unused	Unused
4	Unused	Unused	Unused
5	Unused	Unused	Unused
6	Unused	Unused	Unused
7	Unused	Unused	Unused

MAC [192]

- Author: Mahaa Santeep G, Shylashree N, Ravish Aradhya H V, RV College Of Engineering, Sneha R V, PES University
- Description: Design and Implementation of MAC Unit Using Dadda Multiplier and Kogge-Stone Adder
- GitHub repository
- HDL project
- Mux address: 192
- Extra docs
- Clock: 100 Hz

Credits : We gratefully acknowledge the COE in Integrated Circuits and Systems (ICAS) and Department of ECE. Our special thanks to Dr K S Geetha (Vice Principal) and, Dr. K N Subramanya (principal) for their constant support and encouragement to do TAPEOUT in Tiny Tapeout 8 .

How it works

The tt_um_mac module is a Multiply-Accumulate (MAC) unit designed for high-performance digital signal processing and embedded system applications. This module integrates a Dadda multiplier and a Kogge-Stone adder to achieve efficient and fast computations. The MAC unit performs a sequence of multiplication and accumulation operations, which are essential in various digital signal processing tasks, such as filtering and convolution.

Functional Description

Input and Output Ports

- Inputs:
 - o ui_in (8-bit): Dedicated input for the first operand.
 - o uio_in (8-bit): Input/Output interface for the second operand.
 - o clk (1-bit): Clock signal to synchronize all operations.
 - o rst_n (1-bit): Active-low reset signal to initialize the internal state of the MAC unit.
- Outputs:
 - o uo_out (8-bit): Output that holds the final accumulated result.
 - o uio_oe (8-bit): Output enable signal, set to 0 indicating the uio is used as input.
 - o uio_out (8-bit): Unused output path in the current context.

1. Dadda Multiplier The Dadda multiplier is a high-speed multiplier designed for efficient computation. It reduces the partial products in a sequence of reduction stages until the final product is obtained. In this design, a 4x4 Dadda multiplier is used to compute the 8-bit product of the two 4-bit operands, A and B.
2. Pipeline Registers Pipeline registers are implemented to enhance the performance of the MAC unit by storing intermediate results at each stage of the operation. This design uses two pipeline registers:
 - Prod_stage: Holds the product of the multiplication.
 - Sum_stage: Holds the result of the accumulation.

3. Kogge-Stone Adder The Kogge-Stone adder is a parallel-prefix form of a carry-lookahead adder, known for its high speed and efficiency in handling large bit-width additions. It computes the sum of the product and the current accumulator value (Acc), which is stored in the Sum_stage register.
4. Accumulator The accumulator (Acc) is a key component that stores the ongoing sum of the products. It is updated with the result from the Kogge-Stone adder on each clock cycle, allowing the MAC unit to perform repeated accumulation operations. Reset Behavior When the reset signal (`rst_n`) is asserted low, the pipeline registers (`Prod_stage`, `Sum_stage`) and the accumulator (Acc) are cleared, resetting the MAC unit to its initial state.

How to test

How to Test

To verify the functionality of the `tt_um_mac` module, a testbench (`tt_um_mac_tb`) has been provided. The testbench simulates different input scenarios and observes the output behavior of the `tt_um_mac` module to ensure that it works correctly.

- The testbench will output the results of the simulation, including the values of the inputs and the resulting output for each test case.
- Monitor the output in the console or waveform viewer to ensure the `tt_um_mac` module behaves as expected.

Example Test Scenarios Below is a summary of the test cases used in the `tt_um_mac_tb` testbench, along with their expected results.

Time (ns)	ui_in (Input A)	uio_in (Input B)	Operation	Expected uo_
0-10	00000000 (0)	00000000 (0)	Reset	00000000 (0)
10-30	00000011 (3)	00000010 (2)	Multiply, Accumulate	00000110 (6)
30-50	00000001 (1)	00000100 (4)	Multiply, Accumulate	00001010 (10)
50-70	00000101 (5)	00000011 (3)	Multiply, Accumulate	00011001 (25)
70-90	00000111 (7)	00000010 (2)	Multiply, Accumulate	00100111 (39)
90-110	00000000 (0)	00000000 (0)	No Operation (Idle)	00100111 (39)
110-130	00000001 (1)	00000001 (1)	Multiply, Accumulate	00101000 (40)

Monitoring Output During the simulation, you can monitor the console or waveform outputs for detailed step-by-step results. The testbench uses `$monitor` to display real-time updates of the inputs and the resulting output.

```

initial begin
    $monitor("Time=%0d | ui_in=%b, uio_in=%b | uo_out=%b", $time, ui_in,
end

```

This will provide you with a detailed trace of how the tt_um_mac module processes the inputs to generate the expected outputs.

Pinout

#	Input	Output	Bidirectional
0	ui_in[0]	uo_out[0]	uio_in[0]
1	ui_in[1]	uo_out1	uio_in1
2	ui_in[2]	uo_out2	uio_in2
3	ui_in[3]	uo_out[3]	uio_in[3]
4		uo_out[4]	
5		uo_out[5]	
6		uo_out[6]	
7		uo_out[7]	

DPMU [194]

- Author: Sanjay Kumar M, Shylashree N, Ravish Aradhya H V, RV College Of Engineering, Neha R V, PES Unoversity
- Description: Design and Implementation of Dynamic Power management unit
- GitHub repository
- HDL project
- Mux address: 194
- Extra docs
- Clock: 100 Hz

Credits : We gratefully acknowledge the COE in Integrated Circuits and Systems (ICAS) and Department of ECE. Our special thanks to Dr K S Geetha (Vice Principal) and, Dr. K N Subramanya (principal) for their constant support and encouragement to do TAPEOUT in Tiny Tapeout 8 .

The code provided is a SystemVerilog module that implements a Dynamic Power Management Unit (DPMU) for an SoC (System on Chip). The DPMU dynamically adjusts voltage and frequency levels based on inputs such as performance requirements, temperature, battery level, and workload. The module uses a finite state machine (FSM) to manage transitions between different power states.

Key Components

Inputs and Outputs: Inputs (`ui_in`): The primary input signals include performance requirements, temperature sensor data, battery level, and workload. Outputs (`uo_out`, `ui_o_out`): These include the power-saving indicator, voltage levels, and frequency levels for different cores and memory. I/O (`ui_o_in`, `ui_o_out`, `ui_o_oe`): Handles bidirectional signals; however, in this design, `ui_o_in` is not used, and `ui_o_out` is used for output.
Internal Signals:

State Variables: `state` and `next_state` manage the FSM that controls the DPMU's behavior. **Power and Frequency Controls:** Registers like `vcore1`, `vcore2`, `vmem`, `fcore1`, `fcore2`, and `fmem` store the voltage and frequency settings. **Finite State Machine (FSM):**

States: **NORMAL:** Default operating mode with standard voltage and frequency levels. **PERFORMANCE:** High-performance mode with maximum voltage and frequency levels. **POWERSAVE:** Low-power mode with reduced voltage and frequency levels. **THERMAL_MANAGEMENT:** Mode to handle high temperature by adjusting power levels moderately. **BATTERY_SAVING:** Mode to conserve battery by minimizing voltage and frequency levels.

State Transitions: Transitions between states occur based on the input conditions, such as high performance request, low battery level, high temperature, or low workload.

Detailed Walkthrough Input and Output Mapping:

perf_req: Mapped to the least significant bit (LSB) of ui_in, indicating whether high performance is needed. **temp_sensor:** 2-bit signal derived from ui_in[3:2], providing temperature data. **battery_level:** 2-bit signal derived from ui_in[5:4], indicating the battery's charge status. **workload_core:** 3-bit signal derived from ui_in[7:6], representing the workload of a core.

State Logic:

On each clock cycle (clk), the FSM checks the state and evaluates transitions based on inputs. In NORMAL state, if perf_req is high, the system transitions to PERFORMANCE state. If the battery level is low, it transitions to BATTERY_SAVING state. If the temperature is high, it transitions to THERMAL_MANAGEMENT state. If the workload is low, it transitions to POWERSAVE state. PERFORMANCE state sets all voltages and frequencies to maximum. If perf_req drops, it returns to NORMAL. POWERSAVE state reduces voltages and frequencies to conserve power. If the workload increases, it returns to NORMAL. THERMAL_MANAGEMENT state adjusts power levels to moderate values to manage high temperatures. If the temperature normalizes, it returns to NORMAL. BATTERY_SAVING state minimizes voltages and frequencies to conserve battery. If the battery level increases, it returns to NORMAL.

Output Assignment: The combined voltage (vcore1, vcore2, vmem) and frequency (fcore1, fcore2, fmem) values are assigned to the uio_out and uo_out outputs. The power_save signal is also part of the output, indicating whether the system is in power-saving mode.

Behavior under Reset:

When the reset (rst_n) is low (active), the system resets to the NORMAL state.

Here's a table summarizing the expected output (ui_out, uo_out) based on the input (ui_in) and time using the provided testbench for the tt_um_dpmu module. The table provides the values for different states as the ui_in input changes over time.

Table: Testbench Expected Output

Time (ns)	ui_in (Input)	State	ui_out (Expected Output)	uo_out (Expected Output)
0	11110010	NORMAL	01010110	010_010010 10 00010010
		PERFORMANCE		11111111
111_111111	30	NORMAL	01010110	010_010010 50 11110011
		THERMAL_MANAGEMENT	10101011	011_011011 70 11110010
		NORMAL	01010110	01010110
010_010010	90	THERMAL_MANAGEMENT	11101010	10101011 011_011011
110	11111010	BATTERY_SAVING	00000000	000_000000 130 11111110
		BATTERY_SAVING	00000000	000_000000 150 11111010
				BATTERY_SAVING 00000000 000_000000
				000_000000

Explanation of Table Columns:

Time (ns): The simulation time when the ui_in input is applied. ui_in (Input): The 8-bit input value applied to the design. State: The state of the FSM based on the ui_in input. The states are NORMAL, PERFORMANCE, THERMAL_MANAGEMENT, and BATTERY_SAVING. uio_out (Expected Output): The expected 8-bit output values for the uio_out signals. uio_out[0]: Power save mode indicator. uio_out[2:1], uio_out[4:3], uio_out[6:5]: Voltage controls. uio_out[7]: Part of fcore1[0]. uo_out (Expected Output): The expected 8-bit output values for the uo_out signals. uo_out[0:1]: Part of fcore1[2:1]. uo_out[4:2]: fcore2[2:0]. uo_out[7:5]: fmem[2:0].

Explanation of Key Points: NORMAL State: When the inputs suggest a typical operating environment (e.g., ui_in = 11110010), the design operates with default voltage and frequency levels. PERFORMANCE State: Triggered by a performance request (perf_req = 1), leading to maximum voltage and frequency levels. THERMAL_MANAGEMENT State: Triggered by high temperature (temp_sensor = 10 or 11), moderates the voltage and frequency to prevent overheating. BATTERY_SAVING State: Triggered by low battery level (battery_level = 00 or 01), minimizing power consumption by reducing voltage and frequency to the lowest levels.

Testbench Operation: The testbench applies different ui_in values at specific simulation times. At each time step, it captures the output values (uio_out and uo_out) and compares them with the expected values as per the design's FSM logic. The \$monitor statement continuously logs the input and output values, helping to verify the design's behavior at each time point.

Pinout

#	Input	Output	Bidirectional
0	ui_in[[0]	uo_out[0]	uio_out[0]
1	ui_in[1]	uo_out1	uio_out1
2	ui_in[2]	uo_out2	uio_out2
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]

7 Segment Decode [196]

- Author: Jack Clayton
- Description: ASIC implementation of a university CPLD project which drives 4 multiplexed 7 segment displays, and scans a multiplexed keypad.
- GitHub repository
- HDL project
- Mux address: 196
- Extra docs
- Clock: 5000 Hz

How it works

The serial protocol implemented in this design consists of a simple single byte packet which instructs the CPLD which column of the keypad to multiplex into MISO, which screen should be displaying data, and what number should be displayed on the screen. The screen select signal also doubles up as instructing which row of the keypad to be scanned. Data is sent Most Significant Bit (MSB) first.

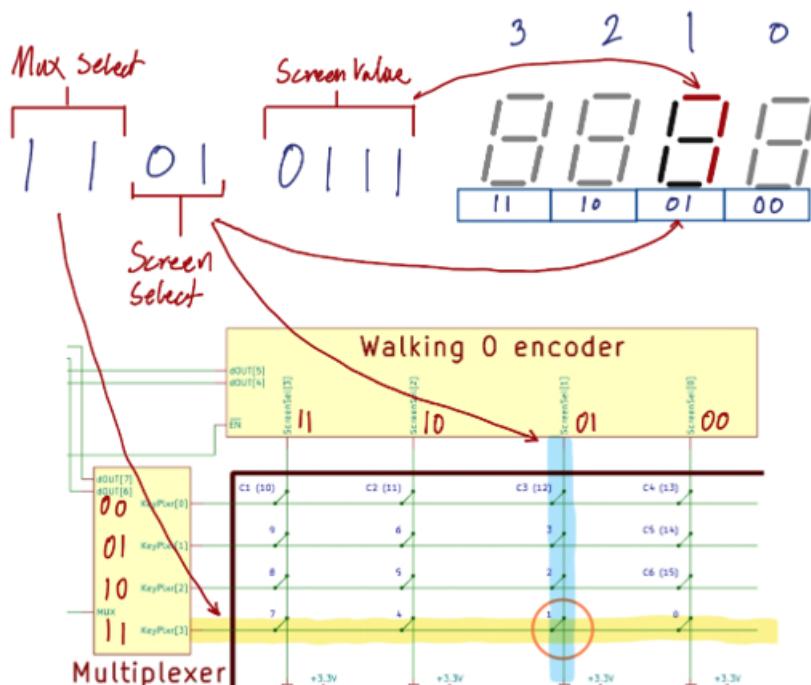
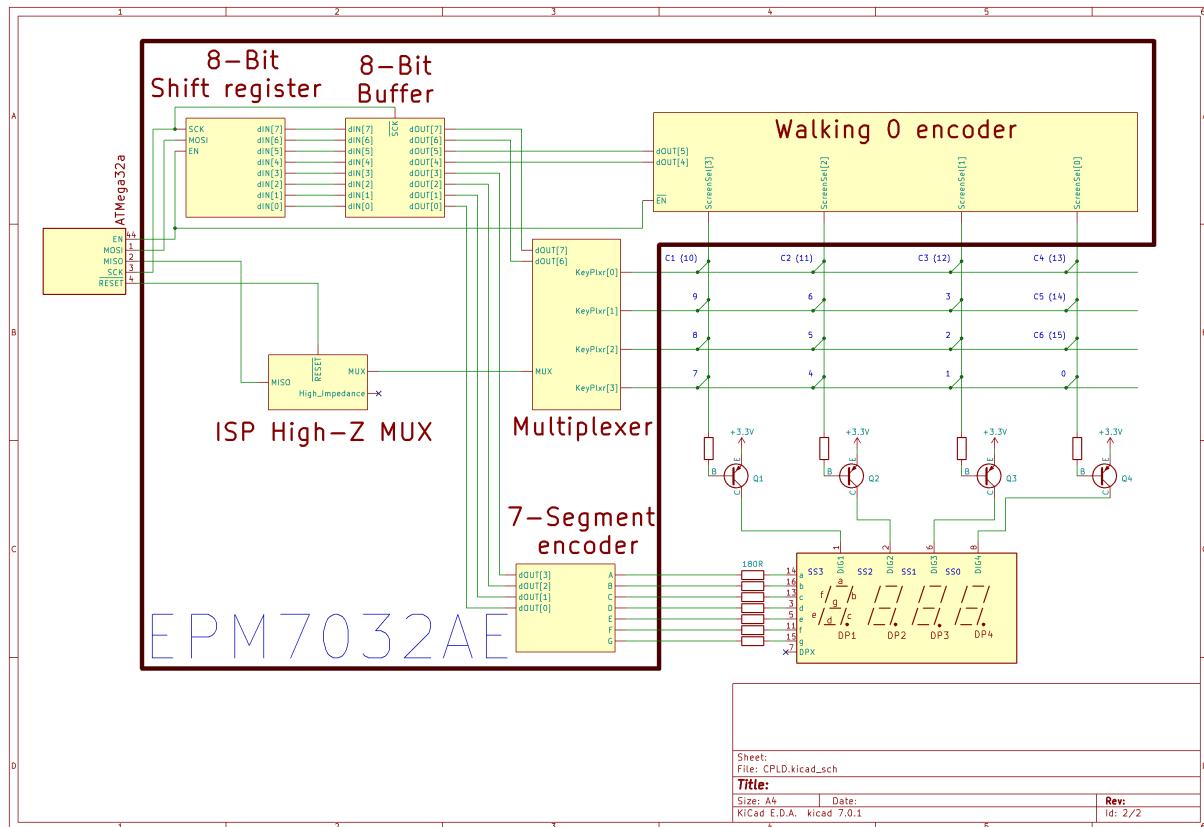
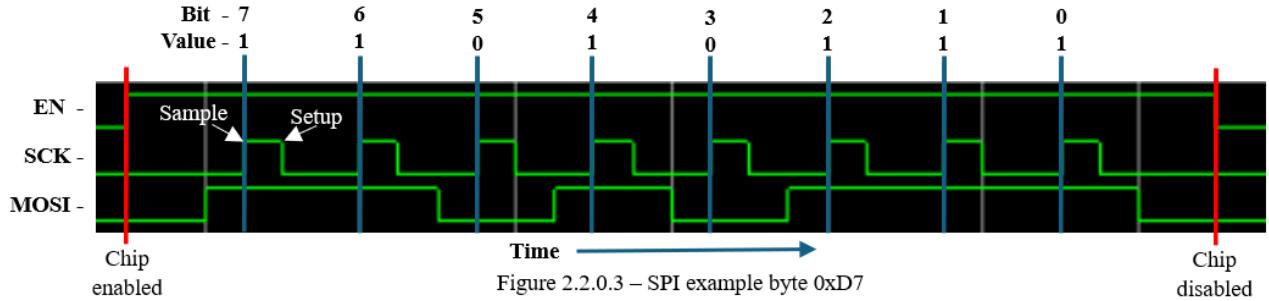


Figure 2.2.0.1 – SPI Protocol visual bit representations

Bit 7	Multiplexer select MSB
Bit 6	Multiplexer select LSB
Bit 5	Screen select MSB
Bit 4	Screen select LSB
Bit 3	Screen data MSB
Bit 2	Screen data
Bit 1	Screen data
Bit 0	Screen data LSB

Figure 2.2.0.2 – SPI Protocol bit definitions

2.2.2 Signal structure



The high impedance programming state is not implemented in this ASIC. It is represented as a bit out instead.

Keep in mind: This system is clocked by the SPI clock, and therefore requires constant clocking to function.

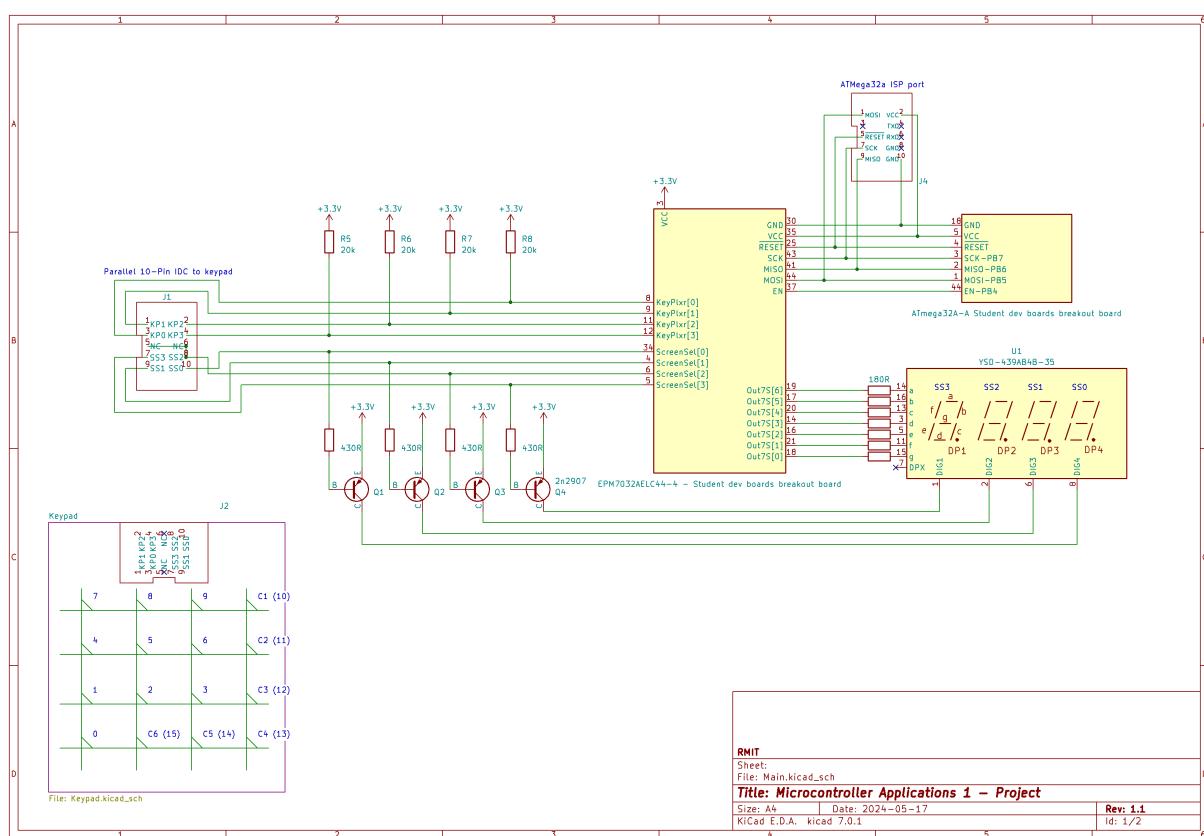
How to test

Build the supporting hardware as described in the schematic found in “External hardware”. Create a system which transmits SPI bytes, according to specifications in “How it works”. The system will display your desired digits on the selected screens.

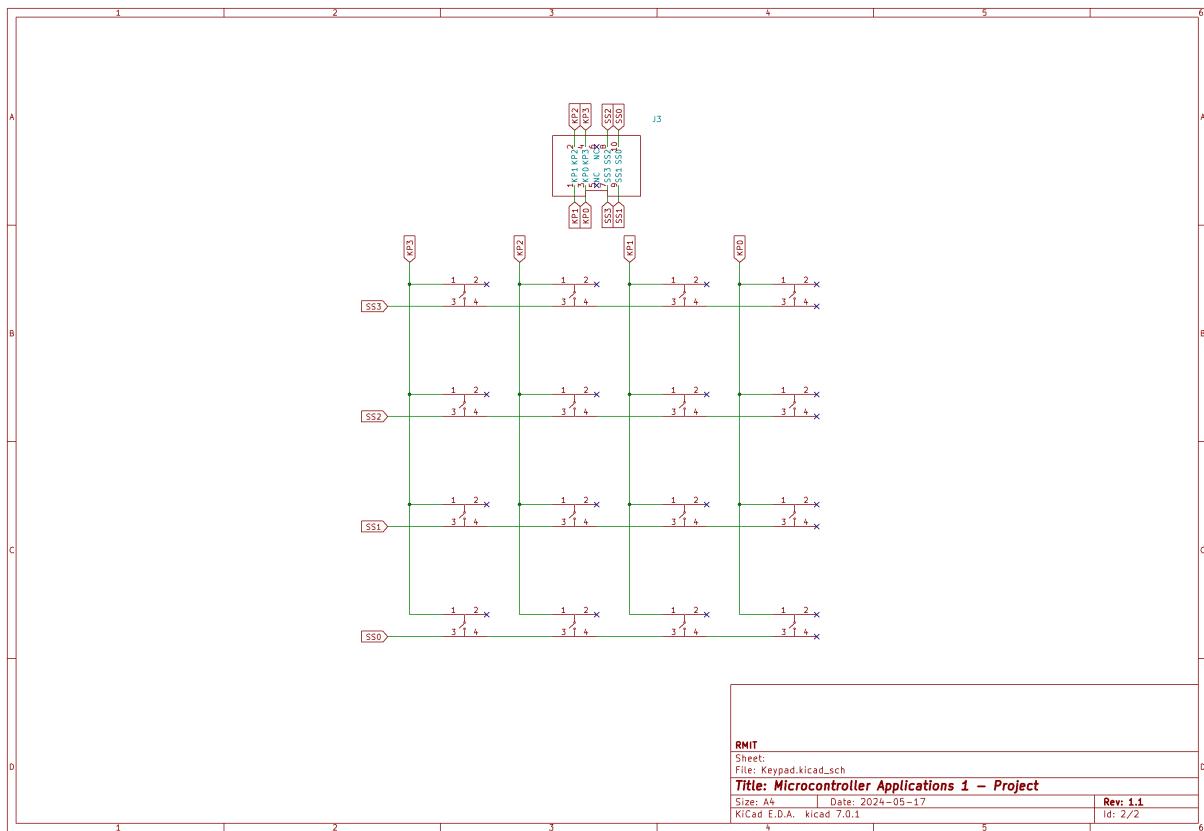
You may also use the MISO to implement a 4x4 keypad, which is interpreted by the system creating the SPI bytes. This will not be detailed as to how to implement.

External hardware

Main external system schematic:



Simple keypad:



Pinout

#	Input	Output	Bidirectional
0		Out7S[0]	ScreenSel[0]
1	MOSI	Out7S1	ScreenSel1
2	EN	Out7S2	ScreenSel2
3	RESET	Out7S[3]	ScreenSel[3]
4	KeyPlxr[0]	Out7S[4]	High-Z
5	KeyPlxr1	Out7S[5]	
6	KeyPlxr2	Out7S[6]	
7	KeyPlxr[3]	MISO	

PS2 Decoder [198]

- Author: Ben Payne
- Description: A PS2 keyboard decoder
- GitHub repository
- HDL project
- Mux address: 198
- Extra docs
- Clock: 25000000 Hz

How it works

This decoder works by first debouncing the inputs to make sure that we get a clean sample of them that is synchronized to our clock. It then looks at the down transition of ps2_clk and reads the value of ps2_data. It shifts this into a 11 bit shift register. When ps2_clk remains high for more than 1/2 of the 10kHz ps2_clk cycle it knows that the end of the data has arrived. It then triggers a valid flag to tell the system that something has arrived. The valid flag, which is exposed on a pin, will trigger the fifo to read the byte of data and it will be stored to retrieval buffer the host. When valid is triggered it will also trigger the interrupt pin. The valid pin is a pulse for one system clock cycle, but the interrupt will remain set until it is cleared. We also include a data_rdy signal that tells the host that there is data to read. This is useful if your interrupt handler needs to read multiple bytes.

When the host wants to read a byte, it asserts the chip select (cs) signal when the system clock goes high. This will result in the uio bus being set with the data value. The uio bus will be put into an output state only when cs is asserted, at all other times it will be an input bus (but we never read it...)

How to test

Simply interface a PS2 keyboard to the PS2 clock and data lines. You will need to level shift these signals to the 3.3v of the chip. At this point you can hit keys and they will be queued in the fifo. Then you would want to interface a retro computer to the CS, interrupt and data lines to read the fifo. This will depend on the system you're using, but note you'll need external address decoding logic and for chips like the m68k you'll need to generate the DTACK and other signals elsewhere.

External hardware

Hook up an PS2 PMOD device to level shift the keyboards 5V to 3.3V for this chip. I have a design for this if anyone wants it.

Pinout

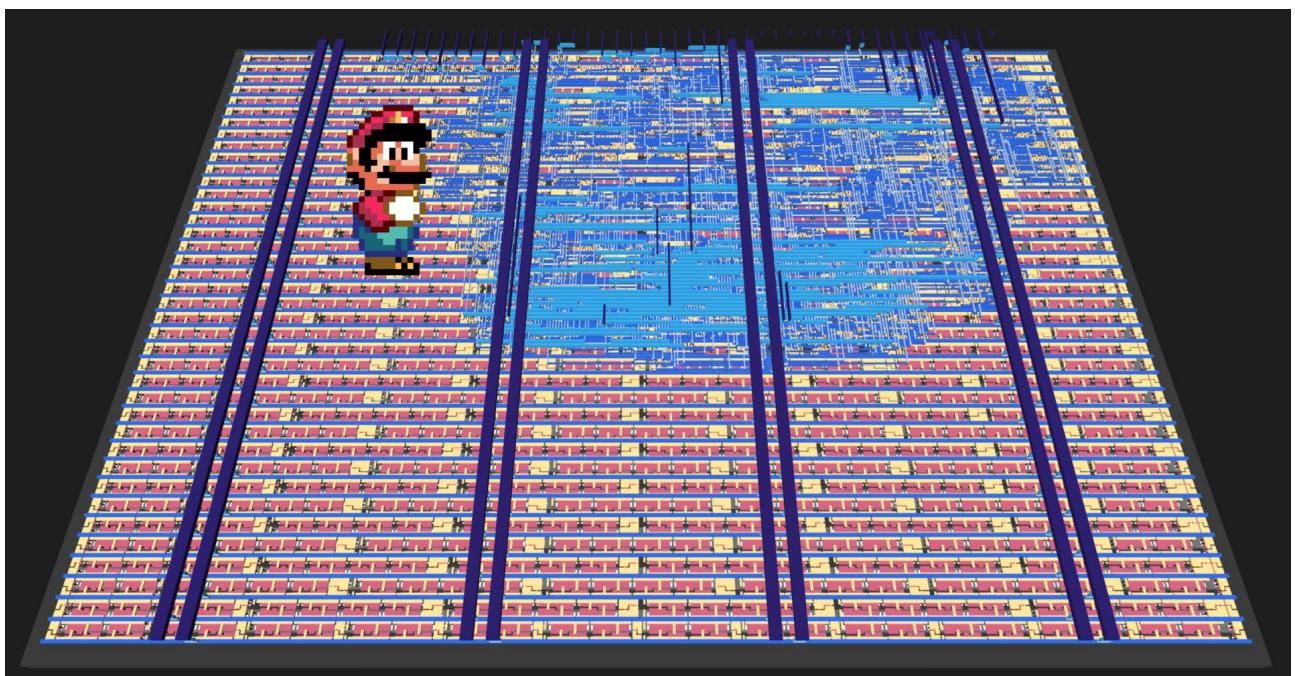
#	Input	Output	Bidirectional
0	ps2_clk	valid	data_out[0]
1	ps2_data	interrupt	data_out1
2	clear_int	data_rdy	data_out2
3	cs		data_out[3]
4			data_out[4]
5			data_out[5]
6			data_out[6]
7			data_out[7]

Super Mario Tune on A Piezo Speaker [200]

- Author: Milosch Meriac
- Description: Plays Super Mario Tune over a Piezo Speaker connected across `ui_out[1:0]`
- GitHub repository
- HDL project
- Mux address: 200
- Extra docs
- Clock: 100000 Hz

How it works

This design will play Super Mario Tune over a Piezo Speaker connected across `bidir[0:1]` and `bidir[7]`. The speaker is driven in differential PWM mode to increase its output power. The changed pinout accomodates for the Tiny Tapeout Audio Pmod.



(see also the interactive version of this design)

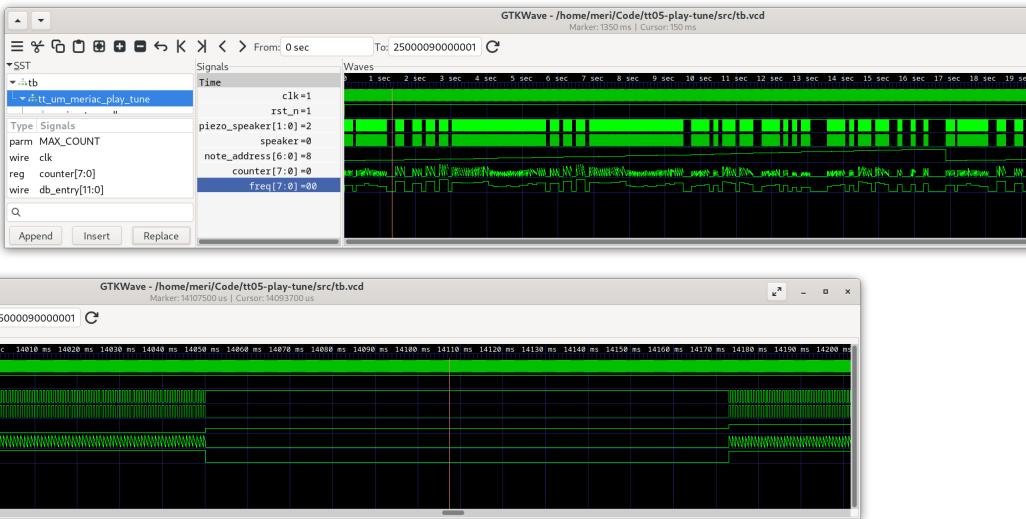
Additionally - for testing purposes, the inputs `ui[7:0]` are copied to the hex segment display 1:1 (`uo[7:0]`).

Verilog Design Files

- Playback Logic

- Autogenerated Super Mario Tune Storage. This project contains a Python-based script for converting a RTTL ringtone into optimized Verilog. An additional script converts TIM-file waveforms from the Verilog simulator back to a WAV-sound file to verify the correctness of the hardware-based player's sound.

PWM Waveform in Verilog Simulation Output Using GTKWave for visualization of Simulation Results:



How to test

Provide 100kHz clock on clk, briefly lower reset (rst_n) and bidir[1:0]/bidir[7] will play a differential sound wave over piezo speaker (Super Mario Tune).

External hardware

Piezo speaker connected across bidir[1:0] (loud) or between bidir[7] and GND (less loud). Alternatively you can connect the Tiny Tapeout Audio Pmod to the bidir port to listen to the music.

Pinout

#	Input	Output	Bidirectional
0	input pin 0	ui[0]	piezo_speaker_p (ui_out[0])
1	input pin 1	ui1	piezo_speaker_n (ui_out1)
2	input pin 2	ui2	GND
3	input pin 3	ui[3]	GND
4	input pin 4	ui[4]	GND

#	Input	Output	Bidirectional
5	input pin 5	ui[5]	GND
6	input pin 6	ui[6]	GND
7	input pin 7	ui[7]	piezo_speaker_n (uio_out[7])

AES Inverse S-box [202]

- Author: Dag Arne Osvik
- Description: Advanced Encryption Standard (AES) Inverse S-box
- GitHub repository
- HDL project
- Mux address: 202
- Extra docs
- Clock: 125000000 Hz

How it works

This circuit computes the inverse S-box of the Advanced Encryption Standard (AES).

How to test

Set the input byte, then read back the result from `uo_out` (unregistered) or `ui0_out` (registered).

External hardware

None.

Pinout

#	Input	Output	Bidirectional
0	$x[0]$	$y[0]$	
1	x_1	y_1	
2	x_2	y_2	
3	$x[3]$	$y[3]$	
4	$x[4]$	$y[4]$	
5	$x[5]$	$y[5]$	
6	$x[6]$	$y[6]$	
7	$x[7]$	$y[7]$	

TT08 - experiments with latch-based shift registers [204]

- Author: Ciro Cattuto
- Description: A 512-bit latch-based shift register in 1 tile
- GitHub repository
- HDL project
- Mux address: 204
- Extra docs
- Clock: 0 Hz

How it works

This is an experiment. A 512-bit shift register (SR) implemented using D latches rather than D flip flops. The shift logic relies on a single pulse rippling along the shift register, from the output latch towards the input latch. The SR has one input, one output, and an edge-triggered control signal that controls the shift update. The SR shifts on either a rising or a falling edge of the control signal.

How to test

Shift zeros into the SR until it contains all zeros. Then shift in any sequence of 1s and 0s and observe it appear on the output of the SR after 512 transitions of the control signal.

External hardware

No external hardware required.

Pinout

#	Input	Output	Bidirectional
0	shift register input	shift register output	
1	shift control (edge-triggered)		
2			
3			
4			
5			
6			

#	Input	Output	Bidirectional
7			

Obstacle Detection [206]

- Author: Emmy Xu
- Description: Does the logic of when to send certain signals when objects are close.
- GitHub repository
- HDL project
- Mux address: 206
- Extra docs
- Clock: 0 Hz

How it works

It takes in two different numbers one for a left sensor and one for a right sensor. There is a threshold value of 1 or 0, if the threshold has been passed, it will have a value of 1. If only one side has a value of one, it will send a 2'b10 meant for a motor to the opposite side. If both sides have a value of one, it will send 2'b01 to both sides meant for motors.

How to test

Set it up so that a value of 1 or 0 is going into the sensor pins and connect the output pins to something that can read what the chip is sending out. The reset pin resets when power is sent to it, which just makes it output 0s to all the outputs.

External hardware

Ultrasonic Sensors, Microcontroller, and Haptic Motors

Pinout

#	Input	Output	Bidirectional
0	sensor_left	left_buzz	
1	sensor_right	right_buzz	
2	reset		
3			
4			
5			

#	Input	Output	Bidirectional
6			
7			

resfuzzy [208]

- Author: roshan
- Description: calculation
- GitHub repository
- HDL project
- Mux address: 208
- Extra docs
- Clock: 0 Hz

How it works

The project implements a fuzzy logic system that estimates “risk” based on rainfall and soil moisture. It uses triangular membership functions to evaluate these inputs as low, medium, or high. Three fuzzy rules fire depending on the overlap between rainfall and soil moisture conditions. The system calculates the weighted average of the rule strengths to produce the risk value. If no rules fire (i.e., denominator is zero), the output risk is zero. The module updates the risk value on the clock edge when ef is enabled.

How to test

To test the fuzzy logic system, simulate different conditions by changing the input data_bus (rainfall/soil moisture data). Test the values of 80, 10, and 50 with ss (sensor select) toggling between 0 and 1 to activate the fuzzy logic. The expected output is a different risk value based on these input scenarios (FF,55,AA) (High,Low,Medium) respectively.

External hardware

8 switches connected to the input ui_in[7:0] pins 1 switch to the uio_in[0] pin 8 bit LED is needed to show the output values uo_out[7:0]

Pinout

#	Input	Output	Bidirectional
0	Input data from the sensors	risk value	sensor select
1	Input data from the sensors	risk value	

#	Input	Output	Bidirectional
2	Input data from the sensors	risk value	
3	Input data from the sensors	risk value	
4	Input data from the sensors	risk value	
5	Input data from the sensors	risk value	
6	Input data from the sensors	risk value	
7	Input data from the sensors	risk value	

CEJMU Beers and Adders [210]

- Author: Prof. Dr.-Ing. Matthias Jung, Philipp Wetzstein, Derek Christ, Jonathan Hager
- Description: Several projects to show in lectures. Includes a simple state-machine, a decoder and two 24 bit adders. Refer to documentation for details
- GitHub repository
- HDL project
- Mux address: 210
- Extra docs
- Clock: 12000000 Hz

How it works

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

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

How to test

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

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

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

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

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

External hardware

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

Pinout

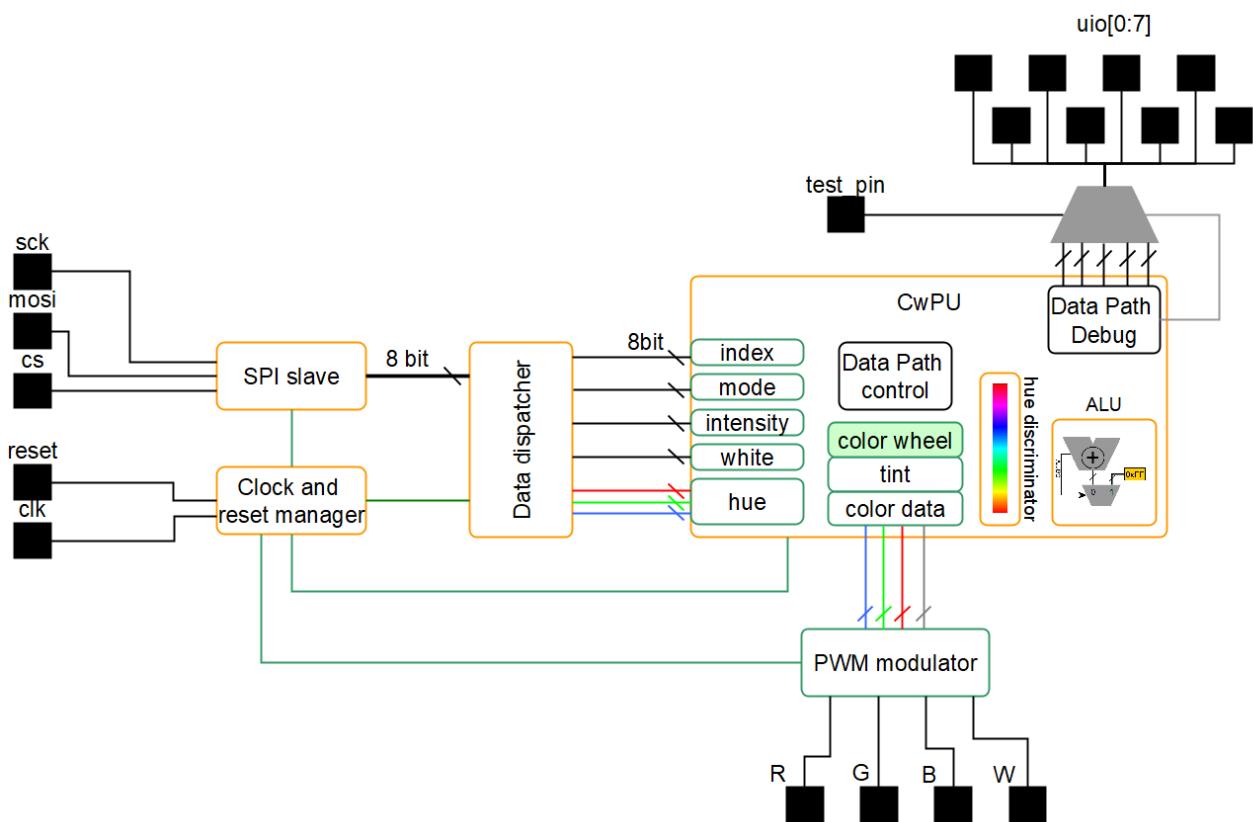
#	Input	Output
0	Multiplexed to all designs (refer to documentation for details)	Multiplexed from all designs
1
2
3
4
5
6
7

RGBW Color Processor [225]

- Author: Enrico Sanino
- Description: Color processor for RGBW LEDs, with generation of hue, tint and intensity based on a color index. Is also a direct SPI to 4 channels PWM converter.
- GitHub repository
- HDL project
- Mux address: 225
- Extra docs
- Clock: 66000000 Hz

How it works

Color generator for RGBW LEDs, with generation of hue, tint and intensity based on a color index. Is also a direct SPI to 4 PWM channels converter, making it flexible to any different kind of use. The system block diagram is as follow:



It is an SPI slave in Mode 0, with SPI protocol consisting of 8 byte long command, discriminated with a preamble sequence (see Protocol and Test for the description).

This payload is unpacked in different data: red, green, blue, white, bypass mode, intensity, color index. This data is then provided to the color wheel processor. If the

bypass mode is activated, the RGBW info from the red, green, blue and white SPI bytes is directly provided as a PWM output in the respective channels.

If bypass mode is not active, only the white, intensity and color index are considered, from which the hue (RGB data) is generated based on the index, then a tint (hue + white) and then the intensity is applied, forming the final color. This is then applied to the PWM outputs to the respective channels.

When bypass mode is not active (color wheel mode), then there is a latency proportional to the “rotation” of the color wheel, i.e. lower the number lower the latency. This is the latency of the color wheel processing unit (CwPU), after which the desired complete color is output on the PWM channels.

Debug pins A debug enable pin, when asserted, will output on the uio pins different internal signals of the CwPU while in operation. This is just to check the internal signals in case the tapeout goes wrong, and for curiosity purposes for fidelity against the gate level simulation.

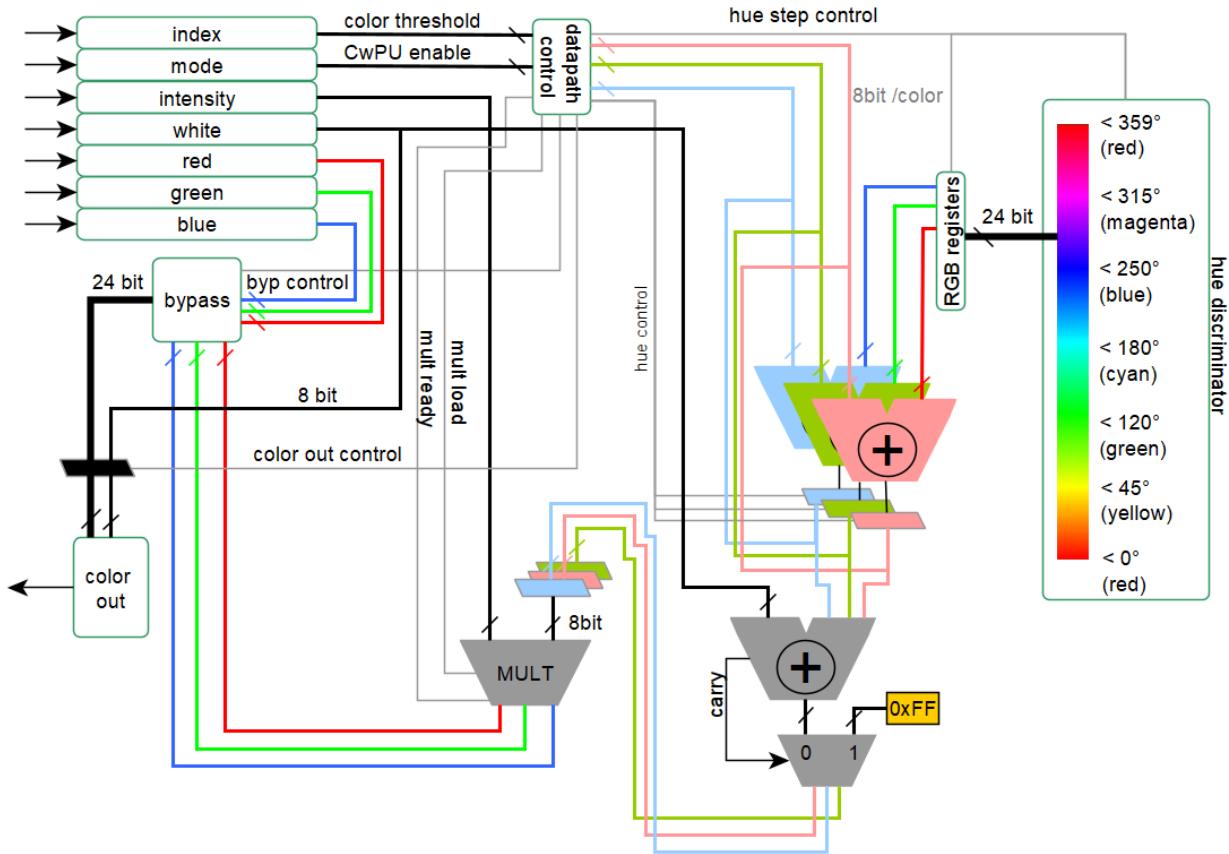
PWM modulator The PWM modulator has a period of $t_{pwm} = t_{clk_presc} * 256$, and a resolution of 1/256 steps. The t_{clk_presc} is the prescaled clock, $t_{clk_presc} = t_{clk} * 2$. Each update is synchronous to the period, hence any change in the duty cycle will happen to the next PWM period without generating artifacts.

Clock and reset manager The clock and reset manager will issue a prescaled clock to the whole system by a factor of 2, except for the multiplicator, which has to run twice as fast w.r.t. the system. A toggle on the reset pin will reset the whole system at the next reset *release*. Meaning, to reset the system, the reset (active low) must go to LOW, then it must be deasserted to HIGH. By doing this, the clock must be always present (sync reset).

When reset is deasserted (HIGH), the manager will start and will keep the rest of the system in reset state for the next 128 t_{clk} cycles (main clock from the pin). This will guarantee that the whole system will be correctly initialized.

Therefore any SPI transaction can take place after at least 128 clock cycles after reset condition is deasserted, otherwise one SPI packet would be lost.

Color wheel processor The logic datapath of the CwPU is shown below:



The CwPU has all the data width of 8 bit, and the energy intensive color discrimination path is active when non in bypass mode only. When active will take the index. Starting from zero, increments the hue progression and compares against this index (i.e. rotates the color wheel) to process at run time with no LUT, the corresponding requested hue. During the rotation, the RGB internal values will also change, increasing and decreasing the hue components to sweep all the combinations to match the requested one. The final value will be used for the next step, which is the tint.

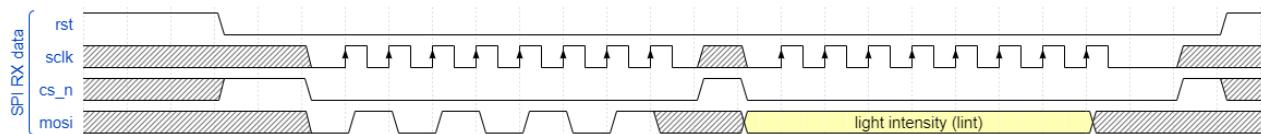
The next step is the sum of the white component, generating a tint, a white adjusted color. It will sum the white up to the maximum value, and the value is output to the intensity multiplier. Also the white is output to the multiplier. This is to not only output an RGB to emulate the white, but to increase the color rendering index (CRI) by allowing to use a single output that can be connected to a pure white generator/phosphor based white LED.

The multiplication for the intensity then takes place with a single multiplicator unit, hence the local control takes care of the data load and synchronization, with 2 clock cycles per operation. Since the multiplicator goes twice as fast, the CwPU has not additional wait states, resulting in 1 CwPU clock cycle delay. Also the white is multiplied. After this step, the output data of each component (R, G, B and W) are 16bit, but the 8 LSB are truncated, generating a final 24 bit color information and 8 bit white.

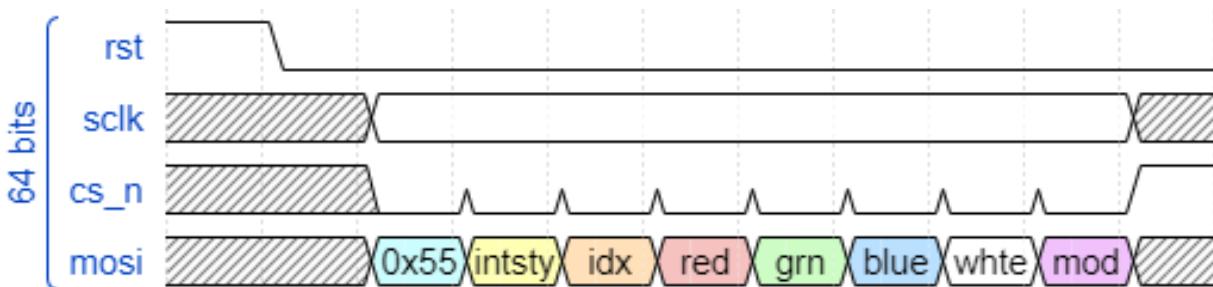
This data is used by the 4 channel PWM modulator.

When in bypass mode, the CwPU will only replicate the same RGBW info in input to the PWM modulator input in one clock cycle.

SPI protocol SPI is Mode 0 as shown in this timing diagram, highlighting the preamble and first byte transfer:



While a whole packet must be compliant with the following diagram:



Which contains:

1. preamble: 0x55
2. intensity: 0x00 - 0xFF
3. color index: 0x00 - 0xFF
4. red: 0x00 - 0xFF
5. green: 0x00 - 0xFF
6. blue: 0x00 - 0xFF
7. white: 0x00 - 0xFF
8. bypass mode: 0xA4 for the color generation, 0x21 bypass

Note that in between each byte is mandatory to toggle the CS signal, since in reality a full transaction is interpreted as 8 individual single byte transactions. Therefore, if the bus gets corrupted, sending any data without preamble with more than 8 bytes, will ensure a clean bus state ready to be synchronized again. Otherwise a reset is an alternative.

How to test

This is normally tested with a micropython script to be interpreted directly from the REPL interface of the TT08 demoboard (see <https://tinytapeout.com/guides/get-started-demoboard/>). To test the design simply setup the demoboard, and run

the script in the test folder. It means it can be simply copy/pasted into the REPL terminal.

To see an output, is suggested to wire some LEDs to the output of the demoboard being careful to not overload the output pins. If you don't know what you are doing, then is better to get like 4x of these for the 4 LEDs tindie.com/products/aleadesigns or any other LED controller that **won't load** more than 4mA on the TT08 chip output pads (see pad spec here).

A custom PMOD will come soon to ease the LED test.

With the RP2040 no input wiring is needed, and the output will be:

uo_out[0] -> Red LED

uo_out1 -> Green LED

uo_out2 -> BLue LED

uo_out[3] -> White LED

What to expect on the outputs

Given the HUE ternary (r,g,b) processed from the index by the CwPU, the final color is $RGBW = ((r,g,b)+w)\text{intensity}$, having a PWM signal per each color channel.

So the white and intensity have a direct impact regardless the hue generated.

The output “color equation” with bypass is $RGBW = \text{spi(red, green, blue, white)}$ with NO intensity, NO automatic white. In this mode, the data provided via SPI is the data taken by the PWM modulator as is.

External hardware

While we're working at a PMOD right now, the external hardware are 4 LEDs, one per each color, connected to the outputs. Be aware that the outputs cannot take more than 4mA!!! So a dedicated circuit is needed (but will be provided soon). Stay tuned.

To control the design, no external controller is needed since it uses the internal RP2040 of the demoboard, see the documentation [here](#) of the test and the REPL script [here](#). Alternatively, a custom firmware and another dedicated python script is provided with the relative STM32 based project, briefly documented [here](#).

Pinout

#	Input	Output	Bidirectional
0		red_pwm	test_out_0
1		green_pwm	test_out_1
2		blue_pwm	test_out_2
3	test_pin	white_pwm	test_out_3
4	cs_n		test_out_4
5	sck		test_out_5
6	mosi		test_out_6
7	clk_div_en		test_out_7

Stochastic Multiplier, Adder and Self-Multiplier [227]

- Author: Ciecen Lestari, Chih-Kuan Ho, David Parent
- Description: Multiplier, Adder and Self-Multiplier using stochastic computing
- GitHub repository
- HDL project
- Mux address: 227
- Extra docs
- Clock: 50000000 Hz

How it works

Design Details

The Stochastic Multiplier, Adder and Self-Multiplier is a digital logic design implementing stochastic arithmetic operations—addition, multiplication, and self-multiplication—using serial 9-bit inputs and outputs. These inputs and outputs are buffered by 1 bit.

Stochastic computing makes use of probability to hold information. LFSRs (Linear feedback serial registers) are used to generate pseudo-random numbers, which are then used by SNGs (Stochastic Number Generators) to generate the stochastic bitstream. The probability of each bit in the stream being a 1 gives the value held by the bitstream, and this is then manipulated by the operations.

Using the bipolar representation, which can represent positive and negative numbers, multiplication can be implemented with an XNOR gate, while addition can be implemented with a MUX.

The design's held inputs are reset every $2^{17}+1$ clock cycle period. The module uses serial-to-parallel input and parallel-to-serial output.

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

Pins Utilization

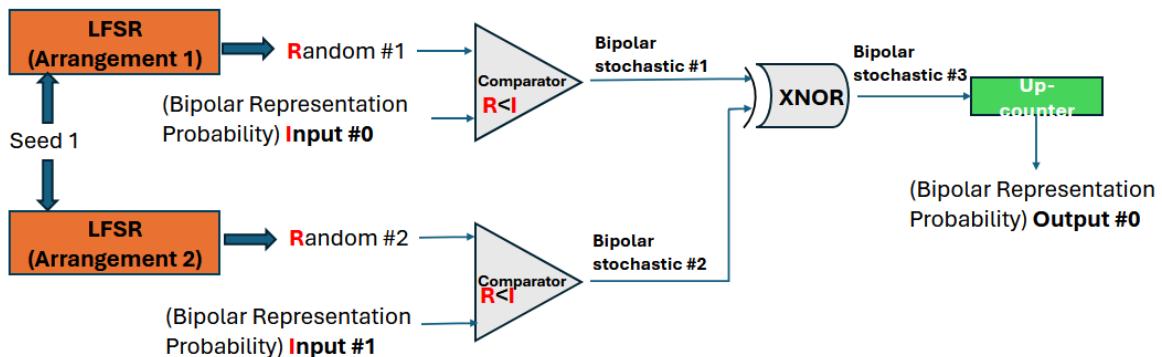
Input Pins:

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

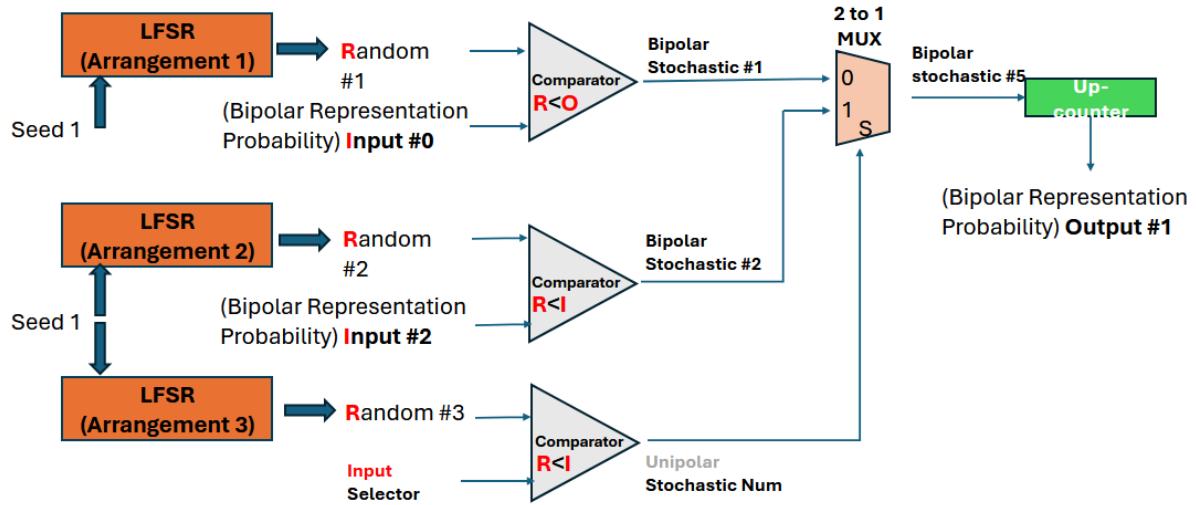
Output Pins:

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

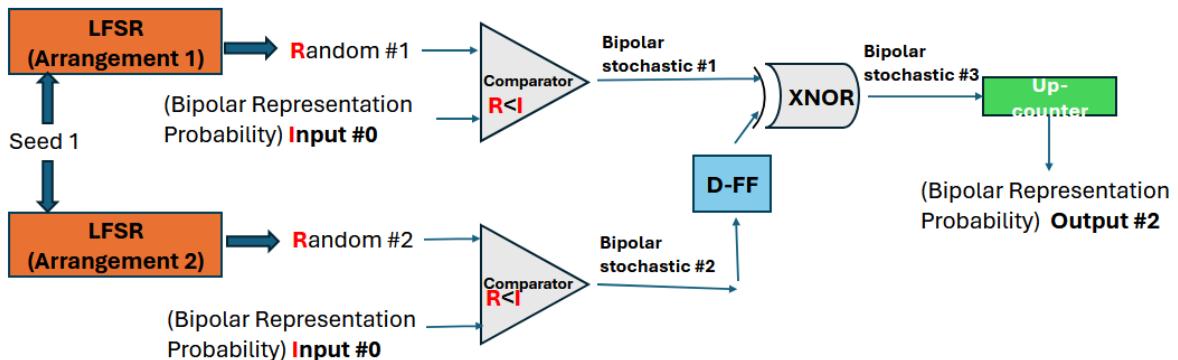
Multiplier: Input 0 * Input 1



Adder: $(\text{Input } 0 + \text{Input } 1) / 2$



Self-Multiplier: $\text{Input } 0 * \text{Input } 1$



D-FF is not necessary but it was included in the design.

REFERENCES USED

General Stochastic Computing Design:

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

B. R. Gaines, "Stochastic computing," in Proceedings of the April 18-20, 1967, spring joint computer conference, in AFIPS '67 (Spring). New York, NY, USA: Association for

Computing Machinery, Apr. 1967, pp. 149–156. doi: 10.1145/1465482.1465505.

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

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

LFSR Design in Stochastic Computing:

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

Digital QIF neuron:

E. J. Basham and D. W. Parent, “Compact digital implementation of a quadratic integrate-and-fire neuron,” 2012 Annual International Conference of the IEEE Engineering in Medicine and Biology Society, San Diego, CA, USA, 2012, pp. 3543–3548, doi: 10.1109/EMBC.2012.6346731.

keywords: {Mathematical model;Clocks;Equations;Vectors;Computational modeling;Field programmable gate arrays;Neurons},

How to test

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

External hardware

ADALM2000

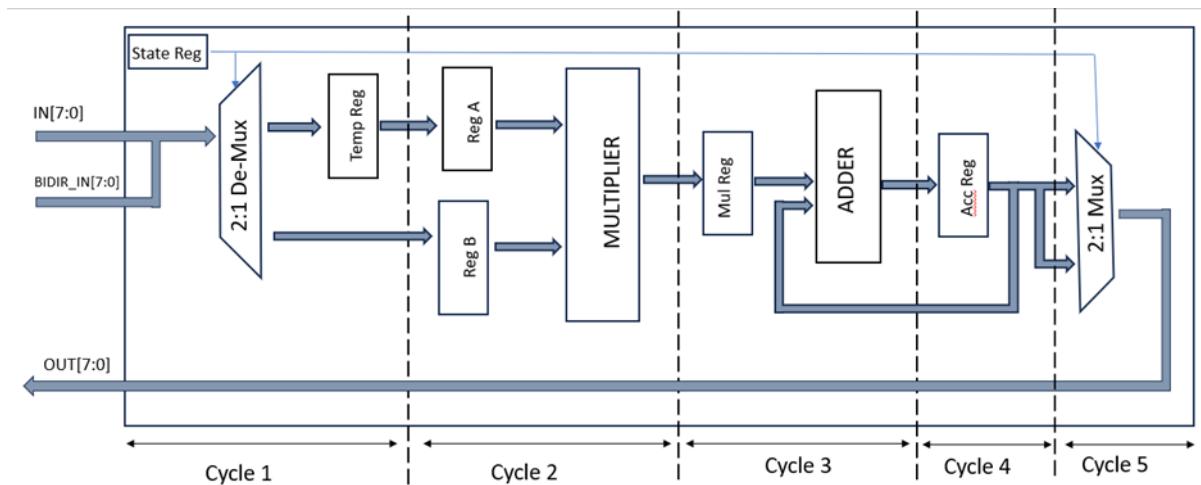
Pinout

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

DL float MAC [229]

- Author: Ananya P & Nidhi M D
- Description: MAC unit for 16 bit DL float data type
- GitHub repository
- HDL project
- Mux address: 229
- Extra docs
- Clock: 40000000 Hz

Design Description



The digital design is a 5 stage pipelined architecture implementation of MAC Operation for 16 bit DLFLOAT numbers. DLFLOAT is a 16-bit floating-point format designed for deep learning training and inference, where speed is prioritized over precision.

Details of DLFLOATs:

Sign bit: 1 bit

Exponent width: 6 bits

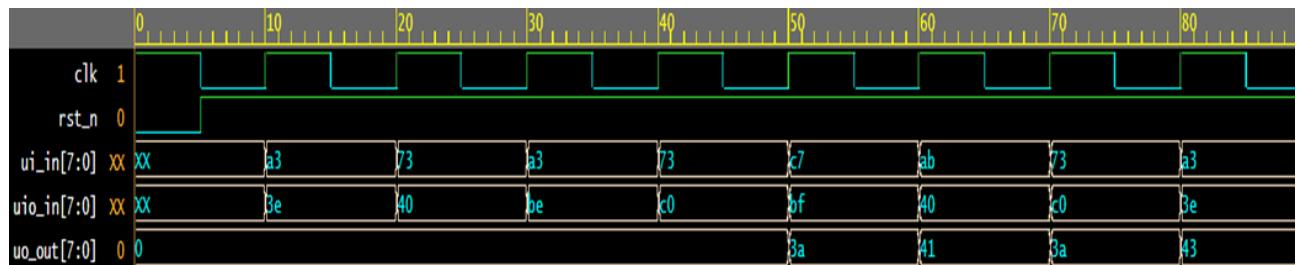
Significand precision: 9 bits

Bias exponent: 31

Value	Binary format
Max normal	S. 111110.111111111
Min normal	S. 000001.000000000
Zero	S. 000000.000000000
Infinity-Nan (combined)	S. 111111.111111111

Work Flow Details:

- The two 16 bit DLFloat input operands are supplied through the ui_in and uio_in (input)pins over two clock cycles getting stored in two registers.
- In the MAC module, the first stage involves multiplying the two inputs, followed by addition of the multiplication result and the accumulated value. The accumulated value in the MAC module starts at zero upon reset.
- After the MAC operation, the 16-bit accumulated result is pushed through uo_out pins over two clock cycles. First the msb 8 bits are pushed out followed by lsb bits.



This arrangement helps in achieving a pipelined architecture where after 5 clock cycles from reset the output values can be pushed out in every cycle.

Here the addition and multiplication follows the IEEE754 algorithm and the MAC operation incorporates handling the special cases like inf, NaN ,subnormals, zero and a full 16 bit precision range.

The Multiplier and Adder blocks also handle overflow and underflow cases with a saturation logic where upon overflow the result is pushed to the largest number that can be represented in the DLFloat format and similarly with underflow the result is pushed to smallest number with the exception that in Multiplier the underflow is pushed to zero to not affect the accumulated results.

How to test

The DLFloat inputs are fed as binary/hexadecimal equivalent of the binary floating point format. The outputs can be read in similar manner

External hardware

An FPGA is required to drive the inputs to the device and needs to be programmed to capture and display the 16-bit result, which arrives as 8 bits over two clock cycles.

Pinout

#	Input	Output	Bidirectional
0	FP16 in[0]	FP16 out[0]/FP16 out[8]	FP16 in[8]
1	FP16 in1	FP16 out1/FP16 out[9]	FP16 in[9]
2	FP16 in2	FP16 out2/FP16 out[10]	FP16 in[10]
3	FP16 in[3]	FP16 out[3]/FP16 out[11]	FP16 in[11]
4	FP16 in[4]	FP16 out[4]/FP16 out[12]	FP16 in[12]
5	FP16 in[5]	FP16 out[5]/FP16 out[13]	FP16 in[13]
6	FP16 in[6]	FP16 out[6]/FP16 out[14]	FP16 in[14]
7	FP16 in[7]	FP16 out[7]/FP16 out[15]	FP16 in[15]

schoolRISCV CPU with Fibonacci program [231]

- Author: Stanislav Zhelnio, Alexander Romanov, Yuri Panchul and Mike Kuskov
- Description: A minimalistic SoC with a schoolRISCV educational CPU and a ROM memory with a program that computes the Fibonacci numbers.
- GitHub repository
- HDL project
- Mux address: 231
- Extra docs
- Clock: 50000000 Hz

How it works

A minimalistic SoC with a schoolRISCV educational CPU and a ROM memory with a program that computes the Fibonacci numbers.

schoolRISCV was originally designed by Stanislav Zhelnio and Alexander Romanov (HSE MIEM) by a suggestion from Yuri Panchul. The goal was to create the simplest possible CPU suitable for the introductory Verilog and FPGA classes. The design was based on a textbook *Digital Design and Computer Architecture* by David Harris and Sarah Harris. Later on Yuri Panchul and Mike Kuskov (Innopolis) adopted the design for the GitHub repositories systemverilog-homework and basics-graphics-music. Now these repos are maintained by the engineers and educators associated with the Verilog Meetup community.

How to test

SystemVerilog testbench A self-checking testbench for the design is located in a directory *test_extra* that contains:

- *clean.bash* - a script to delete temporary files produced by *simulate.bash*.
- *simulate.bash* - a script that simulates the design together with a testbench using Icarus Verilog, producing *log.txt*. Before the simulation, the script compiles assembly *program.s* using the RARS instruction set simulator (ISS) that generates a file *program.hex*. This *program.hex* is used to fill the ROM for both simulation and synthesis.
- *tb.sv* - a self-checking testbench that generates a log and the status *PASS* or *FAIL*.

cocotb testbench The cocotb testbench just runs the simulation for 300 clock cycles checking that the value of the lowest two bits of the dedicated outputs *uo_out* is equal to 01 at the end, which corresponds to self-diagnostics *PASS* and not *FAIL*.

Post silicon After the manufacturing, the design can be manually tested by resetting, driving a clock, and observing the outputs. If the LED connected to the bit 0 of the dedicated outputs (*uo_out*) turns on (*PASS*) and the LED connected to bit 1 turns off (*FAIL*) the design probably works.

Furthermore, you can drive a slow 3 Hz clock and observe the LEDs connected to the bidirectional signals *ui0_out*. Those pins are configured as outputs and they output the lowest 4 bits of the CPU program counter (PC) and the lowest 4 bits of the RISC_V architecture register *a0* (register 10) that contains the currently computed Fibonacci number.

External hardware

LEDs.

Pinout

#	Input	Output	Bidirectional
0	Test pass	CPU reg <i>a0[0]</i>	
1	Test fail	<i>a01</i>	
2		<i>a02</i>	
3		<i>a0[3]</i>	
4		Program Counter <i>pc[0]</i>	
5		<i>pc1</i>	
6		<i>pc2</i>	
7		<i>pc[3]</i>	

Rounding error [233]

- Author: Edwin Török
- Description: Competition entry
- GitHub repository
- HDL project
- Mux address: 233
- Extra docs
- Clock: 25250000 Hz

How it works

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

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

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

The “eye” Z coordinate is animated between 3.5 and 4.5 in 256 steps, where each frame is one step.

The design runs at 640x480@60Hz.

How to test

Set pin ui[0] to 0 to run the default demo. Set pin ui[0] to 1 to show a test image with color bars.

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

The audio is a very simple mix of hsync and vsync signals.

External hardware

Connect according to the Demoscene rules

- VGA output using Leo's VGA PMOD on pins uo[0-7], connected to a monitor supporting 640x480 resolution.
- Audio output using Mike's audio PMOD on ui0[7]

Pinout

#	Input	Output	Bidirectional
0	test mode (0=no, 1=yes)	r1	
1		g1	
2		b1	
3		vsync	
4		r0	
5		g0	
6		b0	
7		hsync	PWM output

SIC-1 8-bit SUBLEQ Single Instruction Computer [234]

- Author: Uri Shaked
- Description: Hardware implementation of the 8-bit Single Instruction Computer
- GitHub repository
- HDL project
- Mux address: 234
- Extra docs
- Clock: 0 Hz

How it works

SIC-1 is an 8-bit Single Instruction computer. The only instruction it supports is SUBLEQ: Subtract and Branch if Less than or Equal to Zero. The instruction has three operands: A, B, and C. The instruction subtracts the value at address B from the value at address A and stores the result at address A. If the result is less than or equal to zero, the instruction jumps to address C. Otherwise, it proceeds to the next instruction.

Memory map The SIC-1 computer has an address space of 256 bytes, and an 8-bit program counter. The first 253 bytes are used for the program memory, and the last 3 bytes are used for input, output, and for halting the computer:

Address	Label	Read	Write
253	@IN	ui pins	Ignored
254	@OUT	Returns 0	uo pins
255	@HALT	Returns 0	Ignored

Setting the program counter to 253, 254, or 255 will halt the computer.

Each instruction is 3 bytes long, and the program counter is incremented by 3 after each instruction, except when a branch is taken.

For more information, check out the SIC-1 Assembly Language Reference.

Execution cycle Each instruction takes 6 cycles to execute, regardless of whether a branch is taken or not. The execution of an instruction is divided into the following stages:

1. Fetch A: Read the value at address PC

2. Fetch B: Read the value at address PC+1
3. Fetch C: Read the value at address PC+2
4. Read valA: Read the value at address A
5. Read valB: Read the value at address B
6. Store: Subtract valB from valA, store the result at A, and branch if the result is less than or equal to zero.

The pseudocode for the execution cycle is as follows:

```
(1) A <= memory[PC]
(2) B <= memory[PC+1]
(3) C <= memory[PC+2]
(4) valA <= memory[A]
(5) valB <= memory[B]
(6) result <= valA - valB
    memory[A] <= result
    if result <= 0:
        PC = C
    else:
        PC = PC + 3
```

Control signals The ui pins are used to load a program into the computer, and to control the computer:

ui pin	Name	Type	Description
0	run	input	Start the computer
1	halted	output	Computer has halted
2	set_pc	input	Set the program counter to the value on ui pins
3	load_data	input	Load the value from the ui pins into the memory at the PC
4	out_strobe	output	Pulsed for one clock cycle when the computer writes to @OUT (

Programming the SIC-1

You can use the [<https://jaredkrinke.itch.io/sic-1>] (online SIC-1 app) to compile and simulate your SIC-1 programs. Click on “Run game” and then “Apply for the job”, close the “Electronic mail” popup. Paste the code and click on “Compile” (on the bottom left). You’ll see the compiled code in the “Memory” window on the right, and will be able to step through the code.

To load a program and run a program, follow this sequence:

1. Set the ui pins to 0 (target address)
2. Pulse the the load_pc pin
3. Set the ui pins to the value you want to load
4. Pulse the load_data pin
5. Repeat steps 3-4 until you have loaded the entire program
6. Set the ui pins to the address you want to start at (usually 0)
7. Pulse the set_pc pin
8. Set the run pin to 1. The computer will start running the program, and the halted pin will go high when the program is done.

If you want to step through the program, you can pulse the run pin to advance one instruction at a time.

Pinout

#	Input	Output	Bidirectional
0	in[0]	out[0]	run
1	in1	out1	halted
2	in2	out2	set_pc
3	in[3]	out[3]	load_data
4	in[4]	out[4]	out_strobe
5	in[5]	out[5]	
6	in[6]	out[6]	
7	in[7]	out[7]	

Sea Battle [235]

- Author: Yuri Panchul
- Description: Sea Battle is a VGA game with sprites for the Tiny Tapeout Demoscene competition.
- GitHub repository
- HDL project
- Mux address: 235
- Extra docs
- Clock: 23000000 Hz

How it works

Sea Battle is a VGA game with sprites for the Tiny Tapeout Demoscene competition.

The *Sea Battle* design is used as a part of basics-graphics-music GitHub repository of Verilog examples, which is maintained by the Verilog Meetup community.

The game uses two keys, *left* and *right*, to control a torpedo. Pressing any key starts the movement. The goal is to hit the moving target.

The design is supposed to work on a 23 MHz frequency and connect to a VGA display using a Tiny VGA board with 2 bits per color channel.

How to test

The design was tested on several FPGA boards and has no self-checking Verilog test-bench for simulation. We just hope it is going to work on ASIC silicon as is.

External hardware

Buttons and a Tiny VGA connector.

Pinout

#	Input	Output	Bidirectional
0	Key right	VGA red 1	
1	Key left	VGA green 1	
2		VGA blue 1	
3		VGA vsync	

#	Input	Output	Bidirectional
4		VGA red [0]	
5		VGA green [0]	
6		VGA blue [0]	
7		VGA hsync	

Comm_IC [237]

- Author: Bhavuk
- Description: Communication protocols: UART, SPI, I2C
- GitHub repository
- HDL project
- Mux address: 237
- Extra docs
- Clock: 20000000000 Hz

How it works

Top module for the Comm_IC project. Submitted for the TinyTapeout8 (TT8).

Designed by: Bhavuk

Github ID: Bhavuk-HDL

Date of creation: 04-Sept-2024

Code version: V01

This project combines three different communication protocols, namely:

1. UART: Universal Asynchronous Receiver Transmitter
2. SPI: Serial Peripheral Interface
3. I2C: Inter Integrated Circuit

To communicate with this project, there is 'data_en' signal.

data_en should be low by default. When it gets high it receives 4 bit data from data_in (MSB first) based on the clk rising edge.

First 4-bits of data bits will decide the comm. protocol and readwrite.

data_in = 4'bab_cd:

ab: 00-> Read

ab: 11-> Write

cd: 00-> UART

cd: 01-> SPI

cd: 10-> I2C

ab: 10-> Use previous settings: valid only in 'write mode'.

Second 4-bits will have two directions: 'read mode' or 'write mode'.

Read mode: data will be read from the comm protocol and interrupt will be set to '0'.

Write mode: if cd was set to '11' in the last cycle, we use previous settings for the communication. Otherwise we use fresh settings.

Next few 4-bit sequences will be used to send the data to resp. module.

How to test

Refer to the test_bench folder in src for test cases.

External hardware

Not applicable

Pinout

#	Input	Output	Bidirectional
0	UART_RX	UART_TX	SDA_out
1	MISO	SEN	new_uart
2	data_en	SCLK	data_out[0]
3		MOSI	data_out1
4		SCL	data_out2
5		busy_uart	data_out[3]
6		busy_spi	error_i2c
7		busy_i2c	

16 Mic Beamformer [239]

- Author: Armaan Gomes
- Description: A 0 delays fixed delay and sum beamformer that can utilize up to 16 input microphones
- GitHub repository
- HDL project
- Mux address: 239
- Extra docs
- Clock: 0 Hz

How it works

It does stuff (testing) Explain how your project works

How to test

You can test it (testing) Explain how to use your project

External hardware

You need 16 digital microphones, a clock generator (can be a raspberry pi, microcontroller, etc.), and something that receives the I2S output (this can be a raspberry pi or most audio output devices). List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Pinout

#	Input	Output	Bidirectional
0	PDM Input Mics 0,1	I2S Out	Bit Clock (3.072 MHz)
1	PDM Input Mics 2,3		LR Clock (48kHz)
2	PDM Input Mics 4,5		
3	PDM Input Mics 6,7		
4	PDM Input Mics 8,9		
5	PDM Input Mics 10,11		
6	PDM Input Mics 12,13		
7	PDM Input Mics 14,15		

PDM Pitch Filter [241]

- Author: Armaan Gomes
- Description: It uses a moving average filter and decimator to filter out a specific frequency
- GitHub repository
- HDL project
- Mux address: 241
- Extra docs
- Clock: 0 Hz

How it works

Explain how your project work This project pitch filters a microphone input stream. Because the bitstream is pdm (1 or -1 at 3.072 Mhz) a sine wave of certain frequencies has a certain length at which its average energy is 0. By making a moving average filter of that length we can eliminate that frequency and its harmonics

How to test

Connect a microphone to the pin and use the spi port to se thte decimator and filter length . Inprogress Explain how to use your project

External hardware

A pdm microphone spi input and clock generator List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Pinout

#	Input	Output	Bidirectional
0	Bit Clock (3.072 Mhz)	PCM Out Mic 0	Delay Select 0
1	LR Clock (48Khz)	PCM Out Mic 1	Delay Select 1
2	PDM Input Mics 0,1	PCM Out Mic 2	Delay Select 2
3	PDM Input Mics 2,3	PCM Out Mic 3	Delay Select 3
4	PDM Input Mics 4,5	PCM Out Mic 4	Delay Select 4
5	PDM Input Mics 6,7	PCM Out Mic 5	Beamformed PCM Output
6		PCM Out Mic 6	Mic Clock

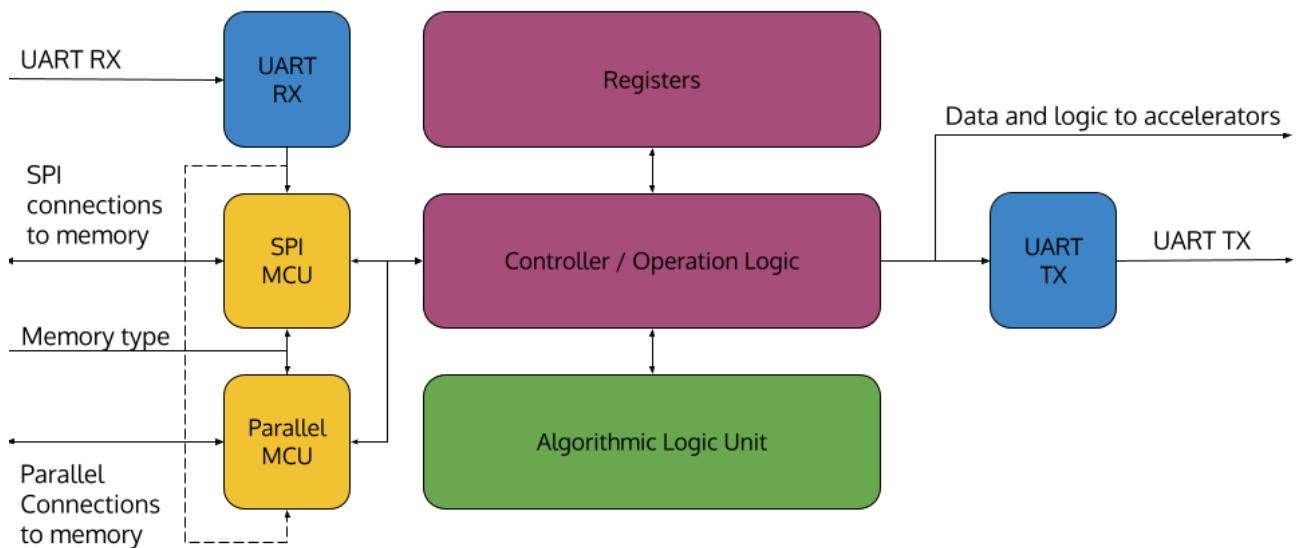
#	Input	Output	Bidirectional
7		PCM Out Mic 7	

Zoom Zoom [242]

- Author: Justin T, Andrew H, Simon Y, Kellen Y, Vallabh A, Nicole C
- Description: Custom Cpu with custome external memory bus and sha-3 and CORDIC accelerators
- GitHub repository
- HDL project
- Mux address: 242
- Extra docs
- Clock: 60000000 Hz

What is Zoom Zoom?

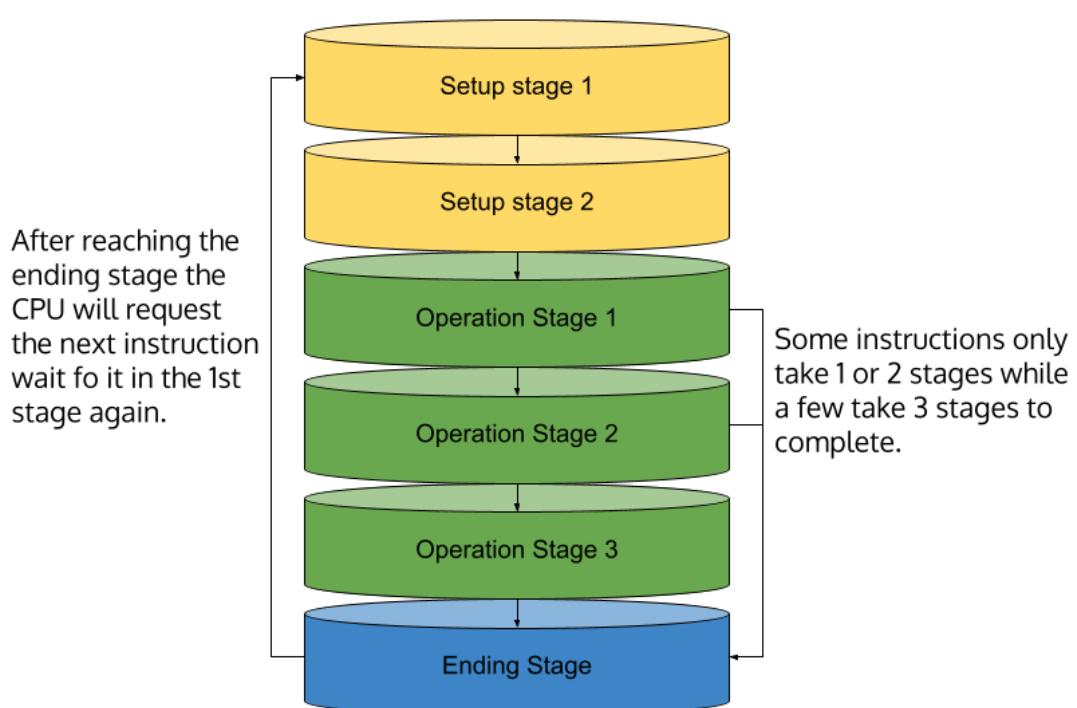
Zoom Zoom is a custom, 16-bit, barebones CPU. We store memory externally using either a custom parallel connection or SPI. We also have a simple UART protocol implemented on the CPU as well as numerous accelerators(that may not be included in the final design due to size constraints). (Link to Document with helpful coding info)

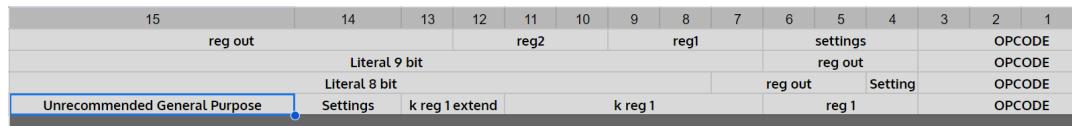


Detailed List of Features

- Custom Architecture and ISA
 - 16-bit instructions
 - 5 types of instructions
- 6 general purpose registers
 - 1 flag register
 - 1 zero register
- UART Interface
- SPI and Custom Parallel Mememoy Interface
 - 16 bit memory address
 - supports up to 65536 memory addresses(2^{16})
- Flexible design easy integration of accelerators as instructions

The Architecture





Instruction Layout

General Instructions

Instruction	Name	Type	Opcode	Settings	Description
nop	No Operation	0	0000		
ld	Load	A	0101	0	reg out = mem[mem[inst addr + 1]]
ldr	Load Register	A	0101	1	reg out = mem[reg1]
str	Store	A	0110	0	mem[mem[inst addr + 1]] = reg2
strr	Store Register	A	0110	1	mem[reg1] = reg2
ldi	Load Immediate	L	0111		reg out = L9[7:15]

ALU Instructions

Instruction	Name	Type	Opcode	Settings	Description
add	Add	A	0001	000	reg out = reg1 + reg2
sub	Subtract	A	0001	001	reg out = reg1 - reg2
mult	Multiply	A	0001	010	reg out = reg1[0:7] * reg 2[0:7]
nand	NAND	A	0001	011	reg out = !(reg1 & reg2)
addi	Add Immediate	I	0010	0	reg out = register 2 + L8[0:7]
multi	Multiply Immediate	I	0010	1	reg out = register 2 * L8[0:7]
shl	Shift Left	A	0001	100	reg out = reg1 « 0
shr	Shift Right	A	0001	101	reg out = reg1 » 0

Branching Instructions

Instruction	Name	Type	Opcode	Settings	Description
jmp	Jump	A	0100	000	inst addr = reg1
jmpz	Jump if Zero	A	0100	001	reg_out = inst addr; if (ZF)
jmpg	Jump if Greater	A	0100	010	reg_out = inst addr; if (GF)
jmpe	Jump if Equal	A	0100	111	reg_out = inst addr; if (EF)
jmpl	Jump if Less	A	0100	011	reg_out = inst addr; if (!GF)
jppm	Jump if Memory Flagged	A	0100	100	reg_out = inst addr; if (MF)
jpu	Jump if UART Flagged	A	0100	101	reg_out = inst addr; if (UF)
jpi	Jump Immediate	A	0100	110	inst addr = mem[inst addr + 1]

Programming the CPU

Memory Address 769 is reserved: The Assembler does not give a warning currently!

To assemble, we use custoasm with installation instructions here. We recommend installation via rust's package manager by running cargo install customasm. You can then compile an assembly file by running customasm -o <outputfilename> <filename>. The format for the assembly file is to add #include "x3q16_ruleset.asm" to the top of each .asm file as well as that file which is located here. Instruction memory and General Purpose are all located in the same place. Thus, to store general values in memory, just jump to wherever you store it in memory.

Accelerators

Many are still a work in progress or aren't supported by the assembler

Keccakf1600 Approximately 50% of the computational time for the Kyber Algorithm is hashing needed for random number generation. The Kyber algorithm uses SHA-3 and SHAKE algorithms to generate cryptographically secure random polynomials and numbers. Both of these algorithm rely on the keccakf1600 state permutation which target to accelerate. More information on the keccak algorithm can be found here and the kyber algorithm here.

The branch keccak_integration holds a complete state permuation accelerator however this is not included in main since it's too big to fit for tinytapeout. A smaller accelerator is currently being worked on.

How to test

Generate the binary file from test/x3q16 and load it into memory. Reset the chip and see if anything is written in memory.

External hardware

Either a SPI ram chip or a MCU emulator or parallel storage with custom protocol

Pinout

#	Input	Output	Bidirectional
0	lower_byte_in	write_enable	DATA0
1	upper_byte_in	register_enable	DATA1
2	rx	read_enable	DATA2
3	IN3	lower_bit	DATA3
4	IN4	tx	DATA4
5	IN5	upper_bit	DATA5
6	IN6	OUT6	DATA6
7	IN7	OUT7	DATA7

PDM Correlator [243]

- Author: Armaan Gomes
- Description: A chip that performs either cross or auto correlation on PDM microphone inputs
- GitHub repository
- HDL project
- Mux address: 243
- Extra docs
- Clock: 0 Hz

How it works

It performs an XOR on two input bitstreams and sums the result. The lower this value is the higher correlation. Explain how your project works

How to test

Connect microphones to pins and stuff Explain how to use your project

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	Bit Clock (3.072 Mhz)	PCM Out Mic 0	Delay Select 0
1	LR Clock (48Khz)	PCM Out Mic 1	Delay Select 1
2	PDM Input Mics 0,1	PCM Out Mic 2	Delay Select 2
3	PDM Input Mics 2,3	PCM Out Mic 3	Delay Select 3
4	PDM Input Mics 4,5	PCM Out Mic 4	Delay Select 4
5	PDM Input Mics 6,7	PCM Out Mic 5	Beamformed PCM Output
6		PCM Out Mic 6	Mic Clock
7		PCM Out Mic 7	

SPI Logic Analyzer with Charlieplexed Display [258]

- Author: ParallelLogic-
- Description: Displays contents of register map on charlieplexed display. Generates waveforms for PWM, UART, WS2812 in response to trigger.
- GitHub repository
- HDL project
- Mux address: 258
- Extra docs
- Clock: 10000000 Hz

How it works

The bi-directional pins are used to drive a charlieplexed 8*7 LED display. A SPI serial connection is used to set the values in a register map. Auxilary functions are implemented, space/time permitting, ex: LFSR, PWM, freqency counting, ultrasonic distance sensing

How to test

Use SPI to read/write values to the register map, observe the output on the LEDs and/or in the serial response. CS active low SPI MODE 0 SPI_CLK <= SYS_CLK/2 Most signigicant bit is exchanged first

External hardware

Charlielexed 7*8 LED display

Pinout

#	Input	Output	Bidirectional
0	CS	ASIC_OUT_0	MAT0
1	SCLK	ASIC_OUT_1	MAT1
2	MOSI	ASIC_OUT_2	MAT2
3	TRIGGER	ASIC_OUT_3	MAT3
4	ASIC_IN_0	ASIC_OUT_4	MAT4
5	ASIC_IN_1	ASIC_OUT_5	MAT5
6	ASIC_IN_2	ASIC_OUT_6	MAT6

#	Input	Output	Bidirectional
7	ASIC_IN_3	MISO	MAT7

Find The Damn Issue [259]

- Author: Leonel Gouveia Ergin (Synogate), Michael Offel (Synogate)
- Description: USB to UART/SPI/I2C/JTAG/GPIO adapter
- GitHub repository
- HDL project
- Mux address: 259
- Extra docs
- Clock: 12000000 Hz

How it works

It is a bit bang device to interface that can be used to communicate to various devices over UART, SPI, 3wire, I2C, JTAG, GPIO or many custom protocols. To the host it registers as a USB Communication Device Class (CDC) device.

UART mode It is in UART mode by default. In UART mode it can be used as a standard CDC device with configurable baud rate and 8 data bits. You can make use of any tool that supports COM ports, like Visual Studio's Serial Monitor, to send and receive data or configure the baud rate.

Pins TX, RX, DTR and RTS are used in UART mode. DTR and RTS can be set by most tools and can be used as GPIO. There is no flow control implemented.

BitBang mode In BitBang mode, the device can be used similar to an FTDI MPSSE. **To enter BitBang mode set the baud rate to 57600 and parity to even.** A description of the protocol and its commands can be found in `/libs/gatery/doc/BitBangEngine/BitBangEngine.md`. There is also a collection of examples and a c++ header-only API in `/example/`. In contrast to the FTDI chips this is not a clone. It does not pretend to be from FTDI, nor does it support the FTDI driver or API. It acts as a standard CDC device in BitBang mode and can be used by and program that supports writing and reading to serial ports.

Note that on tiny tapeout we choose to follow the pinout templates of the tiny tapeout wiki. The documentation is written for the default pinout. Instead, refer to the pinout table below for each pins function.

How to test

1. Connect a device to communicate to. A good one is a LIS2DH12, for its wide range of protocols and available example code in this repo.
2. Connect USB_DP and USB_DN to your computer's USB. And attach the external pull up of 1.5k ohm to 3.3V.
3. Compile and run /example/LIS2DH12.cpp using CMake on Linux or Windows.
4. You should see sensor readings from the device on your screen.

External hardware

An external pull up of 1.5k to 3.3V on USB_DP is required.

Pinout

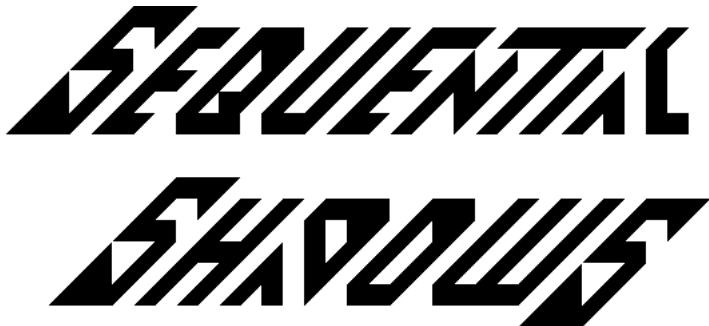
#	Input	Output	Bidirectional
0	GPIOh0	GPIOh0/DTR	GPIOI0-CS
1	GPIOh1	GPIOh1/RTS	GPIOI1-MOSI/TX
2	GPIOh2	GPIOh2	GPIOI2-MISO/RX
3	GPIOh3	GPIOh3	GPIOI3-CLK
4	GPIOh4	GPIOh4	GPIOI4-TMS
5	GPIOh5	GPIOh5	GPIOI5-WAIT
6	GPIOh6	GPIOh6	USB_DP
7	GPIOh7	GPIOh7	USB_DN

Sequential Shadows Deluxe [TT08 demo competition] [262]

- Author: Toivo Henningsson
- Description: My contribution to the TT08 demo competition, extended version
- GitHub repository
- HDL project
- Mux address: 262
- Extra docs
- Clock: 50400000 Hz

Intro

Curly / Medieval presents



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

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

How to test

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

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

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

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

If using A Pmod VGA as output, set `rgb444_mode` unless you want the original RGB222 experience.

For the demo competition, set `pmod_vga_pinout` and `rgb444_mode` if you have a Pmod VGA, and please consider if you can still hook up the sound. Don't set any of the other inputs.

External hardware

This project needs

- either
 - a TinVGA VGA Pmod.
 - Mike's audio Pmod.
- or a Pmod VGA

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

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

Pinout

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

DDC [266]

- Author: Armaan Gomes
- Description: Converts I2S input to PDM output
- GitHub repository
- HDL project
- Mux address: 266
- Extra docs
- Clock: 0 Hz

How it works

It uses an inverted cic filter and modulator to convert an i2s signal to pdm Explain how your project works

How to test

Can and I2s output and a pdm input deive Explain how to use your project

External hardware

I2s Output device and pdm input device

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

Pinout

#	Input	Output	Bidirectional
0	Bit Clock (3.072 Mhz)	PCM Out Mic 0	Delay Select 0
1	LR Clock (48Khz)	PCM Out Mic 1	Delay Select 1
2	PDM Input Mics 0,1	PCM Out Mic 2	Delay Select 2
3	PDM Input Mics 2,3	PCM Out Mic 3	Delay Select 3
4	PDM Input Mics 4,5	PCM Out Mic 4	Delay Select 4
5	PDM Input Mics 6,7	PCM Out Mic 5	Beamformed PCM Output
6		PCM Out Mic 6	Mic Clock
7		PCM Out Mic 7	

mulmul [270]

- Author: JJ Wong
- Description: Small 4-bit vector multiplication engine
- GitHub repository
- HDL project
- Mux address: 270
- Extra docs
- Clock: 0 Hz

How it works

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

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

How to test

You can run the testbench tests in the test dir.

External hardware

Will be programmed by RP2040. No other external hardware.

Pinout

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

#	Input	Output	Bidirectional
7	op1	out[7]	

Warp [274]

- Author: sylefeb
- Description: Demo on TinyTapeout? Let's do something!
- GitHub repository
- HDL project
- Mux address: 274
- Extra docs
- Clock: 25000000 Hz

Warp

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

How it works

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

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

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

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

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

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

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

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

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

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

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

External hardware

- VGA PMOD
- Audio PMOD

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

Pinout

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

Supermic [275]

- Author: Armaan Gomes, Asmi Sawant, Ria Saheta, Vikhaash Kanagavel Chithra, Morgan Packard, Sanjay Ravishankar
- Description: A 8 channel customizable beamforming signal processor
- GitHub repository
- HDL project
- Mux address: 275
- Extra docs
- Clock: 0 Hz

How it works

Cool stuff makes cool stuff happen Explain how your project works

How to test

Plug cool stuff into the chip and it will output cool stuff Explain how to use your project

External hardware

You need some cool microphones and a cool clock generator and a cool i2s receiver
List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Pinout

#	Input	Output	Bidirectional
0	Bit Clock (3.072 Mhz)	PCM Out Mic 0	Delay Select 0
1	LR Clock (48Khz)	PCM Out Mic 1	Delay Select 1
2	PDM Input Mics 0,1	PCM Out Mic 2	Delay Select 2
3	PDM Input Mics 2,3	PCM Out Mic 3	Delay Select 3
4	PDM Input Mics 4,5	PCM Out Mic 4	Delay Select 4
5	PDM Input Mics 6,7	PCM Out Mic 5	Beamformed PCM Output
6		PCM Out Mic 6	Mic Clock
7		PCM Out Mic 7	

DPM_Unit [289]

- Author: Sanjay Kumar M, Shylashree N, Ravish Aradhya H V, RV College Of Engineering, Neha R V, PES University
- Description: Design and Implementation of Dynamic Power management unit
- GitHub repository
- HDL project
- Mux address: 289
- Extra docs
- Clock: 100 Hz

Credits : We gratefully acknowledge the COE in Integrated Circuits and Systems (ICAS) and Department of ECE. Our special thanks to Dr K S Geetha (Vice Principal) and, Dr. K N Subramanya (principal) for their constant support and encouragement to do TAPEOUT in Tiny Tapeout 8 .

The code provided is a SystemVerilog module that implements a Dynamic Power Management Unit (DPMU) for an SoC (System on Chip). The DPMU dynamically adjusts voltage and frequency levels based on inputs such as performance requirements, temperature, battery level, and workload. The module uses a finite state machine (FSM) to manage transitions between different power states.

Key Components

Inputs and Outputs: Inputs (ui_in): The primary input signals include performance requirements, temperature sensor data, battery level, and workload. Outputs (uo_out, uio_out): These include the power-saving indicator, voltage levels, and frequency levels for different cores and memory. I/O (uio_in, uio_out, uio_oe): Handles bidirectional signals; however, in this design, uio_in is not used, and uio_out is used for output.

Internal Signals:

State Variables: state and next_state manage the FSM that controls the DPMU's behavior. **Power and Frequency Controls:** Registers like vcore1, vcore2, vmem, fcore1, fcore2, and fmem store the voltage and frequency settings. **Finite State Machine (FSM):**

States: NORMAL: Default operating mode with standard voltage and frequency levels. PERFORMANCE: High-performance mode with maximum voltage and frequency levels. POWERSAVE: Low-power mode with reduced voltage and frequency levels. THERMAL_MANAGEMENT: Mode to handle high temperature by adjusting power levels moderately. BATTERY_SAVING: Mode to conserve battery by minimizing voltage and frequency levels.

State Transitions: Transitions between states occur based on the input conditions, such as high performance request, low battery level, high temperature, or low workload.

Detailed Walkthrough Input and Output Mapping:

perf_req: Mapped to the least significant bit (LSB) of ui_in, indicating whether high performance is needed. **temp_sensor:** 2-bit signal derived from ui_in[3:2], providing temperature data. **battery_level:** 2-bit signal derived from ui_in[5:4], indicating the battery's charge status. **workload_core:** 3-bit signal derived from ui_in[7:6], representing the workload of a core. **State Logic:**

On each clock cycle (clk), the FSM checks the state and evaluates transitions based on inputs. In NORMAL state, if perf_req is high, the system transitions to PERFORMANCE state. If the battery level is low, it transitions to BATTERY_SAVING state. If the temperature is high, it transitions to THERMAL_MANAGEMENT state. If the workload is low, it transitions to POWERSAVE state. PERFORMANCE state sets all voltages and frequencies to maximum. If perf_req drops, it returns to NORMAL. POWERSAVE state reduces voltages and frequencies to conserve power. If the workload increases, it returns to NORMAL. THERMAL_MANAGEMENT state adjusts power levels to moderate values to manage high temperatures. If the temperature normalizes, it returns to NORMAL. BATTERY_SAVING state minimizes voltages and frequencies to conserve battery. If the battery level increases, it returns to NORMAL.

Output Assignment: The combined voltage (vcore1, vcore2, vmem) and frequency (fcore1, fcore2, fmem) values are assigned to the uio_out and uo_out outputs. The power_save signal is also part of the output, indicating whether the system is in power-saving mode. **Behavior under Reset:**

When the reset (rst_n) is low (active), the system resets to the NORMAL state.

Here's a table summarizing the expected output (ui_out, uo_out) based on the input (ui_in) and time using the provided testbench for the tt_um_dpmu module. The table provides the values for different states as the ui_in input changes over time.

Table: Testbench Expected Output

Time (ns)	ui_in (Input)	State	ui_out (Expected Output)
0	11110010	NORMAL	01010110
10	00010010	PERFORMANCE	11111111
30	11110010	NORMAL	01010110
50	11110011	THERMAL_MANAGEMENT	10101011
70	11110010	NORMAL	01010110
90	11101010	THERMAL_MANAGEMENT	10101011
110	11111010	BATTERY_SAVING	00000000
130	11111110	BATTERY_SAVING	00000000
150	11111010	BATTERY_SAVING	00000000

Time (ns)	ui_in (Input)	State	uio_out (Expected Output)
-----------	---------------	-------	---------------------------

Explanation of Table Columns:

Time (ns): The simulation time when the ui_in input is applied. ui_in (Input): The 8-bit input value applied to the design. State: The state of the FSM based on the ui_in input. The states are NORMAL, PERFORMANCE, THERMAL_MANAGEMENT, and BATTERY_SAVING. uio_out (Expected Output): The expected 8-bit output values for the uio_out signals. uio_out[0]: Power save mode indicator. uio_out[2:1], uio_out[4:3], uio_out[6:5]: Voltage controls. uio_out[7]: Part of fcore1[0]. uo_out (Expected Output): The expected 8-bit output values for the uo_out signals. uo_out[0:1]: Part of fcore1[2:1]. uo_out[4:2]: fcore2[2:0]. uo_out[7:5]: fmem[2:0].

Explanation of Key Points: NORMAL State: When the inputs suggest a typical operating environment (e.g., ui_in = 11110010), the design operates with default voltage and frequency levels. PERFORMANCE State: Triggered by a performance request (perf_req = 1), leading to maximum voltage and frequency levels. THERMAL_MANAGEMENT State: Triggered by high temperature (temp_sensor = 10 or 11), moderates the voltage and frequency to prevent overheating. BATTERY_SAVING State: Triggered by low battery level (battery_level = 00 or 01), minimizing power consumption by reducing voltage and frequency to the lowest levels.

Testbench Operation: The testbench applies different ui_in values at specific simulation times. At each time step, it captures the output values (uio_out and uo_out) and compares them with the expected values as per the design's FSM logic. The \$monitor statement continuously logs the input and output values, helping to verify the design's behavior at each time point.

Pinout

#	Input	Output	Bidirectional
0	ui_in[[0]]	uo_out[0]	uio_out[0]
1	ui_in[1]	uo_out1	uio_out1
2	ui_in[2]	uo_out2	uio_out2
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]

Generate VGA output for Color Blindness Test [291]

- Author: Krushnasis Pradhan, Aniruddha Ranade
- Description: Generate VGA output which shall display the pattern similar to Ishihara Plates
- GitHub repository
- HDL project
- Mux address: 291
- Extra docs
- Clock: 0 Hz

How it works

Generates VGA output for displaying pattern similar to Ishihara plates used for conducting a color blindness test. (Disclaimer: Note that this is not an approved medical test and test setup. The pattern generated is for purely experimental purpose.)

How to test

This project will work out of the box. Just connect a VGA display via TinyVGA PMOD.

External hardware

TinyVGA PMOD to connect to a VGA display

Pinout

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

4-bit CLA [293]

- Author: Wei Zhang
- Description: A 4 bit carry look-ahead adder
- GitHub repository
- HDL project
- Mux address: 293
- Extra docs
- Clock: 0 Hz

How it works

This is a 4-bit CLA. It can be used to construct the adder with higher bit.

How to test

The design has 3 input ports: a, b and ci. It has 2 output ports: s and co. a: an addend. b: the other addend. ci: the carry signal for the input. s: the output sum. co: the carry signal for the output.

External hardware

This project was tested by an U250 FPGA.

Pinout

#	Input	Output	Bidirectional
0	a[0]	s[0]	ci
1	a1	s1	
2	a2	s2	
3	a[3]	s[3]	
4	b[0]	co	
5	b1		
6	b2		
7	b[3]		

SkyKing Demo [295]

- Author: Nicklaus Thompson
- Description: Types some text over an image of a plane flying into the sunset
- GitHub repository
- HDL project
- Mux address: 295
- Extra docs
- Clock: 25200000 Hz

How it works

The project presents an RGB222 VGA signal to the output port.

How to test

Runs automaticaly.

External hardware

VGA PMOD on UO.

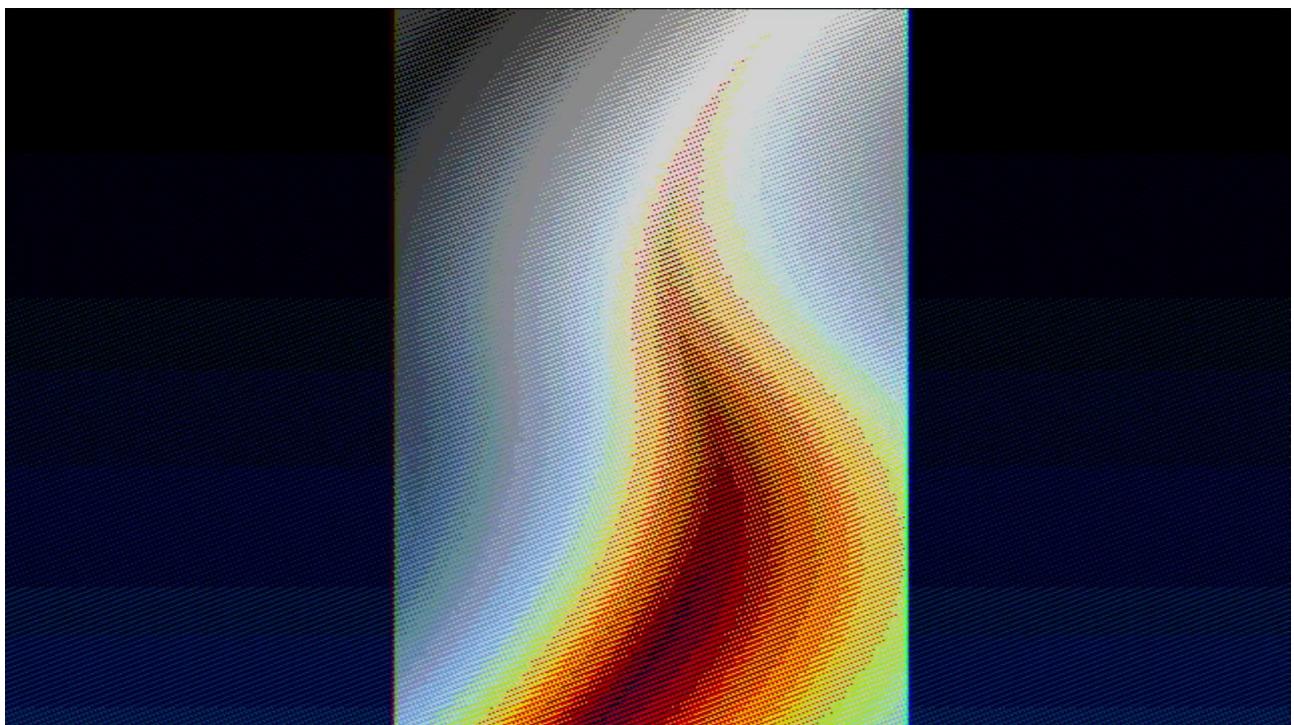
Pinout

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

Flame demo [297]

- Author: Konrad Beckmann & Linus Mårtensson
- Description: Flame demo
- GitHub repository
- HDL project
- Mux address: 297
- Extra docs
- Clock: 25000000 Hz

Flame - Konrad & Linus tinytapeout08 demo compo entry



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

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

External hardware Follows the democompo hardware rules:

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

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

Pinout

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

Metaballs [299]

- Author: Johannes Hoff
- Description: You can't prove it's not metaballs
- GitHub repository
- HDL project
- Mux address: 299
- Extra docs
- Clock: 50000000 Hz

How it works

An attempt at metaballs on a very rushed timeline. Keep your hopes down. Including for this documentation.

How to test

Should work like other VGA projects. No sound.

External hardware

VGA PMOD

Pinout

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

Simple Stopwatch [301]

- Author: Fabio Ramirez Stern
- Description: A simple stopwatch counting in 100th seconds and outputting it via SPI to a MAX7219 chip controlling an 8 digit 7-segment display.
- GitHub repository
- HDL project
- Mux address: 301
- 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 its 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
0	start/stop	SPI MOSI
1	lap time	SPI CS (active low)

#	Input	Output
2	reset (active high)	SPI CLK
3	skip display setup (only output time, active high during reset)	stopwatch enabled (counting)
4		display enabled (goes low when)
5		
6		
7		

PWM generator [303]

- Author: Matea Samuel
- Description: Generate pwm signal with configurable period - 12-bit - and duty cycle - 1%-99%.
- GitHub repository
- HDL project
- Mux address: 303
- Extra docs
- Clock: 50000000 Hz

How it works

This design intend to be used like a PWM generator. It contains two 12-bit registers: one for duty cycle(duty_reg) respectively one for period(period_reg). When sel signal is set to “0” the duty_reg will be selected and when sel is “1” the period_reg is selected. If values for duty/period is set at the input, the value is written in the regs only after wr_en is set to “1”. For duty cycle, will be used only 7 bits(from 0 to 6) the rest of the bits being 0 hardcoded. the value for period_rescan be set between 2-4095(on 12 bits).

How to test

Connect the output to the oscilloscope and verify if the frequency and duty cycle correspond with your expectation.

External hardware

The external hardware will be only the wire used for the pwm_out and 14 inputs (uController, digital pattern generator etc.) to set the period, duty cycle, set and wr_en signals.

Pinout

#	Input	Output	Bidirectional
0	in[0]	pwm_out	in[8]
1	in1		in[9]
2	in2		in[10]

#	Input	Output	Bidirectional
3	in[3]		in[11]
4	in[4]		
5	in[5]		
6	in[6]		sel
7	in[7]		wr_en

DMTD [305]

- Author: Armaan Gomes
- Description: A Dual Mixer Timer Differential
- GitHub repository
- HDL project
- Mux address: 305
- Extra docs
- Clock: 0 Hz

How it works

Cool stuff makes cool stuff happen Explain how your project works

How to test

Plug cool stuff into the chip and it will output cool stuff Explain how to use your project

External hardware

You need some cool microphones and a cool clock generator and a cool i2s receiver
List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Pinout

#	Input	Output	Bidirectional
0	Bit Clock (3.072 Mhz)	PCM Out Mic 0	Delay Select 0
1	LR Clock (48Khz)	PCM Out Mic 1	Delay Select 1
2	PDM Input Mics 0,1	PCM Out Mic 2	Delay Select 2
3	PDM Input Mics 2,3	PCM Out Mic 3	Delay Select 3
4	PDM Input Mics 4,5	PCM Out Mic 4	Delay Select 4
5	PDM Input Mics 6,7	PCM Out Mic 5	Beamformed PCM Output
6		PCM Out Mic 6	Mic Clock
7		PCM Out Mic 7	

I2S to PWM [307]

- Author: Armaan Gomes
- Description: An 8-bit I2S to PWM convertor
- GitHub repository
- HDL project
- Mux address: 307
- Extra docs
- Clock: 0 Hz

How it works

Cool stuff makes cool stuff happen Explain how your project works

How to test

Plug cool stuff into the chip and it will output cool stuff Explain how to use your project

External hardware

You need some cool microphones and a cool clock generator and a cool i2s receiver
List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Pinout

#	Input	Output	Bidirectional
0	Bit Clock (3.072 Mhz)	PCM Out Mic 0	Delay Select 0
1	LR Clock (48Khz)	PCM Out Mic 1	Delay Select 1
2	PDM Input Mics 0,1	PCM Out Mic 2	Delay Select 2
3	PDM Input Mics 2,3	PCM Out Mic 3	Delay Select 3
4	PDM Input Mics 4,5	PCM Out Mic 4	Delay Select 4
5	PDM Input Mics 6,7	PCM Out Mic 5	Beamformed PCM Output
6		PCM Out Mic 6	Mic Clock
7		PCM Out Mic 7	

Basys 3 Over UART Link [322]

- Author: Devin Atkin
- Description: Run the main Basys 3 Peripherals over a 115200 Uart Link
- GitHub repository
- HDL project
- Mux address: 322
- Extra docs
- Clock: 50000000 Hz

How it works

The Basys 3 is a normal board for learning FPGA design or prototyping certain designs. This project runs the main peripherals over a 115200 UART link. This code includes the main block that takes 16 “Led” inputs, 16 “Switch” Outputs, 12 “7 Segment Display” inputs, and 5 “Button” outputs; the block then gives a UART RX and UART TX which are routed to the bi-directional PMOD bus.

How to test

Use the associated PMOD board or interact with the UART. The following are the expected elements on the UART.

- “LD: 0xFFFF” Coming from this design going to the peripheral
- “SW: 0xFFFF” Coming from the peripheral going to the design
- “7S: 0xFFFF” Coming from this design going to the peripheral
- “BT: 0xFFFF” Coming from the peripheral going to the design

External hardware

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

Pinout

#	Input	Output	Bidirectional
0			uart_tx
1			uart_rx
2			uart_tx_ready

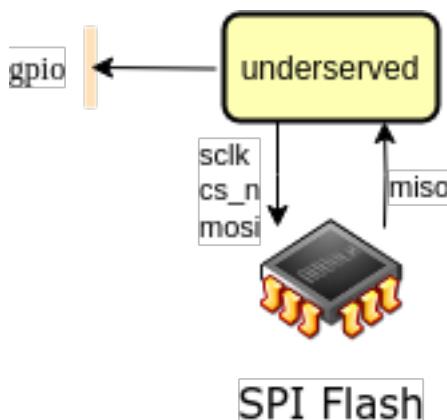
#	Input	Output	Bidirectional
3			uart_tx_valid
4			uart_rx_valid
5			uart_rx_ready
6			
7			

ITS-RISCV [326]

- Author: Bambang T. Wibowo, Chazim Fikri A., Hernanda A. P., M. Hafidzh, Figo A. M., and Faiz S. K.
- Description: ITS RISC V based on the underserved TinyTapeout 07.
- GitHub repository
- HDL project
- Mux address: 326
- Extra docs
- Clock: 20000000 Hz

How it works

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



How to test

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

Underserved can easiest be run locally using FuseSoC.

Install FuseSoC

```
pip install fusesoc
```

Create and enter a new workspace

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

Register underserved as a library in the workspace

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

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

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

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

```
fusesoc core list
```

Show info about underserved

```
fusesoc core show underserved
```

Run linting (static code checks) using Verilator

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

Run underserved testbench

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

Run with modelsim instead of default tool (icarus)

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

External hardware

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

Pinout

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

Zilog Z80 [330]

- Author: ReJ aka Renaldas Zioma
- Description: Z80 open-source silicon. Goal is to become a silicon proven, pin compatible, open-source replacement for classic Z80.
- GitHub repository
- HDL project
- Mux address: 330
- Extra docs
- Clock: 16000000 Hz

How it works

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

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

The future work

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

Z80 technical capabilities

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

Z80 registers

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

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

Z80 Pinout

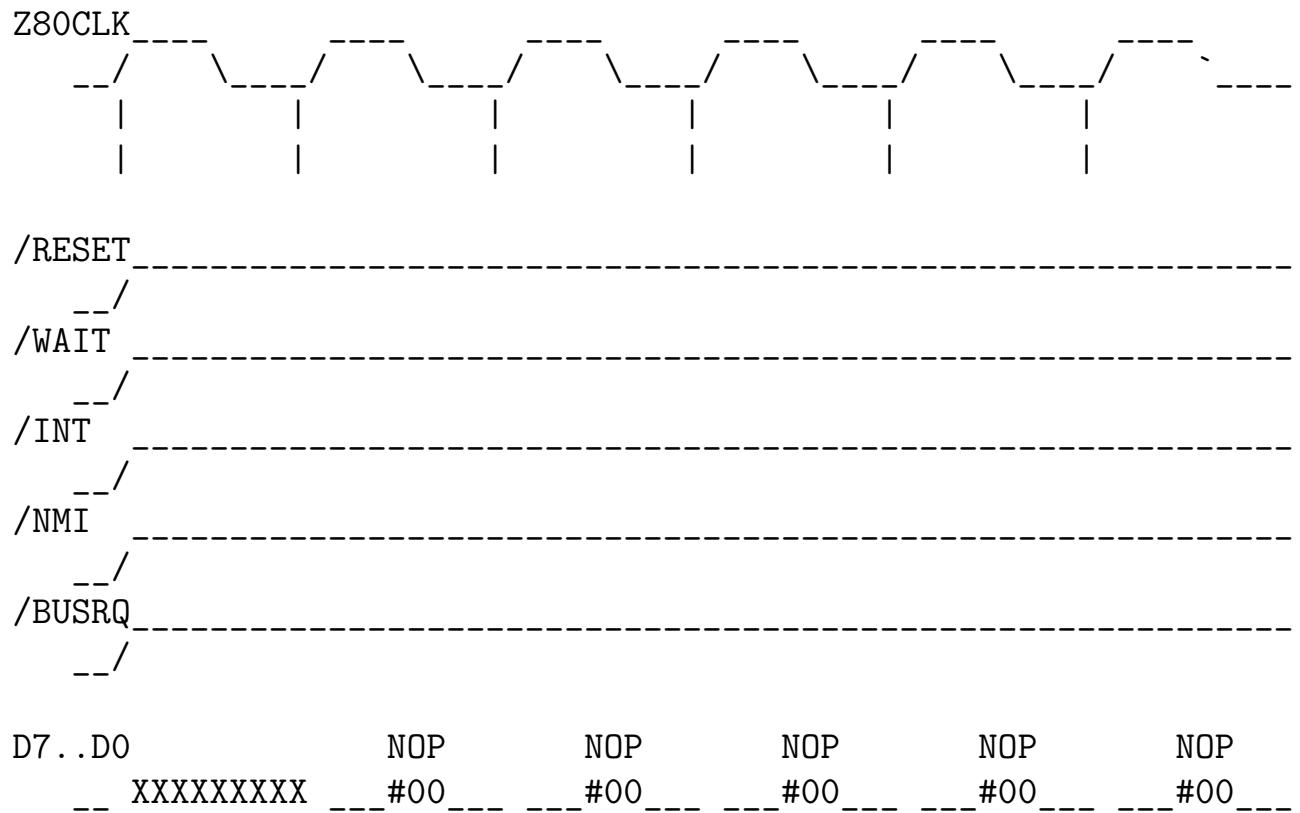
<--	A11	1		40	A10	-->
<--	A12	2		39	A9	-->
<--	A13	3	Z80 CPU	38	A8	-->
<--	A14	4		37	A7	-->
<--	A15	5		36	A6	-->
-->	CLK	6		35	A5	-->
-->	D4	7		34	A4	-->
-->	D3	8		33	A3	-->
-->	D5	9		32	A2	-->
-->	D6	10		31	A1	-->
	VCC	11		30	A0	-->
-->	D2	12		29	GND	
-->	D7	13		28	/RFSH	-->
-->	D0	14		27	/M1	-->
-->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

How to test

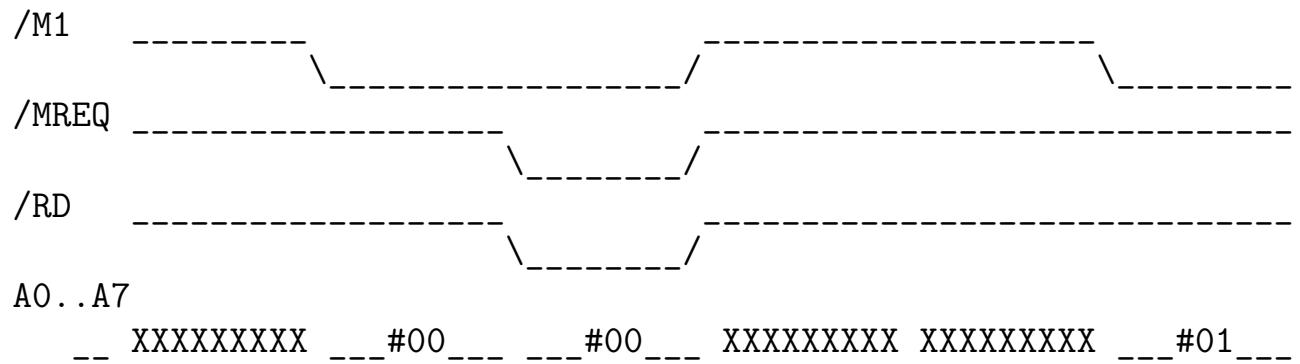
Hold all bidirectional pins (**Data bus**) low to make CPU execute **NOP** instruction. **NOP** instruction opcode is 0. Hold all input pins high to disable interrupts and signal that data bus is ready.

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

Timing diagram, input pins



Expected signals on output pins



External hardware

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

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

Pinout

#	Input	Output	Bidirectional
0	/WAIT	/M1, A0, A8	D0
1	/INT	/MREQ, A1, A9	D1
2	/NMI	/IORQ, A2, A10	D2
3	/BUSRQ	/RD, A3, A11	D3
4		/WR, A4, A12	D4
5		/RFSH, A5, A13	D5
6	MUX – address lo/hi bits on the output pins	/HALT, A6, A14	D6
7	MUX – control signals on the output pins	/BUSA _K , A7, A15	D7

2048 sliding tile puzzle game (VGA) [334]

- Author: Uri Shaked
- Description: Slide numbered tiles on a grid to combine them to create a tile with the number 2048.
- GitHub repository
- HDL project
- Mux address: 334
- Extra docs
- Clock: 25175000 Hz

How it works

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

The game is played on a 4x4 grid, with numbered tiles that slide when a player moves them using `ui_in` pins or using a SNES compatible controller along with the Gamepad Pmod.

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

How to test

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

<code>ui_in</code> pin	Direction
0	Up
1	Down
2	Left
3	Right

Or use a SNES compatible controller along with the Gamepad Pmod. The game will automatically detect the presence of the Pmod and switch to controller input mode.

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

The game offers two color themes: modern and retro. You can switch between the two themes using the select button on the gamepad or by setting both `ui_in[4]` and `ui_in[5]` to 1.

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

External hardware

- TinyVGA Pmod
- Optional: Gamepad Pmod

Pinout

#	Input	Output	Bidirectional
0	<code>btn_up</code>	<code>R1</code>	<code>debug_cmd</code>
1	<code>btn_down</code>	<code>G1</code>	<code>debug_cmd</code>
2	<code>btn_left</code>	<code>B1</code>	<code>debug_cmd</code>
3	<code>btn_right</code>	<code>VSync</code>	<code>debug_cmd</code>
4	<code>gamepad_latch</code>	<code>R0</code>	<code>debug_data</code>
5	<code>gamepad_clk</code>	<code>G0</code>	<code>debug_data</code>
6	<code>gamepad_data</code>	<code>B0</code>	<code>debug_data</code>
7	<code>debug_mode</code>	<code>HSync</code>	<code>debug_data</code>

ChatGPT-generated Spiking Neural Network with Delays [335]

- Author: Paola Vitolo
- Description: ChatGPT-generated Spiking Neural Network with Delays
- GitHub repository
- HDL project
- Mux address: 335
- Extra docs
- Clock: 50000000 Hz

Overview

How it works

This project implements 18 programmable digital LIF neurons with programmable delays and a total of 144 synapses. The neurons are arranged in 3 layers (8 inputs + FC (8 neurons) + FC (8 neurons) + FC (2 neurons) + 2 outputs). Spikes_in directly maps to the inputs of the first layer neurons. When an input spike is received, it is first multiplied by an 2-bit weight, programmable from an SPI interface, 1 per input neuron. This 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 8 neurons in the next layer. There are 144 ($8 \times 8 + 8 \times 8 + 8 \times 2$) programmable weights describing the connectivity between the input spikes and the first layer (64 weights = 8×8), the first and second layers (64 weights = 8×8), and the second and third layers (16 weights = 8×2).

Through a configurable selection signal via SPI, it is possible to read any of the membrane potentials from any neuron in any layer, or the output spikes from any layer.

How to test

After reset, program the neuron threshold, decay rate, and refractory period. Additionally program the first, second, and third layer weights and delays. Once programmed activate spikes_in to represent input data, track spikes_out synchronously.

Memory Map Overview Each parameter (decay, refractory period, membrane potential threshold, weights, and delays) and each configuration signal (value for the configurable clock divider and output select signal) is accessible via SPI in specific byte addresses. The memory is organized as follows:

Parameter	Bit Range / Byte	Address (Hex)	Address (Decimal)	Description
decay	5:0 bits in 2nd byte	0x00	0	Decay control
refractory_period	5:0 bits in 3rd byte	0x01	1	Refractory period
threshold	5:0 bits in 4th byte	0x02	2	Membrane threshold
div_value	5th byte	0x03	3	Division value
weights	36 bytes (5th to 40th)	0x04 - 0x27	4 - 39	Synaptic weights
delays	72 bytes (41st to 112th)	0x28 - 0x6F	40 - 111	Synaptic delays
output_config	8 bits in 113th byte	0x70	112	Output configuration

Simulations

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	input_spike[0]	output[0]	CS
1	input_spike1	output1	MOSI
2	input_spike2	output2	MISO
3	input_spike[3]	output[3]	SCLK
4	input_spike[4]	output[4]	input_ready
5	input_spike[5]	output[5]	output_ready
6	input_spike[6]	output[6]	SNN_en
7	input_spike[7]	output[7]	spi_instruction_done

Space Invaders ASIC [338]

- Author: Lukas Krupp, Adam Gebregziabher
- Description: HDL implementation of the retro game Space Invaders
- GitHub repository
- HDL project
- Mux address: 338
- Extra docs
- Clock: 25000000 Hz

How it works

Implementation of the Space Invaders game in Verilog. Use inputs 0 and 1 to move the player to the right and to the left respectively. To shoot at the aliens, press button 2. It is intended that the game is played using the TinyTapeout Commander app. Via the app the keyboard of the computer can be used to set the input pins of the ASIC.

How to test

Set the inputs and check the outputs. TinyTapeout VGA Playground is the preferred way of testing.

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	input1	out1	
1	input2	out2	
2	input3	out3	
3		out4	
4		out5	
5		out6	
6		out7	
7		out8	

Demo by a1k0n [339]

- Author: Andy Sloane
- Description: Tiny Tapeout demo competition entry
- GitHub repository
- HDL project
- Mux address: 339
- Extra docs
- Clock: 48000000 Hz

a1k0n's tinytapeout08 demo compo entry



How it works It's a standalone VGA+sound demo that fits in two tiles; you'll just have to see.

This was developed with a 48MHz clock, so it's in a funky VGA video mode – it's standard 640x480@60Hz VGA timing and 4:3 aspect ratio, but with 1220 horizontal

pixels instead of 640. All graphics are dithered down to RGB222 with a Bayer matrix which alternates each frame. Because of the dithering and the weird resolution, it looks best on a real CRT, but any VGA monitor ought to work.

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

I will add more info here after the deadline passes, as the demo is in flux as I try to fit effects into 2 tiles...

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

External hardware Follows the democompo hardware rules:

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

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

Pinout

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

Clock Divider [353]

- Author: Armaan Gomes
- Description: A clock divider with lag correction
- GitHub repository
- HDL project
- Mux address: 353
- Extra docs
- Clock: 0 Hz

How it works

Cool stuff makes cool stuff happen Explain how your project works

How to test

Plug cool stuff into the chip and it will output cool stuff Explain how to use your project

External hardware

You need some cool microphones and a cool clock generator and a cool i2s receiver
List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Pinout

#	Input	Output	Bidirectional
0	Bit Clock (3.072 Mhz)	PCM Out Mic 0	Delay Select 0
1	LR Clock (48Khz)	PCM Out Mic 1	Delay Select 1
2	PDM Input Mics 0,1	PCM Out Mic 2	Delay Select 2
3	PDM Input Mics 2,3	PCM Out Mic 3	Delay Select 3
4	PDM Input Mics 4,5	PCM Out Mic 4	Delay Select 4
5	PDM Input Mics 6,7	PCM Out Mic 5	Beamformed PCM Output
6		PCM Out Mic 6	Mic Clock
7		PCM Out Mic 7	

TinyFPGA resubmit for TT08 [355]

- Author: Emilian Miron
- Description: TinyFPGA
- GitHub repository
- HDL project
- Mux address: 355
- Extra docs
- Clock: 0 Hz

How it works

Configure the FPGA then look at outputs.

How to test

Configure the FPGA then look at outputs. See the tests.

External hardware

No special hardware needed.

Pinout

#	Input	Output	Bidirectional
0	input0	output0	n/a
1	input1	output1	n/a
2	input2	output2	n/a
3	input3	output3	n/a
4		output4	n/a
5		output5	n/a
6	cmd0	output6	n/a
7	cmd1	output7	n/a

Dummy Counter [357]

- Author: Chinmay
- Description: A 16-bit counter
- GitHub repository
- HDL project
- Mux address: 357
- Extra docs
- Clock: 0 Hz

How it works

Like a 16-bit counter

How to test

Like a 16-bit counter

External hardware

NA

Pinout

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

RGB Mixer [359]

- Author: Tianmin (Kevin) Kong
- Description: First ASIC Project!
- GitHub repository
- HDL project
- Mux address: 359
- 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

32x8 LED Matrix Animation [361]

- Author: Ayla Lin, Pavit Thakur, Lauren Low
- Description: An animation using a 32x8 matrix, switching between a beaver logo and the letters ‘BWSI’
- GitHub repository
- HDL project
- Mux address: 361
- Extra docs
- Clock: 33000000 Hz

How it works

This project contains 3 components:

- SPI for sending instructions and bitmaps to MAX7219/7221.
- ROM for storing bitmaps to display.
- Controller for instructing SPI and ROM.

How to test

Test using FPGA and a breadboard

External hardware

- 32x8 LED Matrix.
- MAX7219/7221.
- 5V battery.

Pinout

#	Input	Output	Bidirectional
0		spi_clk	
1		spi_cs_n	
2		spi_mosi	
3			
4			
5			

#	Input	Output	Bidirectional
6			
7			

TT09Ball VGA Screensaver [363]

- Author: Rebecca G. Bettencourt; Uri Shaked
- Description: THE STRONGEST DVD style screen saver (640x480, TinyVGA Pmod)
- GitHub repository
- HDL project
- Mux address: 363
- Extra docs
- Clock: 0 Hz

How it works

Displays THE STRONGEST bouncing logo on the screen, with animated color gradient.



How to test

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

- tile (ui_in[0]) to repeat the logo and tile it across the screen,
- solid_color (ui_in1) to use a solid color instead of an animated gradient.

External hardware

TinyVGA PMOD

Pinout

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

Color Bars [365]

- Author: Rebecca G. Bettencourt
- Description: VGA demo resembling NTSC color bars
- GitHub repository
- HDL project
- Mux address: 365
- Extra docs
- Clock: 0 Hz

How it works

Displays a test pattern on the screen resembling NTSC color bars. Optionally, you can add a station ID, make the ID scroll, and make the color bars scroll.

The colors displayed are NOT accurate to actual NTSC color bars. This cannot be used to adjust NTSC video equipment; it's just for fun.



How to test

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

- `show_id` (`ui_in[0]`) to add a station ID,
- `custom_id` (`ui_in[1]`) to use a custom ID (address on `uio_out`, data on `ui_in[7:4]`),
- `scroll_id` (`ui_in[2]`) to make the ID scroll,
- `scroll_bars` (`ui_in[3]`) to make the color bars scroll.

External hardware

TinyVGA PMOD

Pinout

#	Input	Output	Bidirectional
0	<code>show_id</code>	R1	A0 (custom id)
1	<code>custom_id</code>	G1	A1 (custom id)
2	<code>scroll_id</code>	B1	A2 (custom id)
3	<code>scroll_bars</code>	VSync	A3 (custom id)
4	D3 (custom id)	R0	A4 (custom id)
5	D2 (custom id)	G0	A5 (custom id)
6	D1 (custom id)	B0	A6 (custom id)
7	D0 (custom id)	HSync	A7 (custom id)

Hardware UTF Encoder/Decoder [367]

- Author: Rebecca G. Bettencourt
- Description: Converts Unicode code points between UTF-8, UTF-16, and UTF-32.
- GitHub repository
- HDL project
- Mux address: 367
- Extra docs
- Clock: 0 Hz

How it works

This project contains hardware logic to convert between the UTF-8, UTF-16, and UTF-32 encodings for Unicode text.

It will detect and raise an error signal on overlong encodings, out of range code point values, and invalid byte sequences.

(You can optionally disable range checking if you wish to use the original UTF-8 spec that supports values up to 0x7FFFFFFF.)

Basic operation

- In the initial state, all dedicated inputs should be set HIGH.
- At any time, set /RESET (`rst_n`) LOW and pulse CLK to reset all inputs and outputs to initial state.
- At any time, set /ROUT (input 0) LOW and pulse CLK to seek to the beginning of the output.
- You can set ERRS or /PROPS (input 1) HIGH to get an error status on the dedicated outputs.
- You can set ERRS or /PROPS (input 1) LOW to get character properties on the dedicated outputs.
- You can set CHK (input 2) HIGH to raise an error signal when the code point value is out of range (0x110000).
- You can set CHK (input 2) LOW to ignore out of range code point values and encode/decode values up to 0x7FFFFFFF.
- You can set CBE (input 3) HIGH to specify big endian order for UTF-32 and UTF-16 input and output.
- You can set CBE (input 3) LOW to specify little endian order for UTF-32 and UTF-16 input and output.

Inputting UTF-32

1. Set READ or /WRITE (input 4) LOW.
2. Set /CIO (input 5, character I/O) LOW.
3. Set bidirectional I/O to the first byte of the UTF-32 word and pulse CLK.
4. Set bidirectional I/O to the second byte of the UTF-32 word and pulse CLK.
5. Set bidirectional I/O to the third byte of the UTF-32 word and pulse CLK.
6. Set bidirectional I/O to the fourth byte of the UTF-32 word and pulse CLK.
7. Set /CIO (input 5, character I/O) HIGH.
8. Set READ or /WRITE (input 4) HIGH.
9. If READY (output 0) is HIGH and ERROR (output 5) is LOW, the input and output are both valid.
10. If READY (output 0) is LOW or ERROR (output 5) is HIGH, the input was out of range (0x110000 or, if CHK is LOW, 0x80000000).

Inputting UTF-16

1. Set ERRS or /PROPS (input 1) LOW.
2. Set READ or /WRITE (input 4) LOW.
3. Set /UIO (input 6, UTF-16 I/O) LOW.
4. Set bidirectional I/O to the first byte of the first UTF-16 word and pulse CLK.
5. Set bidirectional I/O to the second byte of the first UTF-16 word and pulse CLK.
6. If HIGHCHAR (output 3) is LOW, skip to step 9.
7. Set bidirectional I/O to the first byte of the second UTF-16 word and pulse CLK.
8. Set bidirectional I/O to the second byte of the second UTF-16 word and pulse CLK.
9. Set /UIO (input 6, UTF-16 I/O) HIGH.
10. Set READ or /WRITE (input 4) HIGH.
11. Set ERRS or /PROPS (input 1) HIGH.
12. If READY (output 0) is HIGH and ERROR (output 5) is LOW, the input and output are both valid.
13. If RETRY (output 1) is HIGH, the first word was a high surrogate but the second word was not a low surrogate. The output will be the high surrogate only; the last word will need to be processed again.

Inputting UTF-8

1. Set READ or /WRITE (input 4) LOW.
2. Set /BIO (input 7, byte I/O) LOW.
3. Set bidirectional I/O to the current byte of the UTF-8 sequence and pulse CLK.

4. Repeat step 3 until READY (output 0) or ERROR (output 5) is HIGH.
5. If READY (output 0) is HIGH and ERROR (output 5) is LOW, the input and output are both valid.
6. If RETRY (output 1) is HIGH, the UTF-8 sequence was truncated (not enough continuation bytes). The output will be the truncated sequence only; the last byte will need to be processed again.
7. If INVALID (output 2) is HIGH, the UTF-8 sequence was a single continuation byte or invalid byte (0xFE or 0xFF).
8. If OVERLONG (output 3) is HIGH, the UTF-8 sequence was an overlong encoding.
9. If NONUNI (output 4) is HIGH, the UTF-8 sequence was out of range (0x110000).

Outputting UTF-32

1. Set READ or /WRITE (input 4) HIGH.
2. Set /CIO (input 5, character I/O) LOW.
3. Pulse CLK and read the first byte of the UTF-32 word from the bidirectional I/O.
4. Pulse CLK and read the second byte of the UTF-32 word from the bidirectional I/O.
5. Pulse CLK and read the third byte of the UTF-32 word from the bidirectional I/O.
6. Pulse CLK and read the fourth byte of the UTF-32 word from the bidirectional I/O.
7. Set /CIO (input 5, character I/O) HIGH.
8. If the UTF-32 word is within range, the input and output are both valid.
9. If the UTF-32 word is not within range, then the input was either incomplete or invalid.

Outputting UTF-16

1. Set READ or /WRITE (input 4) HIGH.
2. If UEOF (output 6) is HIGH, then the input was either incomplete or invalid.
3. Set /UIO (input 6, UTF-16 I/O) LOW.
4. Pulse CLK and read the next byte of the UTF-16 sequence from the bidirectional I/O.
5. Repeat step 4 until UEOF (output 6) is HIGH.
6. Set /UIO (input 6, UTF-16 I/O) HIGH.

Outputting UTF-8

1. Set READ or /WRITE (input 4) HIGH.
2. If BEOF (output 7) is HIGH, then the input was either incomplete or invalid.
3. Set /BIO (input 7, byte I/O) LOW.
4. Pulse CLK and read the next byte of the UTF-8 sequence from the bidirectional I/O.
5. Repeat step 4 until BEOF (output 7) is HIGH.
6. Set /BIO (input 7, byte I/O) HIGH.

Error status

When ERRS or /PROPS (input 1) is HIGH, the dedicated outputs will be:

#	Name	Meaning
0	READY	The input and output are complete sequences.
1	RETRY	The previous input was invalid or the start of another sequence and was ignored.
2	INVALID	The input and output are invalid.
3	OVERLONG	The UTF-8 input was an overlong sequence.
4	NONUNI	The code point value is out of range (0x110000). (This is set independently of the other error codes.)
5	ERROR	Equivalent to (RETRY or INVALID or OVERLONG or (NONUNI and CHK))

If all of these outputs are LOW, the accumulated input is incomplete and more input is required (underflow).

Character properties

When ERRS or /PROPS (input 1) is LOW, the dedicated outputs will be:

#	Name	Meaning
0	NORMAL	The code point value is valid and not a C0 or C1 control character, surrogate, or noncharacter.
1	CONTROL	The code point value is valid and a C0 or C1 control character (0x00-0x1F).
2	SURROGATE	The code point value is valid and a UTF-16 surrogate (0xD800-0xDFFF).
3	HIGHCHAR	The code point value is valid and either a high surrogate (0xD800-0xDBFF) or a high surrogated code point (0xDC00-0xDFFF).
4	PRIVATE	The code point value is valid and either a private use character (0xE000-0xF000) or a private surrogated code point (0xED00-0xF400).
5	NONCHAR	The code point value is valid and a noncharacter (0xFDD0-0xFDEF or the code point value is invalid).

If all of these outputs are LOW, there is no valid code point in the output.

How to test

The test.py file covers a comprehensive set of test cases which are listed in a separate file to avoid bloating the TT09 manual.

External hardware

Any device that needs to process Unicode text.

Pinout

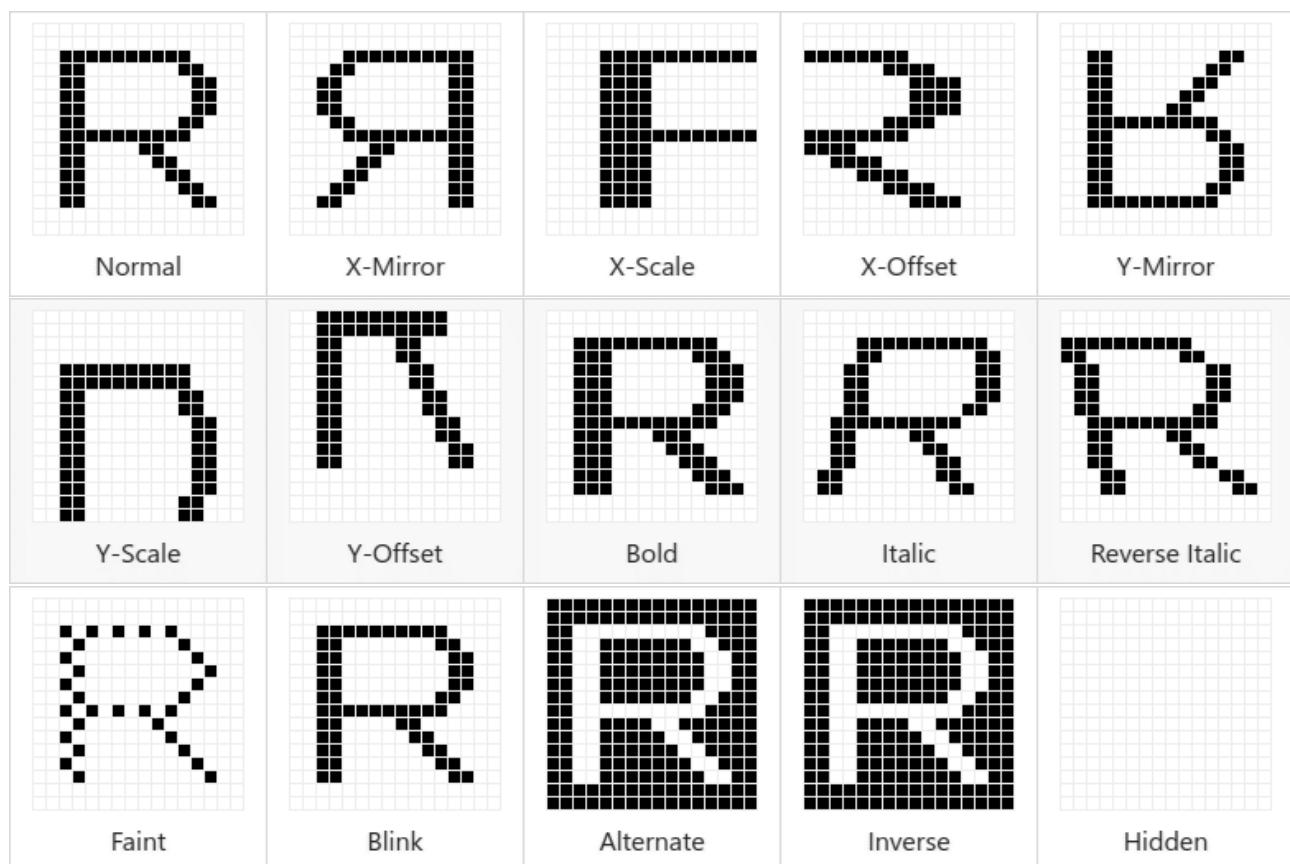
#	Input	Output	Bidirectional
0	/ROUT	READY; NORMAL	I/O LSB
1	ERRS, /PROPS	RETRY; CONTROL	I/O
2	CHK	INVALID; SURROGATE	I/O
3	CBE, /CLE	OVERLONG; HIGHCHAR	I/O
4	READ, /WRITE	NONUNI; PRIVATE	I/O
5	/CIO	ERROR; NONCHAR	I/O
6	/UIO	UEOF	I/O
7	/BIO	BEOF	I/O MSB

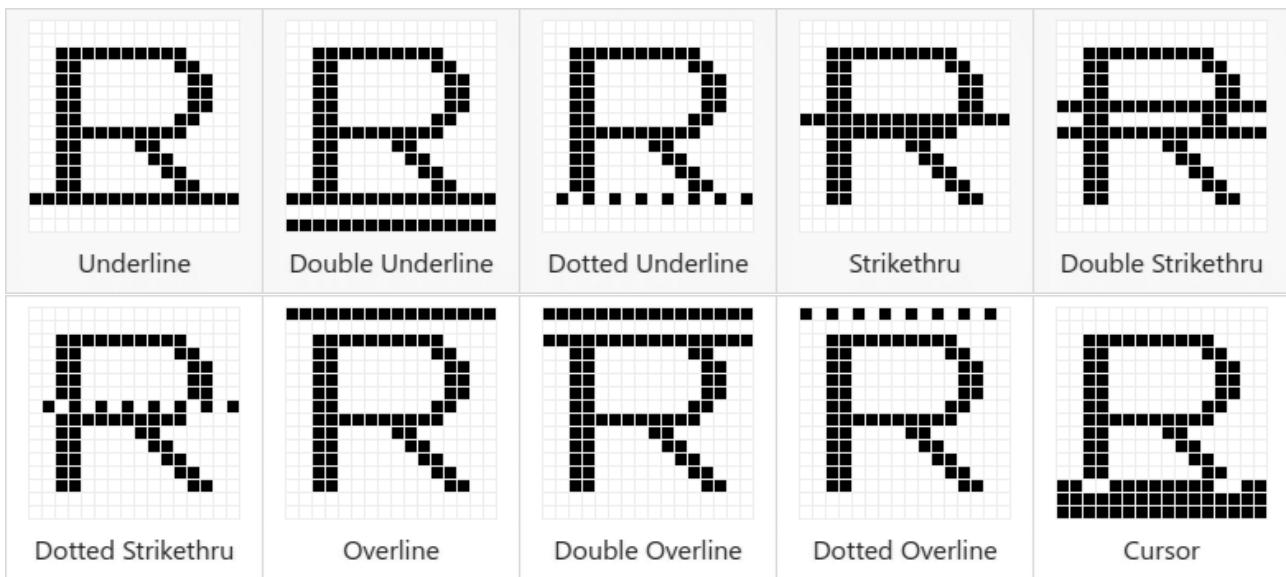
Styler [369]

- Author: Rebecca G. Bettencourt
- Description: 16x16 bitmap manipulation based on text mode attributes.
- GitHub repository
- HDL project
- Mux address: 369
- Extra docs
- Clock: 0 Hz

How it works

The styler chip is used to transform a 16x16 character glyph bitmap based on a set of text mode attributes. It consists of a 4-bit scanline register, an 8-bit control register, a 16-bit bitmap register, and a 25-bit attribute register. Additionally, three independent input lines are used to control polarity of faint text (even or odd pixels), text and cursor blink rate, and cursor position.





Typical use of the styler chip follows these steps:

1. Set output enable (input 6) HIGH and write enable (input 7) LOW.
2. Set the address (inputs 0-2) to 0.
3. Set the bidirectional pins to the physical scanline number.
4. Pulse clk.
5. Set output enable (input 6) LOW and write enable (input 7) HIGH.
6. Read the logical scanline number from the bidirectional pins.
7. Set output enable (input 6) HIGH and write enable (input 7) LOW.
8. Set the address (inputs 0-2) to 2.
9. Set the bidirectional pins to the right half of the row of the character bitmap corresponding to the logical scanline number.
10. Pulse clk.
11. Set the address (inputs 0-2) to 3.
12. Set the bidirectional pins to the left half of the row of the character bitmap corresponding to the logical scanline number.
13. Pulse clk.
14. Set output enable (input 6) LOW and write enable (input 7) HIGH.
15. Set the address (inputs 0-2) to 2.
16. Read the right half of the final character bitmap from the bidirectional pins.
17. Set the address (inputs 0-2) to 3.
18. Read the left half of the final character bitmap from the bidirectional pins.

You can also read from the dedicated output pins without changing output enable or write enable.

The register layout is as follows:

Address	Bits	Description
0	0-3	Input: physical scanline number; output: logical scanline number.
0	4-7	Input: ignored; output: 0.
1	0	Show cursor at bottom of character cell.
1	1	Show cursor at top of character cell.
1	2	Enable cursor blink.
1	3	Enable cursor.
1	4	Enable character underline, strikethrough, overline attributes.
1	5	Enable character blink, alternate attributes.
1	6	Reserved.
1	7	Reserved.
2	0-7	Right half of character glyph bitmap.
3	0-7	Left half of character glyph bitmap.
4	0	X offset. (Determines which half of a double-width character.)
4	1	Double width.
4	2	Y offset. (Determines which half of a double-height character.)
4	3	Double height.
4	4	X premirror (flip input bitmap horizontally).
4	5	X postmirror (flip output bitmap horizontally).
4	6	Y premirror (invert physical scanline).
4	7	Y postmirror (invert logical scanline).
5	0	Bold.
5	1	Faint.
5	2	Italic.
5	3	Reverse italic.
5	4	Blink (text only, VT100-style).
5	5	Alternate (text and background, Apple II-style).
5	6	Inverse.
5	7	Hidden.
6	0	Underline.
6	1	Double underline.
6	2	Dotted underline.
6	3	Strikethrough.
6	4	Double strikethrough.
6	5	Dotted strikethrough.
6	6	Overline.
6	7	Double overline.
7	0	Dotted overline.
7	1-7	Input: ignored; output: 0.

The input pin assignments are as follows:

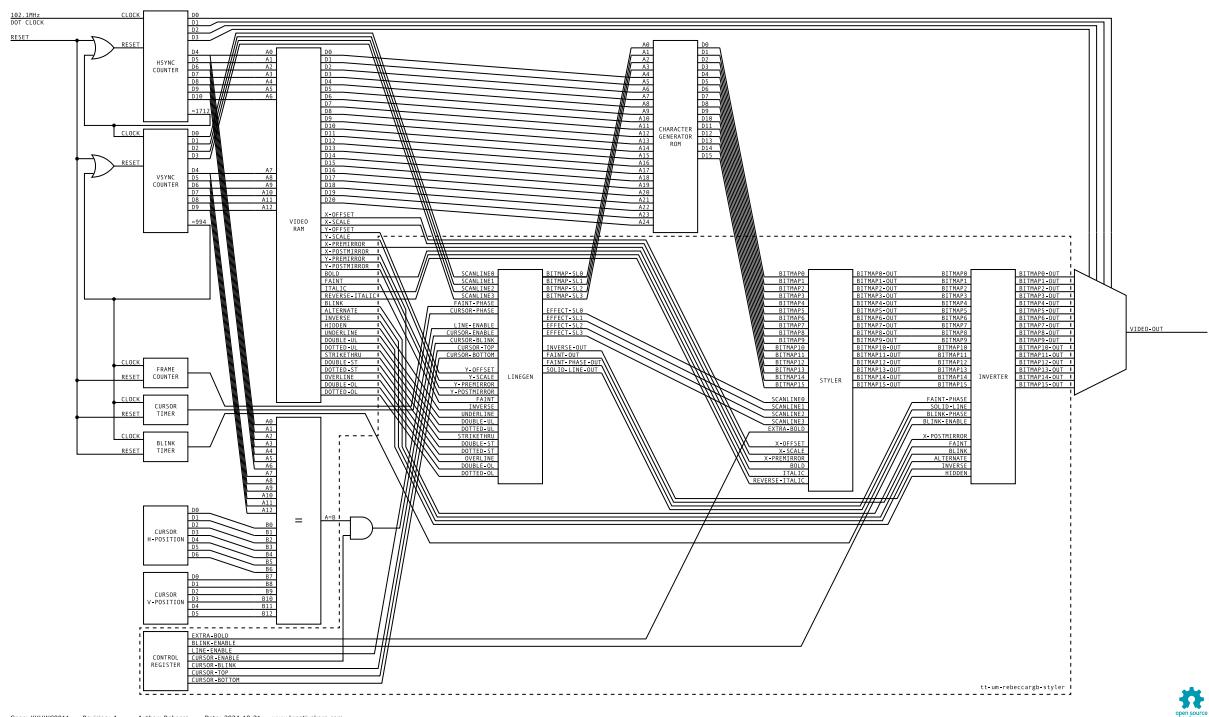
Pin	Description
0	A0 (address line 0).
1	A1 (address line 1).
2	A2 (address line 2).
3	Faint text polarity (even or odd pixels).
4	Blink phase.
5	Cursor enable.
6	/OE (output enable).
7	/WE (write enable).

How to test

The test.py file covers a variety of test cases.

External hardware

The styler chip is intended to be used as part of a larger text mode video display hardware project.



Pinout

#	Input	Output	Bidirectional
0	A0 (address)	D0	D0
1	A1 (address)	D1	D1
2	A2 (address)	D2	D2
3	faint text polarity	D3	D3
4	blink phase	D4	D4
5	cursor enable	D5	D5
6	/OE (output enable)	D6	D6
7	/WE (write enable)	D7	D7

VGA Timing Experiments [371]

- Author: Rebecca G. Bettencourt
- Description: Configurable VGA signal generator for experimentation purposes.
- GitHub repository
- HDL project
- Mux address: 371
- Extra docs
- Clock: 0 Hz

How it works

Generates VGA signals. All signal timings (display area, front porch, back porch, hsync, vsync, polarity) are fully configurable and several test patterns are included to enable experimentation.

How to test

Connect to a VGA monitor. Set `ui_in[3:0]` all LOW and pulse `ui_in[7]` to set signal timings to a “known good” configuration of 640×480 at 60Hz. Observe the vertical color bars. Set either `ui_in[0]` or `ui_in[1]` HIGH and pulse `ui_in[7]` to change the displayed test pattern.

Set `ui_in[3:0]` to a register address, set `{ui_in[6:4], ui_in}` to a register value, and pulse `ui_in[7]` to change individual timing values. (When setting hsync width or vsync height, set `ui_in[6]` HIGH for positive polarity or LOW for negative polarity.)

Address	Description	Default
0	Reset.	
1	Next pattern.	
2	Previous pattern.	
3	Pattern number.	31
4	Horizontal visible width.	640
5	Horizontal front porch (right border).	16
6	Horizontal sync width (polarity on <code>ui_in[6]</code>).	96
7	Horizontal back porch (left border).	48
8	Vertical visible height.	480
9	Vertical front porch (bottom border).	10
10	Vertical sync height (polarity on <code>ui_in[6]</code>).	2

Address	Description	Default
11	Vertical back porch (top border).	33
12	Pattern color.	0
13	Next color.	
14	Previous color.	
15	Reset.	

Pattern	Description
0	Solid color.
1	1×1 pixel checkerboard.
2	2×2 pixel checkerboard.
3	4×4 pixel checkerboard.
4	8×8 pixel checkerboard.
5	16×16 pixel checkerboard.
6	32×32 pixel checkerboard.
7	64×64 pixel checkerboard.
8	8×8 pixel grid.
9	16×16 pixel grid.
10	32×32 pixel grid.
11	64×64 pixel grid.
12	1×1 pixel color table.
13	2×2 pixel color table.
14	4×4 pixel color table.
15	8×8 pixel color table.
16	16×16 pixel color table.
17	32×32 pixel color table.
18	1×1 pixel color antidiagonal lines.
19	2×2 pixel color antidiagonal lines.
20	4×4 pixel color antidiagonal lines.
21	8×8 pixel color antidiagonal lines.
22	16×16 pixel color antidiagonal lines.
23	32×32 pixel color antidiagonal lines.
24	1×1 pixel color diagonal lines.
25	2×2 pixel color diagonal lines.
26	4×4 pixel color diagonal lines.
27	8×8 pixel color diagonal lines.
28	16×16 pixel color diagonal lines.
29	32×32 pixel color diagonal lines.
30	Horizontal color bars.
31	Vertical color bars.

External hardware

TinyVGA PMOD

Pinout

#	Input	Output	Bidirectional
0	A0	R1	D0
1	A1	G1	D1
2	A2	B1	D2
3	A3	VSync	D3
4	D8	R0	D4
5	D9	G0	D5
6	D10	B0	D6
7	WE	HSync	D7

JTAG TAP [385]

- Author: Sean Patrick O'Brien
- Description: JTAG TAP with Boundary Scan
- GitHub repository
- HDL project
- Mux address: 385
- Extra docs
- Clock: 0 Hz

How it works

A simple “inner project” drives a seven-segment display from either an internal 4-bit counter or from a 4-bit value presented on `ui[4:1]`.

The “outer project” adds a boundary scan register and JTAG TAP that supports the following instructions:

- IDCODE
- SAMPLE/PRELOAD
- EXTEST
- INTEST
- CLAMP
- BYPASS

How to test

At startup, the project will drive the seven-segment display from either the internal 4-bit counter (if `ui[0]` is low) or from `ui[4:1]` (if `ui[0]` is high).

A BSDL file is provided for testing the TAP and boundary scan register. A tool like UrJTAG can be used to control the output pins (via the EXTEST instruction) or to test the inner project (via the INTEST instruction).

External hardware

JTAG adapter connected to `uio[7:4]`

Pinout

#	Input	Output	Bidirectional
0	output_mode	seven_segment	
1	output_value[0]	seven_segment	
2	output_value1	seven_segment	
3	output_value2	seven_segment	
4	output_value[3]	seven_segment	TCK
5		seven_segment	TMS
6		seven_segment	TDI
7		seven_segment	TDO

7-segment with LFSR [387]

- Author: Jun-ichi OKAMURA
- Description: TEST 7-segment/LFSR
- GitHub repository
- HDL project
- Mux address: 387
- Extra docs
- Clock: 50000000 Hz

How it works

This design uses a set of registers to divide the clock and combinational logic to convert binary values into decimal for display.

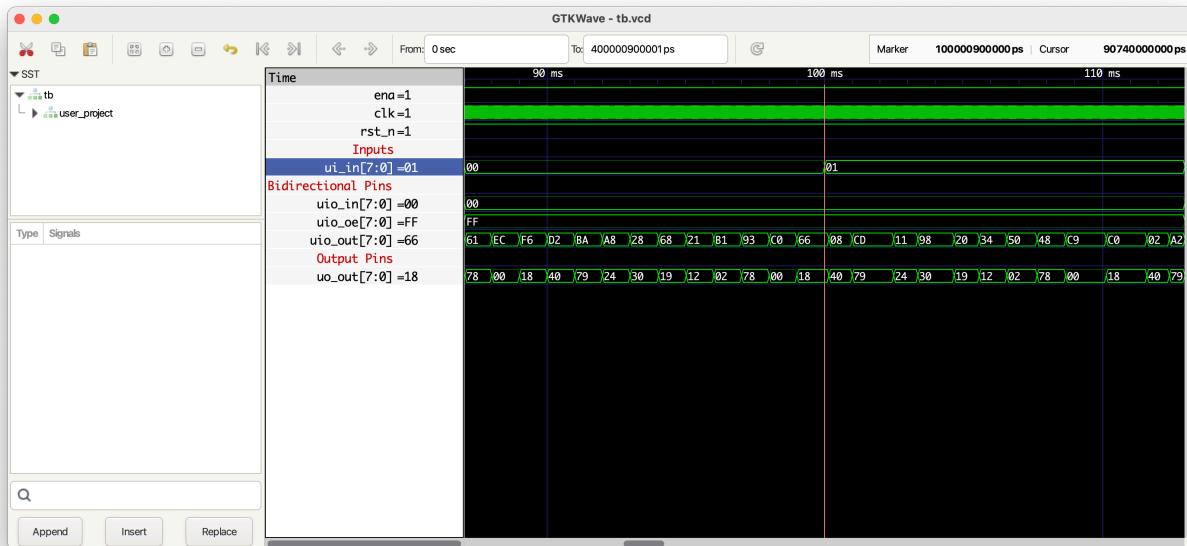
- Inputs[0]: Selects between a fixed period or a pseudo-random period for counting up the seven-segment display.
- Inputs1: Chooses between displaying either a hexadecimal sequence ("0" to "F") or the text "-OPENSUSI-AISol-" on the display.
- Bidirectional Outputs: The bottom 8 bits of a 24-bit counter are placed on the bidirectional outputs.

Fixed Period Mode:

- The internal 16-bit compare register is set to 10,000.
- This results in the display incrementing by one each second in case the CLK input is 10KHz.

Pseudo-Random Period Mode:

- If Input[0] is set to 1, an 8-bit pseudo-random value is used as bits 6 to 12 of the 16-bit compare register, introducing variation in the counting period.



How to test

After reset, the counter will increment by one every second, assuming a 10MHz input clock.

You can experiment by modifying Inputs[1:0] to:

- Change the display characters
- Adjust the pseudo-random sequence of periodic speed

External hardware

Only TT-EVB.

Note

This is the first test project designed by “jun1okamura”, supported by OpenSUSI (non-profit) and AIST Solutions inc. in Japan.

#	Input	Output	Bidirectional
---	-------	--------	---------------

Pinout

#	Input	Output	Bidirectional
0	sel0	segment a	counter bit 0
1	sel1	segment b	counter bit 1
2	compare bit 13	segment c	counter bit 2
3	compare bit 14	segment d	counter bit 3
4	compare bit 15	segment e	counter bit 4
5	compare bit 16	segment f	counter bit 5
6	compare bit 17	segment g	counter bit 6
7	compare bit 18	dot	counter bit 7

TT10 HPDL 1414 Uart [389]

- Author: Andrew Tudoroi
- Description: Uart interface for 4xHPDL 1414 PMOD Module
- GitHub repository
- HDL project
- Mux address: 389
- Extra docs
- Clock: 12000000 Hz

How it works

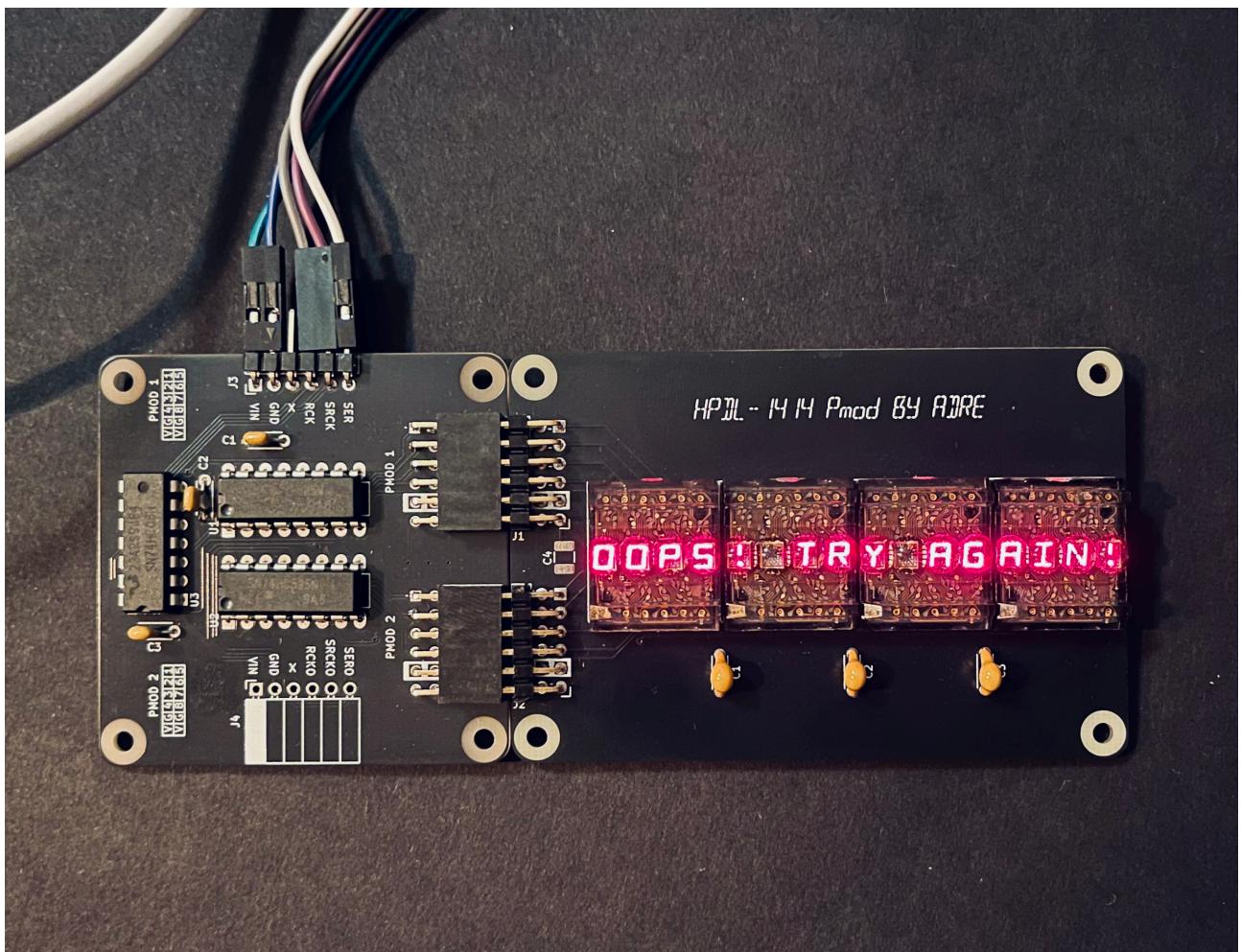
This project is a compact UART transceiver with an integrated display update mechanism. It operates at 115200 baud and stores received data in a 16-byte internal buffer. The data is asynchronously transferred to four HPDL-1414 alphanumeric LED modules. When new characters arrive and the buffer is full, the existing characters shift left to make space. A blinking cursor indicates the current input position, and backspace (Ctrl-H) is supported for navigating back and editing.

How to test

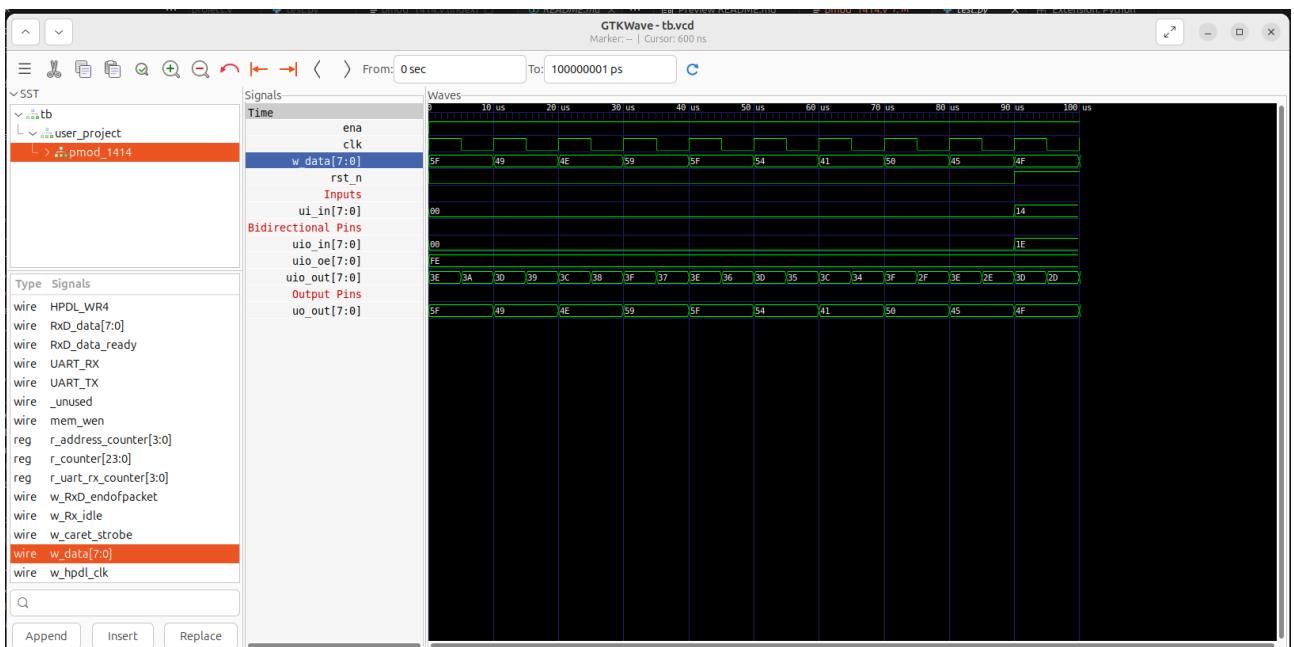
1. Connect the UART interface to a computer or microcontroller configured at 115200 baud.
2. Send ASCII characters over the UART interface.
3. Observe the received characters displayed on the HPDL-1414 modules.
4. Test the character shifting behavior by exceeding 16 characters.
5. Use Ctrl-H to test the backspace functionality.

External hardware

HPDL-1414 Pmod module <https://github.com/ADDTDR/HPDL-1414-Pmod-Module>



Signal test



Pinout

#	Input	Output	Bidirectional
0		HPDL_D0	HPDL_A0
1		HPDL_D1	HPDL_A1
2		HPDL_D2	HPDL_WR1
3		HPDL_D3	HPDL_WR2
4		HPDL_D4	HPDL_WR3
5		HPDL_D5	HPDL_WR4
6		HPDL_D6	UART_TX
7			UART_RX

KCH CD101 Saw Synth [391]

- Author: Johannes Pfau
- Description: Fork of the KCH CD101 Synth Generating Saw Waves
- GitHub repository
- HDL project
- Mux address: 391
- Extra docs
- Clock: 25000000 Hz

How it works

This project provides a simple digital synth. It consists of a pulse-wave generator with programmable frequency, ADSR amplitude modulation with adjustable ADSR parameters and a simple one-pole IIR filter with programmable cutoff frequency. Digital audio output is generated using a simple first-order delta-sigma modulator. The lowpass filter for the reconstruction of this signal must be realized externally.

All parameters can be programmed using a simple SPI slave interface. Sound generation for a note starts when asserting the trigger signal and stops when the trigger is deasserted again. Triggering can happen both via a dedicated input or via SPI, which enables fully customizable operation using the SPI port. A VST/CLAP plugin is provided to generate the SPI commands from DAWs.

This project is educational and therefore makes some decisions that might not lead to an optimal design. For example, the structural presentation of the signal processing pipeline is directly realized in hardware. Some parts of the design are therefore clocked with a frequency as low as the sample rate and some even lower. This wastes a lot of possible performance, but it is easier for students to map the audio application to the circuit mentally, compared to introducing a more complex microcontroller-based system (which might be a more efficient design). The design shows how to realize a serial-parallel multiplier, use negative edge clocking, use simple small clock dividers, use multiple clocks etc.

More detailed information on all these topics will be provided later on.

How to test

As the design generates a lot of data on the single serial output pin, testing generates a lot of data. The cocotb testbench simulates and external lowpass and stores the audio data to a .s16 file which you can convert to .wav using ffmpeg:

Play the output file to assess whether the output is reasonable.

In addition, the testbench also compares to a golden reference output datastream, that was generated from behavioral simulation. This test is used to determine if there are any differences for the final implemented designs.

External hardware

- Button PMOD
- Audio PMOD
- SPI Master (No PMOD available. Use Adafruit board)

TODO: More detailed information about these things.

Pinout

#	Input	Output	Bidirectional
0	Trigger	Audio Data	
1	SPI CLK		
2	SPI MOSI		
3	SPI nSS		
4			
5			
6			
7			

tt10_zhouzhouthezhou_adder [393]

- Author: Kyle Zhou
- Description: Adds 2 4bit numbers and displays the output on the 8 segment
- GitHub repository
- HDL project
- Mux address: 393
- Extra docs
- Clock: 16000 Hz

How it works

Adds 2 4bit numbers together Verilog then goes into an always block where it maps the sum to a number on the 8 segment If the sum is a number the 8 segment cannot display, it will show a . Overflow is not handled

How to test

Use the dip switches to set the input, will add the upper 4 bits to the lower 4 bits

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	ui_0	uo_1	
1	ui_1	uo_2	
2	ui_2	uo_3	
3	ui_3	uo_4	
4	ui_4	uo_5	
5	ui_5	uo_6	
6	ui_6	uo_7	
7	ui_7	uo_8	

Asynchronous Locking Unit [395]

- Author: Tórur Biskopstø Strøm
- Description: An asynchronous lock with 8 request signals and 8 grant signals
- GitHub repository
- HDL project
- Mux address: 395
- Extra docs
- Clock: 0 Hz

How it works

Each input corresponds to a separate request for the lock. The lock is given to the single output signal that goes high.

Pinout

#	Input	Output	Bidirectional
0	req0	ack0	
1	req1	ack1	
2	req2	ack2	
3	req3	ack3	
4	req4	ack4	
5	req5	ack5	
6	req6	ack6	
7	req7	ack7	

XOR Cipher [397]

- Author: Damian
- Description: Simple XOR Cipher with UART
- GitHub repository
- HDL project
- Mux address: 397
- Extra docs
- Clock: 50000000 Hz

How it works

You send a char via a uart port (8bit data, no parity bit) and it sends an “encrypted” char back

To update the key : briefly activate a 1 signal on updateKey port, the circuit will now wait for the next input and set it as the new key once received

the activation signal for updateKey should be held down before sending the new key otherwise the circuit will stay in the updateKey state

The default key is b10101010.

How to test

A python file containing a code to communicate with the serial port may be transformed to work with the TT board.

External hardware

It requires an input clock of 50Mhz

It has two inputs : ui[0]: “updateKey” ui[7]: “rx”

and one output uo[0]: “tx”

#	Input	Output	Bidirectional
---	-------	--------	---------------

Pinout

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

Verilog based clock to 7-segment counter [399]

- Author: EnJens
- Description: Verilog based clock to 7-segment counter
- GitHub repository
- HDL project
- Mux address: 399
- Extra docs
- Clock: 1 Hz

How it works

I have no idea yet! But something something counting and showing it on 7-segment.

How to test

Enable it, get a clock and watch it work.

External hardware

7-segment display from demo board

Pinout

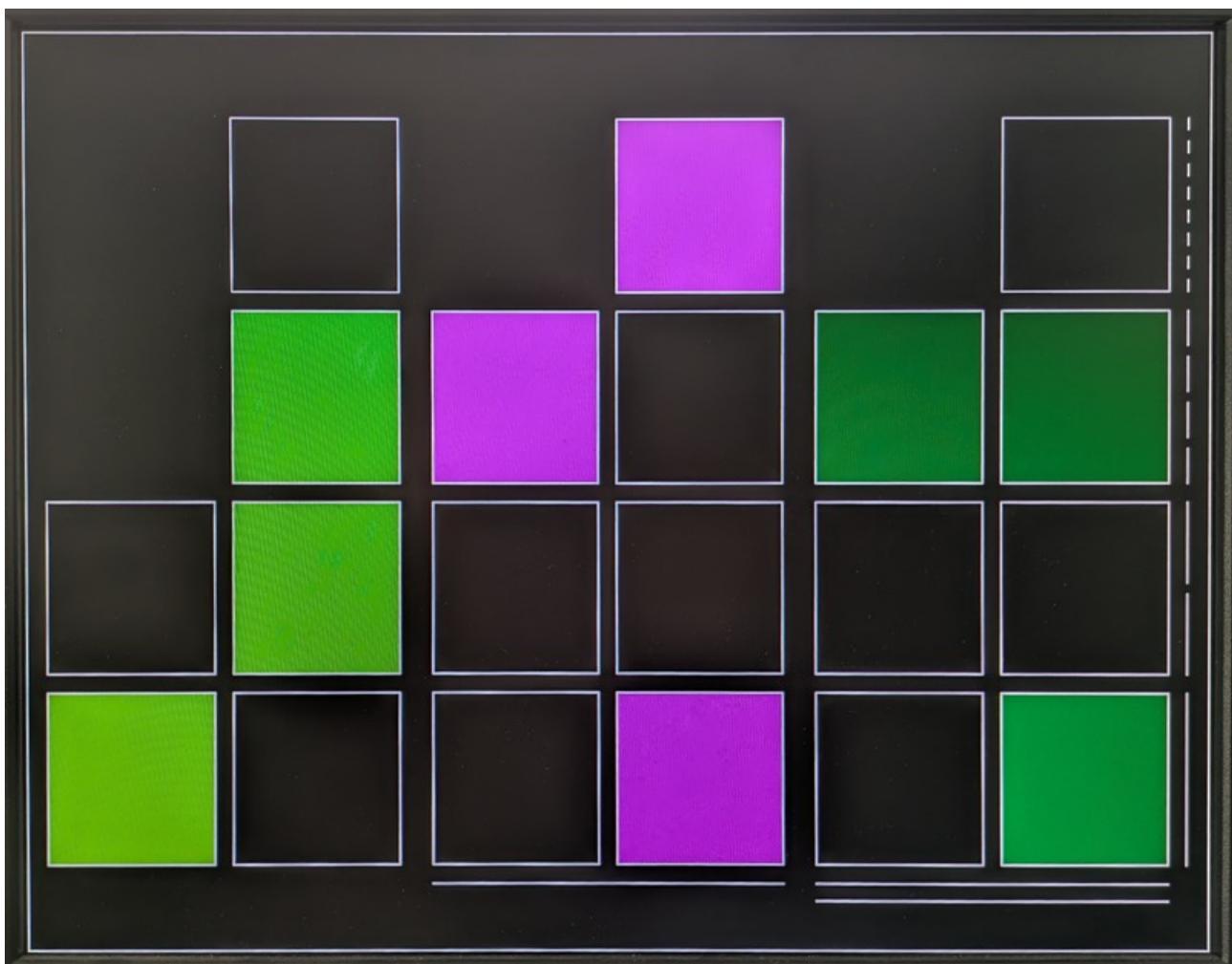
#	Input	Output	Bidirectional
0	IN0	OUT0	
1	IN1	OUT1	
2	IN2	OUT2	
3	IN3	OUT3	
4	IN4	OUT4	
5	IN4	OUT5	
6	IN5	OUT6	
7	IN6	OUT7	

TT10_Luke_Clock [401]

- Author: Luca Pezzarossa
- Description: A VGA based binary clock
- GitHub repository
- HDL project
- Mux address: 401
- Extra docs
- Clock: 25175000 Hz

How it works

The **TT10 Luke Clock** project implements a **VGA-based binary clock** using Chisel. The clock displays hours, minutes, and seconds in a binary format using a matrix of squares, which updates every second. Each column shows the binary value of a digit of the time in the format HH:MM:SS.



Facts

- The clock supports multiple time-clock sources (internal and external) selectable via input settings and allows user interaction through input buttons (see description in the user input below).
- The display colors change randomly every day (randomized with LFSR) or with input by the user. The display layout can also be changed by the user.
- The VGA output generates a 640x480 image at 60 fps.
- All user inputs (buttons and switches) are debounced internally.

Module inputs (ui_in)

- **ui_in(0): Select time clock source [bit 0]**
 - 00: Use internal clock (if set at 25.175 MHz)
 - 01: Use internal clock (if set at 25 MHz)
 - 10: Use external 32.768 kHz clock from ui_in(2)
 - 11: Use external 1 Hz clock from ui_in(3)
- **ui_in(2): Time clock 1Hz**
 - External 1 Hz time clock input
- **ui_in(3): Time clock 32768Hz**
 - External 32.768 kHz time clock input
- **ui_in(4): Plus** – Used to set the clock or change layout (depending on select mode inputs)
- **ui_in(5): Minus (button)** – Used to set the clock or change layout (depending on select mode inputs)
- **ui_in(6): Select mode [bit 0]**
- **ui_in(7): Select mode [bit 1]**
 - 00: Adjust seconds (Plus: clear seconds, Minus: clear seconds)
 - 01: Adjust minutes (Plus: increase minutes, Minus: decrease minutes)
 - 10: Adjust hours plus: (Plus: increase hours, Minus: decrease hours)
 - 11: Switch layout/color (Plus: changes layout, Minus: changes colors)

Module outputs (uo_out) The outputs are used to connect the VGA monitor. They are compatible with the **TT VGA PMOD** interface.

- **uo_out(0): Red [bit 1]**
- **uo_out(1): Green [bit 1]**

- `uo_out(2)`: **Blue [bit 1]**
- `uo_out(3)`: **Vertical sync**
- `uo_out(4)`: **Red [bit 0]**
- `uo_out(5)`: **Green [bit 0]**
- `uo_out(6)`: **Blue [bit 0]**
- `uo_out(7)`: **Horizontal sync**

Module bidirectionals (`ui0`) The following bidirectionals (all used as output) provide debugging information for internal signals.

- `ui0_out(0)`: **(Debug output) tClk**
- `ui0_out(1)`: **(Debug output) cntReg [bit 0]**
- `ui0_out(2)`: **(Debug output) cntReg [bit 1]**
- `ui0_out(3)`: **(Debug output) cntReg [bit 2]**
- `ui0_out(4)`: **(Debug output) cntReg [bit 3]**
- `ui0_out(5)`: **(Debug output) inDisplayArea**
- `ui0_out(6)`: **(Debug output) modeReg [bit 0]**
- `ui0_out(7)`: **(Debug output) modeReg [bit 1]**

How to test

1. **Connect VGA:** Connect the **TT VGA PMOD** to the board and to the VGA monitor.
2. **Connect buttons and switches:** Ensure all input buttons are wired correctly to control clock settings and layout changes.
3. **Select time source:** Use the input settings to choose between the internal clock, an external 32.768 kHz clock, or an external 1 Hz clock.
4. **Set time and layout:** Use the **Plus** and **Minus** buttons and the **Select mode** switches to adjust hours, minutes, and seconds, or switch between different display layouts and colors.
5. **Observe VGA output:** The binary clock should now display the current time, with automatic updates and color changes occurring daily.
6. **Test debugging outputs:** If required, observe **`ui0_out`** signals to verify internal timing and display area status.
7. **Enjoy the clock:** Watch the binary time representation update in real-time on the VGA display.

External hardware

- TT VGA PMOD modules

- Buttons and switches
- (*Optional*) Real-Time Clock (RTC)
 - For precision timekeeping, an RTC generating 1 Hz or 32.768 kHz can be connected as an external time-clock input.

Pinout

#	Input	Output	Bidirectional
0	select time clock source [bit 0] (switch)	red [bit 1]	(debug output) tClk
1	select time clock source [bit 1] (switch)	green [bit 1]	(debug output) cntReg [bit 0]
2	time clock 1Hz	blue [bit 1]	(debug output) cntReg [bit 1]
3	time clock 32768Hz	vsync	(debug output) cntReg [bit 2]
4	plus (button)	red [bit 0]	(debug output) cntReg [bit 3]
5	minus (button)	green [bit 0]	(debug output) inDisplayArea
6	select mode [bit 0] (switch)	blue [bit 0]	(debug output) modeReg [bit 0]
7	select mode [bit 1] (switch)	hsync	(debug output) modeReg [bit 1]

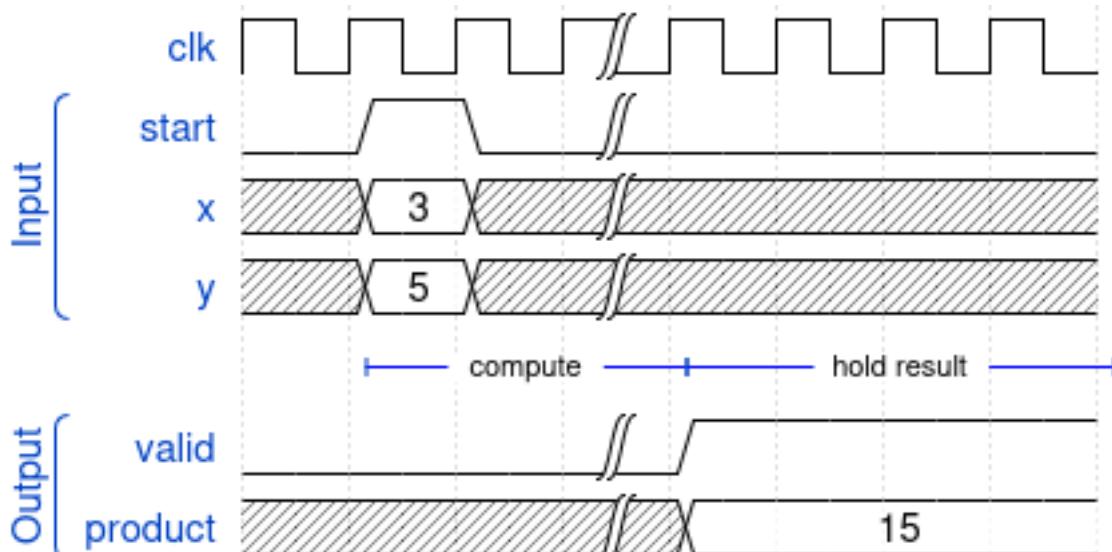
SSMCI [403]

- Author: Oliver Keszocze
- Description: Slow 3-bit unsigned multiplier
- GitHub repository
- HDL project
- Mux address: 403
- Extra docs
- Clock: 0 Hz

How it works

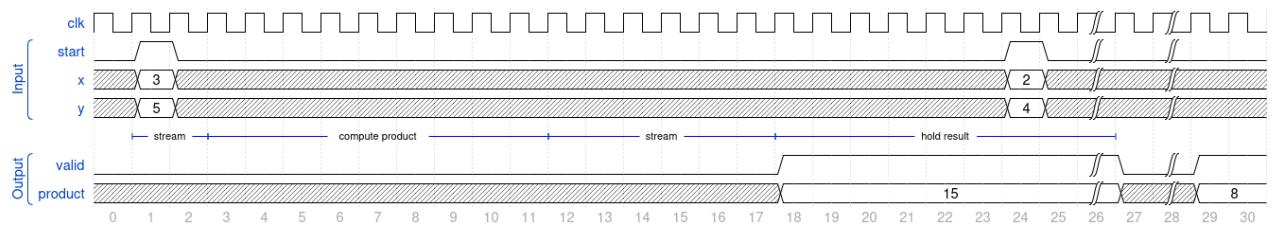
This design contains a three-bit multiplier that aims to be area/resource efficient at the expense of using multiple clock cycles to compute the product.

When start is asserted, the values at the x and y inputs are then being streamed into the actual multiplier, taking 3 cycles in total. The multiplier computes the product in 9 cycles and then streams back the 6-bit product in 6 cycles. The result can be seen at the output bit in cycle 18. This is illustrated by the simplified wave trace below.



Simplified wavetrace of the 3-bit multiplier

The result will be held until the next multiplication starts streaming back its result (the streaming aspect is fully hidden from the end user in this design!). The following wave trace shows this in full detail (there might be an off-by-one error there; I tend to make those. The general mode of operation is correctly captured by the trace!).



How to test

The input to the multiplier can conveniently be controlled using the web interface of the motherboard. Alternatively, you can connect the input/output pmods as shown in the table in the pinout section below.

Pinout

#	Input	Output	Bidirectional
0	$y[0]$	$product[0]$	
1	y_1	$product_1$	
2	y_2	$product_2$	
3	$x[0]$	$product[3]$	
4	x_1	$product[4]$	
5	x_2	$product[5]$	
6			
7	start	valid	

Configurable Logic Block [416]

- Author: Gary Mejia
- Description: A small CLB with a LUT3
- GitHub repository
- HDL project
- Mux address: 416
- Extra docs
- Clock: 0 Hz

How it works

The chip takes in two 8-bit inputs `uin_in`, this is the three arguments to the boolean function, write enable of the LUT, and clock enable of the CLB, and `ui0_in` is the actual boolean function. The single output is the evaluation of the boolean function given the argument.

How to test

A simple hardware test would be to set the `ui0_in` to 01111111 to get a NAND3. Use `uin_in[3]` to program the LUT with the seed and use `uin_in[4]` to make the output synchronous. Use `uin_in[2:0]` to input values into the NAND3.

External hardware

Switches on all inputs and leds on all outputs.

Common Boolean Functions and Seeds

Function	Seed
NAND3	01111111
NOR3	00000001
NOT	01010101
XOR2	01100110
Majority	11101000
Even Parity	01101001
One Hot	00010110

Pinout

#	Input	Output	Bidirectional
0	ui_in[0]	uo_out[0]	ui_out[0]
1	ui_in1	uo_out1	ui_out1
2	ui_in2	uo_out2	ui_out2
3	ui_in[3]	uo_out[3]	ui_out[3]
4	ui_in[4]	uo_out[4]	ui_out[4]
5	ui_in[5]	uo_out[5]	ui_out[5]
6	ui_in[6]	uo_out[6]	ui_out[6]
7	ui_in[7]	uo_out[7]	ui_out[7]

Gamepad Pmod Demo [417]

- Author: Uri Shaked
- Description: Gamepad Pmod + Tiny VGA demo from VGA Playground
- GitHub repository
- HDL project
- Mux address: 417
- Extra docs
- Clock: 25175000 Hz

How it works

This project demonstrates how to use the Gamepad Pmod to get input from a gamepad and display it on a VGA monitor.

How to test

Connect the TinyVGA and Gamepad Pmods to the Tiny Tapeout board, activate the project, reset it, and start pressing buttons on the gamepad.

When you press a button on the gamepad, its corresponding symbol will appear in green on the VGA display.

External hardware

- TinyVGA Pmod
- Gamepad Pmod

Pinout

#	Input	Output	Bidirectional
0		R1	
1		G1	
2		B1	
3		VSync	
4	gamepad_latch	R0	
5	gamepad_clk	G0	
6	gamepad_data	B0	

4-bit up/down binary counter [418]

- Author: claudiotalarico
- Description: 4-bit up/down binary counter with enable and test mode
- GitHub repository
- HDL project
- Mux address: 418
- Extra docs
- Clock: 50000000 Hz

How it works

4 bit up/down binary counter with enable

Pin Mapping

direction	pin name	function
in	clk	clk
in	rst_n	rst_n
in	ui_in[0]	test (test mode)
in	ui_in[1]	ud (up/down)
in	ui_in[2]	en (enable)
out	ui_out[3:0]	cnt[3:0] (count)

How to test

Connect input pin EN to VDD

Connect input pin TEST to GND

Connect input pin UD to VDD or GND through a switch

Connect input pin RST_N to an R-C startup circuit

Connect input pin CLK to a 50 MHz square waveform

Connect the output pins CNT[3:0] to 4 LEDs

External hardware

switch 4 LEDs R-C startup circuit

Pinout

#	Input	Output	Bidirectional
0	test	cnt[0]	
1	ud	cnt1	
2	en	cnt2	
3		cnt[3]	
4			
5			
6			
7			

6Digit7SegClock [419]

- Author: Patrick Lampl
- Description: A digital clock multiplexed to a 6 digit 7 segment display
- GitHub repository
- HDL project
- Mux address: 419
- Extra docs
- Clock: 32768 Hz

How it works

This project is a six digit clock displaying time in a hh:mm:ss format. Two active high pushbuttons are available to increment both the hours and minutes for setting the time. The dot between the hour and minute numbers as well as between the minute and second numbers are blinking in the interval of a second. The outputs are active low control signals for a common anode seven segment display. The signals are multiplexed for all six digits and PMOS or PNP transistors are intended to enable the six digits/anodes.

How to test

Supply a clock of 32768 Hz clock to the circuit and connect two push buttons to input pins 0 and 1, connect a 7-segment display to the eight output ports. The 8-bits are coded from MSB to LSB: dot, segments a, b, c, d, e, f and g. Connect the bidirectional ports (all configured as outputs) to the digit enable transistors. The coding for the 6-bits is as follows: enable hour_tens, hour_ones, minute_tens, minute_ones, second_tens, second_ones.

External hardware

Two active high push buttons with pull down resistors, a six digit seven segment display, six PMOS or PNP transistors to enable the digits. Mounted on a breadboard or a custom PCB.

Pinout

#	Input	Output	Bidirectional
0	minute increment	segment g	digit ena second ones
1	hour increment	segment f	digit ena second tens
2		segment e	digit ena minute ones
3		segment d	digit ena minute tens
4		segment c	digit ena hour ones
5		segment b	digit ena hour tens
6		segment a	
7		segment dot	

Team 17's 8 bit DAC [420]

- Author: Vance Wiberg
- Description: This 8 bit digital to analogue converter uses a SAR to convert signals from Digital into Analogue
- GitHub repository
- HDL project
- Mux address: 420
- Extra docs
- Clock: 0 Hz

How it works

Uses nonlinear sampling to convert a input coming from a comparator to a digital signal

How to test

In put comparator values, check for desired digital outputs

External hardware

Analog comparator and resistor array

Pinout

#	Input	Output	Bidirectional
0	A[0]	Z[0]	O[0]
1	A1	Z1	
2	A2	Z2	
3	A[3]	Z[3]	
4	A[4]	Z[4]	
5	A[5]	Z[5]	
6	A[6]	Z[6]	
7	A[7]	Z[7]	

MAC Operation [421]

- Author: Sachin & Sandeep
- Description: Multiplication & accumulation
- GitHub repository
- HDL project
- Mux address: 421
- Extra docs
- Clock: 100000 Hz

How it works

It multiply and accumulates the data

How to test

Using Memory Operation

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	ui[0]	uo[0]	
1	ui1	uo1	
2	ui2	uo2	
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]	

Tiny Registers [422]

- Author: Roni Kant, Jeremy Kam
 - Description: Various Registers for 8-bit CPU
 - GitHub repository
 - HDL project
 - Mux address: 422
 - Extra docs
 - Clock: 50000000 Hz

How it works

The various registers used for a basic 8-bit CPU design. Consists of a simple general purpose register, a memory address register, and an instruction register. The 3 registers are selected using the 6th and 7th uio pins. | uio[7] | uio[6] | Selected Register | —————|————|————| | 0 | 0 | General Purpose Register | | 0 | 1 | Memory Address Register | | 1 | 0 | Instruction Register |

Design Specifications

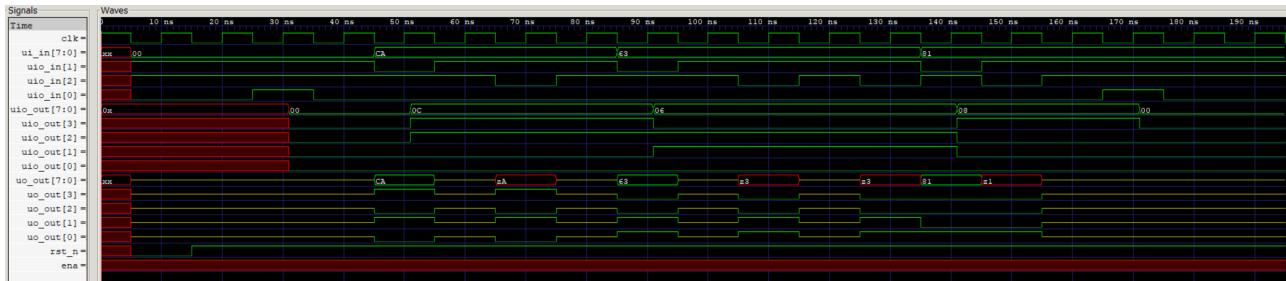
Instruction Register

Label	Input/Output	Description
CLK [1 bit]	Input	Clock signal. Executes actions on rising edges.
W bus [8 bit]	Input	Takes 8 bits with the most significant 4 bits rep
\LI [1 bit]	Input	Control signal that decides whether to read from
\EI [1 bit]	Input	Control signal that decides tri-state buffer output
CLR [1 bit]	Input	Clears the instruction register's data.
Instruction register[3:0] [4 bit]	Output	Output to W bus
Instruction register[7:4] [4 bit]	Output	Output to controller/sequences

Pinouts when instruction register is selected

Test Input Name	Description
clk	CLK
ui_in[7:0]	W bus
uios_in [4]	\LI
uios_in [5]	\EI
rst_n	CLR

Test Input Name	Description
ui_out[3:0]	Instruction register[7:4]
uo_out[3:0]	Instruction register[3:0]



- **Note:** All simulations pictured in this document were run using a 10 ns clock. The actual design will have a 100 ns clock.

Test Input Connections (as seen in waveform)

Test Input Name	Description
clk	CLK
ui_in[7:0]	W bus
ui_out[11:0]	\LI
ui_out[2:0]	\EI
ui_out[0:0]	CLR
uo_out[3:0]	Instruction register[7:4]
uo_out[0:0]	Instruction register[3:0]

Output Register

Label	Input/Output	Description
CLK [1 bit]	Input	Clock signal. Executes actions on rising edges.
W bus [8 bit]	Input	Data from the bus lines that are to be written to the O
\LO [1 bit]	Input	Control signal that decides whether to read from the bu
Output register [8 bit]	Output	Register data that will be written to the binary display.

Pinouts when output register is selected

Test Input Name	Description
clk	CLK
ui_in[7:0]	W bus
ui_o_in [4]	\LO
uo_out[7:0]	Output register



Test Input Connections (as seen in waveform)

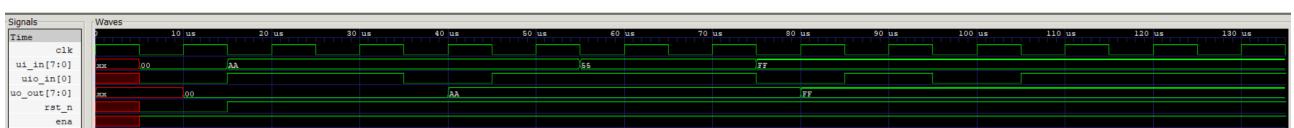
Test Input Name	Description
clk	CLK
ui_in[7:0]	W bus
ui_o_in [0]	\LO
uo_out[7:0]	Output register

B Register

Label	Input/Output	Description
CLK [1 bit]	Input	Clock signal. Executes actions on rising edges.
W bus [8 bit]	Input	Data from the bus lines that are to be written to the B register.
\LB [1 bit]	Input	Control signal that decides whether to read from the bus and write to the B register.
B register [8 bit]	Output	Register data that will be written to adder/subtractor.

Pinouts when b register is selected

Test Input Name	Description
clk	CLK
ui_in[7:0]	W bus
ui_o_in [4]	\LB
uo_out[7:0]	B register



Test Input Connections (as seen in waveform)

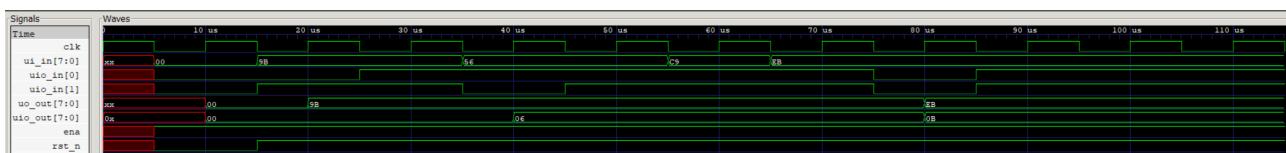
Test Input Name	Description
clk	CLK
ui_in[7:0]	W bus
ui_o_in [0]	\LMD
uo_out[7:0]	B register

Input and MAR

Label	Input/Output	Description
CLK [1 bit]	Input	Clock signal. Executes actions on rising edges.
W bus [8 bit]	Input	Data from the bus lines that are to be written either Input or Output.
\LMD [1 bit]	Input	Control signal that decides if W bus data is to be written.
\LMA [1 bit]	Input	Control signal that decides if W bus data is to be written.
Input register [8 bit]	Output	Register data to be written to memory.
MAR [4 bit]	Output	Register data taken by RAM that controls where the data is stored.

Pinouts when input and mar register is selected

Test Input Name	Description
clk	CLK
ui_in[7:0]	W bus
ui_o_in [4]	\LMD
ui_o_in [5]	\LMA
uo_out[7:0]	Input register
uo_out[3:0]	MAR



Test Input Connections (as seen in waveform)

Test Input Name	Description
clk	CLK

Test Input Name	Description
ui_in[7:0]	W bus
ui_o_in [0]	\LMD
ui_o_in 1	\LMA
uo_out[7:0]	Input register
uo_out[3:0]	MAR

Pinout

#	Input	Output	Bidirectional
0	in_0	out_0	extra_output_0
1	in_1	out_1	extra_output_1
2	in_2	out_2	extra_output_2
3	in_3	out_3	extra_output_3
4	in_4	out_4	extra_input_0
5	in_5	out_5	extra_input_1
6	in_6	out_6	register_select_0
7	in_7	out_7	register_select_1

Xor-Logic [423]

- Author: Haohua Li
- Description: A xor logic from input to output pins
- GitHub repository
- HDL project
- Mux address: 423
- Extra docs
- Clock: 0 Hz

How it works

It uses SPI interface for serial input and convert 8 bit data to parallel output.

How to test

Use RP2040 or other boards to communicate.

External hardware

WIP.

Pinout

#	Input	Output	Bidirectional
0	ui_in[0]	uo_out[0]	ui_out[0]
1	ui_in1	uo_out1	ui_out1
2	ui_in2	uo_out2	ui_out2
3	ui_in[3]	uo_out[3]	ui_out[3]
4	ui_in[4]	uo_out[4]	ui_out[4]
5	ui_in[5]	uo_out[5]	ui_out[5]
6	ui_in[6]	uo_out[6]	ui_out[6]
7	ui_in[7]	uo_out[7]	ui_out[7]

Leaky Integrate Fire Neuron [424]

- Author: Rocky Lim
- Description: Simulates a Leaky Integrate Fire Neuron based on snnTorch's implementation
- GitHub repository
- HDL project
- Mux address: 424
- Extra docs
- Clock: 0 Hz

How it works

This chip takes in an 8-bit number voltage to simulate a Leaky Fire Integrate (LIF) Network. The 8-bit number is split into two different neurons in which they have their respective layers, and it takes that voltage to act as an input current to the LIF neurons. Each neuron generates a spike when the threshold, defined to be 8, is reached or surpassed. Once an input current is passed through, each neuron will decay the value over each clock cycle by shifting the bits of the current state once as it constantly takes the input current. The idea behind the layers is for more significant spikes to be able to reach the output states while less significant events would not affect the output.

How to test

N/A

External hardware

N/A

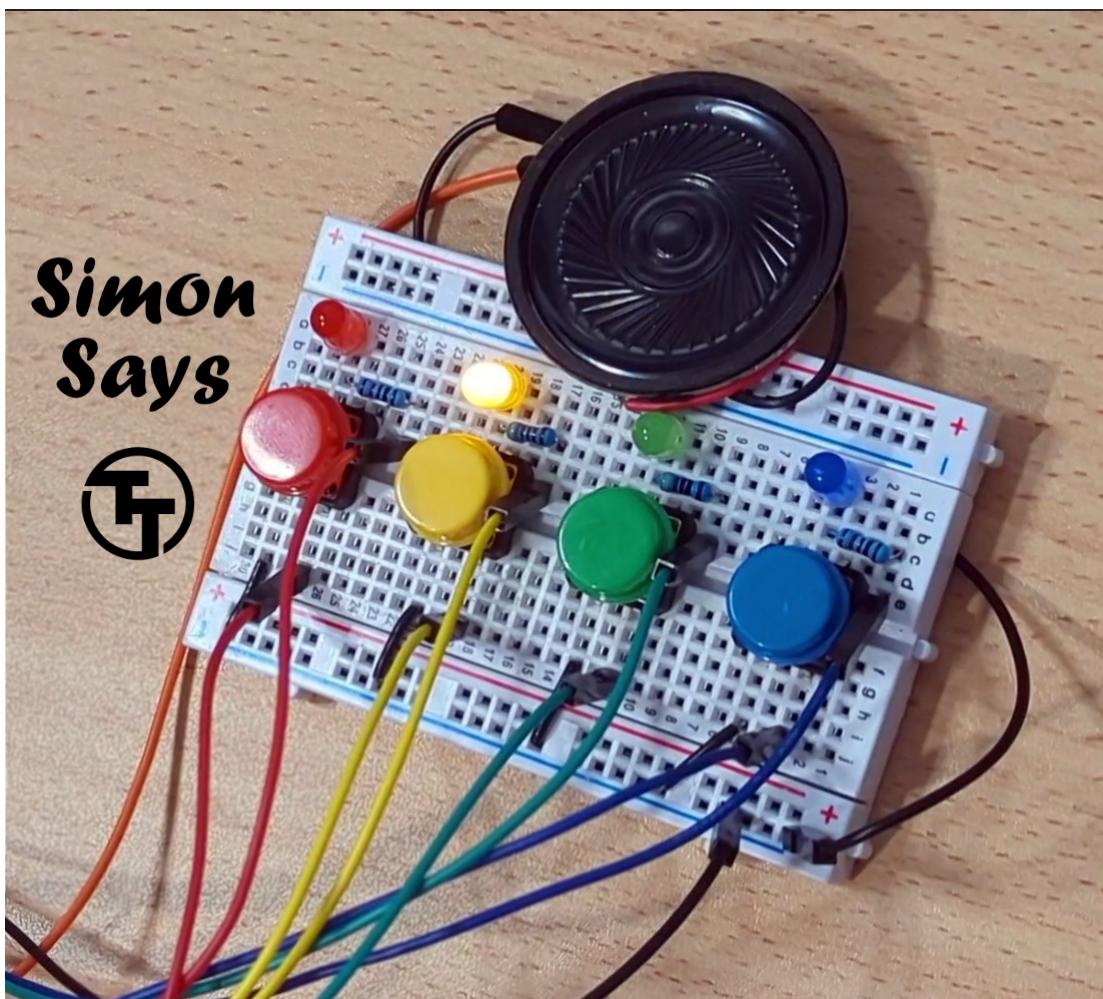
Pinout

#	Input	Output	Bidirect
0	Input Current Bit [0] (Input Neuron 1)	State Variable Bit [0] (Output Neuron 1)	Spike B
1	Input Current Bit 1 (Input Neuron 1)	State Variable Bit 1 (Output Neuron 1)	Spike B
2	Input Current Bit 2 (Input Neuron 1)	State Variable Bit 2 (Output Neuron 1)	Spike B
3	Input Current Bit [3] (Input Neuron 1)	State Variable Bit [3] (Output Neuron 1)	Spike B
4	Input Current Bit [4] (Input Neuron 2)	State Variable Bit [4] (Output Neuron 2)	Spike B

#	Input	Output	Bidirect
5	Input Current Bit [5] (Input Neuron 2)	State Variable Bit [5] (Output Neuron 2)	Spike B
6	Input Current Bit [6] (Input Neuron 2)	State Variable Bit [6] (Output Neuron 2)	Spike B
7	Input Current Bit [7] (Input Neuron 2)	State Variable Bit [7] (Output Neuron 2)	Spike B

Simon Says memory game [425]

- Author: Uri Shaked
 - Description: Repeat the sequence of colors and sounds to win the game
 - GitHub repository
 - HDL project
 - Mux address: 425
 - Extra docs
 - Clock: 50000 Hz



How it works

Simon says is a simple electronic memory game: the user has to repeat a growing sequence of colors. The sequence is displayed by lighting up the LEDs. Each color also has a corresponding tone.

In each turn, the game will play the sequence, and then wait for the user to repeat the sequence by pressing the buttons according to the color sequence. If the user repeated

the sequence correctly, the game will play a “leveling-up” sound, add a new color at the end of the sequence, and move to the next turn.

The game continues until the user has made a mistake. Then a game over sound is played, and the game restarts.

Check out the online simulation at <https://wokwi.com/projects/408757730664700929> (including wiring diagram).

Clock settings

The `clk_sel` input selects the clock source:

- 0: external 50 KHz clock, provided through the `clk` input.
- 1: internal clock, generated by the `ring_osc` module, with a frequency of ~55 KHz.

The internal clock is generated by a 13-stage ring oscillator, divided by 16384 to get the desired frequency. The divider value was determined by running the ring oscillator simulation in `<xschem/simulation/ring_osc.spice>`.

When using the internal clock, its signal is also output on the `uo_out[7]` pin for debugging purposes.

How to test

Use a Simon Says Pmod to test the game.

Provide a 50 KHz clock input, reset the game, and enjoy!

If you don't have the Pmod, you can still connect the hardware manually as follows:

1. Connect the four push buttons to pins `btn1`, `btn2`, `btn3`, and `btn4`. Also connect each button to a pull down resistor.
2. Connect the LEDs to pins `led1`, `led2`, `led3`, and `led4`, matching the colors of the buttons (so `led1` and `btn1` have the same color, etc.). Don't forget current-limiting resistors!
3. Connect the speaker to the `speaker` pin (optional).
4. Connect the seven segment display as follows: `seg_a` through `seg_g` to individual segments, `dig1` to the common pin of the tens digit, `dig2` to the common pin of the ones digit. Set `seginv` according to the type of 7 segment display you have: high for common anode, low for common cathode.
5. Reset the game, and then press any button to start it. Enjoy!

External Hardware

Simon Says Pmod or four push buttons (with pull-down resistors), four LEDs, and optionally a speaker/buzzer and two digit 7-segment display.

Pinout

#	Input	Output	Bidirectional
0	btn1	led1	seg_a
1	btn2	led2	seg_b
2	btn3	led3	seg_c
3	btn4	led4	seg_d
4	seginv	speaker	seg_e
5		dig1	seg_f
6		dig2	seg_g
7	clk_sel	clk_internal	

Tiny Tapeout Group 7 Lab D [426]

- Author: Will and Andrea
- Description: Our project implements a 4x4 array multiplier
- GitHub repository
- HDL project
- Mux address: 426
- Extra docs
- Clock: 0 Hz

How it works

Our program works by using a 4x4 array multiplier computes the product of two 4-bit binary numbers, m and q, through bitwise multiplication and summing partial products. Each bit of q is multiplied by every bit of m, generating partial products that are shifted based on their significance. Full adders (FA) then sum these partial products. At each stage, the full adders combine two partial product bits and any carry from the previous stage. As the process progresses through the rows, the number of bits to sum increases, which is managed by additional full adders. The final output is an 8-bit product p, with the least significant bit produced by the sum of the first row and the most significant bit formed by the final carry after all additions.

How to test

To test the 4x4 multiplier feed the multiplier two 4 bit inputs. From here the partial products will be calculated and the remaining product should be a binary representation of the decimal product. To verify you can convert final products between binary and decimal and compare expected values.

External hardware

Tiny Tapeout design

Pinout

#	Input	Output	Bidirectional
0	m[0]	p[0]	
1	m1	p1	

#	Input	Output	Bidirectional
2	m2	p2	
3	m[3]	p[3]	
4	q[0]	p[4]	
5	q1	p[5]	
6	q2	p[6]	
7	q[3]	p[7]	

SPI 7-segment display [427]

- Author: Garima Bajpayi
- Description: A small SPI slave module driving the 7-segment display on the TT-Carrier board
- GitHub repository
- HDL project
- Mux address: 427
- Extra docs
- Clock: 0 Hz

How it works

A small SPI slave device, receives 6-bit messages to display the lower 4-bits on the 7-segment display on the TinyTapeout Carrier board

How to test

- SCLK, SS and MOSI is provided through the inputs clk, ui[0] and ui[1] respectively.
- First two bits in 6-bit message serve as the command, and can be 01 or 10.
- 01 causes the decimal point ($uo[7]$) to turn on and display the next 4 bits on the 7-segment display.
- 10 behaves exactly the same, just switches the decimal point off.
- In case of malformed instructions (11 or 00), the decimal point switches on, with the rest of the display off.

External hardware

- Carrier board Seven Segment Display
- Microcontroller to drive the SPI slave device

Pinout

#	Input	Output	Bidirectional
0	SS	Segment A	
1	MOSI	Segment B	
2		Segment C	

#	Input	Output	Bidirectional
3		Segment D	
4		Segment E	
5		Segment F	
6		Segment G	
7	SCLK	Segment DP	

8-bit-CARRY_SKIP [428]

- Author: Aaquil Kasham, Temiloluwa Omomuwasan
- Description: 8 bit input adder
- GitHub repository
- HDL project
- Mux address: 428
- Extra docs
- Clock: 0 Hz

How it works

This project implements an 8-bit carry-skip adder using a combination of ripple-carry and skip logic for enhanced performance. The adder is divided into two 4-bit sections. The lower 4 bits compute the initial partial sum and generate a carry-out, which is then either passed directly to the upper 4-bit section or skipped, depending on the carry-propagate signal. This design reduces the delay associated with carry propagation, making it more efficient than a conventional ripple-carry adder. The final 8-bit sum is registered and outputted in sync with the clock signal.

How to test

To test the carry-skip adder:

Load the design into your simulation environment.

Set the ui_in and uio_in inputs with the desired 8-bit values for addition.

The result of the addition will appear on uo_out after each rising edge.

Verify that the output matches expected values by comparing uo_out with the expected result.

For more extensive testing, a testbench can be used to automate input combinations and check results across various cases.

External hardware

No external hardware is required for this project. List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Pinout

#	Input	Output	Bidirectional
0	a[0]	sum[0]	b[0]
1	a1	sum1	b1
2	a2	sum2	b2
3	a[3]	sum[3]	b[3]
4	a[4]	sum[4]	b[4]
5	a[5]	sum[5]	b[5]
6	a[6]	sum[6]	b[6]
7	a[7]	sum[7]	b[7]

AtomNPU [429]

- Author: Aakash Apoorv
- Description: A 4-bit Neural Processing Unit performing multiply-accumulate operations
- GitHub repository
- HDL project
- Mux address: 429
- Extra docs
- Clock: 100000000 Hz

How it works

The **AtomNPU** (Neural Processing Unit) is a compact, 4-bit processing module designed to perform basic **multiply-accumulate (MAC)** operations, essential for neural network computations. This NPU efficiently processes input activations and weights to produce quantized output results.

Functional Components

1. Inputs:

- **input_data [3:0] (ui_in[3:0])**: Represents the 4-bit activation vector input to the NPU.
- **weight [3:0] (uio_in[3:0])**: Represents the 4-bit weight vector applied to the activation vector.
- **start (uio_in[4])**: A control signal that initiates the MAC operation.

2. Outputs:

- **output_data [3:0] (uo_out[3:0])**: The 4-bit result of the multiply-accumulate operation.
- **done (uo_out[4])**: A status signal indicating the completion of the MAC operation.

3. Control Signals:

- **clk (Clock)**: Synchronizes the operations within the NPU.
- **rst_n (Reset)**: An active-low signal that resets the NPU to its initial state.

Operational Workflow

1. Initialization (**IDLE** State):

- Upon receiving a **reset (rst_n low)**, the NPU enters the **IDLE** state.
- All internal registers, including the accumulator and bit counter, are reset to 0.
- The done signal is deasserted (0).

2. Start Operation (**CALC** State):

- When the **start** signal is asserted (1), the NPU transitions from **IDLE** to **CALC**.
- The **input_data** and **weight** vectors are loaded into their respective registers.
- The accumulator is initialized to 0, and the bit counter is reset to 0.

3. Multiply-Accumulate Process:

- The NPU processes each bit of the **weight** vector in a **shift-add** manner over **4 clock cycles** (one for each bit).
- **For each bit (bit_count from 0 to 3):**
 - **Check the Least Significant Bit (LSB) of weight:**
 - * If the LSB (**weight[0]**) is 1, the **input_data** is **left-shifted** by the current **bit_count** and **added** to the accumulator.
 - * This effectively multiplies the **input_data** by the corresponding bit weight.
 - **Shift Right:** The **weight** is shifted right by 1 bit to process the next bit in the subsequent cycle.
 - **Increment Bit Counter:** Moves to the next bit.

4. Completion (**DONE** State):

- After processing all 4 bits, the NPU transitions to the **DONE** state.
- **Clamping Logic:**
 - If the **accumulator** exceeds 15 (8'd15), the **output_data** is clamped to 15 to maintain the 4-bit width.
 - Otherwise, the accumulator's lower 4 bits are assigned to **output_data**.
- The **done** signal is asserted (1) to indicate the operation's completion.
- The NPU returns to the **IDLE** state, ready for the next operation.

Clamping Mechanism To prevent overflow and ensure the output remains within the 4-bit constraint, the NPU incorporates a **clamping mechanism**:

- **Condition:** If the **accumulator** value after the MAC operation exceeds 15.
- **Action:** The **output_data** is set to 15 (4'd15).
- **Else:** The **output_data** reflects the accumulator's value.

How to test

Below is a step-by-step guide to facilitate thorough testing.

Testing Procedure

1. Initialization:

- **Power Up:** Ensure the ASIC is properly powered.
- **Reset:** Press the **reset button** (asserting `rst_n` low) to initialize the NPU.

2. Setting Inputs:

- **Input Data (`input_data [3:0]`):**
 - Use **switches/buttons** connected to `ui_in[3:0]` to set the 4-bit activation vector.
- **Weight (`weight [3:0]`):**
 - Use **switches/buttons** connected to `uio_in[3:0]` to set the 4-bit weight vector.

3. Initiating Operation:

- Press the **start button** (connected to `uio_in[4]`) to begin the MAC operation.
- The **start** signal is internally connected to initiate the NPU's state machine.

4. Observing Outputs:

- **output_data [3:0] (`uo_out[3:0]`):**
 - Observe the **LEDs** connected to `uo_out[3:0]` to view the resulting 4-bit output.
- **done Signal (`uo_out[4]`):**
 - The **status LED** connected to `uo_out[4]` will illuminate (1) once the operation is complete.

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	input_data[0]	output_data[0]	weight[0]
1	input_data1	output_data1	weight1
2	input_data2	output_data2	weight2
3	input_data[3]	output_data[3]	weight[3]
4			
5			
6			
7			

Semana UCU Verilog [430]

- Author: Universidad Católica del Uruguay
- Description: Union of projects done in class
- GitHub repository
- HDL project
- Mux address: 430
- Extra docs
- Clock: 0 Hz

Summary

This project is a compilation of designs created by students with little to no knowledge in electronics, as a part of a hands-on learning course during * SEMANA UCU* , with their outputs multiplexed so we can test all. There were 15 total projects submitted, based on 3 different guidelines. Select the project using `mux_in[0:3]`.

Guidelines

1- Basic Project

- Description: A shift register with `ui_in[0]` as input and `ui_in1` as external clock. When the shift register contains a specific key chosen by the students, `ui_out[0]` is driven to 1.
- How to test: Connect `ui_in1` with an external clock and insert the key via `ui_in1` from MSB to LSB

2- Advanced Project N°1

- Description: Decoder from 3 bits to 7 segment display with `ui_in[2:0]` as inputs. Some groups upped it to 4 bits
- How to test: Input a 3 bit number through `ui_in[2:0]` and check if the output lights up the correct number (Watch out, most groups made `ui_in[0]` be the MSB and `ui_in2` be the LSB of your input)

3- Advanced Project N°2

- Description: A 3 bit counter, driven by an external clock through ui_in[0], connected to the 3 bits to 7 segment display decoder from Advanced Project N°1. Once again some groups upped both the counter and the decoded to 4 bits.
- How to test: Connect ui_in[0] to an external clock and check if the 7 segment display lights up correctly.

Projects (Ordered by mux value)

Group 0

- Member(s): Locatelli, Roldós
- Wokwi: <https://wokwi.com/projects/410732069226456065>
- Guideline chosen for project: Advanced Project N°1
- Details: Uses 4 bits as input, and a common cathode display

Group 1

- Member(s): Giacometti, Salvo, Varela
- Wokwi: <https://wokwi.com/projects/410463015062285313>
- Guideline chosen for project: Advanced Project N°2
- Details: Uses a common cathode display, and counts up to 10 and overflows.

Group 2

- Member(s): Raposo
- Wokwi: <https://wokwi.com/projects/410724169008053249>
- Guideline chosen for project: Advanced Project N°1
- Details: Uses 3 bits as input, and a common cathode display

Group 3

- Member(s): Bava, Perez
- Wokwi: <https://wokwi.com/projects/410732939207035905>
- Guideline chosen for project: Advanced Project N°1
- Details: Uses 3 bits as input, and a common cathode display

Group 4

- Member(s): Firpo, Pursals
- Wokwi: <https://wokwi.com/projects/410570046815176705>
- Guideline chosen for project: Advanced Project N°1
- Details: Uses 3 bits as input, and a common cathode display

Group 5

- Member(s): Martinez
- Wokwi: <https://wokwi.com/projects/410640428205329409>
- Guideline chosen for project: Advanced Project N°2
- Details: Uses a common cathode display, and counts up to 8 and overflows.

Group 6

- Member(s): Nasso, Juarez
- Wokwi: <https://wokwi.com/projects/410553650788005889>
- Guideline chosen for project: Advanced Project N°2
- Details: Uses a common cathode display, and counts up to 16 and overflows.
(Only displays correctly up to 9)

Group 7

- Member(s): Lenzuen, Gauthier
- Wokwi: <https://wokwi.com/projects/410463710171875329>
- Guideline chosen for project: Advanced Project N°2
- Details: Uses a common anode display, and counts up to 8 and overflows. In this case, clock is driven by ui_in1, and ui_in[0] sets the counter to 7

Group 8

- Member(s): Mendez, Vago
- Wokwi: <https://wokwi.com/projects/410463176068023297>
- Guideline chosen for project: Advanced Project N°2
- Details: Uses a common cathode display, and counts up to 16 and overflows.
(Only displays correctly up to 9)

Group 9

- Member(s): Albín
- Wokwi: <https://wokwi.com/projects/410462842465590273>
- Guideline chosen for project: Basic Project
- Details: Key is 0x11

Group 10

- Member(s): Muniz
- Wokwi: <https://wokwi.com/projects/410463191701250049>
- Guideline chosen for project: Basic Project
- Details: Key is 0xB2

Group 11

- Member(s): Cerizola, Mesa
- Wokwi: <https://wokwi.com/projects/410555856765101057>
- Guideline chosen for project: Basic Project
- Details: Key is 0x80

Group 12

- Member(s): Romano, Ventós
- Wokwi: <https://wokwi.com/projects/410463349567547393>
- Guideline chosen for project: Basic Project
- Details: Both 0x7F and 0xFF work as key

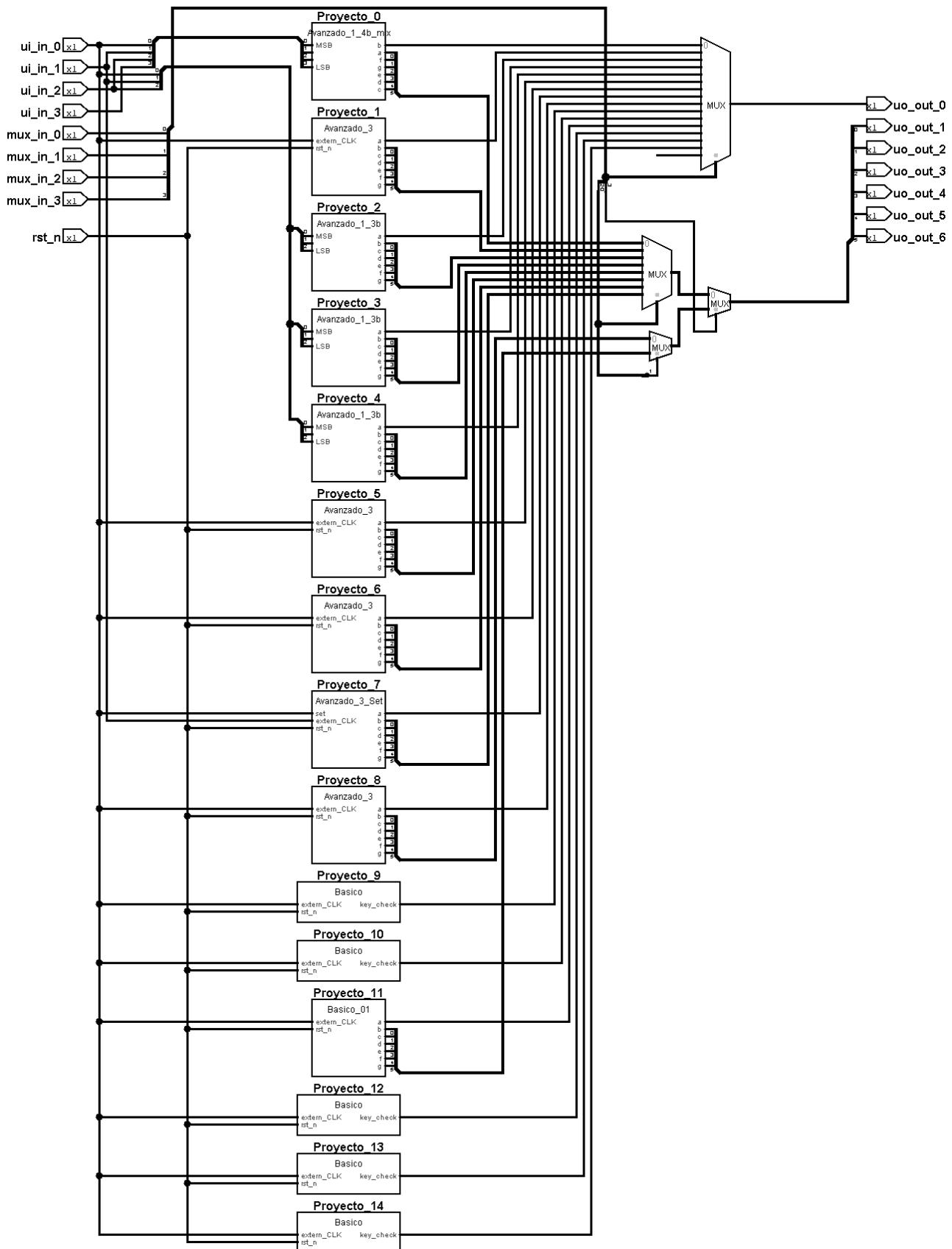
Group 13

- Member(s): Locatelli, Roldós
- Wokwi: <https://wokwi.com/projects/410639448686247937>
- Guideline chosen for project: Basic Project
- Details: Key is 0x49

Group 14

- Member(s): Hernández, Pedron
- Wokwi: <https://wokwi.com/projects/410643958389030913>
- Guideline chosen for project: Basic Project
- Details: Key is 0x55

Schematic



External hardware

7 segment displays (common anode and common cathode) LEDs

Pinout

#	Input	Output	Bidirectional
0	ui_in[0]	uo_out[0]	
1	ui_in1	uo_out1	
2	ui_in2	uo_out2	
3	ui_in[3]	uo_out[3]	
4	mux_in[0]	uo_out[4]	
5	mux_in1	uo_out[5]	
6	mux_in2	uo_out[6]	
7	mux_in[3]		

Enigma - 52-bit Key Length [431]

- Author: Virantha Ekanayake
- Description: Silicon implementation of an Enigma I machine with a limited plugboard supporting 3 wires
- GitHub repository
- HDL project
- Mux address: 431
- Extra docs
- Clock: 10000000000 Hz

How it works

Background This project features a silicon implementation of a 52-bit equivalent key model of the WWII-era Enigma code machine used by the Germans. The British, led by Alan Turing (as depicted in The Imitation Game), cracked this code, giving the Allies a crucial advantage in the war.

This electronic version is accurate and will match any simulator you can find on the web¹². Although almost every Enigma operates on similar principles, the particular model implemented here is the *Enigma I*³⁴ used by the German Army and Air Force; it comes with 3 rotor slots, the 5 original Rotors, the UKW-B Reflector, and plugboard. The only limitation is **that the plugboard only supports 3 wires**, whereas the actual wartime procedure was to use up to 10 wires. **This limits the key length of this implementation to 52-bits**. The calculation is shown below.

Key-length Calculation The Enigma is a symmetric⁵⁶ encryption engine, and the equivalent key length is comprised of the different settings and ways the rotors and plugboard can be arranged. See the excellent analysis⁷⁸ from Dr. Ray Miller at NSA for more details on the calculations below:

1. Selecting the three rotors, which can be arranged from right to left in any order:

¹<https://piotte13.github.io/enigma-cipher/>

²<https://www.dcode.fr/enigma-machine-cipher>

³<https://www.cryptomuseum.com/crypto/enigma/i/index.htm>

⁴<https://www.cryptomuseum.com/crypto/enigma/i/index.htm>

⁵<https://crypto.stackexchange.com/questions/33628/how-many-possible-enigma-machine-settings>

⁶<https://crypto.stackexchange.com/questions/33628/how-many-possible-enigma-machine-settings>

⁷https://www.nsa.gov/portals/75/documents/about/cryptologic-heritage/historical-figures-publications/publications/wwii/CryptoMathEnigma_Miller.pdf

⁸https://www.nsa.gov/portals/75/documents/about/cryptologic-heritage/historical-figures-publications/publications/wwii/CryptoMathEnigma_Miller.pdf

$$5 \times 4 \times 3 = \mathbf{60} \text{ possible ways}$$

2. Starting position of each rotor:

$$26 * 26 * 26 = \mathbf{17576}$$

3. Ring of each rotor (only two right rotors matter):

$$26 * 26 = \mathbf{676}$$

4. Plugboard with 3 wires (see table on p.9 for $p=3^9$ ¹⁰):

$$= 26! / (26-6)! / 3! / 8 = \mathbf{3,453,450} \text{ ways to plug in 3 wires}$$

The total ways (# of keys) to set up this particular Enigma is therefore:

$$60 * 17576 * 676 * 3,453,450 = 2,461,904,276,832,000 \text{ ways}$$

yielding a key length of ~52-bits.

Implementation Using the Python-based hardware description tool Amaranth HDL¹¹¹² for the first time made building, testing, and generating the Verilog implementation much easier. Given the complexity of the rotors' input-output mappings, I would've needed to write Python scripts anyway to generate Verilog logic. Amaranth streamlined this process and allowed seamless integration with my reference Python implementation for test generation.

- Rotor design: three separate combinational hardware rotors was too large and lacked
 - Meeting the tight area requirements involved several design iterations that narrowly missed targets late in the cycle. Amaranth's flexibility made re-architecting much simpler. For example, my initial approach of implementing

configurability. I ultimately created a single reconfigurable rotor block that processes data over six cycles, effectively forming a six-rotor pipeline (three forward, three backward after reflection).

- Plugboard Design:

⁹<https://www.cryptomuseum.com/crypto/enigma/i/index.htm>

¹⁰<https://www.cryptomuseum.com/crypto/enigma/i/index.htm>

¹¹<https://amaranth-lang.org/docs/amaranth/latest/>

¹²<https://amaranth-lang.org/docs/amaranth/latest/>

- Initial Attempt: A 26-entry, 5-bit lookup table using DFFs, which proved too large.
- Next Approach: A scan-chain-based design, but the hold-fix buffers and comparison logic made it even larger.
- Final Solution: A 26-entry, 5-bit lookup table using Skywater 130 standard-cell latches. This worked well since the plugboard functions like a ROM, with only a few initial writes to set the configuration. These writes are precisely pulsed using the state machine.

Key statistics	
Utilization	81%
Cells	1583
DFF	67
Latches	130
Frequency	35MHz

Operation The Enigma is designed to accept an 8-bit input (command plus data) at the clk edge. The internal state machine then takes a varying number of clk cycles to respond, raising the “Ready” signal when it’s ready to accept the next command. If the command generates an output, the raw value will be output on the bidir pins, and the LCD display will show the character generated.

Pinouts

Description	Width	Direction	Signal(s)
Command	3	in	ui_in[7:5]
Data	5	in	ui_in[4:0]
Scrambled output char	5	out	uo_out[4:0]
Ready	1	out	uo_out[5]
7-segment LCD	7	out	uo_out[6:0]

Commands The machine accepts the following 8 commands:

Encoding ¹³	Command	Data	Description
000	NOP	N/A	Do nothing
001	LOAD_START	Setting 0-25 (A-Z)	Set the start position of a rotor. [
010	LOAD_RING	Setting 0-25 (A-Z)	Set the ring setting of a rotor. Do

Encoding ¹³	Command	Data	Description
011	RESET	N/A	Go back to the initial state
100	SCRAMBLE	Input char 0-25 (A-Z)	Run a letter through the rotor. TH
101	LOAD_PLUG_ADDR	Src 0-25 (A-Z)	Set an internal register to where t
110	LOAD_PLUG_DATA	Dst 0-25 (A-Z)	Set the other end of the plug. No
111	SET_ROTORS	Rotor 0-4	Pick the Rotor type for each slot \

Sample run At some point, I'll have some code ready for running on the RPi on the PC, but for now, here is the pseudo code for setting up and scrambling/descrambling with this machine:

```

# Install the rotors
send_command(SET_ROTORS, 0)    # Set slot 0 to Rotor I
send_command(SET_ROTORS, 1)    # Set slot 0 to Rotor II
send_command(SET_ROTORS, 2)    # Set slot 0 to Rotor III

# Dial start position of the rotors
send_command(LOAD_START, 15)   # Set rotor 0 start position to P
send_command(LOAD_START, 5)    # Set rotor 1 start position to F
send_command(LOAD_START, 1)    # Set rotor 2 start position to B

# Dial ring position of the rotors
send_command(LOAD_RING, 18)    # Set rotor 0 start position to S
send_command(LOAD_RING, 5)     # Set rotor 1 start position to F
send_command(LOAD_RING, 24)    # Set rotor 2 start position to Y

# Set up the plugboard
# First, configure the plugboard default configuration with
# no swizzling of letters
for i in range(26):
    send_command(LOAD_PLUG_ADDR, i)
    send_command(LOAD_PLUG_DATA, i)

# Now, plug in three wires
send_command(LOAD_PLUG_ADDR, 0)    # connect A -> N
send_command(LOAD_PLUG_DATA, 13)

send_command(LOAD_PLUG_ADDR, 13)    # connect N -> A

```

¹³See the src/defines.py file

```

send_command(LOAD_PLUG_DATA, 0)

send_command(LOAD_PLUG_ADDR, 3)      # connect D -> E
send_command(LOAD_PLUG_DATA, 4)

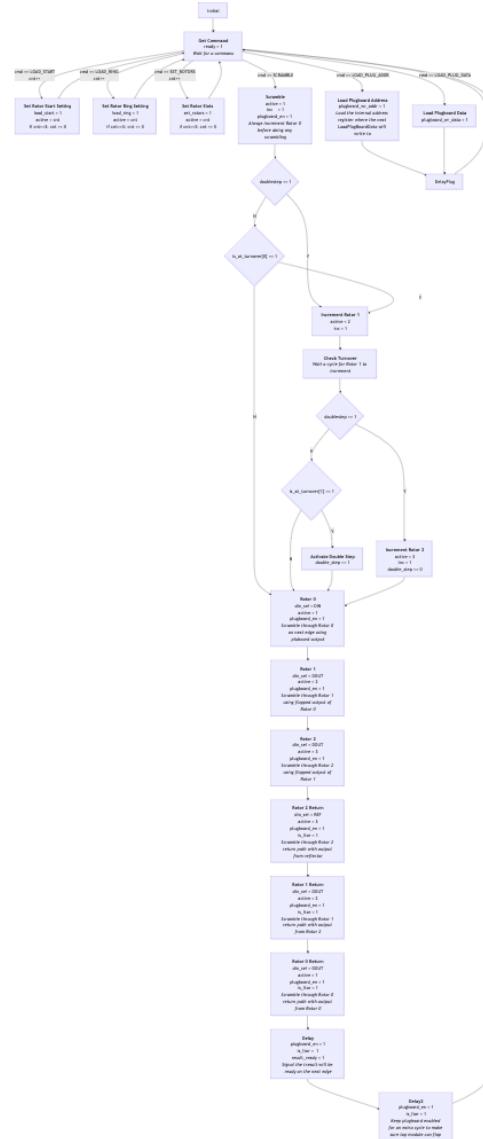
send_command(LOAD_PLUG_ADDR, 4)      # connect E -> D
send_command(LOAD_PLUG_DATA, 3)

send_command(LOAD_PLUG_ADDR, 25)     # connect Z -> B
send_command(LOAD_PLUG_DATA, 1)

send_command(LOAD_PLUG_ADDR, 1)      # connect D -> Z
send_command(LOAD_PLUG_DATA, 25)

# Now, enter letters into the machine and watch the coded char
# appear on the display
send_command(SCRAMBLE, 11)  # 'L' -> 'X'
send_command(SCRAMBLE, 14)  # 'O' -> 'K'

.
.
.
```



Control FSM

The state machine diagram source can be found on github¹⁴¹⁵.

How to test

Design verification

1. Generate the verilog from the Amarangth HDL source

```
cd tt10-enigma
python -m src.top
```

This will write a file `src/am_top.v` with the Enigma block. This block is connected to the TinyTapeout harness using `src/project.v`

¹⁴<https://github.com/virantha/tt10-enigma/blob/main/docs/fsm.md>

¹⁵<https://github.com/virantha/tt10-enigma/blob/main/docs/fsm.md>

2. Run the functional test

```
cd test  
make
```

3. Run the gate-level tests: After hardening (synthesis/pnr/gds), copy the gate_level_netlist.v into the test/ directory. Then:

```
make -B GATES=yes
```

External hardware

None. Uses the built-in 7-segment display on the PCB.

Pinout

#	Input	Output	Bidirectional
0	din[0]	seg[0]	dout[0]
1	din1	seg1	dout1
2	din2	seg2	dout[3]
3	din[3]	seg[3]	dout[4]
4	din[4]	seg[4]	dout[5]
5	cmd[0]	seg[5]	ready
6	cmd1	seg[6]	
7	cmd2	GND	

Frequency Encoder and Decoder [432]

- Author: Miguel Robles
- Description: Simple implementation of an 8-bit frequency encoder/decoder for a 1 bit frequency channel
- GitHub repository
- HDL project
- Mux address: 432
- Extra docs
- Clock: 10000000 Hz

How it works

Takes an 8-bit input voltage and treats it as a current injection to a LIF neuron

How to test

Do something

External hardware

NA

Pinout

#	Input	Output
0	Input frequency channel for decoder OR input bit for encoder [0]	LSB output of decoder [0]
1	Input encoder bit 1	Output encoder bit 1
2	Input encoder bit 2	Output encoder bit 2
3	Input encoder bit [3]	Output encoder bit [3]
4	Input encoder bit [4]	Output encoder bit [4]
5	Input encoder bit [5]	Output encoder bit [5]
6	Input encoder bit [6]	Output encoder bit [6]
7	Input encoder bit [7]	Output encoder bit [7]

synth_simple [433]

- Author: MM
- Description: Simple monophonic synth with PWM output
- GitHub repository
- HDL project
- Mux address: 433
- Extra docs
- Clock: 50000000 Hz

How it works

Simple monophonic synth with PWM output: 5 inputs, 8 switches to choose sounds/notes, left/right output channels

How to test

Connect the

External hardware

5 IR modules to generate notes, 8 switches to choose sounds/notes, two low-pass filters to generate left/right audio signals

Pinout

#	Input	Output	Bidirectional
0	note_enn_i[0]	pwm_left	sw_i[0]
1	note_enn_i1	pwm_right	sw_i1
2	note_enn_i2		sw_i2
3	note_enn_i[3]		sw_i[3]
4	note_enn_i[4]		sw_i[4]
5			sw_i[5]
6			sw_i[6]
7			sw_i[7]

carry skip adder [434]

- Author: Dron Sankhala
- Description: two 8-bit input adder
- GitHub repository
- HDL project
- Mux address: 434
- Extra docs
- Clock: 0 Hz

How it works

This project implements an 8-bit carry-skip adder using a combination of ripple-carry and skip logic for enhanced performance. The adder is divided into two 4-bit sections. The lower 4 bits compute the initial partial sum and generate a carry-out, which is then either passed directly to the upper 4-bit section or skipped, depending on the carry-propagate signal. This design reduces the delay associated with carry propagation, making it more efficient than a conventional ripple-carry adder. The final 8-bit sum is registered and outputted in sync with the clock signal.

How to test

To test the carry-skip adder:

1. Load the design into your simulation environment.
2. Set the `ui_in` and `uo_in` inputs with the desired 8-bit values for addition.
3. The result of the addition will appear on `uo_out` after each rising edge.
4. Verify that the output matches expected values by comparing `uo_out` with the sum of the inputs.

For more extensive testing, a testbench can be used to automate input combinations and check results across various cases.

External hardware

No external hardware is required for this project.

Pinout

#	Input	Output	Bidirectional
0	a[0]	sum[0]	b[0]
1	a1	sum1	b1
2	a2	sum2	b2
3	a[3]	sum[3]	b[3]
4	a[4]	sum[4]	b[4]
5	a[5]	sum[5]	b[5]
6	a[6]	sum[6]	b[6]
7	a[7]	sum[7]	b[7]

VGA clock [435]

- Author: Matt Venn
- Description: Shows the time on a VGA screen
- GitHub repository
- HDL project
- Mux address: 435
- Extra docs
- Clock: 31500000 Hz

How it works

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

Every minute the colours cycle.

How to test

Hook up a VGA monitor to the outputs and provide a clock at 31.5 MHz.

Adjust time with the inputs[2:0], and choose the type of VGA PMOD with the input[3].

External hardware

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

- <https://github.com/mole99/tiny-vga>
- <https://github.com/TinyTapeout/tt-vga-clock-pmod>

Set input[3] low to use tiny-vga and high to use vga-clock

#	Input	Output	Bidirectional
---	-------	--------	---------------

Pinout

#	Input	Output	Bidirectional
0	adjust hours	hsync / R1	
1	adjust minutes	vsync / G1	
2	adjust seconds	B0 / B1	
3	PMOD type select	B1 / VS	
4		G0 / R0	
5		G1 / G0	
6		R0 / B0	
7		R1 / HS	

Crossyroad [449]

- Author: Matt, Jovan, Ryan
- Description: Crossyroad game recreated on an ASIC
- GitHub repository
- HDL project
- Mux address: 449
- Extra docs
- Clock: 25000000 Hz

How it works

Chicken tries to cross the never ending roads with dangerous cars

Move chicken up by pressing the move_btn on ui_in[0]

Score increments while pressing the movement button

How to test

Hook up ui_in[0] to a debounced button, supply clk @25Mhz, connect to vga display

External hardware

- Debouned button to ui_in[0]
- 25Mhz clock to clk pin

Pinout

#	Input	Output	Bidirectional
0	move_btn	red	
1		green	
2		blue	
3		vsync	
4		red	
5		green	
6		blue	
7		hsync	

zc-sushi-demo [451]

- Author: Zachary Chen
- Description: sushi running
- GitHub repository
- HDL project
- Mux address: 451
- Extra docs
- Clock: 25200000 Hz

How it works

Sushi running across the screen

How to test

connect using TinyVGA PMOD and reset

External hardware

TinyVGA PMOD

Pinout

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

kch cd101 [453]

- Author: Johannes Pfau
- Description: KCH Things
- GitHub repository
- HDL project
- Mux address: 453
- Extra docs
- Clock: 25000000 Hz

How it works

This project provides a simple digital synth. It consists of a pulse-wave generator with programmable frequency, ADSR amplitude modulation with adjustable ADSR parameters and a simple one-pole IIR filter with programmable cutoff frequency. Digital audio output is generated using a simple first-order delta-sigma modulator. The lowpass filter for the reconstruction of this signal must be realized externally.

All parameters can be programmed using a simple SPI slave interface. Sound generation for a note starts when asserting the trigger signal and stops when the trigger is deasserted again. Triggering can happen both via a dedicated input or via SPI, which enables fully customizable operation using the SPI port. A VST/CLAP plugin is provided to generate the SPI commands from DAWs.

This project is educational and therefore makes some decisions that might not lead to an optimal design. For example, the structural presentation of the signal processing pipeline is directly realized in hardware. Some parts of the design are therefore clocked with a frequency as low as the sample rate and some even lower. This wastes a lot of possible performance, but it is easier for students to map the audio application to the circuit mentally, compared to introducing a more complex microcontroller-based system (which might be a more efficient design). The design shows how to realize a serial-parallel multiplier, use negative edge clocking, use simple small clock dividers, use multiple clocks etc.

More detailed information on all these topics will be provided later on.

How to test

As the design generates a lot of data on the single serial output pin, testing generates a lot of data. The cocotb testbench simulates and external lowpass and stores the audio data to a .s16 file which you can convert to .wav using ffmpeg:

Play the output file to assess whether the output is reasonable.

In addition, the testbench also compares to a golden reference output datastream, that was generated from behavioral simulation. This test is used to determine if there are any differences for the final implemented designs.

External hardware

- Button PMOD
- Audio PMOD
- SPI Master (No PMOD available. Use Adafruit board)

TODO: More detailed information about these things.

Pinout

#	Input	Output	Bidirectional
0	Trigger	Audio Data	
1	SPI CLK		
2	SPI MOSI		
3	SPI nSS		
4			
5			
6			
7			

SimpleSPIdev [455]

- Author: Zedulo
- Description: SPI device with GPIO/mem interface
- GitHub repository
- HDL project
- Mux address: 455
- Extra docs
- Clock: 0 Hz

How it works

See `readme.md`

How to test

Use e.g. a USB to SPI interface such as an MCP2210 based board. Interface to the device SPI interface.

External hardware

None for the board.

Pinout

#	Input	Output	Bidirectional
0	SPI MOSI	SPI MISO	
1	SPI CLK	debug	
2	SPI CS	system clk mirror	
3			
4			
5			
6		status	
7		spi reset	status

RNG_test [457]

- Author: Ba-Anh Dao
- Description: trying to implement lightweight all digital trng
- GitHub repository
- HDL project
- Mux address: 457
- Extra docs
- Clock: 50000000 Hz

How it works

Trial of Implementing the RO-based TRNG design

How to test

Just power up and get the data from the UART terminal

External hardware

UART terminal, PC to run the randomness test, Oscilloscope to measure the RO output

Pinout

#	Input	Output	Bidirectional
0	RX_Serial	TX_Serial	
1		o_RO	
2		o_RG	
3		led	
4			
5			
6			
7			

15bit GCD [459]

- Author: stephan
- Description: Greatest common denominator
- GitHub repository
- HDL project
- Mux address: 459
- Extra docs
- Clock: 100000 Hz

How it works

This project is a 15 bit Greatest Common Divisor module. Hand it two integers and it will calculate the GCD and output it.

15 bits input on ui and uio, where ui[0] is lsb, and uio[6] is MSB. So:

MSB

uio[6] uio[5] uio[4] uio[3] uio2 uio1 uio[0] ui[7] ui[6] ui[5] ui[4] ui[3] ui2 ui1 ui0

uio[7] is used as request signal to signal when first number and second number has been inputted. Request should be hold high when second number has inputted.

uo[7] is used as acknowledge signal, signalling when first input has been received and when GCD has been calculated.

uo[0] to uo[6] will output the GCD when acknowledge is high.

MSB	LSB
uo[6] uo[5] uo[4] uo[3] uo2 uo1 uo[0]	

How to test

Assign and hold an integer to the first 15 bits of ui_in and uio_in. Set REQ high. Wait for ACK. Set REQ low. Assign and hold an integer to the first 15 bits of ui_in and uio_in. Set REQ high. Wait for ACK. Read out the GCD. Release REQ to allow for a new calculation.

External hardware

Buttons and LEDs.

Pinout

#	Input	Output	Bidirectional
0	ui_in[0]	uo_out[0]	ui_out[0]
1	ui_in1	uo_out1	ui_out1
2	ui_in2	uo_out2	ui_out2
3	ui_in[3]	uo_out[3]	ui_out[3]
4	ui_in[4]	uo_out[4]	ui_out[4]
5	ui_in[5]	uo_out[5]	ui_out[5]
6	ui_in[6]	uo_out[6]	ui_out[6]
7	ui_in[7]	uo_out[7]	ui_out[7]

XY Spacewar [461]

- Author: Nicklaus Thompson
- Description: A scaled-down Spacewar game with a VGA mode. Might be Asteroids instead.
- GitHub repository
- HDL project
- Mux address: 461
- Extra docs
- Clock: 50000000 Hz

How it works

It's just the controller example from VGA playground, but you can also move a little square on the screen. It would have been an entire game if I hadn't taken a break when the Efabless shutdown was announced.

How to test

Operating the controller D-Pad will move the square around the screen.

External hardware

It accepts the VGA PMOD, Audio PMOD, and SNES PMOD. The pinout is configured for the demoscene competition.

Pinout

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

16-bit Logarithmic Approximate Floating Point Multiplier [463]

- Author: Anwesh Rao, B S Gurucharan, Shreyas M Iliger, Tushar M, Shylashree N, RV College of Engineering
- Description: A 16-bit floating-point multiplier utilizing logarithmic approximation to achieve fast and power-efficient floating-point multiplication with reduced hardware.
- GitHub repository
- HDL project
- Mux address: 463
- Extra docs
- Clock: 50000000 Hz

Credits

We sincerely acknowledge the **Center of Excellence in Integrated Circuits and Systems (CoE-ICAS)** and the **Department of Electronics and Communication Engineering, RV College of Engineering, Bengaluru**, for their invaluable support in providing us with the necessary knowledge and training.

We extend our special gratitude to **Dr. H V Ravish Aradhya (HoD, ECE)**, **Dr. K S Geetha (Vice Principal)** and **Dr. K N Subramanya (Principal)** for their continuous encouragement and support, enabling us to achieve **TAPEOUT** in **Tiny Tapeout 10**.

We are also deeply grateful to **Mahaa Santeep G (RVCE Alumni)** for his mentorship and invaluable guidance throughout the completion of this project.

The code provided is a Verilog module that implements a 16-bit logarithmic approximate floating-point multiplier. This module utilizes logarithmic approximation techniques to perform floating-point multiplication efficiently, reducing computational complexity while maintaining accuracy. It incorporates a state machine that processes the LSB(lower 8 bits) of the inputs in the first cycle and the MSB(upper 8 bits) in the next cycle, subsequently producing the LSB(lower 8 bits) of the output first, followed by the MSB(upper 8 bits) in the next cycle.

Key Components

The Logarithmic Approximate Floating-Point Multiplier (LAFPM) is a hardware-efficient multiplier that processes two 16-bit floating-point numbers using logarithmic

approximation techniques. Instead of traditional multiplication, this design reduces complexity by leveraging logarithmic transformations, shifts, and additions. This approach significantly lowers power consumption and area, making it ideal for resource-constrained applications such as machine learning accelerators and embedded systems. The multiplier operates using a finite state machine (FSM) that progresses through several key states:

IDLE – The system remains in this state until a non-zero input is detected. Once an input is received, it transitions to the next stage.

COLLECT – The multiplier collects the two 8-bit portions of each operand over multiple cycles to reconstruct the 16-bit floating-point numbers. After both parts are received, it moves to processing.

PROCESS_1 – The floating-point components, including the sign, exponent, and mantissa, are extracted for further computation.

PROCESS_2 – The mantissas undergo logarithmic approximation through bit-shifting techniques, reducing the complexity of multiplication.

PROCESS_3 – The approximated mantissas are added together using a logarithmic-based summation.

PROCESS_4 – A carry-out bit is determined, which helps adjust the exponent in the next stage.

PROCESS_5 – The new exponent is computed, and an additional approximation step refines the mantissa for better accuracy.

PROCESS_6 – The final floating-point result is assembled, combining the computed sign, exponent, and mantissa.

OUTPUT – The computed result is transmitted over multiple cycles. Once completed, the system returns to the IDLE state, ready for the next operation.

Inputs and Clock Frequency

u_in and **ui_o_in** are used to receive operands A and B through multiple cycles.

rst_n is the active-low reset signal.

clk operates at a 50 MHz frequency.

Table: State Transition for FP-16 Multiplication of (0x43BC)*(0x4190) | **Time (ns)**

ui_in (Input A)	ui_o_in (Input B)	Reset	State	uo_out (Output)
Clock				
		0	00000000	00000000 0 Reset xxxxxxxx 0
10	00000000	00000000 0	Reset 00000000 1	20 10111100 10010000
1	Reset 00000000	0	30 10111100	10010000 1 IDLE 00000000 1

	50	10111100	10010000	1	COLLECT_1	00000000	1		60	01000011
	01000001	1	COLLECT_1	00000000	0		70	01000011	01000001	1
	COLLECT_2	00000000	1		90	01000011	01000001	1	PROCESS_1	
	00000000	1		110	01000011	01000001	1	PROCESS_2	00000000	1
	130	01000011	01000001	1	PROCESS_3	00000000	1		150	01000011
	01000001	1	PROCESS_4	00000000	1		170	01000011	01000001	1
	PROCESS_5	00000000	1	190	01000011	01000001	1	PROCESS_6		
	00000000	1		210	01000011	01000001	1	OUTPUT_1	01110101	1
	220	01000011	01000001	1	OUTPUT_1	01110101	0		230	01000011
	01000001	1	OUTPUT_2	01001001	1		240	01000011	01000001	1
	OUTPUT_2	01001001	0							

Other Operands can also given as follows: Table: Multiple Operands with Expected

output	Input A	Input B	Output	0x4871
0x482e	0x54a6	0x41bd	0x46ef	0x4d31
0x44df	0x483d	50x12c		

Pinout

#	Input	Output	Bidirectional
0	A[0]/A[8]	P[0]/P[8]	B[0]/B[8]
1	A1/A[9]	P1/P[9]	B1/B[9]
2	A2/A[10]	P2/P[10]	B2/B[10]
3	A[3]/A[11]	P[3]/P[11]	B[3]/B[11]
4	A[4]/A[12]	P[4]/P[12]	B[4]/B[12]
5	A[5]/A[13]	P[5]/P[13]	B[5]/B[13]
6	A[6]/A[14]	P[6]/P[14]	B[6]/B[14]
7	A[7]/A[15]	P[7]/P[15]	B[7]/B[15]

TT_spiralPattern [465]

- Author: rkarl
- Description: Creates A Controllable Spiral Pattern
- GitHub repository
- HDL project
- Mux address: 465
- Extra docs
- Clock: 25000000 Hz

How it works

Creates a arcemedes spiral ($r=\theta$) using vga.

How to test

Set the frequency to 25 MHz and connect a pmod to vga to work. Changing the input values alters the foreground/background and speed of the spiral.

External hardware

PMOD VGA Switches (optional for control)

Pinout

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

ledtest [467]

- Author: jellyant
- Description: Turns one output ON.
- GitHub repository
- HDL project
- Mux address: 467
- Extra docs
- Clock: 10000000 Hz

How it works

It turns ON an output.

How to test

Don't do anything.

External hardware

No need.

Pinout

#	Input	Output	Bidirectional
0		A[0]	
1		A1	
2		A2	
3		A[3]	
4		A[4]	
5		A[5]	
6		A[6]	
7		A[7]	

I2C and SPI [480]

- Author: Vidyamol and Arun A V
- Description: Design of I2C and SPI communication protocols
- GitHub repository
- HDL project
- Mux address: 480
- Extra docs
- Clock: 400000 Hz

How it works

I2C and SPI protocol

How to test

send data enable clk

External hardware

No external hardware

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			

VGA Screensaver with Tiny Tapeout Logo [481]

- Author: Uri Shaked
- Description: Tiny Tapeout Logo bouncing around the screen (640x480, TinyVGA Pmod)
- GitHub repository
- HDL project
- Mux address: 481
- Extra docs
- Clock: 25175000 Hz

How it works

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



How to test

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

- tile (ui_in[0]) to repeat the logo and tile it across the screen,
- solid_color (ui_in1) to use a solid color instead of an animated gradient.

If you have a Gamepad Pmod connected, you can also use the following controls:

- Start button: start/pause bouncing

- Left/right/up/down: change the bouncing direction (if bouncing) or move the logo around the screen (if paused)

External hardware

- Tiny VGA Pmod
- Optional: Gamepad Pmod

Pinout

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

Perceptron Neuron [482]

- Author: Michael Chun
- Description: Makes a NAND gate with a perceptron neuron
- GitHub repository
- HDL project
- Mux address: 482
- Extra docs
- Clock: 0 Hz

How it works

Placeholder

How to test

Placeholder

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	Input current bit [0]	State variable bit [0]	
1	Input current bit 1	State variable bit 1	
2	Input current bit 2	State variable bit 2	
3	Input current bit [3]	State variable bit [3]	
4	Input current bit [4]	State variable bit [4]	
5	Input current bit [5]	State variable bit [5]	
6	Input current bit [6]	State variable bit [6]	
7	Input current bit [7]	State variable bit [7]	Output bit

SPI test [483]

- Author: Matt Venn
- Description: SPI test design
- GitHub repository
- HDL project
- Mux address: 483
- Extra docs
- Clock: 50000000 Hz

How it works

SPI test design based from https://github.com/calonso88/tt07_alu_74181

How to test

See https://github.com/calonso88/tt07_alu_74181

External hardware

Nothing required

Pinout

#	Input	Output	Bidirectional
0	cpol		
1	cpha		
2			
3		spi_miso	
4		spi_cs_n	
5		spi_clk	
6		spi_mosi	
7			

Histogramming [484]

- Author: isil isiksalan
- Description: histogramming unit
- GitHub repository
- HDL project
- Mux address: 484
- Extra docs
- Clock: 0 Hz

Histogramming on Chip for Short Luminescence Signals

Isil Isiksalan

09 November 2024

Background To measure the lifetime of short luminescence signals effectively, Time-Correlated Single Photon Counting (TCSPC) is commonly used. TCSPC measures the time intervals between photon pulse detections and a synchronized reference signal, usually from a laser. This data is used to create a histogram of photon arrival times, which helps in calculating luminescence lifetimes.

This module is useful in TCSPC systems, particularly after a Time-to-Digital converter or other time-tagging components. It processes 6-bit time-tagged data by bins. Designed for systems capable of supporting up to 64 bins, our module uses 32 numbered bins, mapping data into 32 bins to save space. This approach allows for the omission of missing bins after the decay fitting process.

Main Idea

The main idea is to integrate histogramming functionality within a digital signal processing system.

Overview of thettumhistogrammingModule

The `thttumhistogrammingmodule` is designed for digital signal processing, particularly for applications that require data binning based on their values. Implemented in Verilog, it takes an 8-bit input stream, using the last 6 bits to classify values, and outputs a histogram.

results of its operations through a finite state machine with states IDLE, RESETBINS.

Description of the Module

Inputs and Outputs:

- Inputs:

- uiin[7:0]: Main 8-bit input where binning is derived from the last
- uioin[7:0]: Auxiliary input, not used in the current logic.
- clk: Clock input for synchronization.
- rstn: Active-low reset signal.
- ena: Enable signal to activate histogramming.

- Outputs:

- uoout[7:0]: Outputs the count of the current bin.
- uioout[7:0]: Provides status flags including data validity, last binness for new data.
- uiooke[7:0]: Output enable signal for uioout.

Working Principle:

1. Initialization and Resetting: Clears bins to zero and sets the module for new data intake.

2. Data Handling and Binning: Receives data, determines the bin index, and updates bin counts according to the input conditions.

3. State Management: Manages data output and resets based on binning outcomes.

Module Testing

The module underwent thorough testing using a testbench that simulated various scenarios, including:

- Initial reset and setup.
- Ignoring even-numbered inputs.

- Filling multiple odd bins and managing overflow conditions.
- Checking reset functionality after data output.

- Testing operational robustness with manipulated enable signals.

All tests verified the module's functionality.

Pinout

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

Huffmann_Coder [485]

- Author: Marvin Barth
- Description: compresses ASCII input into Huffman codes
- GitHub repository
- HDL project
- Mux address: 485
- Extra docs
- Clock: 0 Hz

How it works

This ASIC compresses ASCII characters into Huffman codes, using a lookup table.

How to test

Send an ASCII character to ui[6:0], set ui[7] = 1 (Load), wait for valid_out = 1, then read the Huffman code from uo and uio.

External hardware

To communicate with the ASIC, you need either the RP2040 or an external MCU to send ASCII input and read the compressed Huffman output.

Pinout

#	Input	Output	Bidirectional
0	ASCII_in[0]	Huffman_out[0]	Huffman_out[8]
1	ASCII_in1	Huffman_out1	Huffman_out[9]
2	ASCII_in2	Huffman_out2	Valid_out
3	ASCII_in[3]	Huffman_out[3]	Bit_length[0]
4	ASCII_in[4]	Huffman_out[4]	Bit_length1
5	ASCII_in[5]	Huffman_out[5]	Bit_length2
6	ASCII_in[6]	Huffman_out[6]	Bit_length[3]
7	Load	Huffman_out[7]	

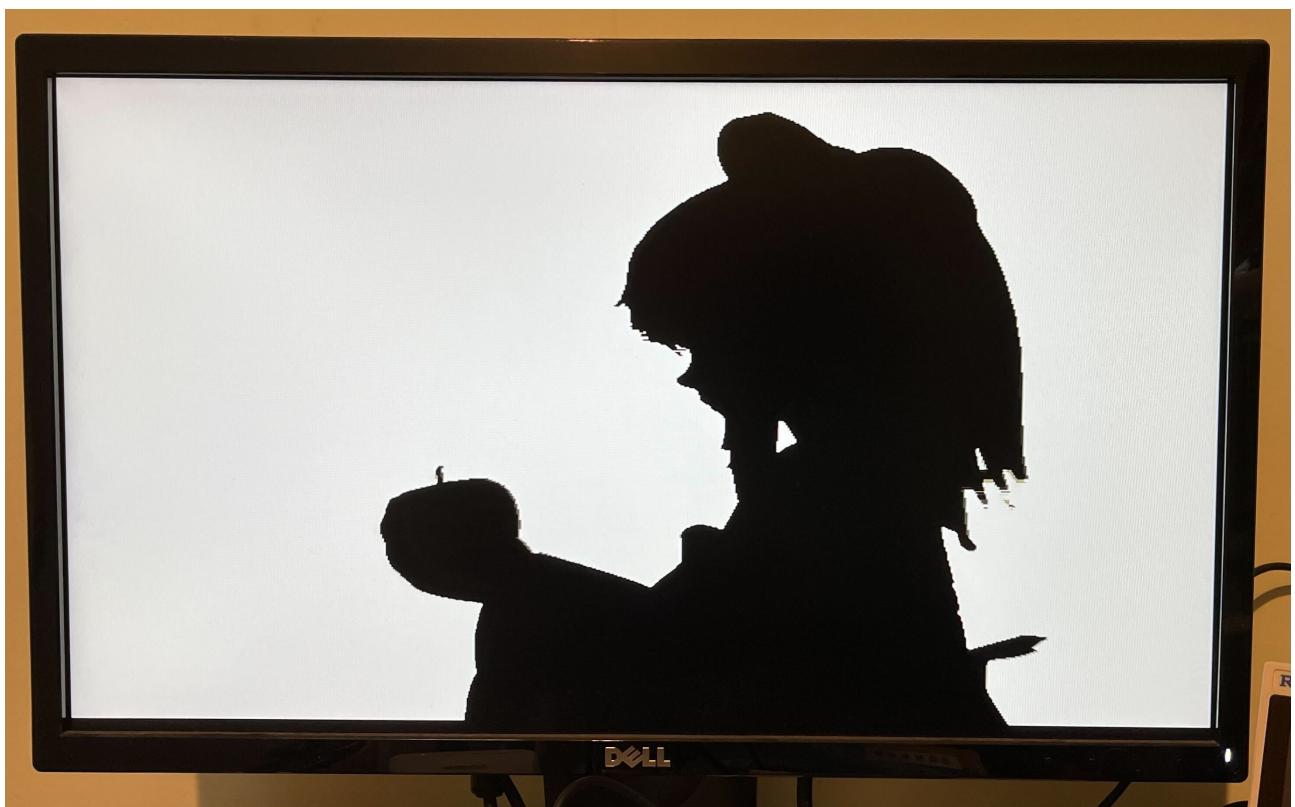
RLE Video Player [486]

- Author: Mike Bell
- Description: Reads run length encoded data from QSPI flash, displays on VGA
- GitHub repository
- HDL project
- Mux address: 486
- Extra docs
- Clock: 24000000 Hz

How it works

A 6bpp run length encoded image or video is read from a W25Q128JV or similar QSPI flash, and output to 640x480 VGA.

This is perfect for displaying the Bad Apple music video.



Run Length Encoding The encoding uses 16-bit words. Most words are a run length in the top 10 bits, and a colour in the bottom 6 bits. A run must come to the end at the end of each row.

A run must be at least 2 pixels, and any group of 3 consecutive runs within a row must be at least 12 pixels, otherwise the data buffer will empty.

8-bit mono audio data can be interleaved into the video stream. The PWM output value is updated by the value `0xC000 + sample`, these must be at the end of a row, but do not have to be present on every row. With a 24MHz project clock the row clock is exactly 30kHz.

To compress the audio slightly, sample deltas can also be used, packing 2, 3 or 4 samples into one command. These add a signed offset to the current sample value at the end of the next 2, 3 or 4 rows:

- `0xD000 + (offset1 < 6) + offset2` with 2 6-bit signed offsets
- `0xE000 + (offset1 < 8) + (offset2 < 4) + offset3` with 3 4-bit signed offsets
- `0xF000 + (offset1 < 9) + (offset2 < 6) + (offset3 < 3) + offset4` with 4 3-bit signed offsets

This means that quieter audio takes less space!

Note that row and frame repeat, which were supported on the TT07 and TT IHP 0p2 versions are not supported here because audio data is interleaved into the video data.

The data is read starting at address 0. The special word `0xBFC0` causes the player to stop and restart from address 0 at the beginning of the next frame, restarting the video. This could also be used to display a still image.

How to test

Create a RLE binary file (docs/scripts to do this TBD) and load onto the flash. The pinout matches the QSPI Pmod. This should be plugged into the audio Pmod, and then the audio Pmod plugged into the bidir pins. Note the flash must support the h6B Fast Read Quad Output command, with 8 dummy cycles between address and data.

Connect the Tiny VGA PMOD to the output pins.

Inputs 2-0 set the read latency for the SPI in half clock cycles, it's likely that will need to be set to 2 (set input 1 high and inputs 0 and 2 low). This latency depends on the total round trip time through the mux and out to the flash and back. Valid values are 1 to 4.

Run with a 24MHz clock.

Maximum file size The 16MB flash is only enough for the first minute of Bad Apple. But because the flash read is just one very long read it would be straightforward to supply the data stream from the RP2040 or other external source. To make it easier to do this from the demo board RP2040, the QSPI pin configuration can be modified by setting in3 high so that the 4 data pins are contiguous.

External hardware

- QSPI PMOD plugged into Audio PMOD
- Tiny VGA PMOD

Pinout

#	Input	Output	Bidirectional
0	SPI latency[0]	R1	CS
1	SPI latency1	G1	SD0 / SCK
2	SPI latency2	B1	SD1 / SD0
3	Select QSPI pinout	vsync	SCK / SD1
4		R[0]	SD2
5		G[0]	SD3
6		B[0]	Unused CS
7		hsync	PWM audio

Vedic multiplier [487]

- Author: Vivek Chiranjit
- Description: 8-bit binary unsigned multiplier
- GitHub repository
- HDL project
- Mux address: 487
- Extra docs
- Clock: 0 Hz

How it works

This is a 8-bit binary multiplier

Inputs: mul_ip_A mul_ip_B

Outputs: prod_low prod_high

How to test

Give the two 8-bit inputs through ui_in and uio_in. The multiplied data will be received at uo_out and uio_out

External hardware

None

Pinout

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

8-Bit CPU [488]

- Author: University of Waterloo - Fall 2024 ECE 298A
- Description: A basic 8-bit CPU design building off the SAP-1
- GitHub repository
- HDL project
- Mux address: 488
- Extra docs
- Clock: 50000000 Hz

How it works

This project is a basic 8-bit CPU design building off the SAP-1. It is a combination of various modules developed as a part of the ECE298A Course at the University of Waterloo.

The control block is implemented using a 6 stage sequential counter for sequencing micro-instructions, and a LUT for corresponding op-code to operation(s).

The program counter enumerates all values between 0 and F (15) before looping back to 0 and starting again. The counter will clear back to 0 whenever the chip is reset.

The Instruction register stores the current instructions and breaks it up into the opcode and address, which are passed into corresponding locations

The 16 Byte memory module consists of 16 memory locations that store 1 byte each. The memory allows for both read and write operations, controlled by input signals, as well as data supplied by the MAR.

The MAR is a register which handles RAM interactions, namely specifying the address for store/load, as well as the data to be stored.

The 8-bit ripple carry adder assumes 2s complement inputs and thus supports addition and subtraction. It pushes the result to the bus via tri-state buffer. It also includes a zero flag and a carry flag to support conditional operation using an external microcontroller. These flags are synchronized to the rising edge of the clock and are updated when the adder outputs to the bus.

The Accumulator register functions to store the output of the adder. It is synchronized to the positive edge of the clock. The accumulator loads and outputs its value from the bus and is connected via tri-state buffer. The accumulator's current value is always available as an output (and usually connected to the Register A input of the ALU)

The B register stores the second operand for ALU operations which is loaded from RAM.

The Output register outputs the value from register A onto the `uo_out` pins.

The 8 Bit Bus is driven by various blocks. We allow multiple blocks that are able to write using tri-state buffers.

Supported Instructions

Mnemonic	Opcode	Function
HLT	0x0	Stop processing
NOP	0x1	No operation
ADD {address}	0x2	Add B register to A register, leaving result in A
SUB {address}	0x3	Subtract B register from A register, leaving result in A
LDA {address}	0x4	Put RAM data at {address} into A register
OUT	0x5	Put A register data into Output register and display
STA {address}	0x6	Store A register data in RAM at {address}
JMP {address}	0x7	Change PC to {address}

Instruction Notes

- All instructions consist of an opcode (most significant 4 bits), and an address (least significant 4 bits, where applicable)

Control Signal Descriptions

Control Signal	Array	Component	Function
Cp	14	PC	Increments the PC by 1
Ep	13	PC	Enable signal for PC to drive the bus
Lp	12	PC	Tells PC to load value from the bus
nLma	11	MAR	Tells MAR when to load address from the bus
nLmd	10	MAR	Tells MAR when to load memory from the bus
nCE	9	RAM	Enable signal for RAM to drive the bus
nLr	8	RAM	Tells RAM when to load memory from the MAR
nLi	7	IR	Tells IR when to load instruction from the bus
nEi	6	IR	Enable signal for IR to drive the bus
nLa	5	A Reg	Tells A register to load data from the bus
Ea	4	A Reg	Enable signal for A register to drive the bus
Su	3	ALU	Activate subtractor instead of adder
Eu	2	ALU	Enable signal for Adder/Subtractor to drive the bus
nLb	1	B Reg	Tells B register to load data from the bus

Control Signal	Array Component	Function
nLo	0	Output Reg Tells Output register to load data from the bus

Sequencing Details

- The control sequencer is negative edge triggered, so that control signals can be steady for the next positive clock edge, where the actions are executed.
- In each clock cycle, there can only be one source of data for the bus, however any number components can read from the bus.
- Before each run, a CLR signal is sent to the PC and the IR.

Instruction Micro-Operations

Stage	HLT	NOP	STA	JMP
T0	Ep, nLma	Ep, nLma	Ep, nLma	Ep, nLma
T1	Cp	Cp	Cp	Cp
T2	nCE, nLi	nCE, nLi	nCE, nLi	nCE, nLi
T3	**	-	nEi, nLma	nEi, Lp
T4	-	-	Ea, nLmd	
T5	-	-	nLr	

Stage	LDA	ADD	SUB	OUT
T0	Ep, nLma	Ep, nLma	Ep, nLma	Ep, nLma
T1	Cp	Cp	Cp	Cp
T2	nCE, nLi	nCE, nLi	nCE, nLi	nCE, nLi
T3	nEi, nLma	nEi, nLma	nEi, nLma	Ea, nLo
T4	nCE, nLa	nCE, nLb	nCE, nLb	-
T5	-	Eu, nLa	Su, Eu, nLa	-

Instruction Micro-Operations Notes

- First three micro-operations are common to all instructions.
- NOP operation executes only the first three micro-operations.
- Cp signal is not asserted during the HLT instruction in T2.
- ** Halt internal register is set to 1. More on this later

Programmer

Stage	Control Signals	Programmer specific signals
T0	Ep, nLMA	ready = 1
T1	Cp	ready = 0
T2	-	-
T3	nLmd	read_ui_in = 1
T4	nLr	read_ui_in = 0, done_load = 1
T5	-	done_load = 0

Detailed Overview T0: Control Signals the same as the typical default microinstruction \– load the MAR with the address of the next instruction. Assert ready signal to alert MCU programmer (off chip) that CPU is ready to accept next line of RAM data.

T1: Increment the PC, the same as the typical default microinstruction. De-assert ready signal since the MCU programmer is polling for the rising edge.

T2: Do nothing to allow an entire clock cycle for programmer to prepare the data.

T3: Load the MAR with the data from the bus. Also, assert the read_ui_in signal which controls a series of tri-state buffers, attaches the ui_in pins straight to the bus.

T4: Load the RAM from the MAR. De-assert the read_ui_in signal (disconnect the ui_in pins from driving the bus since the ui_in pin data might be now inaccurate). Assert the done_load signal to indicate to the MCU that the chip is done with the ui_in data.

T5: De-assert done_load signal.

Programmer Notes The MCU must be able to provide the data to the ui_in pins (steady) between receiving the ready signal (assume worst case end of T0), and the bus needing the values (assume worst case beginning of T3).

Therefore, the MCU must be able to provide the data at a maximum of 2 clock periods.

IO Table: CB (Control Block)

Name	Verilog	Description	I/O	Width	Trigger
clk	clk	Clock signal	I	1	Edge Transition
resetn	rst_n	Set stage to 0	I	1	Active Low
opcode	opcode	Opcode from IR	I	4	NA
out	control_signals	Control Signal Array	O	15	NA
programming	programming	Programming mode	I	1	Active High
done_load	done_load	Executed Load during prog	O	1	Active High
read_ui_in	read_ui_in	Push ui_in onto bus	O	1	Active High
ready	ready_for_ui	Ready to prog next byte	O	1	Active High
HF	HF	Halting flag	O	1	Active High

IO Table: PC (Program Counter)

Name	Verilog	Description	I/O	Width	Trigger
bus	bus[3:0]	Connection to bus	IO	4	NA
clk	clk	Clock signal	I	1	Falling Edge
clr_n	rst_n	Clear to 0	I	1	Active Low
cp	Ep	Allow counter increment	I	1	Active High
ep	Cp	Output to bus	I	1	Active High
lp	Lp	Load from bus	I	1	Active High

PC (Program Counter) Notes

- Counter increments only when Cp is asserted, otherwise it will stay at the current value.
- Ep controls whether the counter is being output to the bus. If this signal is low, our output is high impedance (Tri-State Buffers).
- When CLR is low, the counter is cleared back to 0, the program will restart.
- The program counter updates its value on the falling edge of the clock.
- Lp indicates that we want to load the value on the bus into the counter (used for jump instructions). When this is asserted, we will read from the bus and instead of incrementing the counter, we will update each flip-flop with the appropriate bit and prepare to output.
- The least significant 4 bits from the 8-bit bus will be used to store the value on the program counter (0-15). Will be read from (JMP asserted) and written to (Ep asserted).
- clr_n has precedence over all.
- Lp takes precedence over Cp.

IO Table: Instruction Register (IR)

Name	Verilog	Description	I/O	Width	Trigger
bus	bus	Connection to bus	IO	8	NA
clk	clk	Clock signal	I	1	Rising Edge
clear	~rst_n	Clear to 0	I	1	Active High
opcode	opcode	Opcode from IR	O	4	NA
n_load	nLi	Load from Bus	I	1	Active Low
n_enable	nEi	Output to bus	O	1	Active Low

Instruction Register (IR) Notes

- The A Register updates its value on the rising edge of the clock.
- nEi controls whether the instruction is being output to the bus[3:0]. If this signal is high, our output is high impedance (Tri-State Buffers).
- nLi indicates that we want to load the value on the bus into the IR. When this is low, we will read from the bus and write to the register.
- When clear is high, the opcode is cleared back to NOP.
- IR always outputs the current value of the register to CB.

IO Table: RAM

Name	Verilog	Description	I/O	Width	Trigger
addr	mar_to_ram_addr	Address for read/write	I	4	NA
data_in	mar_to_ram_data	Data for write	I	8	NA
data_out	bus	Connection to bus	O	8	NA
lr_n	nLr	Load data from MAR	I	1	Active Low
ce_n	nCE	Output to bus	I	1	Active Low
clk	clk	Clock Signal	I	1	Rising edge
rst_n	'1'	Clear RAM	I	1	Active Low

RAM Notes

- Addressing: The memory is 4-bit addressable, where the address specifies which register (out of 16) is being accessed for reading or writing.
- Write operation: A byte of data is written to specific register in RAM, where the location is determined by the address. Requires write enable lr_n signal as active (low) and the clock edge to occur.

- Read operation: Data can be read from a specific register in RAM determined by the input address. Requires chip enable ce_n signal as active (low). The data is output on the bus, and it is updated on the clock edge.
- Output: Data is presented on the bus line when the chip is enabled for reading, and high-impedance (Z) otherwise.
- RAM is never reset, rather, we always flash it.

IO Table: MAR

Name	Verilog	Description	I/O	Width	Trigger
bus	bus	Connection to bus	IO	8	NA
clk	clk	Clock signal	I	1	Rising Edge
addr	mar_to_ram_addr	Address for read/write	O	4	NA
data	mar_to_ram_data	Data for write	O	8	NA
n_load_data	nLmd	Load data from Bus	I	1	Active Low
n_load_addr	nLma	Load address from Bus	I	1	Active Low

MAR Notes

- The MAR updates its value on the rising edge of the clock.
- nLmd indicates that we want to load the value on the bus into the data register. When this is low, we will read from the bus and write to the register.
- nLma indicates that we want to load the value on the bus[3:0] into the address register. When this is low, we will read from the bus and write to the register.
- MAR always outputs the current value of the data and address registers to the RAM module.

IO Table: ALU (Adder/Subtractor)

Name	Verilog	Description	I/O	Width	Trigger
clk	clk	Clock Signal	I	1	Rising edge
enable_out	Eu	Output to bus	I	1	Active High
Register A	reg_a	Accumulator Register	I	8	NA
Register B	reg_b	Register B	I	8	NA
subtract	sub	Perform Subtraction	I	1	Active High
bus	bus	Connection to bus	O	8	NA
Carry Out	CF	Carry-out flag	O	1	Active High
Result Zero	ZF	Zero flag	O	1	Active High

ALU (Adder/Subtractor) Notes

- Ea controls whether the counter is being output to the bus. If this signal is low, our output is high impedance (Tri-State Buffers).
- A Register and B Register always provide the ALU with their current values.
- When sub is not asserted, the ALU will perform addition: Result = A + B
- When sub is asserted, the ALU will perform subtraction by taking 2s complement of operand B: Result = A - B = A + !B + 1
- Carry Out and Result Zero flags are updated on rising clock edge.

IO Table: Accumulator (A) Register

Name	Verilog	Description	I/O	Width	Trigger
clk	clk	Clock Signal	I	1	Rising edge
bus	bus	Connection to bus	IO	8	NA
load	nLa	Load from bus	I	1	Active Low
enable_out	Ea	Output to bus	I	1	Active High
Register A	reg_a	Accumulator Register	O	8	NA
clear	rst_n	Clear Signal	I	1	Active Low

Accumulator (A) Register Notes

- The A Register updates its value on the rising edge of the clock.
- Ea controls whether the counter is being output to the bus. If this signal is low, our output is high impedance (Tri-State Buffers).
- nLa indicates that we want to load the value on the bus into the A Register. When this is low, we will read from the bus and write to the register.
- When CLR is low, the register is cleared back to 0.
- (Register A) always outputs the current value of the register to the ALU.

IO Table: B Register

Name	Verilog	Description	I/O	Width	Trigger
bus	bus	Connection to bus	IO	8	NA
clk	clk	Clock Signal	I	1	Rising edge
n_load	nLb	Load from bus	I	1	Active Low
value	reg_b	B Register value	O	8	NA

B Register Notes

- The B Register updates its value on the rising edge of the clock.
- nLb indicates that we want to load the value on the bus into the B Register. When this is low, we will read from the bus and write to the register.
- B Register always outputs the current value of the register to the ALU.

IO Table: Output Register

Name	Verilog	Description	I/O	Width	Trigger
bus	bus	Connection to bus	IO	8	NA
clk	clk	Clock Signal	I	1	Rising edge
n_load	nLo	Load from bus	I	1	Active Low
value	uo_out	B Register value	O	8	NA

Output Register Notes

- The Output Register updates its value on the rising edge of the clock.
- nLo indicates that we want to load the value on the bus into the B Register. When this is low, we will read from the bus and write to the register.

How to test

Provide input of op-code. Check that the correct output bits are being asserted/deasserted properly.

Setup

1. **Power Supply:** Connect the chip to a stable power supply as per the voltage specifications.
2. **Clock Signal:** Provide a stable clock signal to the clk pin.
3. **Reset:** Ensure the rst_n pin is properly connected to allow resetting the chip.

Testing Steps

1. Initial Reset:

- Perform a sync reset by pulling the `rst_n` pin low, waiting for 1 clock signal, and then pulling the `rst_n` high to initialize the chip.

2. Load Program into RAM:

- Use the `ui_in` pins to load a test program into the RAM. Ensure the programming pin is high during this process.
- Perform a sync reset by pulling the `rst_n` pin low, waiting for 1 clock signal, and then pulling the `rst_n` high to initialize the chip.
- Wait for the `ready_for_ui` signal to go high, indicating that the CPU is ready to accept data.
- Provide the first byte of data on the `ui_in` pins.
- Wait for the `done_load` signal to go high, indicating that the data has been successfully loaded into the RAM.
- Repeat the process for each byte of data:
 - Wait for `ready_for_ui` to go high.
 - Provide the next byte of data on the `ui_in` pins.
 - Wait for `done_load` to go high.
- Example program data:

```
0x10, # NOP
0x73, # JMP 0x3
0x00, # HLT
0x4F, # LDA 0xF
0x2E, # ADD 0xE
0x6D, # STA 0xD
0x50, # OUT
0x3F, # SUB 0xF
0x50, # OUT
0x4D, # LDA 0xD
0x50, # OUT
0x72, # JMP 0x2
0x10, # NOP
0x00, # Padding/empty instruction
0x02, # Constant 2 (data)
0x01 # Constant 1 (data)
```

3. Run Test Program:

- Set the programming pin low to exit programming mode.
- Perform a sync reset by pulling the `rst_n` pin low, waiting for 1 clock signal, and then pulling the `rst_n` high to initialize the chip.
- Monitor the `uo_out` and `ui_out` pins for expected outputs.
- Verify the control signals and data outputs at each clock cycle.

4. Functional Tests:

- Perform specific functional tests for each instruction (e.g., ADD, SUB, LDA, STA, JMP, HLT).
- Verify the correct execution of each instruction by checking the output and control signals.

Example Test Cases

- **HLT Instruction:** Example program data:

```
0x4E, # LDA 0xE
0x50, # OUT
0x00, # HLT
0x4F, # LDA 0xF
0x50, # OUT
0x00, # HLT
0x00, # Padding/empty instruction
0x09, # Constant 9 (data)
0xFF # Constant 255 (data)
```

This program should first output 9 and then NOT change that to 255. HF should be set to 1

- **NOP Instruction:** Example program data:

```
0x42, # LDA 0x2
0x50, # OUT
0x10, # NOP / Constant 16 (data)
```

```

0x1F, # NOP
0x4E, # LDA 0xF
0x50, # OUT
0x1F, # NOP
0x1F, # NOP / Constant 31 (data)

```

This program should flash the lower 4 bits of the output register on and off with different on/off times

- **NOP Instruction:** Example program data:

```

0x42, # LDA 0x2
0x50, # OUT
0x10, # NOP / Constant 16 (data)
0x1F, # NOP
0x4E, # LDA 0xF
0x50, # OUT
0x1F, # NOP
0x1F, # NOP / Constant 31 (data)

```

This program should flash the lower 4 bits of the output register on and off with different on/off times

- **ADD Instruction** Example program data:

```

0x50, # OUT
0x2E, # ADD 0xE
0x70, # JMP 0x0

```

```
0xFF, # Padding/empty instruction
0x01, # Constant 1 (data)
0xFF, # Padding/empty instruction
```

This program should add 1 to the A register, display it and loop back to the start. The output should be a counter from 0 to 255, then repeat.

CF should be set to 1 when the A register overflows, and 0 when it doesn't. CF=1 happens when the A register is 255 and 1 is added to it.

ZF should be set to 1 when the A register is 0, and 0 otherwise.

- **SUB Instruction** Example program data:

```
0x50, # OUT
0x3E, # SUB 0xE
0x70, # JMP 0x0
0xFF, # Padding/empty instruction
0x01, # Constant 1 (data)
0xFF, # Padding/empty instruction
```

This program should subtract 1 to the A register, display it and loop back to the start. The output should be a counter from 255 to 0, then repeat.

CF should be set to 1 when the A register overflows, and 0 when it doesn't. CF=0 happens when the A register is 0 and 1 is subtracted from it.

ZF should be set to 1 when the A register is 0, and 0 otherwise.

- **LDA Instruction**

See above for example program data.

- **OUT Instruction**

See above for example program data.

- **STA Instruction**

Example program data:

```
0x4E, # LDA 0xE
0x2F, # ADD 0xF
0x5F, # OUT
0x6E, # STA 0xF
0x2F, # ADD 0xE
0x5F, # OUT
0x00, # HLT
0xFF, # Padding/empty instruction
0x09, # Constant 9 (data)
0xFF # Constant 255 (data) -> Constant 8 (data)
```

This program should load 9 to the A register, add 255 to it, resulting in 8 (CF should set to 1) display it, store it in 0xF, add 9 to it, resulting in 17 (CF should set to 0) and display it. Then, it should halt, and set HF to 1.

- **JMP Instruction**

Example program data:

```
0x44, # LDA 0x4
0x5F # OUT
0x7D, # JMP 0xD
0x0F, # HLT
0x00, # Constant 0 (data)
0xFF, # Constant 5 (data)
0xFF, # Padding/empty instruction
```

```

0xFF, # Padding/empty instruction
0x45, # LDA 0x5
0x5F # OUT
0x0F, # HLT

```

This program should load 0x4 (0) to the A register, display it, NOT HALT, jump to 0xD, then load 0x5 (255) to the A register, display it, and halt. HF should be set to 1.

Acknowledgements

- Darius Rudaitis, Eshann Mehta: RAM
- Evan Armoogan, Catherine Ye: PC
- Damir Gazizullin, Owen Golden: ALU, Accumulator
- Roni Kant, Jeremy Kam: MAR, B Register, Output Register, Instruction Register
- Gerry Chen, Siddharth Nema: Control Block and Programmer
- ECE 298A Course Staff: Prof. John Long, Prof. Vincent Gaudet, Refik Yalcin

Pinout

#	Input	Output	Bidirectional
0	prog_in_0	output_register_0	in_programming
1	prog_in_1	output_register_1	out_ready_for_ui
2	prog_in_2	output_register_2	out_done_load
3	prog_in_3	output_register_3	out_CF
4	prog_in_4	output_register_4	out_ZF
5	prog_in_5	output_register_5	out_HF
6	prog_in_6	output_register_6	
7	prog_in_7	output_register_7	

Tiny piano [489]

- Author: Kenneth Petersen
- Description: A tiny musical note generator with 16 notes across 4 octaves and tremolo effect
- GitHub repository
- HDL project
- Mux address: 489
- Extra docs
- Clock: 10000000 Hz

How it works

This digital tone generator turns binary inputs into musical notes through the ancient art of frequency division:

- 20-bit counter that ticks away until it's time to toggle a square wave
- 16 musical notes to choose from
- 4 octaves, because sometimes you want to annoy dogs, sometimes submarines
- Tremolo effect for when regular beeping isn't dramatic enough
- 7-segment LED display that dances along, pretending to be a visualizer

Really it's just a binary counter that gets impatient at musically-appropriate intervals. Seemed simple enough since time sort of ran out from this project.

How to test

1. **Note selection:** Set ui_in[3:0] to pick your poison:

- 0 = C
- 9 = A/440Hz
- The others are somewhere in between

2. **Octave selection:** ui_in[5:4] lets you choose your pitch range:

- 00: Standard frequencies (for normal people)
- 01: One octave higher (for annoying people)
- 10: Two octaves higher (for annoying pets)
- 11: Three octaves higher (for annoying bats)

3. **Master switch:** ui_in[6] = 1 turns it on. Set to 0 for silence.

4. **Tremolo:** ui_in[7] = 1 adds cool effects.

The main square wave output comes out from `uo_out[7]`, while `uo_out[6:0]` provides visual confirmation that yes, you are indeed making noise, while also letting you know which kind. Solder jumpers to all pins except the DP since that one is the audio (`uo_out[7]`) I think it should be able to control the inputs from the Pi Pico, and maybe I could make a small keyboard for attaching also to control it.

External hardware

you'll need:

- RC low-pass filter (probably $1\text{k}\Omega + 0.1\mu\text{F}$ will do)
- DC blocking capacitor (unless playing AC/DC)
- Speaker or headphones
- Audio amplifier (optional)

Pinout

#	Input	Output	Bidirectional
0	Note select bit 0	Audio out	
1	Note select bit 1	Note LED 1	
2	Note select bit 2	Note LED 2	
3	Note select bit 3	Note LED 3	
4	Octave select bit 0	Note LED 4	
5	Octave select bit 1	Note LED 5	
6	Enable tone	Note LED 6	
7	Enable tremolo	Note LED 7	

carry_select [490]

- Author: Juan, Leyang
- Description: This project designs a 8-bit carry select adder.
- GitHub repository
- HDL project
- Mux address: 490
- Extra docs
- Clock: 0 Hz

How it works

The 8-bit carry select adder works through the full adder and mux. The Carry Select Adder works by essentially using two ripple adders, with one having $\text{cin} = 0$ and the other $\text{cin} = 1$. Through this procedure, we are able to speed up the calculation of selecting which sum depending on our cin .

The ripple adder works by using a cascade of several full adders connected in series with each other. Each full adder is responsible for their adding their corresponding bits from both inputs and outputs their carryout to the carryin of the next full adder until both inputs have been fully added together. The ripple adder, and by extension the carry select adder is simple to implement and requires minimal logic gates to implement, making it inexpensive and space-efficient compared to other methods of addition. However, there is a delay due to the carry propagation which limits the ripple adder (and therefore the carry-select adder) in its effective speed with larger bitwidth inputs. However, for this application (8-bits), this adder is very efficient in both space and speed.

This project uses '<https://github.com/FCHXWH823/Verilog-Adders\XeTeXglyph\numexpr\XeTeXcharglyph>'0027\relax{} as reference.

How to test

We tested all the combinations. This means two 8 bits input sum to a 8 bit output, and we ignore the carry out bit.

Therefore, we expect both the input and the output to be in the range of 0 to 255.

External hardware

We did not use any external hardware.

Pinout

#	Input	Output	Bidirectional
0	a[0]	sum[0]	b[0]
1	a1	sum1	b1
2	a2	sum2	b2
3	a[3]	sum[3]	b[3]
4	a[4]	sum[4]	b[4]
5	a[5]	sum[5]	b[5]
6	a[6]	sum[6]	b[6]
7	a[7]	sum[7]	b[7]

Asynchronous I2C Registerfile Interface [491]

- Author: DTU
- Description: async. i2c if to regfile
- GitHub repository
- HDL project
- Mux address: 491
- Extra docs
- Clock: 0 Hz

How it works

Async I2C to register file, shows up as an I2C interface. To be used to configure analog blocks.

How to test

Address as I2C peripheral, write and read words.

External hardware

LED

Pinout

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

test_friday2 [492]

- Author: Niles Peter
- Description: class
- GitHub repository
- HDL project
- Mux address: 492
- Extra docs
- Clock: 0 Hz

8-bit KoggeStone Adder

Author: Niles Villaverde, Joshua Cho Language: Verilog

How it works

The KoggeStone Adder computes in parallel, first the sum from the two different inputs and then computes the carry-out for each bit. Then uses the calculated carry-out and sum of each bit to compute the final result of the adder. *Note: No carry-out so values greater than 255 can not be outputted*

In the project.v file, there are 5 different modules: BigCircle, SmallCircle, Square, Triangle, and tt_um_koggestone_adder8.

Shown in figure 1 below is the block diagram for the flow for the KoggeStone Adder

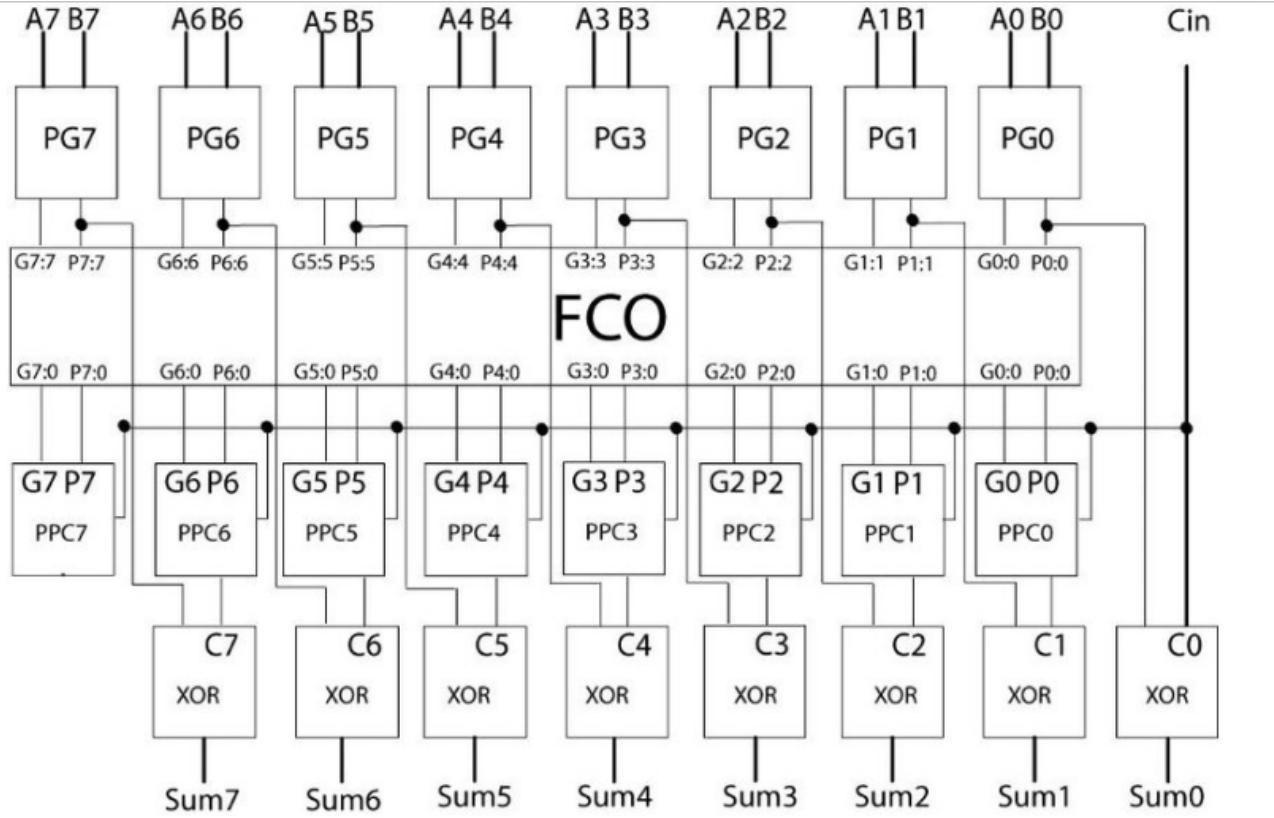


Figure 1: KoggeStone Adder Block Diagram

BigCircle Module The BigCircle module represents the carry generator for the KoggeStone Adder. It calculates the generated and propagated signal in each bit stage in the Adder. In comparison to carry-ripple adders, the KoggeStone adder allows for the carry information to propagate efficiently to multiple bit positions. This allows for the number of sequential steps in calculating the final carry-out to be reduced.

The BigCircle takes in the generate and propagate signals from the current position in the adder and the previous position in the adder. Using these signals, BigCircle updates the generate signal for the bit position to reflect if the carry is generated from this bit position or propagated from the previous. Then calculates the propagation signal to decide whether if a carry can be passed through this position.

SmallCircle Module The SmallCircle module passes the carry in signal and generated carry signal to the next position

Square Module The Square module calculates the current generate and propagate signal by ANDing the inputs A and B as well as XORing the inputs A and B respectively.

Triangle Module The Triangle module calculates the sum bit by XORing the propagate bit with the previous carry-in bit.

tt_um_koggstone_adder8 Module The tt_um_koggstone_adder8 module takes in two 8-bit inputs, ui_in and uio_in. The module also outputs an 8-bit output, uo_out. **Input Signals:** Two 8-bit, a and b which are mapped to ui_in and uio_in, respectively. Cin, carry-in for the addition which is set to zero. g and p, generate and propagation signal for each bit. c, carries for each bit position.

The first sequence is to use the Square Module to create the initial generate and propagation calculations. Then uses the BigCircle Module to calculate the intermediate generate and propagation signals of each bit. In the second stage of the BigCircle Module, by combining the signals over groups of 4 bits, it further propagates the carry. In the third stage of the BigCircle Module, it continues the carry propagation over an even wider spans of bits. Then using the SmallCircle Module, the final Carry-Out signals for each position are calculated. Then the final sum is calculated using the Triangle Modules.

How to test

The two different inputs, ui_in[7:0] and uio_in[7:0] are iterated through each possible combination of 8-bit numbers to test all corner cases. The outputs are set to the calculated values calculated by the KoggeStone Adder. If the sum between the two values are greater than 255, the test is skipped as limitations on the hardware prevent us from having a carry-out value.

External hardware

no external hardware

Pinout

#	Input	Output	Bidirectional
0	a[0]	sum[0]	b[0]
1	a1	sum1	b1
2	a2	sum2	b2
3	a[3]	sum[3]	b[3]
4	a[4]	sum[4]	b[4]
5	a[5]	sum[5]	b[5]

#	Input	Output	Bidirectional
6	a[6]	sum[6]	b[6]
7	a[7]	sum[7]	b[7]

Tappu [493]

- Author: Tobias Jensen
- Description: A simple 8 bit CPU inspired by the esoteric language BF
- GitHub repository
- HDL project
- Mux address: 493
- Extra docs
- Clock: 20 Hz

How it works

This project implements the Tappu CPU with a ROM memory that turns blinks all outputs

How to test

Watch the output blink :)

External hardware

None

Pinout

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

Perceptron [494]

- Author: Mimi Rapoport
- Description: Simulates a perceptron
- GitHub repository
- HDL project
- Mux address: 494
- Extra docs
- Clock: 0 Hz

How it works

The perceptron takes in three inputs, multiplies them by weights and then sums the products. It then weighs the sum against a threshold to decide whether to output 1 or 0. The perceptron also takes in a desired output and performs a weight update when the desired output and actual output don't match. .

How to test

Make sure that the clock and reset are working.

External hardware

None

Pinout

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

mp_LIF_neuron [495]

- Author: Andreas Schorer
- Description: Mixed precision Leaky integrate and fire (LIF) neuron
- GitHub repository
- HDL project
- Mux address: 495
- Extra docs
- Clock: 0 Hz

How it works

This project implements a mixed precision leaky integarte and fire (LIF) neuron. It either computes one neuron at a time with 24bit membrane value and 8 bit weights or two neurons with 12bit membrane values and 4 bit weights.

How to test

Apply the weight at the input and observe spikes. <- Will be updated later

External hardware

none

Pinout

#	Input	Output	Bidirectional
0	w0	spk_upper	ready
1	w1	spk_lower	
2	w2		
3	w3		
4			
5			
6			
7			

Hopfield Network with Izhikevich-type RS and FS Neurons [496]

- Author: Daniel Solis
- Description: An on-chip implementation of a Hopfield neural network using Izhikevich-type regular spiking (RS) and fast spiking (FS) neurons with on-chip Hebbian learning for pattern storage and retrieval.
- GitHub repository
- HDL project
- Mux address: 496
- Extra docs
- Clock: 16000000 Hz

How it works

It is a leaky Integrated Neuron

How to test

Just Test

External hardware

No

Pinout

#	Input	Output	Bidirectional
0	learning_enable		spikes[0]
1	pattern_input[0]		spikes1
2	pattern_input1		spikes2
3	pattern_input2		spikes[3]
4	pattern_input[3]		spikes[4]
5			spikes[5]
6			spikes[6]
7			

digital LIF Neuron [497]

- Author: Kosmas Wernhard
- Description: digital LIF Neuron
- GitHub repository
- HDL project
- Mux address: 497
- Extra docs
- Clock: 0 Hz

How it works

Digital LIF Neuron, Reset by subtraction, Set Leakage and Threshold

How to test

Input Weight and Spike, See if it spikes

External hardware

No external Hardware needed

Pinout

#	Input	Output	Bidirectional
0	Weight0	Spike_OUT	
1	Weight1		
2	Weight2		
3	Weight3		
4			
5			
6			
7	Spike_IN		

Tinysynth [498]

- Author: Erling Rennemo Jellum
- Description: A tiny square wave oscillator accepting MIDI commands.
- GitHub repository
- HDL project
- Mux address: 498
- Extra docs
- Clock: 50000000 Hz

How it works

Accepts MIDI commands over UART, generates a corresponding square wave signal using PWM.

How to test

External hardware

A Pmod AMP2 connected to the PMOD connector.

Pinout

#	Input	Output	Bidirectional
0	a	x0	b0
1	b	x1	b1
2	c	x2	b2
3	d	x3	b3
4	e	x4	b4
5	f	x5	b5
6	g	x6	b6
7	h	x7	b7

Hero on Tape [499]

- Author: Marcus Sand
- Description: Echoes out a predefined text onto a 16x2 character LCD.
- GitHub repository
- HDL project
- Mux address: 499
- Extra docs
- Clock: 0 Hz

How it works

Echoes out a predefined text onto a 16x2 character LCD.

How to test

Connect up a character LCD according to the pinout, and set the clock.

External hardware

A 16x2 character LCD (LCD1602)

Pinout

#	Input	Output	Bidirectional
0	CLK	RS	
1		E	
2		D4	
3		D5	
4		D6	
5		D7	
6			
7			

16 Bit Izhikevich Neuron [512]

- Author: Noah Williams
- Description: Izhikevich neuron model with 16 bit arithmetic.
- GitHub repository
- HDL project
- Mux address: 512
- Extra docs
- Clock: 0 Hz

How it works

Izhikevich model

The Izhikevich model is a simple spiking neuron model that builds on the dynamics of the simplistic leaky integrate-and-fire model, adding complexity of the Hodgkin-Huxley model with minimal computational cost.

The model is described by the following system:

```
v' = 0.04*v^2 + 5*v + 140 - u + I  
u' = a*(b*v - u)  
if v >= 30 then {v = c; u = u + d}
```

where:

a, b, c, d = dimensionless constants

Regular Spiking (RS) Excitatory Neuron:
a = 0.02, b = 0.2, c = -65, d = 8

v = membrane potential
u = membrane recovery (Na and K, neg feedback to v)
a = time scale of the recovery variable u (small = slow recovery)
b = sensitivity of the recovery variable u to v
Larger values increase sensitivity and lead to more spiking behavior. b<a(b>a) is saddle-node
c = after spike reset value of v
caused by fast K⁺ channels
d = after spike reset value of u

caused by slow Na⁺ & K⁺ channels
I = input current

The constants for the model differential equation v' are experimental determined by fitting the model to the desired neuron behavior. In the original paper (from which the equations are taken), the model was fit to experimental data from Regular Spiking of a rat corticospinal neuron.

References:

<https://www.izhikevich.org/publications/spikes.pdf>

How to test

To test the model, use the supplied test-bench. The test-bench will run three different scenarios. The first case is the reset test case, which ensures that the model resets properly given a reset condition (res_n). The next test case checks to make sure that the model doesn't spike when the input current is below threshold. The spike value for each of the included non-spike test cases should be 0. The final test case is the test that ensures the model spikes when the input is above the threshold. This includes a test for the maximum current to test overflow conditions. This is checked with an assert statement.

External hardware

N/A at the moment :)

Pinout

#	Input	Output	Bidirectional
0	Input current bit [0]	State variable bit [0]	
1	Input current bit 1	State variable bit 1	
2	Input current bit 2	State variable bit 2	
3	Input current bit [3]	State variable bit [3]	
4	Input current bit [4]	State variable bit [4]	
5	Input current bit [5]	State variable bit [5]	
6	Input current bit [6]	State variable bit [6]	

#	Input	Output	Bidirectional
7	Input current bit [7]	State variable bit [7]	Spike bit

dff_mem [514]

- Author: dmrudait
- Description: 16 byte RAM built out of DFFs
- GitHub repository
- HDL project
- Mux address: 514
- Extra docs
- Clock: 0 Hz

How it works

This project implements a 16 Byte memory module (it consists of 16 memory locations that store 1 byte each). The memory allows for both read and write operations, controlled by input signals. The module requires a 4-bit address input, control signals lr_n and ce_n, a clock, 8-bit data input (for writes), and a reset signal.

Signals

- ui_in[7:0]: Dedicated input line for all control signals
- ce_n (Active Low): Chip enable signal for reading data.
- lr_n (Active Load): Load/write signal, enabling writing to memory.
- uio_in[7:0]: Bidirectional IO line for input (used as the input line for data)
- uio_out[7:0]: Bidirectional IO line for output.
- uio_oe (Active High): Used to set the bidirectional IO line to an input to be able to input data
- ena (Active High): Tiny Tapeout signal for enabling the module
- clk: global clock. Operations happen on the positive edge
- rst_n (Active low): Resets all contents in RAM to NULL.
- uo_out[7:0]: Dedicated output line (outputs ram contents when ce_n is low (active)).

Addressing: The memory is 4-bit addressable, where the address specifies which register (out of 16) is being accessed for reading or writing.

Write operation: A byte of data is written to specific register in RAM, where the location is determined by the address. Requires write enable lr_n signal as active (low) and the clock edge to occur.

Read operation: Data can be read from a specific register in RAM determined by the input address. Requires chip enable ce_n signal as active (low). The data is output on the uo_out ports, and it is updated asynchronously (independant of the clock edge).

Output: Data is presented on the `uo_out` line when the chip is enabled for reading, and high-impedance (Z) otherwise.

How to test

To test, set the address and corresponding inputs to desired values. Clear `lr_n` for a write operation and `ce_n` for a read operation. Then pulse the clock to run signals.

The CocoTB testbenches located in the `test.py` file, test various scenarios for the module. First, it tests a write operation to each address in the module followed by a read operation at each address, to ensure correct behaviour. The script then sets `ui_in`, `lr_n` high and clears `ce_n` to setup for a Read with RAM output enabled. It then iterates over and reads from each address, comparing the received value (`uo_out`), to the expected byte from that address. If there are any mismatches, an assertion error is raised, specifying the faulty address and value.

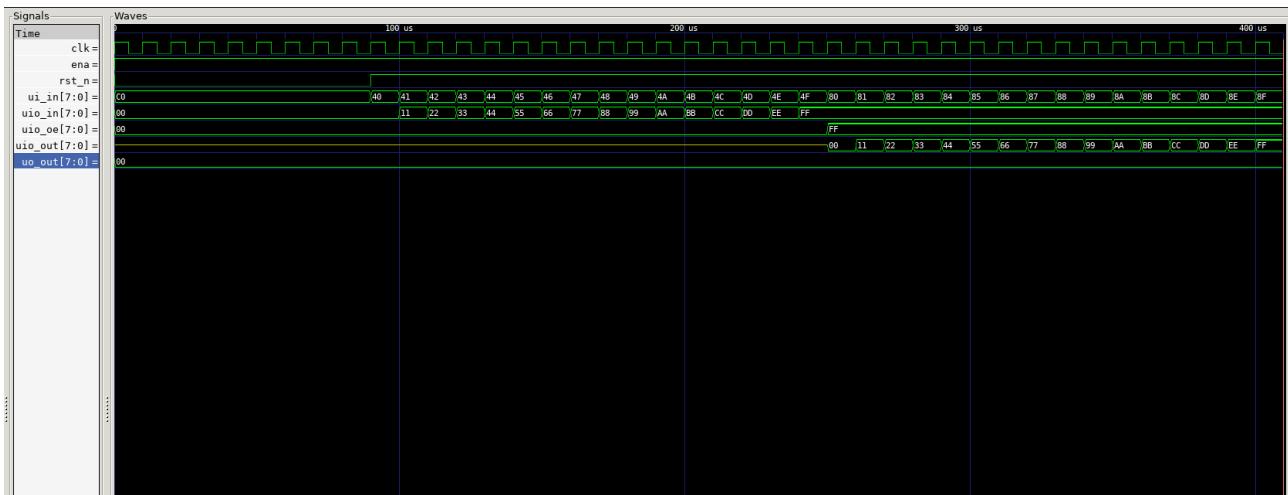


Figure 1: Gate level Test

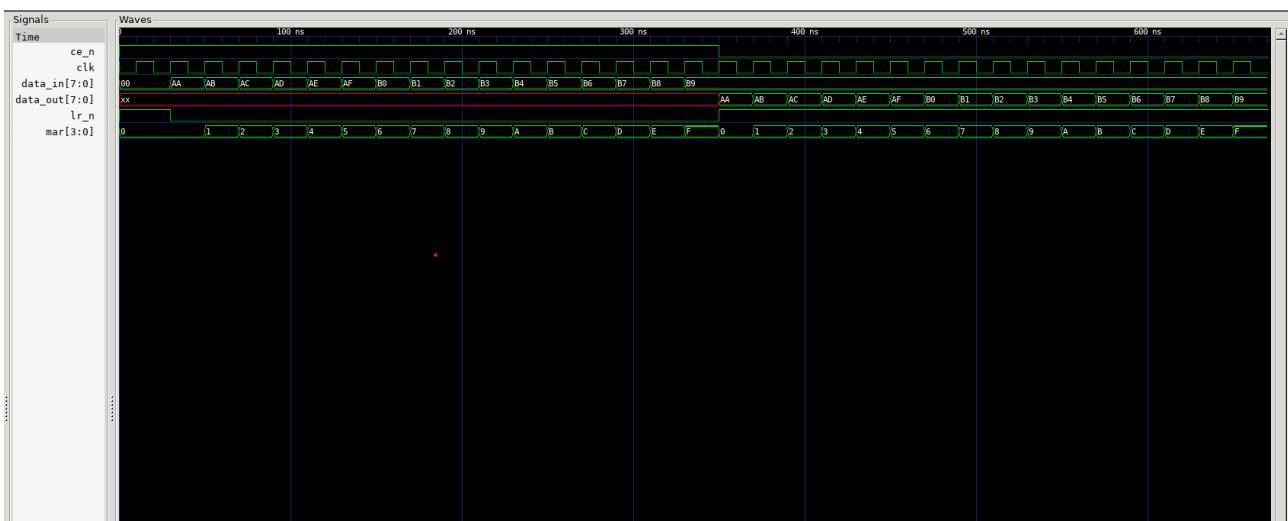


Figure 2: Ideal Test

External hardware

This RAM module is intended to be integrated into an 8-bit processor. However, it is being submitted to TT as an individual tile for testing. An external MAR would thus be required to program RAM and subsequently read memory. The MAR would act as a programmer according to the above described specifications.

Pinout

#	Input	Output	Bidirectional
0	addr[0]	out[0]	in[0]
1	addr1	out1	in1
2	addr2	out2	in2
3	addr[3]	out[3]	in[3]
4	addr[4]	out[4]	in[4]
5		out[5]	in[5]
6	lr_n	out[6]	in[6]
7	ce_n	out[7]	in[7]

Verilog ring oscillator V2 [516]

- Author: algofoogle (Anton Maurovic)
- Description: Multiple simple ring oscillators by instantiating sky130 inv_2 inverter rings
- GitHub repository
- HDL project
- Mux address: 516
- Extra docs
- Clock: 0 Hz

What is this?

Everyone has done a ring oscillator using inverter cells. Now it's my turn!

I already submitted tt09-ring-osc on TT09 and rather than muck that up with extra stuff I decided to submit this alternate version which features:

- 4 simple independent rings instead of 1, hoping to run at different speeds:
 - ring_125: 125 inverters, *maybe* 112MHz out? Could be too fast for IO.
 - ring_251: 251 inverters, hopefully good for ~56MHz.
 - ring_501: 501 inverters, ~28MHz.
 - ring_1001: 1001 inverters, ~14MHz.
- Some other PWM experiments on faster ring oscillators.

Approximate frequencies are estimated on the assumption that each inverter introduces a delay of ~70ps.

These use Verilog to instantiate the rings of (an odd number of) sky130_fd_sc_hd__inv_2 cells – **UPDATE**: Actually, since this is targeting IHP instead, there is a polyfill that somebody else wrote to map sky130 cells to generic cells (that OpenLane will then map to IHP cells).

Pinout

#	Input	Output	Bidirectional
0	pwm2_in[0]	ring_125	dummy
1	pwm2_in1	ring_251	pwm3a_out
2	pwm3_in[0]	ring_501	
3	pwm3_in1	ring_1001	

#	Input	Output	Bidirectional
4		c0_3	
5	pwm3a_in[0]	c1_3	
6	pwm3a_in1	c2_5	pwm2_out
7	pwm3a_in2	c3_5	pwm3_out

Basic model for Systolic array implementation of LIF [518]

- Author: Sulaiman Islam
- Description: A model for systolic array implementation of LIF neurons. Hazard cases have been taken into account such as overstimulation of LIF neurons with bypass cases.
- GitHub repository
- HDL project
- Mux address: 518
- Extra docs
- Clock: 0 Hz

How it works

The model represents how a generative implementation of SNN training can be implemented in verilog. Hazards that were predicted were that of LIF overstimulation due to excessive accumulation of the multiply accumulate MAC operations. In order to prevent this I implemented bypass conditions that made regulated the LIF inputs to choose between 0 and the output of the MAC operations. The LIF would only take in the value of the MAC operation when the threshold for firing was reached by the output of the MAC. One clock cycle later the MAC Accumulation would be reset. This was a weight stationary implementation that had fixed constant weights on each of the MACS. Several Blocks of these weight stationary MACS could be implemented with their respective LIFS in theory, however due to size restrictions there is only one small block in the Top module. The user input ui_in is used to drive the MAC inputs. The uo_out individual bits were used to drive the spike to indicate that the bypass had occurred and that the LIF had spiked.

How to test

Using ui_in to vary X and checking uo_out for expected behavior based on the MAC operations.

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	input current bit [0]	State variable bit [0]	
1	input current bit 1	State variable bit 1	
2	input current bit 2	State variable bit 2	
3	input current bit [3]	State variable bit [3]	
4	input current bit [4]	State variable bit [4]	
5	input current bit [5]	State variable bit [5]	
6	input current bit [6]	State variable bit [6]	
7	input current bit [7]	State variable bit [7]	spike bit

Leaky integrate and fire spiking neural network [520]

- Author: Aliyaa Islam
- Description: simulates a lif neuron
- GitHub repository
- HDL project
- Mux address: 520
- Extra docs
- Clock: 0 Hz

How it works

It takes input voltages and treats that as the input injection to the LIF neuron

How to test

Do something

External hardware

NA

Pinout

#	Input	Output	Bidirectional
0	Input current bit [0]	State varibale bit [0]	
1	Input current bit 1	State varibale bit 1	
2	Input current bit 2	State varibale bit 2	
3	Input current bit [3]	State varibale bit [3]	
4	Input current bit [4]	State varibale bit [4]	
5	Input current bit [5]	State varibale bit [5]	
6	Input current bit [6]	State varibale bit [6]	
7	Input current bit [7]	State varibale bit [7]	Spike bit

tinydsp-lol [522]

- Author: Tassilo Tanneberger
- Description: testing digital dsp
- GitHub repository
- HDL project
- Mux address: 522
- Extra docs
- Clock: 2000000 Hz

How it works

This is just a test project to explore Chisel.

How to test

It should have a ChiselTest to run with sbt test.

External hardware

Nothing at the moment.

Pinout

#	Input	Output	Bidirectional
0	a	x0	b0
1	b	x1	b1
2	c	x2	b2
3	d	x3	b3
4	e	x4	b4
5	f	x5	b5
6	g	x6	b6
7	h	x7	b7

Shifter [524]

- Author: Ethan Sifferman
- Description: Input » Inout
- GitHub repository
- HDL project
- Mux address: 524
- Extra docs
- Clock: 0 Hz

How it works

$\text{Output} = \text{Input}[7:0] \gg \text{Inout}[7:0]$

How to test

The LEDs will output the shifted value of $\text{Output} = \text{Input}[7:0] \gg \text{Inout}[7:0]$.

External hardware

Switches on the inputs, LEDs on the outputs

Pinout

#	Input	Output	Bidirectional
0	ui_in[0]	uo_out[0]	ui_out[0]
1	ui_in1	uo_out1	ui_out1
2	ui_in2	uo_out2	ui_out2
3	ui_in[3]	uo_out[3]	ui_out[3]
4	ui_in[4]	uo_out[4]	ui_out[4]
5	ui_in[5]	uo_out[5]	ui_out[5]
6	ui_in[6]	uo_out[6]	ui_out[6]
7	ui_in[7]	uo_out[7]	ui_out[7]

LRC - Longitudinal Redundancy Check generator [526]

- Author: Steve Jenson <stevej@gmail.com>
- Description: LRC implementation for Tiny Tapeout 09
- GitHub repository
- HDL project
- Mux address: 526
- Extra docs
- Clock: 0 Hz

How it works

Calculates a running error correcting code. For each new byte applied to the input pins, calculates a running longitudinal redundancy code.

How to test

Supply a byte to ui_in, read the LRC on uo_out. Keep feeding it bytes and you'll keep getting new LRC codes. Code resets when the chip resets.

External hardware

No external hardware needed.

Pinout

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

Workshop demo [528]

- Author: Tommy Thorn
- Description: Just a demo
- GitHub repository
- HDL project
- Mux address: 528
- Extra docs
- Clock: 50000000 Hz

How it works

It's magic

How to test

Connect the TX pin to your favorite terminal (more to be written)

External hardware

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

Pinout

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

A Tale of Two NCOs [530]

- Author: Mike Ng
- Description: Two NCOs enter, one signal leaves
- GitHub repository
- HDL project
- Mux address: 530
- Extra docs
- Clock: 50000000 Hz

How it works

This design contains two NCOs, implemented with phase accumulators and sine lookup tables. The outputs of the NCOs are multiplied together by default. NCO B can be bypassed to a constant “one” or “half”. There is also a boxcar filter for funsies.

When operating at 50 MHz, it should be possible to tune NCO A from 24.8 MHz to 0.195 MHz. NCO B has one less bit in its increment control, so it can only go up to 12.3 MHz.

How to test

The output is intended for something simple like an R-2R DAC. Don’t expect it to be pretty at high frequency.

External hardware

- DAC
- Oscilloscope

Pinout

#	Input	Output	Bidirectional
0	phase_incr_A[0]	OUT0	phase_incr_B[0]
1	phase_incr_A1	OUT1	phase_incr_B1
2	phase_incr_A2	OUT2	phase_incr_B2
3	phase_incr_A[3]	OUT3	phase_incr_B[3]
4	phase_incr_A[4]	OUT4	phase_incr_B[4]
5	phase_incr_A[5]	OUT5	phase_incr_B[5]

#	Input	Output	Bidirectional
6	phase_incr_A[6]	OUT6	low_amplitude_B
7	filter_on	OUT7	bypass_B

Wokwi Group #7 [544]

- Author: Tiny Tapeout
- Description: Combined project to save space for ttihp25a
- GitHub repository
- HDL project
- Mux address: 544
- Extra docs
- Clock: 0 Hz

- SimplePattern by Poorn
- Drew's First Wokwi Design by ReanimationXP
- Not Good BCD Decoder by Erik Shimizu
- JCB First WOKWI Design by Jared Bruce
- Jacks First Project by Jack B
- APTT by Andy
- Trubick - Tiny Tapeout Logic Gate by Zane Trubick
- Metastable Chip by Patrick McDermott
- Yared Fente's Tiny Tapeout by Yared Fente
- project by ahmad
- TT-Farhad by Farhad
- Ripple counter by Marc Mignard
- chip by Olivia
- Full adder Design by Mithun
- sarah's first chip by sarah

Pinout

#	Input	Output	Bidirectional
0	in0	out0	sel0
1	in1	out1	sel1
2	in2	out2	sel2
3	in3	out3	sel3
4	in4	out4	
5	in5	out5	
6	in6	out6	
7	in7	out7	

Wokwi Group #6 [546]

- Author: Tiny Tapeout
- Description: Combined project to save space for ttihp25a
- GitHub repository
- HDL project
- Mux address: 546
- Extra docs
- Clock: 0 Hz
- Input Counter by Benjamin Meyer
- Bad Logic by AaronV
- MuxLED by Alex Moore
- TINY TAPE OUT by Slaiman
- BadeTP by Brandon D
- YoshiTP by Yeshua M (Yoshi)
- Light LED by Baruas
- TinyTapeout1 by Matthew H
- four flip flops by Arjun Vedantham
- Tiniest of tapeouts by J Money
- 3bitFullAdder by Isabella Phung
- 4 bit adder by Angel Lim Hui Yi
- Mini-Adder and Clock Divider by Marcus
- rhTinyTapeout by Raphael Huang
- Tiny_Tapeout_Adder! by Abhinav Chaubey

Pinout

#	Input	Output	Bidirectional
0	in0	out0	sel0
1	in1	out1	sel1
2	in2	out2	sel2
3	in3	out3	sel3
4	in4	out4	
5	in5	out5	
6	in6	out6	
7	in7	out7	

Wokwi Group #5 [548]

- Author: Tiny Tapeout
- Description: Combined project to save space for ttihp25a
- GitHub repository
- HDL project
- Mux address: 548
- Extra docs
- Clock: 0 Hz

- RAYS FIRST TAPEOUT rev 2 by RAY STITS
- joes-first-tiny-tapeout by securelyfitz
- Speller by Aaron Eiche
- OR gate by Joe Merriam
- 1st by HUSSAIN
- Kevin Project by Kevin
- AndLogicPass by James Nguyen
- ovl abc chip by oliver lazaras
- D_flipflop_hold_test by Nicole Ramirez
- one by Neil
- Odd or even by Eliana
- add it by alex b
- Tian TT9 by Tianxin Wu
- 2_bit_7seg by Nathaniel_Laurente
- 2-bit Full Adder by Shreya

Pinout

#	Input	Output	Bidirectional
0	in0	out0	sel0
1	in1	out1	sel1
2	in2	out2	sel2
3	in3	out3	sel3
4	in4	out4	
5	in5	out5	
6	in6	out6	
7	in7	out7	

Wokwi Group #4 [550]

- Author: Tiny Tapeout
- Description: Combined project to save space for ttihp25a
- GitHub repository
- HDL project
- Mux address: 550
- Extra docs
- Clock: 0 Hz

- Shadoff Test by David Shadoff
- 6 Bit shift register by MOMO
- FB GDS by Fahad Bastaki
- First Tapeout Chip - OCR by Owen Robertson
- Andrew Vo - Repository by Andrew Vo
- Tiny Tapeout-Huerta by Fernando Huerta
- 2 Bit Times 2 Bit Plus 4 Bit MAD and 5 Bit Binary to 7 Segment Display by Nathan
- Encoder by Mohammad Almutair
- Steven's Wokwi Test by Steven Abrego
- Four Bit Adder by Anahit
- fulladder by Keoni Gandall
- 2-Bit-Adder by Jamin
- half adder by Adam Wu
- dummy by Naveen
- 2 bit adder by Aadarsha Kandel
- NAND Flip-Flop by Luigi C.

Pinout

#	Input	Output	Bidirectional
0	in0	out0	sel0
1	in1	out1	sel1
2	in2	out2	sel2
3	in3	out3	sel3
4	in4	out4	
5	in5	out5	
6	in6	out6	
7	in7	out7	

Wokwi Group #3 [552]

- Author: Tiny Tapeout
- Description: Combined project to save space for ttihp25a
- GitHub repository
- HDL project
- Mux address: 552
- Extra docs
- Clock: 0 Hz

- achasen workshop validation by adam chasen
- Secret Code by Rex
- And Gates that don't do much by Chris Collins
- comparator by prtx
- Clocked Display by Dooseok
- Encoder by Peilin
- Secret Initial by Kiarash
- My First ASIC by Michael A. Enright
- Hamad's design by Hamad Alwaqayan
- xor gate with registered output by claudiotalarico
- Tiny Tapeout 9 by Maya Choudhury
- Tiny Tapeout 9 Template Version 1 Tata Luka by lukab
- adder-tt09 by Philip Solomatnikov
- Sigma-Delta ADC by Martin Schoeberl
- TinySnake by Ken Pettit
- Broken Two Bit Adder by Mann

Pinout

#	Input	Output	Bidirectional
0	in0	out0	sel0
1	in1	out1	sel1
2	in2	out2	sel2
3	in3	out3	sel3
4	in4	out4	
5	in5	out5	
6	in6	out6	
7	in7	out7	

Wokwi Group #2 [554]

- Author: Tiny Tapeout
- Description: Combined project to save space for ttihp25a
- GitHub repository
- HDL project
- Mux address: 554
- Extra docs
- Clock: 0 Hz
- S-R latch by Albert
- Tiny Tapeout Take 2 by Stephanie Rosales
- 2bit adder by Ya-Chin, Hu
- Half adder by Keyshon Howard
- Full bit adder by Alan
- Light by Natnael Atnafu
- Half Adder by Janani P Srinivasan
- chip_fab by Aleksi
- LCA's first Wokwi design by leahcorbett18
- my First WokWi Design by Mani Rayabarapu
- 7-bit arbiter by Kira Tran
- tt09-4bit-adder-dhags by Danny
- rand by mahi
- gatesoup by Elio Bourcart
- UART TX by Shaokai Lin
- Logic Gates 7-Segment Display by Abdul Karim Tamim

Pinout

#	Input	Output	Bidirectional
0	in0	out0	sel0
1	in1	out1	sel1
2	in2	out2	sel2
3	in3	out3	sel3
4	in4	out4	
5	in5	out5	
6	in6	out6	
7	in7	out7	

Wokwi Group #1 [556]

- Author: Tiny Tapeout
- Description: Combined project to save space for ttihp25a
- GitHub repository
- HDL project
- Mux address: 556
- Extra docs
- Clock: 0 Hz

- Abey's 1st Chip Design by Abey Varghese
- 4-1 mux by zhengfeng wu
- Jordan by Jordan Medina
- Bit Counter by Philip Measor
- GJAA Design by Guadalupe de Jesus Avelar Anguiano
- TinyTapeOut by Siyem Russom
- My First TinyTapeout by Case Kirk
- GDS by Ben
- Vincent's First Design by Vincent Harkins
- NAND-Equ by DanT
- Full Adder by Harish Prabhakaran
- Counter by Alex Solomatnikov
- 7-seg display checker by Ryan Taylor
- 2 input multiplexor by chad
- XorTree by Ammar Ratnani
- gta6 by henry

Pinout

#	Input	Output	Bidirectional
0	in0	out0	sel0
1	in1	out1	sel1
2	in2	out2	sel2
3	in3	out3	sel3
4	in4	out4	
5	in5	out5	
6	in6	out6	
7	in7	out7	

Will It NAND? [558]

- Author: Daniel Samarin
- Description: A bunch of nand gates to test the tool chain... for now.
- GitHub repository
- Wokwi project
- Mux address: 558
- Extra docs
- Clock: 0 Hz

How it works

Yo, it's just a bunch of NANDs.

How to test

Be a man, use your hand to connect up your NAND.

External hardware

Put it in the sand, like it's silicon, because you're a silly con.

Pinout

#	Input	Output	Bidirectional
0	NAND1a	NAND1out	
1	NAND1b	NAND2out	
2	NAND2a	NAND3out	
3	NAND2b	NAND4out	
4	NAND3a		
5	NAND3b		
6	NAND4a		
7	NAND4b		

sphereinabox hello [560]

- Author: Nick Winters
- Description: Hello World
- GitHub repository
- Wokwi project
- Mux address: 560
- Extra docs
- Clock: 0 Hz

How it works

I've built 8-input logic gates, all using each of the 8 inputs.... or will eventually

How to test

Set the inputs 0..7 to your desired 8 inputs.

Observe the outputs the corresponding output pins.

External hardware

No specific external hardware is expected for this project.

Pinout

#	Input	Output	Bidirectional
0	IN0	OUT0	
1	IN1	OUT1	
2	IN2	OUT2	
3	IN3	OUT3	
4	IN4	OUT4	
5	IN5	OUT5	
6	IN6	OUT6	
7	IN7	OUT7	

L display [562]

- Author: Matt Lamparter
- Description: Displays L character on a 7 seg display when 00101111 entered on the input (and pressing the Step button to enable display)
- GitHub repository
- Wokwi project
- Mux address: 562
- Extra docs
- Clock: 0 Hz

How it works

Enter the right combination of bits to display an “L” on the seven segment display. The combination is 0b00101111. Once that combination is entered you’ll need to press the “Step” button in order to display the L on the display. Releasing the Step button will clear the display.

How to test

Try different combinations of inputs. The only time the output should be displayed is when the right bit combination is entered and the Step button is pressed.

External hardware

Requires a 7 segment display, a push button connected to power, and a 8 bit wide DIP switch.

Pinout

#	Input	Output	Bidirectional
0	IN0		
1	IN1		
2	IN2		
3	IN3	OUT3	
4	IN4	OUT4	
5	IN5	OUT5	
6	IN6		

#	Input	Output	Bidirectional
7	IN7		

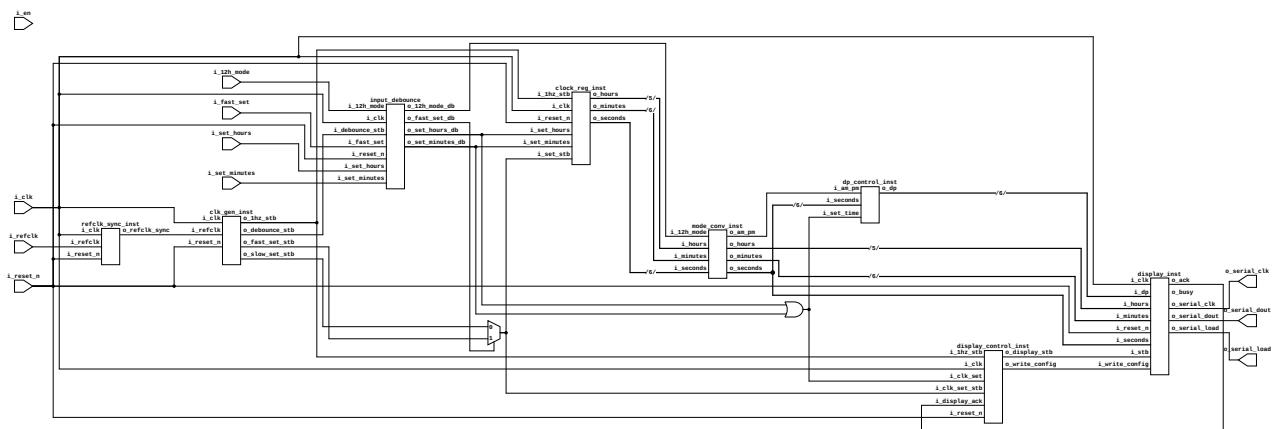
7-Segment Digital Desk Clock [576]

- Author: Samuel Ellicott
 - Description: 7-Segment Desk Clock
 - GitHub repository
 - HDL project
 - Mux address: 576
 - Extra docs
 - Clock: 50000000 Hz

How it works

Simple digital clock, displays hours, minutes, and seconds in either a 24h format. Since there are not enough output pins to directly drive a 6x 7-segment displays, the data is shifted out over SPI to a MAX7219 in 7-segment mode. The time can be set using the `hours_set` and `minutes_set` inputs. If `set_fast` is high, then the the hours or minutes will be incremented at a rate of 5Hz, otherwise it will be set at a rate of 2Hz. Note that when setting either the minutes, rolling-over will not affect the hours setting. If both `hours_set` and `minutes_set` are presssed at the same time the seconds will be cleared to zero.

A block diagram of the system is shown below.



How to test

Apply a 5MHz clock to the clock pin and 32.786Khz signal to the refclk pin. Use the hours_set and minutes_set pins to set the time.

External hardware

Connect the BIDIR PMOD to a MAX7219 7-segment display, For reference Tiny Tapeout SPI

Pinout

#	Input	Output	Bidirectional
0	refclk		Display CS
1			Display MOSI
2	Fast/Slow Set		
3	Set Hours		Display SCK
4	Set Minutes		
5	12-Hour Mode		
6			
7			

Basic Perceptron + ReLU [578]

- Author: UDXS
- Description: Basic Perceptron + ReLU Layer
- GitHub repository
- HDL project
- Mux address: 578
- Extra docs
- Clock: 0 Hz

How it works

It connects a small single-cycle multiply-accumulation unit to a ReLU output.

How to test

Reset and then, for every following cycle, provide pairs of signed 4-bit numbers representing the weight-input pair for a given model layer invocation. The output will change cycle-to-cycle. Sample it while providing your last inputs and then reset to attempt another invocation.

Pinout

#	Input	Output	Bidirectional
0	Weight[0]	ReLU[0]	ReLU[8]
1	Weight1	ReLU1	ReLU[9]
2	Weight2	ReLU2	ReLU[10]
3	Weight[3]	ReLU[3]	ReLU[11]
4	Input[0]	ReLU[4]	ReLU[12]
5	Input1	ReLU[5]	ReLU[13]
6	Input2	ReLU[6]	ReLU[14]
7	Input[3]	ReLU[7]	ReLU[15]

Basic Matrix-Vector Multiplication [580]

- Author: Andy Ly
- Description: Basic matrix and vector multiplier that multiplies a 2x2 matrix with a 2x1 vector. Inputs are limited to 2 bit elements
- GitHub repository
- HDL project
- Mux address: 580
- Extra docs
- Clock: 0 Hz

How it works

Take input voltages and treats them as input current injection to lif neuron

How to test

Test it

External hardware

Possibly

Pinout

#	Input	Output	Bidirectional
0	Input bit [0] for matrix element 11	Output bit [0] for output vector element 1	Output bit [0]
1	Input bit 1 for matrix element 11	Output bit 1 for output vector element 1	Output bit [1]
2	Input bit [0] for matrix element 12	Output bit 2 for output vector element 1	
3	Input bit 1 for matrix element 12	Output bit [3] for output vector element 1	
4	Input bit [0] for matrix element 21	Output bit [4] for output vector element 1	Input bit [0]
5	Input bit 1 for matrix element 21	Output bit [0] for output vector element 2	Input bit [1]
6	Input bit [0] for matrix element 22	Output bit 1 for output vector element 2	Input bit [2]
7	Input bit 1 for matrix element 22	Output bit 2 for output vector element 2	Input bit [3]

8 bit MAC Unit [582]

- Author: Devesh Bhaskaran
- Description: Implementation Of 8-bit MAC Using Vedic Multipliers And Reversible Gates
- GitHub repository
- HDL project
- Mux address: 582
- Extra docs
- Clock: 40000000 Hz

How it works

The project aims to implement a 8-bit MAC unit for unsigned integer data type using Vedic Multipliers and Reversible gates. The two inputs are to be taken in through input pins and bi-directional pins using half a clock cycle and stored in registers. The MAC operation is performed on the values stored in these registers. The multiplier and adder takes half clock cycle each. The result of the operation is then sent through the output and bidirectional pins.

How to test

The project will be used to perform mac operations on 8-bit unsigned integers. This is mainly used in systems with fast computation and also primarily explores the concepts of reversible gates for energy efficiency.

External hardware

No external hardware is used for this project.

Pinout

#	Input	Output	Bidirectional
0	ui_in[0]	uo_out[0]	ui0[0]
1	ui_in1	uo_out1	ui01
2	ui_in2	uo_out2	ui02
3	ui_in[3]	uo_out[3]	ui0[3]
4	ui_in[4]	uo_out[4]	ui0[4]

#	Input	Output	Bidirectional
5	ui_in[5]	uo_out[5]	ui0[5]
6	ui_in[6]	uo_out[6]	ui0[6]
7	ui_in[7]	uo_out[7]	ui0[7]

Programmable PWM Generator [584]

- Author: Anas Alam
- Description: Programmable PWM Generator
- GitHub repository
- HDL project
- Mux address: 584
- Extra docs
- Clock: 0 Hz

How it works

A programmable PWM generator. The desired frequency and duty cycle is programmed by setting `pwm_top` and `pwm_threshold`. A counter counts from 0 to `pwm_top` (over and over), the pwm signal is high as when the counter is $\leq \text{pwm_threshold}$.

`pwm_top` is wired to `ui0` (all of them are used as inputs) `pwm_threshold` is wired to `ui1`

They are encoded as follows

```
pwm_top <= ui(7 downto 5) &< ui(4 downto 0)
```

```
pwm_threshold <= ui(7 downto 5) &< ui(4 downto 0)
```

Resulting frequency of PWM signal is: $f_{out} = \frac{f_{in}}{pwm_{top} + 1}$

Resulting duty cycle is: $f = \frac{pwm_{threshold} + 1}{pwm_{top} + 1}$

The goal is to have wide as possible frequency range while still being able to go from 0% to 100% in duty cycle.

How to test

Use above formulas to determine value of `pwm_threshold` and `pwm_top`, hard wire them to this value or connect through switches. Probe output on oscilloscope

External hardware

Switches and oscilloscope

Pinout

#	Input	Output	Bidirectional
0	pwm_threshold shift_amount[0]	pwm output	input: pwm_top shift_amount[0]
1	pwm_threshold shift_amount1	design is enabled (active high)	input: pwm_top shift_amount1
2	pwm_threshold shift_amount2	wired 0	input: pwm_top shift_amount2
3	pwm_threshold shift_amount[3]	wired 0	input: pwm_top shift_amount[3]
4	pwm_threshold shift_amount[4]	wired 0	input: pwm_top shift_amount[4]
5	pwm_threshold base[0]	wired 0	input: pwm_top base[0]
6	pwm_threshold base1	wired 0	input: pwm_top base1
7	pwm_threshold base[3]	wired 0	input: pwm_top base2

Verilog test project [586]

- Author: Alexander Symons
- Description: It adds the input and the IO pins
- GitHub repository
- HDL project
- Mux address: 586
- Extra docs
- Clock: 0 Hz

How it works

It adds the dedicated input to an internal register every clock cycle Least significant bits: dedicated output Most significant bits: bidirectional output

How to test

Put numbers on the input and see the accumulated value on all the leds

External hardware

Switches on inputs, leds on outputs and bidirectionals

Pinout

#	Input	Output	Bidirectional
0	ui_in[0]	uo_out[0]	ui_out[0]
1	ui_in1	uo_out1	ui_out1
2	ui_in2	uo_out2	ui_out2
3	ui_in[3]	uo_out[3]	ui_out[3]
4	ui_in[4]	uo_out[4]	ui_out[4]
5	ui_in[5]	uo_out[5]	ui_out[5]
6	ui_in[6]	uo_out[6]	ui_out[6]
7	ui_in[7]	uo_out[7]	ui_out[7]

Basic LIF Neuron [588]

- Author: stewedbeef
- Description: This is a basic LIF neuron
- GitHub repository
- HDL project
- Mux address: 588
- Extra docs
- Clock: 0 Hz

How it works

This is a simple leaky integrate-and-fire neuron which performs the integration by addition and leaks by dividing by two every time step. The neuron has an enable pin which causes the neuron to enable and move forward in time roughly once every second when fed a clock of approximately 50 MHz.

How to test

The LED wired up to output seven should turn on and off approximately once every second, with a period of approximately two seconds, to allow synchronisation by the user. Each time the LED switches on or off a time step has occurred. The user should stimulate the neuron by “providing” an input current, which is achieved by switching the inputs manually to indicate to the neuron, in binary, how much current should flow in. With enough stimulus, the neuron will fire a spike, visible on LEDs zero to six, for one time period. The neuron has a timeout which prevents it from having a constant output from overstimulation.

External hardware

Wire switches to all input ports and LEDs to all output ports. Bidirectional ports are unused.

Pinout

#	Input	Output	Bidirectional
0	ui_in[0]	uo_out[0]	
1	ui_in1	uo_out1	

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

Integrate-and-Fire Neuron Circuit [590]

- Author: FNU Ashwine
- Description: A simple integrate-and-fire neuron model implemented in Verilog.
- GitHub repository
- HDL project
- Mux address: 590
- Extra docs
- Clock: 0 Hz

How it works

The Leaky Integrate-and-Fire (LIF) Neuron is a simple model of neuronal behavior. In this design, the neuron receives an input signal (spike) and integrates this input over time by increasing its internal membrane potential. If there is no input, the membrane potential “leaks” or decays gradually over time, simulating the natural loss of charge in biological neurons.

When the membrane potential reaches a defined threshold, the neuron fires a spike output, after which the membrane potential resets to zero. This process emulates the firing and reset cycle of biological neurons, providing a digital approximation of spiking behavior.

LIF Neuron Diagram - https://drive.google.com/uc?export=view&id=19_hF5C_uv8FfWdIOOItIB8326t2pqFBz

How to test

Do something

External hardware

NA

#	Input	Output	Bidirectional
---	-------	--------	---------------

Pinout

#	Input	Output	Bidirectional
0	Input current bit [0]	State variable bit [0]	
1	Input current bit 1	State variable bit 1	
2	Input current bit 2	State variable bit 2	
3	Input current bit [3]	State variable bit [3]	
4	Input current bit [4]	State variable bit [4]	
5	Input current bit [5]	State variable bit [5]	
6	Input current bit [6]	State variable bit [6]	
7	Input current bit [7]	State variable bit [7]	Spike bit

Michaels Tiny Tapeout ALU [592]

- Author: Michael McCulloch
- Description: Should work as a 2 6 bit input ALU, which then can choose from the RISCV ALU opcodes to select the operation which will be outputted in 8bit
- GitHub repository
- HDL project
- Mux address: 592
- Extra docs
- Clock: 0 Hz

How it works

In short the first 6 bits of the bidirection IO is A, then bits 7 and 7 of the bidirectional and bits 0 to 3 of the single way input are B and the last 4 bits are the ALU opcode (based on RISCV) Values get outputted in 8bit from the single way output bus

How to test

Setting the inputs and testing the outputs for certain opcodes

External hardware

None at the moment... Could attach LEDs for testing

Pinout

#	Input	Output	Bidirectional
0	Bit 2 of ALU Input B	Bit 0 of ALU Output	Bit 0 of ALU Input A
1	Bit 3 of ALU Input B	Bit 1 of ALU Output	Bit 1 of ALU Input A
2	Bit 4 of ALU Input B	Bit 2 of ALU Output	Bit 2 of ALU Input A
3	Bit 5 of ALU Input B	Bit 3 of ALU Output	Bit 3 of ALU Input A
4	Bit 0 of ALU OpCode	Bit 4 of ALU Output	Bit 4 of ALU Input A
5	Bit 1 of ALU OpCode	Bit 5 of ALU Output	Bit 5 of ALU Input A
6	Bit 2 of ALU OpCode	Bit 6 of ALU Output	Bit 0 of ALU Input B
7	Bit 3 of ALU OpCode	Bit 7 of ALU Output	Bit 1 of ALU Input B

8-bit CBILBO [594]

- Author: Devesh Bhaskaran, Om Shivshankar Shigarkanti, Garima Bajpayi
- Description: Concurrent Built-In Logic BlockIn Logic Block Observer for Memory Test
- GitHub repository
- HDL project
- Mux address: 594
- Extra docs
- Clock: 40000000 Hz

How it works

In this Verilog code, we implement a BILBO (Built-In Logic Block Observer) shift register with multiple stages, using a combination of logic gates (AND, XOR), D flip-flops (DFF), and multiplexers (MUX) for feedback and shifting operations. We include input and output paths for Tiny Tapeout and support asynchronous reset and clocked logic. The modules interact to store and shift data, providing internal feedback and driving outputs for observation.

How to test

To test this project, we would create a testbench that provides stimulus for the inputs (`ui_in`, `uio_in`, `clk`, `rst_n`) and checks the outputs (`uo_out`, `uio_out`, `uio_oe`). We would simulate the shifting and feedback behavior of the BILBO shift register, verifying that the data is properly shifted and the feedback logic functions correctly across all stages of the register.

External hardware

No external hardware required for this project.

Pinout

#	Input	Output	Bidirectional
0	<code>ui_in[0]</code>	<code>uo_out[0]</code>	<code>uio[0]</code>
1	<code>ui_in1</code>	<code>uo_out1</code>	<code>uio1</code>
2	<code>ui_in2</code>	<code>uo_out2</code>	<code>uio2</code>

#	Input	Output	Bidirectional
3	ui_in[3]	uo_out[3]	uiou[3]
4	ui_in[4]	uo_out[4]	uiou[4]
5	ui_in[5]	uo_out[5]	uiou[5]
6	ui_in[6]	uo_out[6]	uiou[6]
7	ui_in[7]	uo_out[7]	uiou[7]

Wokwi Group #8 [608]

- Author: Tiny Tapeout
- Description: Combined project to save space for ttihp25a
- GitHub repository
- HDL project
- Mux address: 608
- Extra docs
- Clock: 0 Hz

- JonsFirstTapeout by ghangas
- patrick's project by patrick marcus
- TT Test by Austin
- Half Adder by Brendon
- Letter H by Hannah Thoreson
- tinytapeoutkr by kamila ramirez
- Tahiti by Harrison
- Dipankar's first Wowki design by Dipankar Shakya
- Zero to Nine Display Count by Mariano
- tinytapeout by Htun
- print by Syeva
- AND and NOT gate testing by Aman Maldar
- Full Adder by David De La Luz
- seven by nikmign
- Name Speller by Conor Van Bibber

Pinout

#	Input	Output	Bidirectional
0	in0	out0	sel0
1	in1	out1	sel1
2	in2	out2	sel2
3	in3	out3	sel3
4	in4	out4	
5	in5	out5	
6	in6	out6	
7	in7	out7	

Wokwi Group #9 [610]

- Author: Tiny Tapeout
- Description: Combined project to save space for ttihp25a
- GitHub repository
- HDL project
- Mux address: 610
- Extra docs
- Clock: 0 Hz

- Gabe's Big AND by GabeMake
- Project by calculus
- Two PFD by Soumabrata Ghosh
- tt09 kathyhtt by kathyh
- Yohan Tiny Tapeout Project by Juan
- halfadder+not by Vincent Phan
- Encoder by Hoang Le
- Tiny Tapeout by Andy
- Nathan's chip by Nathineal
- Binary to 7 Segment Display Decoder by Robert McLintock
- SK Test Workshop by sreela
- Tiny Tapeout 9 Template by Jason
- Full Adder by Amogha Srinivas
- hello by vishwajeet
- Morse Code for J and R by Jainil Rao

Pinout

#	Input	Output	Bidirectional
0	in0	out0	sel0
1	in1	out1	sel1
2	in2	out2	sel2
3	in3	out3	sel3
4	in4	out4	
5	in5	out5	
6	in6	out6	
7	in7	out7	

Wokwi Group #10 [612]

- Author: Tiny Tapeout
- Description: Combined project to save space for ttihp25a
- GitHub repository
- HDL project
- Mux address: 612
- Extra docs
- Clock: 0 Hz

- Pseudo Random Generator Using 2 Ring Oscillators by Michael Yim
- Redco by Shrikrishna Kaje
- Test_project by Ash
- Lynn's TinyTapeout Design by Lynn Francis
- Encoder by Ryan Schrader
- Big J's Big Circuit by Jonathan Miller
- Logic Gates by Adonai Cruz
- Samson's Tiny Tapout Project by Samson
- 8 bit LFSR by Aaron Nowack
- Kai's Death Adder by Kai Linsley
- Who knows what's happening Tiny Tapeout by Ryan Kuo
- Manchester Encoder by Prajwal Shashidhar Chavadi
- TinyTapeout workshop - Wokwi 8 Bit LFSR by Nate Voorhies
- Kanoa's first Wokwi design Tinytapeout 2024 Nonsense by Kanoa Mignard
- Adbe_Project by Aditya_Bedekar

Pinout

#	Input	Output	Bidirectional
0	in0	out0	sel0
1	in1	out1	sel1
2	in2	out2	sel2
3	in3	out3	sel3
4	in4	out4	
5	in5	out5	
6	in6	out6	
7	in7	out7	

Wokwi Group #11 [614]

- Author: Tiny Tapeout
- Description: Combined project to save space for ttihp25a
- GitHub repository
- HDL project
- Mux address: 614
- Extra docs
- Clock: 0 Hz

- Full Adder by May Wang
- TinyTapeout 4 bit ripple carry adder by Georg Brink Dyvad
- Extremely cool stuff (secret) by Alexander Aakersø
- TinyChipDesign by Oliver
- example1 by Hassan Sirelkhatim
- my_own_chip by Abdala
- 3-bit register print by Emad Maroun
- Hero on Tape by Marcus Sand
- Special code for letter n by Nuno Jorge
- Adder by Ehsan
- Four basic building blocks by Thomas novotny
- First design by Mathias Vestergaard
- Enter-Code by Imad
- ANDNOT by HKG
- Tinytapeout_design_ANP by Ashrfun Naher Pinky
- 4 Bit Adder with Overflow Counter by Yousif

Pinout

#	Input	Output	Bidirectional
0	in0	out0	sel0
1	in1	out1	sel1
2	in2	out2	sel2
3	in3	out3	sel3
4	in4	out4	
5	in5	out5	
6	in6	out6	
7	in7	out7	

Wokwi Group #12 [616]

- Author: Tiny Tapeout
- Description: Combined project to save space for ttihp25a
- GitHub repository
- HDL project
- Mux address: 616
- Extra docs
- Clock: 0 Hz

- tt08-octal-alu by Theo Kachelski
- Simple 8 Bit ALU by Joseph Johnson
- Traffic-light-sequence by Shaurya Sharma
- Logic Test by Eric Ulteig
- Abacus Lock by Raunak Singh
- Counter by Jasmin Mittelman
- simplePass by Shawko
- Emil Njor's Design by Emil Njor
- Holm's TinyTapeOut 4-bit adder by Jakob Holm
- test/15/02/25 by Viktor Hougaard Jørgensen
- NAND by kofi
- DaliaProjekt by Dalia
- 3 Bit Adder by Victor Ding
- Encoder by Damianos
- Simple NAND 2 by Mad

Pinout

#	Input	Output	Bidirectional
0	in0	out0	sel0
1	in1	out1	sel1
2	in2	out2	sel2
3	in3	out3	sel3
4	in4	out4	
5	in5	out5	
6	in6	out6	
7	in7	out7	

triggerer [618]

- Author: Krzysztof Skrzyniecki
- Description: Module capturing timestamps of input signals (triggers) referenced to internal counter
- GitHub repository
- HDL project
- Mux address: 618
- Extra docs
- Clock: 50000000 Hz

How it works

This module is capturing high edges in input lines (triggers) and stores the timestamp when this happened. Period of internal counter is order of 30ms (24b).

Main clk is used for internal logic and timestamp timer. When edge on trigger input is detected, time is captured (max capture frequency is clk/2, but preferably even lower).

When data to read is available it is signalled on data ready output pin.

In order to read the data, first set high enable pin, hold it and while holding start clocking data clk input. (max data clk rate should be over 2 times slower than clk). Read the data on output pin on data clk rising edge. 3 bytes should be read. Most significant bit is transferred first.

How to test

Just hope that this works (tested manually on simulator exactly once..)

External hardware

None, but signal generator bursting a few edges into trig input might be helpful

#	Input	Output	Bidirectional
---	-------	--------	---------------

Pinout

#	Input	Output	Bidirectional
0	DAT_CLK	DAT_RDY	
1	DAT_ENA	DAT_OUT	
2	TRIGG_0		
3			
4			
5			
6			
7			

Wokwi Group #13 [620]

- Author: Tiny Tapeout
- Description: Combined project to save space for ttihp25a
- GitHub repository
- HDL project
- Mux address: 620
- Extra docs
- Clock: 0 Hz

- Test by Bruno
- nothing-yet by elen
- counter_KS by wendelin
- snake by Anton Gerasimov
- TinyTapeoutRocks by Theodote
- Secret code by Kevin Geppert
- TinyTapeoutWorkshop by Torben
- TestFlipflop by Benjamins Sulcas
- Simple Test Project by MZ
- Tape Out and Find Out by Zoe
- spinny by Daniel Rojas
- tinyflipout by Ig
- Synchronous hex counter decoder by Spehro
- OR Gate-Based 7-Segment Display Decoder by Kirill
- Blinking 7 by Milos Lompar
- tt_micha01 by Michael Wiebusch

Pinout

#	Input	Output	Bidirectional
0	in0	out0	sel0
1	in1	out1	sel1
2	in2	out2	sel2
3	in3	out3	sel3
4	in4	out4	
5	in5	out5	
6	in6	out6	
7	in7	out7	

Multiplier Group #1 [622]

- Author: Tiny Tapeout
- Description: Combined project to save space for ttihp25a
- GitHub repository
- HDL project
- Mux address: 622
- Extra docs
- Clock: 0 Hz

- 4 x 4 array multiplier NuKoP by Aiden Li, Mahid Hosen
- 4x4multiplier by hirod nazari, samarth pusegaonkar
- ECE2204 4x4 Array Multiplier by Jason Brandon
- 4x4 Array Multiplier by Adrian Lopez and Jack Verdis
- tt09-C6-array-multiplier by Jonathan Farah and Josef Anurov
- 4-bit multiplier by Annie Huang and Sharon Chi
- my_4bit_multiplier by Terry Mu, Omobolaji Alabi
- Array_Multiplier by Taegahm Kang
- 4-bit Multiplier by Sarp Sevil
- Array multiplier by Wyte wu ,Xintong Hu
- Array Multiplier by Leon Ha, Jegyeoung An
- Array Multiplier by Jeryl Ho & Justin Park
- 4x4 Multiplier by Fajr Baig, Sahana Long
- 4 bit array multiplier by Abdulrahman Albaoud, Joe Leighhardt
- ece2204 project for tapeout by Yiqiao, Geno
- Array Multiplier by Will Shang, Tyler Huynh

Pinout

#	Input	Output	Bidirectional
0	in0	out0	sel0
1	in1	out1	sel1
2	in2	out2	sel2
3	in3	out3	sel3
4	in4	out4	
5	in5	out5	
6	in6	out6	
7	in7	out7	

Multiplier Group #2 [624]

- Author: Tiny Tapeout
- Description: Combined project to save space for ttihp25a
- GitHub repository
- HDL project
- Mux address: 624
- Extra docs
- Clock: 0 Hz

- Array Multiplier by Rebecca Boadu & Sarah Herrera
- 4-bit Array Multiplier by Minjae Kim, Jiawei Ding
- 4-bit-array-multiplier by HenryZ-ErickR
- Lab B Group 1 Array Multiplier by MarcAnthony Williams & Ivy Zheng
- Lab B Group 10 Array Multiplier by Abhinav and Annay
- ECE2204 4x4 Array Multiplier by Jack Li Bill Li
- Array Multiplier by Jaden Daily
- 4x4 Array Multiplier by Marisol and Shahran
- 4 by 4 Array Multiplier by Hanyuan (Bob) Huang
- 4-bit Multiplier by Jeremy Kang, Idris Al-Wazani
- Lab C 4x4 Mult-Array by Justin Morris, Alexa
- 4-bit Multiplier by Asfaq Fahim & Sreeja Ghose
- 4-bit-multiplier by Eric Cheung, Bethel Sisay
- 4bit multiplier by Kylian Yan
- ECE-UY 2204 4x4 Array Multiplier by Jane Manalu, Isabella Menshouse, KJ Moses
- 4-bit Multiplier by Nick Pham, Nathan Macapinlac

Pinout

#	Input	Output	Bidirectional
0	in0	out0	sel0
1	in1	out1	sel1
2	in2	out2	sel2
3	in3	out3	sel3
4	in4	out4	
5	in5	out5	
6	in6	out6	
7	in7	out7	

Multiplier Group #3 [626]

- Author: Tiny Tapeout
- Description: Combined project to save space for ttihp25a
- GitHub repository
- HDL project
- Mux address: 626
- Extra docs
- Clock: 0 Hz
- 4x4 array multiplier by Gabriela Perez, Martha McQuillan
- 4x4 Array Multiplier by Dominic Iafrate
- ece2204_4x4_mult by Eric Wang, Alan Zhu
- ECE-2204 4x4 Array Multiplier by Evan Dworkin, Dante Minasyan
- Array Multiplier by Noah Rivera & Filip Bukowski
- array_multiplier by xg2523_cw4483
- ECE2204MultiplierProject by CaoKeHanMax
- Array Multiplier by Theodore Hua

Pinout

#	Input	Output	Bidirectional
0	in0	out0	sel0
1	in1	out1	sel1
2	in2	out2	sel2
3	in3	out3	sel3
4	in4	out4	
5	in5	out5	
6	in6	out6	
7	in7	out7	

Ternary 128-element Dot Product [640]

- Author: ReJ aka Renaldas Ziomė
- Description: $\text{sum}(A * B)$ where A is a binary vector, B is a ternary vector
- GitHub repository
- HDL project
- Mux address: 640
- Extra docs
- Clock: 0 Hz

How it works

On-chip neural net with ternary weights and 1-bit activations.

How to test

Use commander app to set inputs and weights.

External hardware

No additional hardware needed.

Pinout

#	Input	Output	Bidirectional
0	binary vector A element 0	out (LSB)	(in serial) ternary vector B element ZERO
1	binary vector A element 1	out	(in serial) ternary vector B element SIGN
2	binary vector A element 2	out	out
3	binary vector A element 3	out	out
4	binary vector A element 4	out	out
5	binary vector A element 5	out	out
6	binary vector A element 6	out	out
7	binary vector A element 7	out	out (MSB)

GUS16 CPU [642]

- Author: J. Arias
- Description: 16-bit CPU design
- GitHub repository
- HDL project
- Mux address: 642
- Extra docs
- Clock: 24000000 Hz

How it works

This project includes a 16-bit experimental CPU (GUS16) with a serial port and a few more peripherals (see GUS16_tt.pdf). Memory has to be provided externally. An included bootloader allows the execution of programs loaded through the serial port.

How to test

Connect a serial port 8-bit, no parity, 115200 bps, and send an 'L'. The bootloader code should reply with another 'L'. For more complete tests an external board with SRAM memory and address latches has to be attached to the PMOD ports of the prototype board.

External hardware

A memory board has to be attached to user PMOD connectors (still pending design)

More docs

https://www.ele.uva.es/~jesus/cpu_v2.pdf (older designs, spanish)

<https://www.ele.uva.es/~jesus/GUS16v6.pdf> (current CPU version)

<https://www.ele.uva.es/~jesus/a2.pdf> (CPU usage in a floppy disk emulator for apple-IIIs in FPGAs)

Pinout

#	Input	Output	Bidirectional
0	gpi[0]	xbh	xd[0]
1	gpi1	xlal	xd1
2	gpi2	xlal	xd2
3	rx _d	pwmout	xd[3]
4	gpi[3]	tx _d	xd[4]
5	gpi[4]	gpo	xd[5]
6	gpi[5]	xoeb	xd[6]
7	gpi[6]	xweb	xd[7]

Warp [644]

- Author: sylefeb
- Description: Demo on TinyTapeout? Let's do something!
- GitHub repository
- HDL project
- Mux address: 644
- Extra docs
- Clock: 25000000 Hz

Warp

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

How it works

But does it work?

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

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

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

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

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

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

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

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

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

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

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

External hardware

- VGA PMOD
- Audio PMOD

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

Pinout

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

VGA Drop (audio/visual demo) [646]

- Author: ReJ aka Renaldas Zioma, eriQue aka Erik Hemming, Matthias Kampa
- Description: Tiny 8 part Megademo! TBL^Nesnausk^SonikClique
- GitHub repository
- HDL project
- Mux address: 646
- Extra docs
- Clock: 25200000 Hz

How it works

VGA signal generator

How to test

We are learning how VGA and Sky130 works here

External hardware

VGA PMOD

Pinout

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

Classic 8-bit era Programmable Sound Generator AY-3-8913 [648]

- Author: ReJ aka Renaldas Zioma
- Description: The AY-3-8913 is a 3-voice programmable sound generator (PSG) chip from General Instruments. The AY-3-8913 is a smaller variant of AY-3-8910 or its analog YM2149.
- GitHub repository
- HDL project
- Mux address: 648
- Extra docs
- Clock: 2000000 Hz

How it works

This Verilog implementation is a replica of the classical **AY-3-8913** programmable sound generator. With roughly a 1500 logic gates this design fits on a **single tile** of the TinyTapeout.

The goals of this project

1. closely replicate the behavior and eventually the complete **design of the original AY-3-891x** with builtin DACs
2. provide a readable and well documented code for educational and hardware **preservation** purposes
3. leverage the **modern fabrication** process

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

Chip technical capabilities

- **3 square wave** tone generators
- A single **white noise** generator
- A single **envelope** generator able to produce 10 different shapes
- Chip is capable to produce a range of waves from a **30 Hz** to **125 kHz**, defined by **12-bit** registers.
- **16** different volume levels

Registers The behavior of the AY-3-891x is defined by 14 registers.

Register	Bits used	Function	Description
0	xxxxxxx	Channel A Tone	8-bit fine frequency
1xxx	—//—	4-bit coarse frequency
2	xxxxxxx	Channel B Tone	8-bit fine frequency
3xxx	—//—	4-bit coarse frequency
4	xxxxxxx	Channel C Tone	8-bit fine frequency
5xxx	—//—	4-bit coarse frequency
6	...xxxx	Noise	5-bit noise frequency
7	.CBACBA	Mixer	Tone and/or Noise per channel
8	...xxxx	Channel A Volume	Envelope enable or 4-bit amplitude
9	...xxxx	Channel B Volume	Envelope enable or 4-bit amplitude
10	...xxxx	Channel C Volume	Envelope enable or 4-bit amplitude
11	xxxxxxx	Envelope	8-bit fine frequency
12	xxxxxxx	—//—	8-bit coarse frequency
13xxx	Envelope Shape	4-bit shape control

Square wave tone generators Square waves are produced by counting down the 12-bit counters. Counter counts up from 0. Once the corresponding register value is reached, counter is reset and the output bit of the channel is flipped producing square waves.

Noise generator Noise is produced with 17-bit Linear-feedback Shift Register (LFSR) that flips the output bit pseudo randomly. The shift rate of the LFSR register is controlled by the 5-bit counter.

Envelope The envelope shape is controlled with 4-bit register, but can take only 10 distinct patterns. The speed of the envelope is controlled with 16-bit counter. Only a single envelope is produced that can be shared by any combination of the channels.

Volume Each of the three AY-3-891x channels have dedicated DAC that converts 16 levels of volume to analog output. Volume levels are 3 dB apart in AY-3-891x.

Historical use of the AY-3-891x

The AY-3-891x family of programmable sound generators was introduced by General Instrument in 1978. Soon Yamaha Corporation licensed and released a very similar chip under YM2149 name.

Both variants of the AY-3-891x and YM2149 were broadly used in home computers, game consoles and arcade machines in the early 80ies.

- home computers: Apple II Mockingboard sound card, Amstrad CPC, Atari ST, Oric-1, Sharp X1, MSX, ZX Spectrum 128/+2/+3
- game consoles: Intellivision, Vectrex, Amstrad GX4000

- arcade machines: Frogger, 1942, Spy Hunter and etc.

The AY-3-891x chip family competed with the similar Texas Instruments SN76489.

The original pinout of the AY-3-8913

The **AY-3-8913** was a 24-pin package release of the AY-3-8910 with a number of internal pins left simply unconnected. The goal of AY-3-8913 was to reduce complexity for the designer and reduce the foot print on the PCB. Otherwise the functionality of the chip is identical to AY-3-8910 and AY-3-8912.

```
,--._.--.
GND ---|1    24|<-- /cs*
BDIR -->|2    23|<-- a8*
BC1 -->|3    22|<-- /a9*
DA7 <->|4    21|<-- /RESET
DA6 <->|5    20|<-- CLOCK
DA5 <->|6    19|--- GND
DA4 <->|7    18|--> CHANNEL C OUT
DA3 <->|8    17|--> CHANNEL A OUT
DA2 <->|9    16|    not connected
DA1 <->|10   15|--> CHANNEL B OUT
DA0 <->|11   14|<-- test*
test* <--|12   13|<-- VCC
`-----'
* -- omitted from this Verilog implementation
```

Difference from the original hardware

This Verilog implementation is a completely digital and synchronous design that differs from the original AY-3-8913 design which incorporated internal DACs and analog outputs.

Audio signal output While the original chip had no summation The module provides two alternative outputs for the generated audio signal:

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

Master output channel In contrast to the original chip which had only separate channel outputs, this implementation also provides an optional summation of the channels into a single master output.

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

No /A8, A9 and /CS pins The combination of **/A8, A9** and **/CS** pins originally were intended to select a specific sound chip out the larger array of devices connected to the same bus. In this implementation this mechanism is omitted for simplicity, **/A8, A9** and **/CS** are considered to be tied **low** and chip behaves as always enabled.

Synchronous reset and single phase clock The original design employed 2 phases of the clock and asynchronous reset mechanism for operation of the registers.

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

The reverse engineered AY-3-891x

This implementation would not be possible without the reverse engineered schematics and analysis based on decapped AY-3-8910 and AY-3-8914 chips.

Explain how your project works

How to test

Summary of commands to communicate with the chip

The AY-3-8913 is programmed by updating its internal registers via the data bus. Below is a short summary of the communication protocol of AY-3-891x. Please consult AY-3-891x Technical Manual for more information.

BDIR	BC1	Bus state description
0	0	Bus is inactive
0	1	(Not implemented)
1	0	Write bus value to the previously latched register #
1	1	Latch bus value as the destination register #

Latch register address First, put the destination register address on the bus of the chip and latch it by pulling both **BDIR** and **BC1** pins **high**.

Write data to register Put the desired value on the bus of the chip. Pull **BC1** pin **low** while keeping **BDIR** pin **high** to write the value of the bus to the latched register address.

Inactivate bus by pulling both **BDIR** and **BC1** pins **low**.

Register	Format	Description	Parameters
0,2,4	fffffff	A/B/C tone period	f - low bits
1,3,5	0000FFFF	—//—	F - high bits

Register	Format	Description	Parameters
6	000fffff	Noise period	f - noise period
7	00CBAcba	Noise / tone per channel	CBA - noise off, cba - tone off
8,9,10	000Evvvv	A/B/C volume	E - envelope on, v - volume level
11	fffffff	Envelope period	f - low bits
12	FFFFFFF	—/—	F - high bits
13	0000caAh	Envelope Shape	c - continue, a - attack, A - alternate, h - hold

Note frequency

Use the following formula to calculate the 12-bit period value for a particular note:

$$toneperiod_{cycles} = clock_{frequency}/(16_{cycles} * note_{frequency})$$

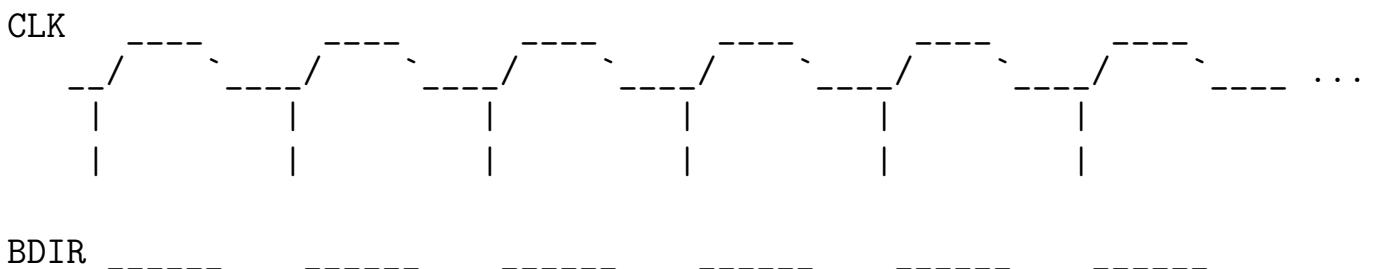
For example 12-bit period that plays 440 Hz note on a chip clocked at 2 MHz would be:

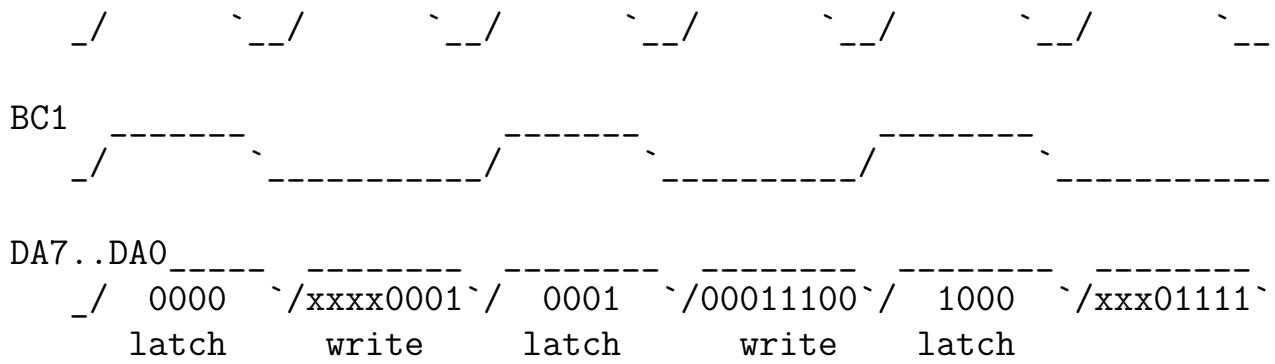
$$toneperiod_{cycles} = 2000000Hz/(16_{cycles} * 440Hz) = 284 = 11C_{hex}$$

An example to play a note at a maximum volume

BDIR	BC1	DA7..DA0	Explanation
1	1	xxxx0000	Latch tone A coarse register address 0 = 0000 _{bin}
1	0	xxxx0001	Write high 4-bits of the 440 Hz note 1 = 0001 _{bin}
1	1	xxxx0001	Latch tone A fine register address 1 _{dec} = 0001 _{bin}
1	0	00011100	Write low 8-bits of the note 1C _{hex} = 00011100 _{bin}
1	1	xxxx1000	Latch channel A volume register address 8 = 1000 _{bin}
1	0	xxx01111	Write maximum volume level 15 _{dec} = 1111 _{bin} with the envelope disabled

Timing diagram





Externally configurable clock divider

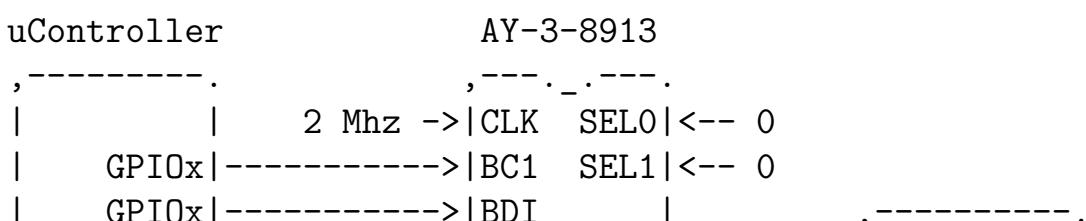
SEL1	SEL0	Description	Clock frequency
0	0	Standard mode, clock divided by 8	1.7 .. 2.0 MHz
1	1	—//—	1.7 .. 2.0 MHz
0	1	New mode for TT05, no clock divider	250 .. 500 kHz
1	0	New mode for TT05, clock div. 128	25 .. 50 MHz

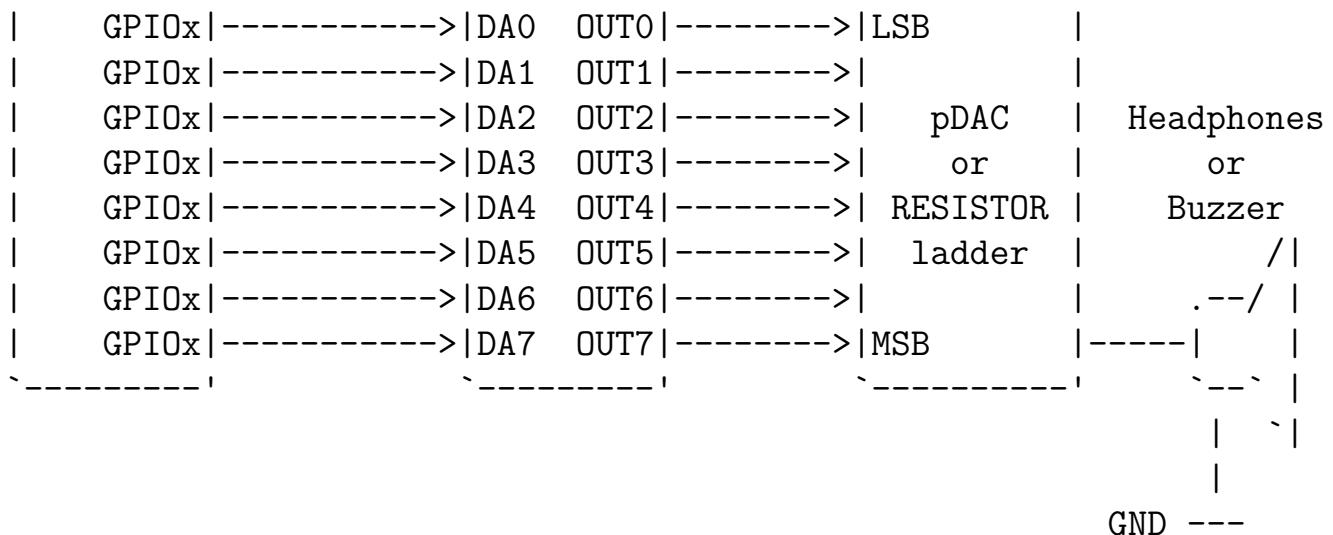
SEL1	SEL0	Formula to calculate the 12-bit tone period value for a note
0	0	$clock_frequency / (16_{cycles} * note_{frequency})$
1	1	—//—
0	1	$clock_frequency / (2_{cycles} * note_{frequency})$
1	0	$clock_frequency / (128_{cycles} * note_{frequency})$

External hardware

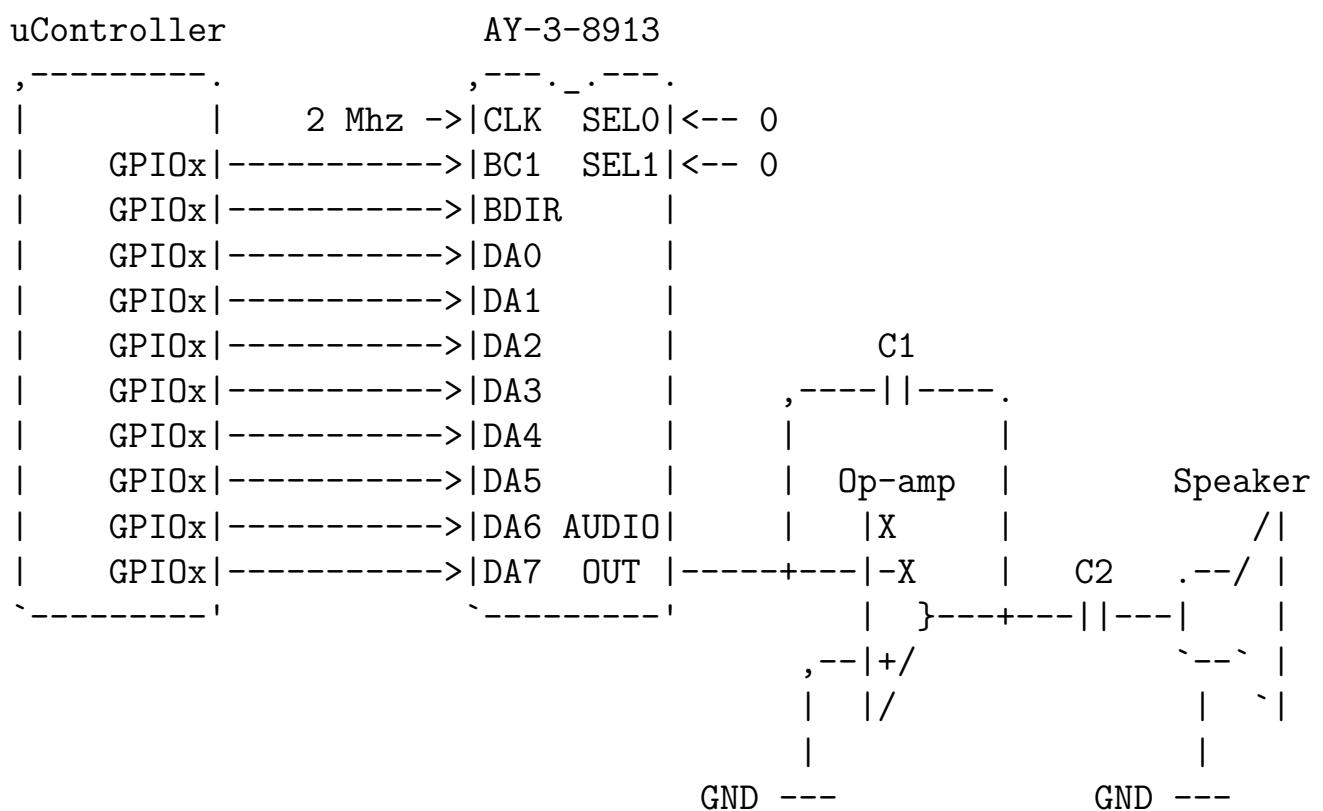
The data bus of the AY-3-8913 chip has to be connected to microcontroller and receive a regular stream of commands. The AY-3-8913 produces audio output and has to be connected to a speaker. There are several ways how the overall schematics can be established.

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

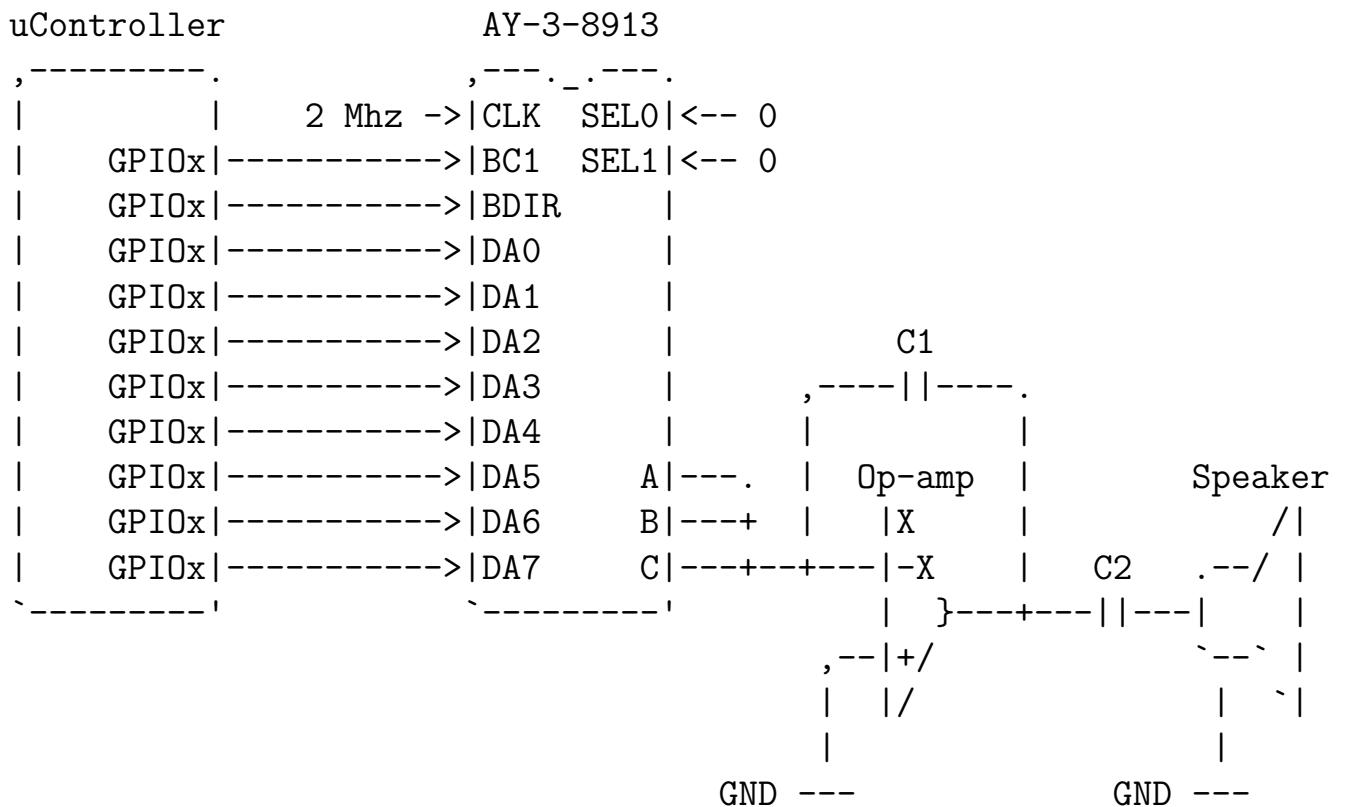




AUDIO OUT through RC filter Another option is to use the Pulse Width Modulated (PWM) AUDIO OUT pin that combines 4 channels with the Resistor-Capacitor based low-pass filter or better the Operation Amplifier (Op-amp) & Capacitor based integrator:



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



Pinout

#	Input	Output	Bidirectional
0	DA0 - multiplexed data/address bus LSB	audio out (PWM)	(in) BC1 bus control
1	DA1 - multiplexed data/address bus	digital audio LSB	(in) BDIR bus direction
2	DA2 - multiplexed data/address bus	digital audio	(in) SEL0 clock divider
3	DA3 - multiplexed data/address bus	digital audio	(in) SEL1 clock divider
4	DA4 - multiplexed data/address bus	digital audio	(out) channel A (PWM)
5	DA5 - multiplexed data/address bus	digital audio	(out) channel B (PWM)
6	DA6 - multiplexed data/address bus	digital audio	(out) channel C (PWM)
7	DA7 - multiplexed data/address bus MSB	digital audio MSB	(out) AUDIO OUT master

SoCET UART with FIFO buffers [650]

- Author: Miguel Isrrael Teran, Yashashwini Singh, Michael Li, Rafael Monteiro Martins Pinheiro, Vito Gamberini
- Description: General-purpose UART with hardware control flow and FIFO buffer capacity developed by Purdue's SoCET team
- GitHub repository
- HDL project
- Mux address: 650
- Extra docs
- Clock: 50000000 Hz

How it works

This project is a UART module that includes FIFO buffers to store bytes of data. The module has standard UART input and output pins, such as rx, tx, cts, and rts. Additional inputs allow the configuration of operation mode(s): **Idle**, **RX**, **TX** or **Buffer Clear**, and desired baud rate is user-configurable through the Control pins. Bidirectional data pins are used to send and receive test data. Additional outputs include an error flag, as well as the TX FIFO's full flag, and the RX FIFO's empty flag.

How to test

Steps for testing are the following:

- 1) Supply a 50 MHz clock signal to the UART
- 2) Configure the control settings: Control[1:0] (ui[3:2]) are used to choose between preloaded baud rates. Here are the following baud rate configurations based on the values of Control[1:0]:

Value of ui[3:2]	Baud rate (bits/s)
0	9600
1	19200
2	38400
3	115200

Control[3:2] (ui[5:4]) set the UART's mode of operation for the current byte of data being processed. Each non-idle control signal must be preceded with an idle

signal to perform a valid transaction/manage the FIFO buffers. Here are the following UART mode configurations determined by the values of Control[3:2]:

Value of ui [5:4]	Mode Configuration
0	IDLE
1	TX
2	RX
3	BUFFER CLEAR

- 3) If you have 2 PCBs with the TT09 ASIC, you can load the same UART design in both and cross-connect their rx, tx, cts, and rts pins as shown in the image below. Then, you can use one of them as a Transmitter and the other as a Receiver. If you only have 1 PCB, you can test the UART with the **FT232RL Mini USB to TTL Serial Adapter Module** (see next section).

External hardware

We suggest using **switches** for the Control pins (this way you can keep the mode of operation stable). Image below shows the **FT232RL** module that can be used for testing and connecting serially to a computer's USB port. More information on the product can be found [here](#).

Pinout

#	Input	Output	Bidirectional
0	rx	tx	data[0]
1	cts	rts	data1
2	Control[0]	err	data2
3	Control1	tx_buffer_full	data[3]
4	Control2	rx_buffer_empty	data[4]
5	Control[3]		data[5]
6			data[6]
7			data[7]

Simon's Caterpillar [652]

- Author: htfab
- Description: Port of Caterpillar Logic to Simon Says PMOD
- GitHub repository
- HDL project
- Mux address: 652
- Extra docs
- Clock: 50000 Hz

How it works

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

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

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

How to test

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

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

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

(The empty sequence is neither valid nor invalid.)

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

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

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

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

Other keys and combinations:

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

External hardware

Simon Says PMOD

Pinout

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

Stochastic Integrator [654]

- Author: Ciecen Lestari, Chih-Kuan Ho, David Parent
- Description: Use stochastic computing to implement integration
- GitHub repository
- HDL project
- Mux address: 654
- Extra docs
- Clock: 50000000 Hz

How it works

The stochastic integrator uses Euler's definition of integration to make it happen in the stochastic domain. This integrator follows unipolar probability.

REFERENCES USED

General Stochastic Integrator Design:

1 S. Liu and J. Han, "Hardware ODE solvers using stochastic circuits," 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, USA, 2017, pp. 1-6, doi: 10.1145/3061639.3062258. keywords: {Radiation detectors;Stochastic processes;Hardware;Generators;Clocks;Energy consumption;Throughput;stochastic integrator;ordinary differential equation;stochastic computing},

LFSR Design in Stochastic Computing:

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

How to test

Set ui_in[0] with a constant high and ui_in1 with constant low to see the equations described.

External hardware

ADALM2000

Pinout

#	Input	Output	Bidirectional
0	serial_input_1	serial_output_seq_integrator_a	
1	serial_input_2	serial_output_seq_integrator_b	
2		serial_output_seq_integrator_c	
3		serial_output_system_integrator_a	
4		serial_output_system_integrator_b	
5		serial_output_test_integrator_a	
6		serial_output_test_integrator_b	
7		output_sn_bit_seq_integrator_c	

E2M0 x INT8 Systolic Array [656]

- Author: ReJ aka Renaldas Zioma
- Description: Systolic array for testing (Septenary and Quinary) 2.6 bits/param packed weights
- GitHub repository
- HDL project
- Mux address: 656
- Extra docs
- Clock: 48000000 Hz

How it works

Reduced precision matrix multiplication base on systolic array architecture. Left side matrix is compressed to 2.6 bits per element.

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

External hardware

External processor (RP2040 for example) 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

#	Input	Output	Bidirectional
6	packed weights	result	(in) activations
7	packed weights MSB	result MSB	(in) activations MSB

VGA Nyan Cat [658]

- Author: Andy Sloane
- Description: Displays the classic nyan.cat animation
- GitHub repository
- HDL project
- Mux address: 658
- Extra docs
- Clock: 25175000 Hz

VGA nyan cat



How it works Outputs nyancat on VGA with music!

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

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

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

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

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

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

Pinout

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

Collatz conjecture brute-forcer [673]

- Author: Vytautas Šaltenis
- Description: Runs a Collatz sequence calculation for a given number
- GitHub repository
- HDL project
- Mux address: 673
- Extra docs
- Clock: 0 Hz

How it works

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

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

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

How to test

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

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

The full loop of computations works like this:

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

Writing input:

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

Reading output:

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

Pinout

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

APA102 to WS2812 Translator [675]

- Author: Squidgeefish
- Description: Convert a 7-LED APA102 stream to a WS2812-compatible one
- GitHub repository
- HDL project
- Mux address: 675
- Extra docs
- Clock: 25000000 Hz

How it works

This is a converter from the SPI-style APA102 LED protocol to the single-line WS2812 protocol.

It's hard-coded to seven LEDs because I needed to set a limit, and this is clearly the simplest possible way to replace the Arduino Micro performing the same task on my 5n4ck3y-7r clone.

It clocks the SPI data on input bit 0 (clock) and bit 1 (data) and waits until it sees a string of 32 low bits to signal a valid start condition. At this point, it starts saving data into an internal shift register, handing that register's contents over to the WS2812 output data feed once all seven LEDs' values have been received. It continues clocking along until recognizing a stop condition (unconditionally 32 bits after the last LED value), at which point it goes back to waiting for a valid start condition. In order to address area concerns, I wound up cutting this down a bit - the internal mirror register was removed entirely, and the SPI reader now also handles discarding the first 8 bits of each 32-bit pixel value. Further tweaks that traded wiring complexity for combinatorics did not make it any better, unfortunately.

I wrote the SPI-parsing and bit-shuffling code from scratch, but the WS2812 output module is lifted from this TT05 submission. I did modify it to read the data stream MSB-first rather than LSB-first since that made my life a lot easier in bit-twiddler land.

Note that the first byte of each APA102 packet encodes an intensity, which I am ignoring since WS2812s do not support such a feature.

How to test

The way I will be testing this is by attaching ui_in[0] to SCK and ui_in[1] to SDO on a DEFCON badge that used APA102 LEDs. Attach uo_out[0] to drive a string of

at least seven WS2812s. I suspect that level shifters will be needed since TinyTapeout ICs run at around 1.8V?

Alternatively, you could probably stream something over in MicroPython.

If you're hand-crafting your packets, a few notes:

- A packet stream must start with a 32-bit start packet (0x00000000)
- APA102s reserve the first byte for intensity: 0b11100000 | <5-bit intensity>. We're ignoring this completely.
- APA102 color order for the remaining three bytes is Blue, Green, Red.

There is also a random feature added in to fill space - there should be a continuous UART output of "Arglius Barglius" on `uo_out[1]` at approximately 115200 baud; this can be read out with a serial bridge or sufficiently advanced logic analyzer.

External hardware

Some sort of SPI driver is necessary, as is a string of at least seven WS2812 LEDs (or I suppose a logic analyzer can verify it if you're allergic to blinkies).

Pinout

#	Input	Output	Bidirectional
0	APA102_CK	WS2812_OUT	
1	APA102_SD	UART_OUT	
2			
3			
4			
5			
6			
7			

pio-ram-emulator example: Julia fractal [677]

- Author: Toivo Henningsson
- Description: Example of using pio-ram-emulator to draw a Julia fractal
- GitHub repository
- HDL project
- Mux address: 677
- Extra docs
- Clock: 50400000 Hz

How it works

This is an example of the using the <https://github.com/toivoh/pio-ram-emulator> RAM emulator for Tiny Tapeout. The RAM is used to store a frame buffer, 320x480 at 2 bits/pixel. The frame buffer is continuously read to output a 640x480 @60 Hz VGA signal. At the same time, the logic computes a Julia fractal, writing 16 bits to the frame buffer for every 8 pixels computed. After about a second, the whole frame buffer is filled in.

For more info about the RAM emulator, see <https://github.com/toivoh/pio-ram-emulator/blob/main/docs/pio-ram-emulator.md>.

The project contains some helper code for working with the RAM emulator:

- `pio_ram_emulator.v` and `pio_ram_emulator.vh` (`sb_io.v` is also need) contain the modules `pio_ram_emu_transmitter` and `pio_ram_emu_receiver`
 - These are used to transmit and receive messages using the RAM emulator's message format
 - The design still has to follow the rules in <https://github.com/toivoh/pio-ram-emulator/blob/main/docs/pio-ram-emulator.md> about which messages can be sent when
 - See `julia_top.v` for an example of how to use these modules
- `test/pio_ram_emulator_model.v` contains a simulation model of the RAM emulator
 - See `test/tb.v` for an example of how to use the simulation model in a test
 - See `verilator/vtop.v` for an example of how to use the simulation model in a verilator setup

- The model will try to detect behavior that violates the rules in <https://github.com/toivoh/pio-ram-emulator/blob/main/docs/pio-ram-emulator.md>, in which case it will set an error flag and stop responding (see the `ERROR_RESPONSE` parameter)
- The simulation model is helpful, but might not capture the full behavior of the RAM emulator. Please try to run your design on an FPGA against the actual RAM emulator as well.

How to test

Plug in a TinyVGA VGA Pmod to the output Pmod. The <https://github.com/toivoh/pio-ram-emulator> RAM emulator must be running on the RP2040. **TODO: Instructions for how to set up.** Start the project.

Controls The appearance of the Julia fractal is controlled by the `C` parameter, which can be seen as a complex value or 2d vector. The `C` parameter can be changed using the `ui_in` port:

- `button_up` / `button_down` / `button_left` / `button_right` move the `C` value.
- `button_incstep` doubles the step length.
- `button_decstep` halves the step length.

A new `ui_in[5:0]` value must be stable for 2^{19} cycles, or approximately 10 ms (at a 50.4 MHz clock rate), before it is accepted. The `use_both_button_dirs` input changes how the input is interpreted:

- When `use_both_button_dirs = 0`, an input is triggered when one of the `button_` signals goes from high to low (and is stable for 10 ms). Recommended if the inputs are connected to buttons.
- When `use_both_button_dirs = 1`, an input is triggered when one of the `button_` signals goes from high to low or low to high (and is stable for 10 ms). Recommended if the inputs are connected to toggle switches.

External hardware

This project needs a TinyVGA VGA Pmod.

Pinout

#	Input	Output	Bidirectional
0	button_up	R1	
1	button_down	G1	
2	button_right	B1	
3	button_left	vsync	
4	button_incstep	R0	tx_out[0]
5	button_decstep	G0	tx_out1
6		B0	rx_in[0]
7	use_both_button_dirs	hsync	rx_in1

Tiny Neural Network Accelerator [678]

- Author: Greg Chadwick
- Description: A toy neural network accelerator targetting CNNs
- GitHub repository
- HDL project
- Mux address: 678
- Extra docs
- Clock: 50000000 Hz

How it works

This project is a neural network accelerator designed for use with convolutional neural networks. The verilog is generated from system verilog source which lives in a separate repository: <https://github.com/GregAC/tiny-nn> which also contains the full DV environment, model, documentation and related utilities and software.

Internally it contains a number of 16-bit floating point add and multiply units (using something approximating the BF16 floating point encoding) that can be configured to work in different ways for different operations. Operations available are

- Convolve - Computes a 4x2 convolution kernel across an image. The parameters are loaded in and held in flops then the image is streamed in one pixel at a time. The most recent 4x2 image pixels are also held in flops so every 2 new pixels giving a new image column computes a new convolution (internally the last image column is dropped and the new column shifted in).
- Accumulate - Sum groups of N input numbers with a fixed bias added and an optional RELU operation (0 if accumulation less than 0 otherwise leaves accumulation untouched). N is provided by the operation word and the bias is only loaded in once. E.g. if you set N = 4 an bias of 1.0 for each 4 numbers input it would sum them together then add the bias and do the optional RELU. Numbers keep streaming in until the operation is terminated

The interface is a fixed 16 bits in and 8 bits out synchronous to the clock. Each operation has a special operand code that starts it that needs to be sent on the 16 input bits. Once started the 16 input bits provide the numbers used for the operation.

The 16-bit numbers output are split over 2 clock cycles for the 8 bit output. With the lower byte output first. The user needs to know when the output is relevant (some cycles the output should be ignored and some it should be captured).

- ui_in, uio_in - 16-bit input, ui_in is top byte
- uo_out - 8-bit output

<https://github.com/GregAC/tiny-nn> should contain full documentation with the details and software to use the accelerator (both a work in progress at tapeout time!).

How to test

There are 3 test modes to test basic input output.

ASCII test Place 16'hFFFF on the input {ui_in, uio_in} and hold it and on the output you will observe a repeating pattern:

- 8'h54
- 8'h2d
- 8'h4e
- 8'h4e

This is 'T-NN' in ASCII

Pulse Test Place 16'hF000 on the input {ui_in, uio_in} and hold it and on the output you will observe a repeating pattern:

- 8'haa
- 8'h55

Count Test Place 16'hF1XX on the input {ui_in, uio_in} where XX is any 8-bit number and on the output you will observe a count down from that number.

Accumulate Operation The simplest operation is the accumulate one. We'll configure it to add two numbers at a time with a -3.5 bias and RELU. Then we'll add 1.0 + 2.0 and 3.0 + 4.0. Put the following on the input over successive clocks

- 16'h2101 # Command word for accumulate operation
- 16'hc060 # -3.5 bias
- 16'h3f80 # 1.0
- 16'h4000 # 2.0
- 16'h4040 # 3.0
- 16'h4080 # 4.0
- 16'hffff # NaN - terminates operation

On the output you should observe:

- 16'hX
- 16'h00
- 16'h00
- 16'h60
- 16'h40

The 16'hX outputs could be anything and should be ignored, the first number output is 0000 representing $0.0 \text{ RELU}(1.0 + 2.0 - 3.5) = 0.0$, the second number output is 4060 representing $3.5 \text{ RELU}(3.0 + 4.0 - 3.5) = 3.5$.

External hardware

No specific external hardware required but it does need some external part to drive the desired sequences, this can be handled by the RP2040 on the demo board.

Pinout

#	Input	Output	Bidirectional
0	ui[0]	uo[0]	uio[0]
1	ui1	uo1	uio1
2	ui2	uo2	uio2
3	ui[3]	uo[3]	uio[3]
4	ui[4]	uo[4]	uio[4]
5	ui[5]	uo[5]	uio[5]
6	ui[6]	uo[6]	uio[6]
7	ui[7]	uo[7]	uio[7]

Fuzzy Search Engine [679]

- Author: Peter Nørlund
- Description: A levenshtein based fuzzy search engine
- GitHub repository
- HDL project
- Mux address: 679
- Extra docs
- Clock: 50000000 Hz

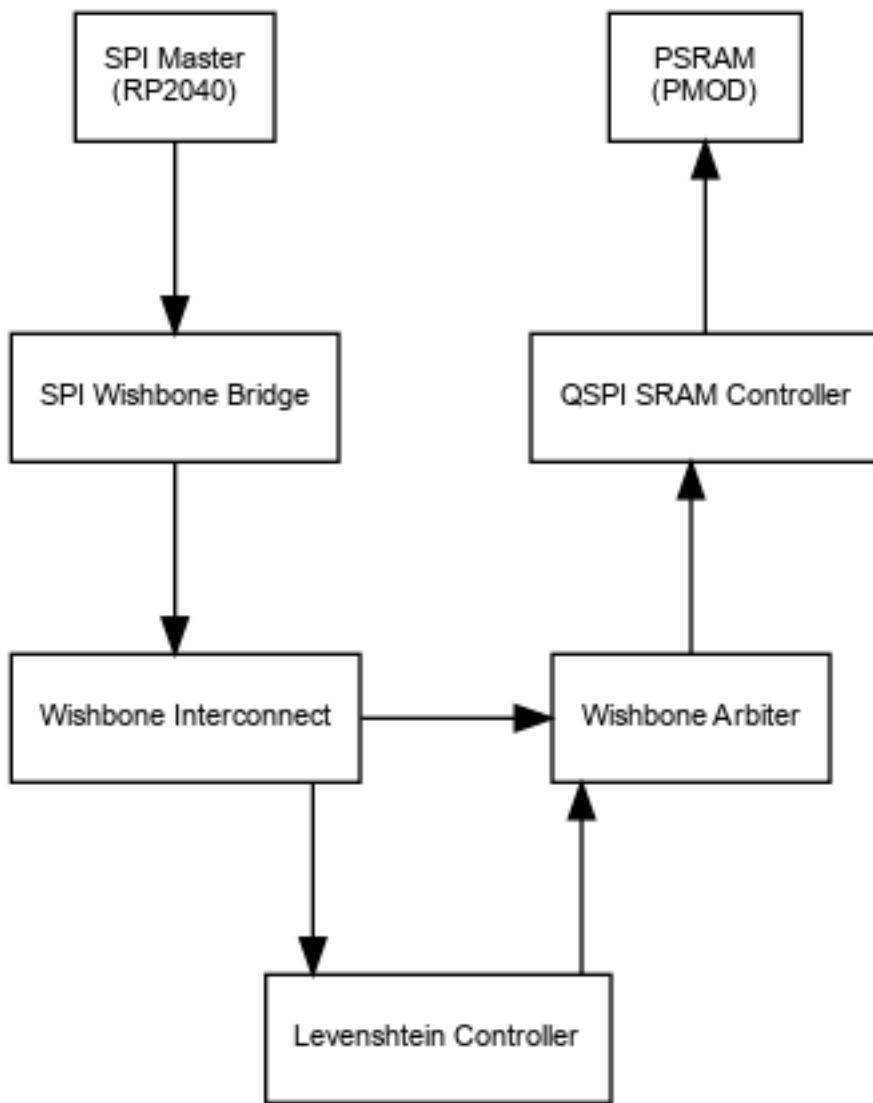
How it works

tt09-levenshtein is a fuzzy search engine which can find the best matching word in a dictionary based on levenshtein distance.

Fundamentally its an implementation of the bit-vector levenshtein algorithm from Heikki Hyyrö's 2003 paper with the title *A Bit-Vector Algorithm for Computing Levenshtein and Damerau Edit Distances*.

Architecture The overall architecture is a Wishbone Classic system with two masters (The levenshtein engine and an SPI controlled master) and two slaves (The levenshtein engine and a QSPI SRAM controller).

Using the SPI interface, you store a dictionary and some bitvectors representing a search word in SRAM and then configures and activates the engine. The engine will then read the dictionary and bitvectors from the SRAM and, ultimately store the index and distance of the word in the dictionary with the lowest levenshtein distance in registers which can be read by the user.



SPI The device is organized as a wishbone bus which is accessed through commands on an SPI bus.

The maximum SPI frequency is 25% of the master clock (12.5MHz when the chip is running at 50MHz).

The bus uses SPI mode 3 (CPOL=1, CPHA=1)

Input bytes:

Byte	Bit	Description
0	7	READ=0 WRITE=1
0	6-0	Address bit 22-16
1	7-0	Address bit 15-8
2	7-0	Address bit 7-0
3	7-0	Byte to write if WRITE, otherwise ignored

Output bytes:

Byte	Bit	Description
0	7-0	Byte read if READ, otherwise just 0x00

Since the SPI bridges to a wishbone bus which is shared by another master and because register and SRAM have different latencies, the response time is variable.

While the bus is working, the output bits will be zero. The final output byte will be preceded by a one-bit.

Note that this means that the value 0x5A can appear 8 different ways on the SPI bus:

01	5A	0000000	1	01011010	
02	B4	000000	1	01011010	0
05	68	00000	1	01011010	00
0A	D0	00000	1	01011010	000
15	A0	00000	1	01011010	0000
2B	40	00000	1	01011010	00000
56	80	00000	1	01011010	000000
AD	00	1	01011010	00000000	

Memory Layout As indicated by the SPI protocol, the address space is 23 bits.

The address space is basically as follows:

Address	Size	Access	Identifier
0x000000	1	R/W	CTRL
0x000001	1	R/W	SRAM_CTRL
0x000002	1	R/W	LENGTH
0x000003	1	R/O	MAX_LENGTH
0x000004	2	R/O	INDEX
0x000006	1	R/O	DISTANCE
0x000200	512	R/W	VECTORMAP
0x000400	8M	R/W	DICT

CTRL

The control register is used to start the engine and see when it has completed.

The layout is as follows:

Bits	Size	Access	Description
0	1	R/W	Enable flag
1-7	7	R/O	Not used

Set the enable flag to start the engine. When the engine is finished, the enable flag is changed to 0

SRAM_CTRL

Controls the SRAM

Bits	Size	Access	Description
0-1	2	R/W	Chip select
2-7	6	R/O	Not used

The chip select flag controls which chip select is used on the PMOD when accessing SRAM

Value	Pin	Notes
0	<i>None</i>	Default value
1	CS	Uses the default CS on the PMOD (Pin 1). Compatible with Machdyne's QQSP
2	CS2	Uses CS2 on the PMOD (pin 6). Compatible with mole99's QSPI Flash/(P)SRA
3	CS3	Uses CS3 on the PMOD (pin 7). Compatible with mole99's QSPI Flash/(P)SRA

LENGTH

Bits	Size	Access	Description
0-7	8	R/W	Word length minus 1

Used to indicate the length of the search word. Note that the word cannot be empty and it cannot exceed 16 characters.

MAX_LENGTH

Bits	Size	Access	Description
0-7	8	R/O	Max word length supported minus 1

This field allows for applications to dynamically detect the size of the bit vector.

DISTANCE

When the engine has finished executing, this address contains the levenshtein distance of the best match.

INDEX

When the engine has finished executing, this address contains the index of the best word from the dictionary in big endian byte order.

VECTORMAP

The vector map must contain the corresponding bitvector for each input byte in the alphabet.

If the search word is application, the bit vectors will look as follows:

Letter	Index	Bit vector
a	0x61	16'b00000000_01000001 (a_____a____)
p	0x70	16'b00000000_00000110 (_pp_____)
l	0x6C	16'b00000000_00001000 (___l_____)
i	0x69	16'b00000001_00010000 (____i____i__)
c	0x63	16'b00000000_00100000 (_____c____)
t	0x74	16'b00000000_10000000 (_____t____)
o	0x6F	16'b00000010_00000000 (_____o_)
n	0x6E	16'b00000100_00000000 (_____n)
*	*	16'b00000000_00000000 (_____)

Each vector is 16 bits in bit endian byte order.

The vectormap is stored in SRAM so the values are indetermined at power up and must be cleared.

DICT

The word list.

The word list is stored of a sequence of words, each encoded as a sequence of 8-bit characters and terminated by the byte value 0x00. The list itself is terminated with the byte value 0x01.

Note that the algorithm doesn't care about the particular characters. It only cares if they are identical or not, so even though the algorithm doesn't support UTF-8 and is limited to a character set of 254 characters, ignoring Asian alphabets, a list of words usually don't contain more than 254 distinct characters, so you can practically just map letters to a value between 2 and 255.

How to test

You can compile the client as follows:

```
mkdir -p build  
cmake -G Ninja -B build .  
cmake --build build
```

Next, you can run the test tool:

```
# Machdyne QQSPI PSRAM  
.build/client/client --interface tt --test --verify-dictionary --verify-  
  
# mole99 PSRAM  
.build/client/client --interface tt --cs cs2 --test --verify-dictionary
```

This will load 1024 words of random length and characters into the SRAM and then perform a bunch of searches, verifying that the returned result is correct.

External hardware

To operate, the device needs a QSPI PSRAM PMOD. The design is tested with the QQSPI PSRAM PMOD from Machdyne, but any memory PMOD will work as long as it supports:

- WRITE QUAD with the command 0x38 in 1S-4S-4S mode and no latency
- FAST READ QUAD with the command 0xE8 in 1S-4S-4S mode and 6 wait cycles
- 24-bit addresses
- Uses pin 0, 6, or 7 for SS#.
- Must be able to run at half the clock speed of the TT chip.

Note that this makes it incompatible with the spi-ram-emu project for the RP2040.

Pinout

#	Input	Output	Bidirectional
0			SRAM QSPI CS
1			SRAM QSPI SIO0/MOSI
2			SRAM QSPI SIO1/MISO
3			SRAM QSPI SCK

#	Input	Output	Bidirectional
4	SPI SS#		SRAM QSPI SIO2
5	SPI SCK		SRAM QSPI SIO3
6	SPI MOSI		SRAM QSPI CS2
7		SPI MISO	SRAM QSPI CS3

VGA Pride [681]

- Author: Rebecca G. Bettencourt
- Description: A VGA demo for showing pride flags
- GitHub repository
- HDL project
- Mux address: 681
- Extra docs
- Clock: 0 Hz

How it works

Displays pride flags on the screen.

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

How to test

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

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

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

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

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

External hardware

TinyVGA PMOD

Pinout

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

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

donut [683]

- Author: Daniel Endrabs
- Description: Showing a Donut
- GitHub repository
- HDL project
- Mux address: 683
- Extra docs
- Clock: 50350000 Hz

How it works

Each ellipse is hand crafted to create a donut.

How to test

Connect the PMOD VGA.

External hardware

TinyVGA PMOD

Pinout

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

UART [685]

- Author: Darryl Miles
- Description: UART
- GitHub repository
- HDL project
- Mux address: 685
- Extra docs
- Clock: 0 Hz

How it works

Docs to follow.

How to test

Docs to follow.

External hardware

Standard Tiny Tapeout PCB. The IC is a UART DTE.

Trying for:

- TxD on UO_OUT[4] for OUT4 on GPIO13 with RP2040 UART0 (main set)
- RxD on UO_IN[3] for IN3 on GPIO12 with RP2040 UART0 (main set)
- RTS on UO_OUT[5] for OUT5 on GPIO14 with RP2040 UART0 (main set)
- CTS on UO_IN[6] for IN6 on GPIO19 with RP2040 UART0 (adjacent set)

Pinout

#	Input	Output	Bidirectional
0	altclk		busData0
1	busMode0		busData1
2	busMode1		busData2
3	rxd	dtr	busData3
4	dsr	txd	busData4
5	dcd	rts	busData5
6	cts	intTx	busData6

#	Input	Output	Bidirectional
7	ri	intRx	busData7

Why not? [687]

- Author: sylefeb
- Description: One tile something
- GitHub repository
- HDL project
- Mux address: 687
- Extra docs
- Clock: 25000000 Hz

How it works

This is a single tile 'demo' hacked on the very last day, basically during coffee breaks. It's using an old rotozoom trick, and is otherwise pretty simple.

Music is ... well it is an attempt ;)

How to test

Plug VGA pmod, power up, enjoy.

External hardware

VGA PMOD, Audio PMOD

Pinout

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

FSK Modem +HDLC +UART (PoC) [689]

- Author: Darryl Miles
- Description: FSK Modem w/ HDLC transciever + UART (PoC digital side)
- GitHub repository
- HDL project
- Mux address: 689
- 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 electronic, 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, this 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 necessary 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 incorporates 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 mechsnism 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 combination, 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 yield 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 architecture makes sense and provided the most options.

Pinout

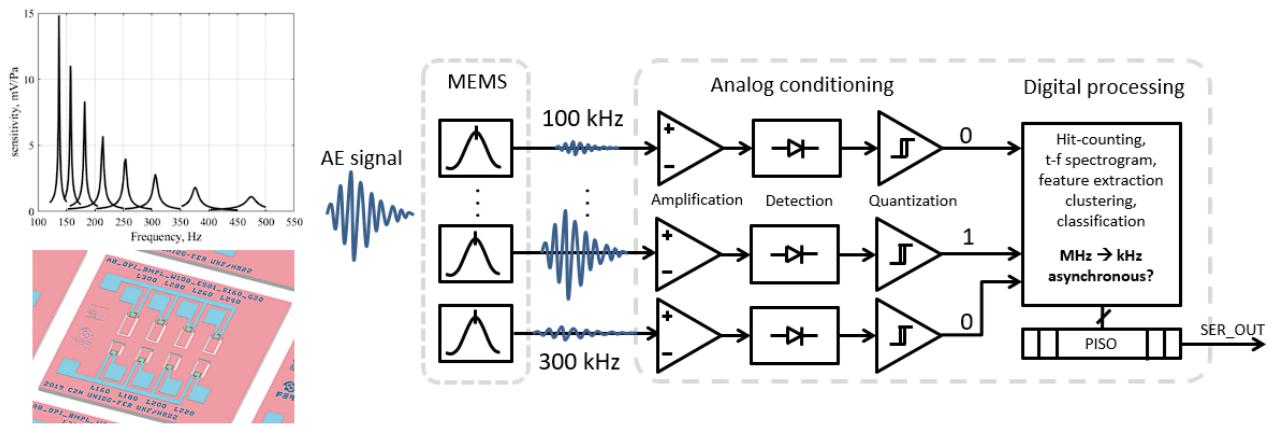
#	Input	Output	Bidirectional
0	Rx Data	UART TX	Rx Clock (bidi)
1	Tx Data	UART CTS	Up/Down (bidi)
2	UART RTS	UART DCD	TableAddr[0]
3	TableData[0]	Rx Error	TableAddr1
4	TableData1	Tx Error	TableAddr2
5	TableData2	Sending	TableAddr[3]
6	TableData[3]		TableAddr[4]
7	UART RX	Tx Clock Stobe	TableAddr[5]

Spectrogram extractor, 2 channels [690]

- Author: Coline Chehense, Dinko Oletic
- Description: Digital part of a time-frequency feature extraction sensor interface, two-channel real-time signal amplitude tracker. 7 input lines per channel represent thermometer code output of a flash ADC. Two-channel serial output.
- GitHub repository
- HDL project
- Mux address: 690
- Extra docs
- Clock: 1000000 Hz

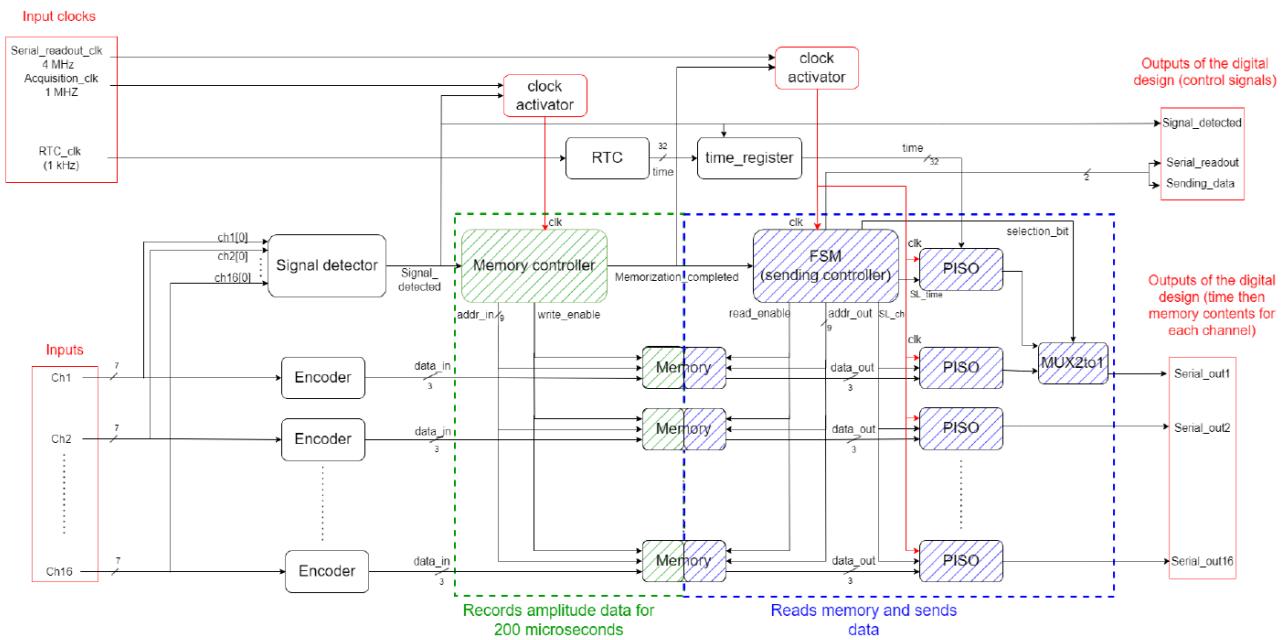
How it works

This is an early work-in progress test implementation of a digital readout, part of a low-power mixed-signal multichannel sensor interface for acoustic emission detection. The sensor interface is developed to support a passive, micromechanically-implemented ultrasonic signal frequency decomposition MEMS device, based on an array of piezoelectric micro-resonators: <https://ieeexplore.ieee.org/document/9139151>.



The digital part implemented here, performs real-time tracking of time-frequency spectrograms of individual acoustic emissions. It is assumed that each input-channel represents a signal amplitude envelope of the associated frequency-component over time, digitized using a flash ADC. The analog part of the ADC is not implemented here. Each input channel is represented by 7 input lines per channel representing the thermometer code output of a flash ADC. This test design implements only two-channels. The amplitude of an signal envelope at each channel is decoded into 3-bit BCD code. Presence of an input signal at any channel (detector of acoustic emission start) initiates event-based 1 MHz sampling of the time-frequency amplitude spectrogram. The sampling lasts for 200 us. Once finished, the state machine controls reads-out the data stored. A double buffer composed of D-bistables is used to manage the storage and

readout simultaneously. The stored data is sent serially for each channel. An RTC module is used to retrieve the time of the acoustic emission start.



This design is part of research activities <https://www.fer.unizg.hr/liss/aemems>. The design is generally applicable as a generic multi-channel time-series feature extraction block, and serve for subsequent clustering or classification, as part of an low-power MEMS-based sensor system-on-chip for acoustic event detection, or non-destructive testing. This is the first TinyTapeout submission of the design.

How to test

Please contact authors for detailed instructions on how to set-up the design.

External hardware

Logics analyzer will be useful for debugging.

Pinout

#	Input	Output	Bidirectional
0	ch1(0)	serial_out(0)	ch2(0)
1	ch1(1)	serial_out(1)	ch2(1)
2	ch1(2)	SL_time	ch2(2)
3	ch1(3)	SL_ch	ch2(3)
4	ch1(4)	signal_detected	ch2(4)

#	Input	Output	Bidirectional
5	ch1(5)	memorization_completed	ch2(5)
6	ch1(6)	serial_readout	ch2(6)
7	RTC_clk(1kHz)	sending_data	serial_readout_clk(4Mhz)

Bouncy Capsule [691]

- Author: htfab
- Description: Demoscene project featuring... well, a bouncy capsule
- GitHub repository
- HDL project
- Mux address: 691
- Extra docs
- Clock: 25000000 Hz

How it works

This is an entry to the Tiny Tapeout demoscene competition

How to test

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

External hardware

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

Pinout

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

TinyTapeout Minimal Branch Predictor [704]

- Author: Tristan Robitaille
- Description: A minimal perceptron-based branch predictor
- GitHub repository
- HDL project
- Mux address: 704
- Extra docs
- Clock: 1000000 Hz

How it works

This project implements a very minimal perceptron-based branch predictor. Using basic SPI, it reads in the lower part of the address of a branch instruction and its ground truth branch direction (taken or not taken). Due to constraints on the memory architecture (namely, 1 byte read per cycle), the predicton is not single-cycle.

The branch predictor is based on this paper: Dynamic Branch Prediction with Perceptrons.

This project uses latch-based memory from Michael Dell, available at: [tt06-memory](#)

It's best to run this project in its Docker container.

The `func_sim` directory contains a C++ functional simulation of the infrastructure. It parses a log file of a simulated execution of a RISC-V reference program and predicts the branch direction on each branch instruction. From the `func_sim` directory:
-Compile the reference: `riscv32-unknown-elf-gcc -O0 start.S reference.c -o reference -march=rv32i_zicsr_zifencei -T link.1d -nostartfiles -nostdlib`
-Disassemble reference: `riscv32-unknown-elf-objdump -d reference > reference_dis.txt` (for info only)
-Run Spike on reference: `spike --log=spike_log.txt --log-commits --isa=rv32i_zicsr_zifencei --priv=m -m128 reference` (to generate execution log)
-Make Makefile: `cmake CMakeLists.txt`
-Compile functional simulation: `make`
-Run: `./build/func_sim ./spike_log.txt`

The tests directory includes all CocoTB test for this design. Run them with `make`.

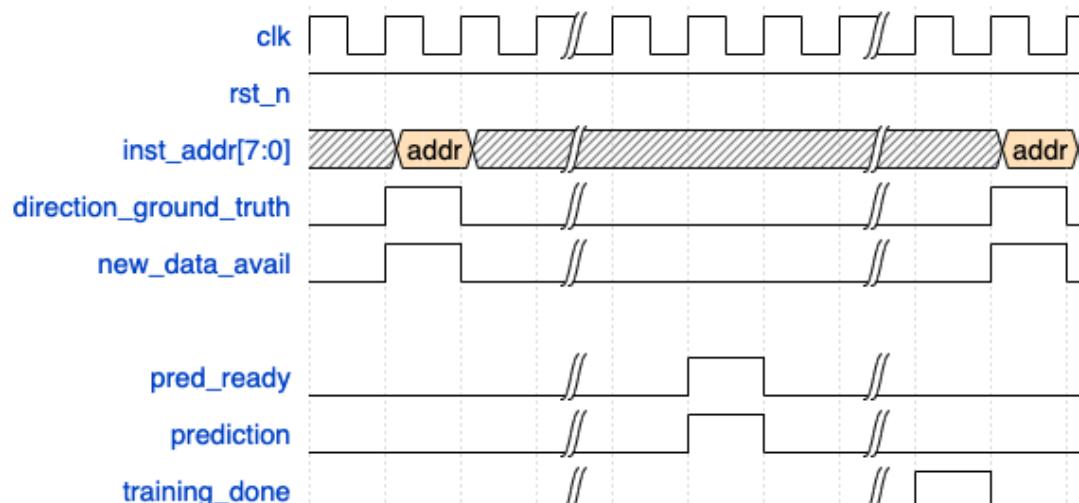
The documentation waveform is generated by Wavedrom.

How to test

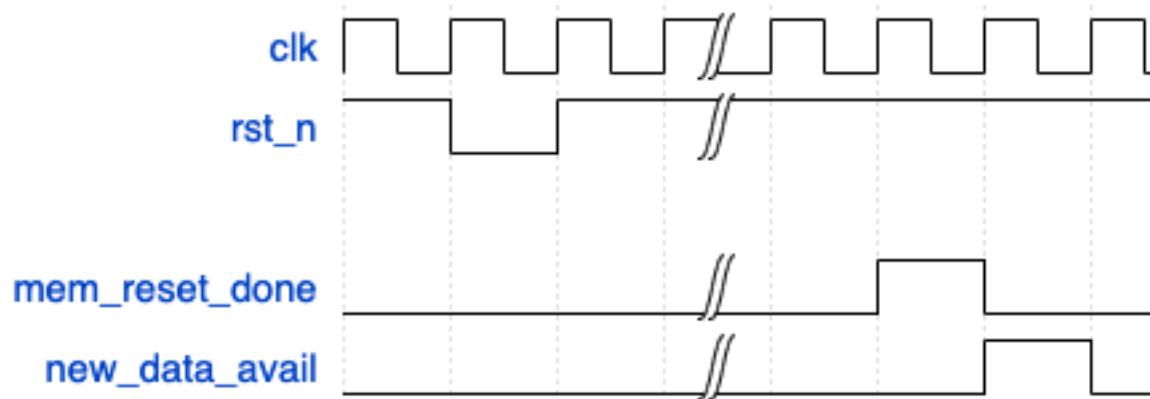
The branch predictor reads the lowest 8b of a 32b RISC-V address on pins `inst_addr`, the branch direction on pin `ui01` and a pulse notifying that new data is available on pin `new_data_avail`. Once the prediction is ready, it pulses pin `pred_ready`, pushes the prediction on pin `prediction`. Once training is complete, it pulses pin `training_done`. You can also request the history buffer by pulsing pin `history_buffer_request`. On the next cycle, the branch history will be sent to pin `DEBUG_history_buffer_output`, one cycle at a time, starting from the most recent.

Notes:

- After `rst_n` goes back high, you must wait for `mem_reset_done` to pulse. This is because the branch predictor resets its own memory. If new data comes in while it is resetting its memory, an inference will not be started.
- If no training is required this round, `training_done` pulses at the same time as `pred_ready`.
- If new data is sent while the predictor is training, it will be ignored and a new inference will not be started.



Prediction waveform:



Reset waveform:

To generate the instructions to use, you can either:

- 1) Parse `func_sim/spike_log.txt` with is a log of all instructions ran for the program `func_sim/reference.c` simulated on RISC-V `rv32i_zicsr_zifencei`. See `func_sim/src/func_sim.cpp` to see how that's done.
- 2) Generate a new log for a program of your choice. For this, you'll need to build the Docker. Word of caution: The Dockert takes >30 min top build and weighs >24GB.
 - 1) From root, run: `docker build -t tt_brand_predictor .`
 - 2) Once complete, run: `docker run -it -v pwd:/tmp tt_brand_predictor`
 - 3) Move to `func_sim`: `cd func_sim`
 - 4) Compile `reference.c`: `riscv32-unknown-elf-gcc -O0 start.S reference.c -o reference -march=rv32i_zicsr_zifencei -T link.ld -nostartfiles -nostdlib`
 - 5) Optionally, view the disassembly with: `riscv32-unknown-elf-objdump -d reference > reference_dis.txt`
 - 6) Generate execution log with Spike: `spike --log=spike_log.txt --log-commits --isa=rv32i_zicsr_zifencei --priv=m -m128 reference`
- 3) Running `func_sim` outputs a list of all branch instruction and the state of important registers in the branch predictor. You can use that to check if the predicted branch outcome is correct: `./build/func_sim ./spike_log.txt`

External hardware

- Some way to drive 10b (Arduino, FPGA, etc.)
- Some way to read 5b (Arduino, FPGA, oscilloscope, etc.)

Pinout

#	Input	Output	Bidirectional
0	<code>inst_addr[0]</code>	<code>pred_ready</code>	<code>new_data_avail</code>
1	<code>inst_addr1</code>	<code>prediction</code>	<code>direction_ground_truth</code>
2	<code>inst_addr2</code>	<code>training_done</code>	<code>DEBUG_perceptron_index[0]</code>
3	<code>inst_addr[3]</code>	<code>mem_reset_done</code>	<code>DEBUG_perceptron_index1</code>
4	<code>inst_addr[4]</code>	<code>DEBUG_new_data_avail_posedge</code>	<code>DEBUG_perceptron_index2</code>
5	<code>inst_addr[5]</code>	<code>DEBUG_state_pred[0]</code>	<code>DEBUG_wr_en</code>
6	<code>inst_addr[6]</code>	<code>DEBUG_state_pred1</code>	<code>DEBUG_history_buffer_output</code>
7	<code>inst_addr[7]</code>	<code>DEBUG_state_RST_mem</code>	<code>history_buffer_request</code>

Moody-mimosa [706]

- Author: D. Levante-Schmidiger
- Description: Moody ASIC reacting to external stimuli
- GitHub repository
- HDL project
- Mux address: 706
- Extra docs
- Clock: 1000000 Hz

How it works

The original idea was quite simple: I wanted to design my first custom digital ASIC and implement a fun concept. Instead of developing something like an adder or encoder, I wanted to push the boundaries of what's possible. A Tamagotchi-like, living creature—that's the vision. A creature with an inner life, capable of interacting with the outside world, and something you can simply have fun with.

This original idea turned out to be a rabbit hole. What if the internal model closely follows biological processes and takes neurotransmitters and hormones into account, for example? Which neurotransmitters and hormones are relevant and what kind of emotions could emerge? How could sleep and hunger be modeled? What does the creature do? What is observable from the outside? How could illness be incorporated? How does something like this fit on the limited space of the chip? How could it be simulated or emulated on an FPGA? How could one still gain a complete view of the inner workings for debugging and testing? What might the hardware look like?

In the meantime, this project has evolved into several subprojects:

Folder	Description
src	The digital design of the Mimosa model, including the Verilog source
test	Testing of the overall design, based on python and cocotb
module_test	Testing of specific verilog submodules, based on python and cocotb
fpga	Additional files (tcl, xdc) for creating the FPGA design for an Artix-7
simulation	A simulation based on Pyverilator and PyQt6 with a graphical interface
misc/mimosa_logger	An STM32 application for an STM32G474 microcontroller to debug the

More details are described in the respective sections below. Whenever possible, I tried to install the required dependencies in the Dockerfile or with an additional batch script or added third-party repositories as git submodules, in order to comply with licensing regulations. However, the FPGA utilities (Vivado Suite from AMD) can only

be installed with a personal account and therefore, you have to install it yourself. I also added makefiles or batch scripts in order to simplify building or running designs or applications.

Moody mimosa

Overview The moody mimosa model depends on several layers of abstraction.

1. Actions: Sleeping, eating, playing, smiling, babbling, kicking legs, doing nothing, crying
2. Emotions: Happiness, excitement, stress, nervousness, boredom, anger, calmness, apathy
3. Stimuli: Input from the outer world, either from the environment (cold, hot, loud, bright) or from an interacting individual (tickle, play with, calm down, talk to, feed).
4. Basic resources:
 - Neurotransmitters: Dopamine, Serotonin, Gamma-aminobutyric acid (Gaba), Norepinephrine
 - Hormones: Cortisol
 - Vital-energy, controlling sleepiness
 - Nourishment, controlling hunger
 - Illness, controlling whether the mimosa is ill or not

From the outer world, only one layer can be influenced directly (stimuli) and only one layer can be observed directly (actions). However, there are various indirect ways of how the stimuli influence the basic resources, emotions and actions and how the actions are influenced by emotions, basic resources and stimuli. The mimosa might cry because it is tired, ill, stressed, starving or angry because it cannot stand that it gets tickled all the time. You just don't know the reason. However, after some time, you develop an understanding of the creature and begin to realize what it might need.

Architecture

Implementation details Each resource consists of a saturating counter (counting up or down) and a regulator, regulating whether it should count up or down, slow or fast, or remain unchanged. The main feedback behaviour is encoded in the regulators. I tried to mimick the biology of the neurotransmitters involved. The first-level, rapid stress response is mediated by norepinephrine (NE). If stress persists, the slower second-level response mediated by the hormone cortisol sets in and leads to long-term stress

effects. The competing triple of serotonin, gaba and dopamine controls the mood and allows emotions such as happiness, excitement, boredom, anxiousness. During elevated periods of stress, all of them start to decrease, basically leading to a depressive state with negative emotions and without motivation. Hunger and tiredness also affects the neurotransmitters and even actions may lead to feedback effects, allowing both bottom-up and top-down emotion regulation (“smiling makes you feel better” vs. “if you feel good, you start smiling”). Although resources are themselves counters with 6-9 bits, only the upper two bits are used for emotion encoding and actions in order to limit gates needed.

Emotions are basically a combinational encoding of the resource levels (0=very low, 1=moderately low, 2=moderately high, 3=very high) and stimuli. In rare cases, several emotions can be present at once. Strictly speaking, emotions would not be necessary for the model. However, it turned out to be much more intuitive and simpler, to decide the resulting action based on emotions rather than on resource levels.

Actions are modelled as a state-machine. State transitions are mediated by stimuli and/or emotions. In rare cases, neurotransmitter levels may even target actions directly. At times, there are several routes how states may change. For example, the mimosa starts to sleep if it is moderately tired and not too stressed. If it is, however, stressed or starving or if you can't stop irritating it or if there are environmental influences (noise, heat), it just can not sleep. It surely will get angry, stressed, nervous and probably starts to cry but it can not sleep. After some time, it will get way too tired (zero vital energy) and start sleeping superficially.

Pinout For the tiny tapeout ASIC, the pins are assigned as described in the following tables:

Pin	Name	Function
clk	clk	Base clock
rst_n	rst_n	Reset, active low
ui_in[0]	stimulus_0	Interaction: Tickle
ui_in1	stimulus_1	Interaction: Play with
ui_in2	stimulus_2	Interaction: Talk to
ui_in[3]	stimulus_3	Interaction: Calm down
ui_in[4]	stimulus_4	Interaction: Feed
ui_in[5]	stimulus_5	Environment: Cool
ui_in[6]	stimulus_6	Environment: Hot
ui_in[7]	stimulus_7	Environment: Quiet

Pin	Dir	Name	Function
ui0_in[0]	0	stimulus_8	Environment: Loud
ui0_in1	0	stimulus_9	Environment: Dark
ui0_in2	0	stimulus_10	Environment: Bright
ui0_out[3]	-	-	-
ui0_out[4]	-	-	-
ui0_out[5]	-	-	-
ui0_out[6]	-	-	-
ui0_out[7]	-	-	-

Pin	Name	Function
uo_out[0]	action_0	Action: Sleeping
uo_out1	action_1	Action: Eating
uo_out2	action_2	Action: Playing
uo_out[3]	action_3	Action: Smiling
uo_out[4]	action_4	Action: Babbling
uo_out[5]	action_5	Action: Kicking legs
uo_out[6]	action_6	Action: Doing nothing
uo_out[7]	action_7	Action: Crying

How to test

Several ways

External hardware

Simulation No hardware required. Just run `python simulation/mimosa_simulation.py`. Make sure that you have run the `scripts/set_up_dependencies.sh` script and that you run a X-Server (e.g. VcXsrv) if you are working with Docker and Windows.

FPGA Following external hardware is required:

- Alchitry Au FPGA board
- Alchitry Br Adapter board
- Custom mimosa PCB [tbd] and USB-C cable

ASIC Apart from the actual ASIC, the following external hardware is required:

- Custom mimosa PCB [tbd] and USB-C cable

Pinout

#	Input	Output	Bidirectional
0	TICKLE	SLEEPING	LOUD
1	PLAY_WITH	EATING	BRIGHT
2	TALK_TO	PLAYING	SPI_MISO
3	CALM_DOWN	SMILING	SPI_SCK
4	FEED	BABBLING	SPI_CS
5	COOL	KICKING_LEGS	SPI莫斯I
6	HOT	DOING NOTHING	UART_TX
7	QUIET	CRYING	CLK_MODEL

Classic 8-bit era Programmable Sound Generator AY-3-8913 [708]

- Author: Eric Farrow - ReJ aka Renaldas Zioma
- Description: The AY-3-8913 is a 3-voice programmable sound generator (PSG) chip from General Instruments. The AY-3-8913 is a smaller variant of AY-3-8910 or its analog YM2149.
- GitHub repository
- HDL project
- Mux address: 708
- Extra docs
- Clock: 2000000 Hz

How it works

This Verilog implementation is a replica of the classical **AY-3-8913** programmable sound generator. With roughly a 1500 logic gates this design fits on a **single tile** of the TinyTapeout.

The goals of this project

1. closely replicate the behavior and eventually the complete **design of the original AY-3-891x** with builtin DACs
2. provide a readable and well documented code for educational and hardware **preservation** purposes
3. leverage the **modern fabrication** process

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

Chip technical capabilities

- **3 square wave** tone generators
- A single **white noise** generator
- A single **envelope** generator able to produce 10 different shapes
- Chip is capable to produce a range of waves from a **30 Hz** to **125 kHz**, defined by **12-bit** registers.
- **16** different volume levels

Registers The behavior of the AY-3-891x is defined by 14 registers.

Register	Bits used	Function	Description
0	xxxxxxx	Channel A Tone	8-bit fine frequency
1xxx	—//—	4-bit coarse frequency
2	xxxxxxx	Channel B Tone	8-bit fine frequency
3xxx	—//—	4-bit coarse frequency
4	xxxxxxx	Channel C Tone	8-bit fine frequency
5xxx	—//—	4-bit coarse frequency
6	...xxxx	Noise	5-bit noise frequency
7	.CBACBA	Mixer	Tone and/or Noise per channel
8	...xxxx	Channel A Volume	Envelope enable or 4-bit amplitude
9	...xxxx	Channel B Volume	Envelope enable or 4-bit amplitude
10	...xxxx	Channel C Volume	Envelope enable or 4-bit amplitude
11	xxxxxxx	Envelope	8-bit fine frequency
12	xxxxxxx	—//—	8-bit coarse frequency
13xxx	Envelope Shape	4-bit shape control

Square wave tone generators Square waves are produced by counting down the 12-bit counters. Counter counts up from 0. Once the corresponding register value is reached, counter is reset and the output bit of the channel is flipped producing square waves.

Noise generator Noise is produced with 17-bit Linear-feedback Shift Register (LFSR) that flips the output bit pseudo randomly. The shift rate of the LFSR register is controlled by the 5-bit counter.

Envelope The envelope shape is controlled with 4-bit register, but can take only 10 distinct patterns. The speed of the envelope is controlled with 16-bit counter. Only a single envelope is produced that can be shared by any combination of the channels.

Volume Each of the three AY-3-891x channels have dedicated DAC that converts 16 levels of volume to analog output. Volume levels are 3 dB apart in AY-3-891x.

Historical use of the AY-3-891x

The AY-3-891x family of programmable sound generators was introduced by General Instrument in 1978. Soon Yamaha Corporation licensed and released a very similar chip under YM2149 name.

Both variants of the AY-3-891x and YM2149 were broadly used in home computers, game consoles and arcade machines in the early 80ies.

- home computers: Apple II Mockingboard sound card, Amstrad CPC, Atari ST, Oric-1, Sharp X1, MSX, ZX Spectrum 128/+2/+3
- game consoles: Intellivision, Vectrex, Amstrad GX4000

- arcade machines: Frogger, 1942, Spy Hunter and etc.

The AY-3-891x chip family competed with the similar Texas Instruments SN76489.

The original pinout of the AY-3-8913

The **AY-3-8913** was a 24-pin package release of the AY-3-8910 with a number of internal pins left simply unconnected. The goal of AY-3-8913 was to reduce complexity for the designer and reduce the foot print on the PCB. Otherwise the functionality of the chip is identical to AY-3-8910 and AY-3-8912.

```
,--._.--.
GND ---|1    24|<-- /cs*
BDIR -->|2    23|<-- a8*
BC1 -->|3    22|<-- /a9*
DA7 <->|4    21|<-- /RESET
DA6 <->|5    20|<-- CLOCK
DA5 <->|6    19|--- GND
DA4 <->|7    18|--> CHANNEL C OUT
DA3 <->|8    17|--> CHANNEL A OUT
DA2 <->|9    16|    not connected
DA1 <->|10   15|--> CHANNEL B OUT
DA0 <->|11   14|<-- test*
test* <--|12   13|<-- VCC
`-----'
* -- omitted from this Verilog implementation
```

Difference from the original hardware

This Verilog implementation is a completely digital and synchronous design that differs from the original AY-3-8913 design which incorporated internal DACs and analog outputs.

Audio signal output While the original chip had no summation The module provides two alternative outputs for the generated audio signal:

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

Master output channel In contrast to the original chip which had only separate channel outputs, this implementation also provides an optional summation of the channels into a single master output.

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

No /A8, A9 and /CS pins The combination of **/A8, A9** and **/CS** pins originally were intended to select a specific sound chip out the larger array of devices connected to the same bus. In this implementation this mechanism is omitted for simplicity, **/A8, A9** and **/CS** are considered to be tied **low** and chip behaves as always enabled.

Synchronous reset and single phase clock The original design employed 2 phases of the clock and asynchronous reset mechanism for operation of the registers.

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

The reverse engineered AY-3-891x

This implementation would not be possible without the reverse engineered schematics and analysis based on decapped AY-3-8910 and AY-3-8914 chips.

Explain how your project works

How to test

Summary of commands to communicate with the chip

The AY-3-8913 is programmed by updating its internal registers via the data bus. Below is a short summary of the communication protocol of AY-3-891x. Please consult AY-3-891x Technical Manual for more information.

BDIR	BC1	Bus state description
0	0	Bus is inactive
0	1	(Not implemented)
1	0	Write bus value to the previously latched register #
1	1	Latch bus value as the destination register #

Latch register address First, put the destination register address on the bus of the chip and latch it by pulling both **BDIR** and **BC1** pins **high**.

Write data to register Put the desired value on the bus of the chip. Pull **BC1** pin **low** while keeping **BDIR** pin **high** to write the value of the bus to the latched register address.

Inactivate bus by pulling both **BDIR** and **BC1** pins **low**.

Register	Format	Description	Parameters
0,2,4	fffffff	A/B/C tone period	f - low bits
1,3,5	0000FFFF	—//—	F - high bits

Register	Format	Description	Parameters
6	000fffff	Noise period	f - noise period
7	00CBAcba	Noise / tone per channel	CBA - noise off, cba - tone off
8,9,10	000Evvvv	A/B/C volume	E - envelope on, v - volume level
11	fffffff	Envelope period	f - low bits
12	FFFFFFF	—/—	F - high bits
13	0000caAh	Envelope Shape	c - continue, a - attack, A - alternate, h - hold

Note frequency

Use the following formula to calculate the 12-bit period value for a particular note:

$$toneperiod_{cycles} = clock_{frequency}/(16_{cycles} * note_{frequency})$$

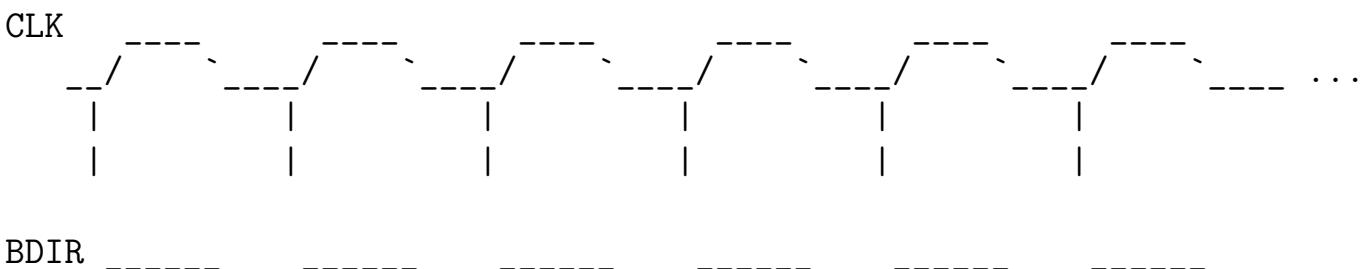
For example 12-bit period that plays 440 Hz note on a chip clocked at 2 MHz would be:

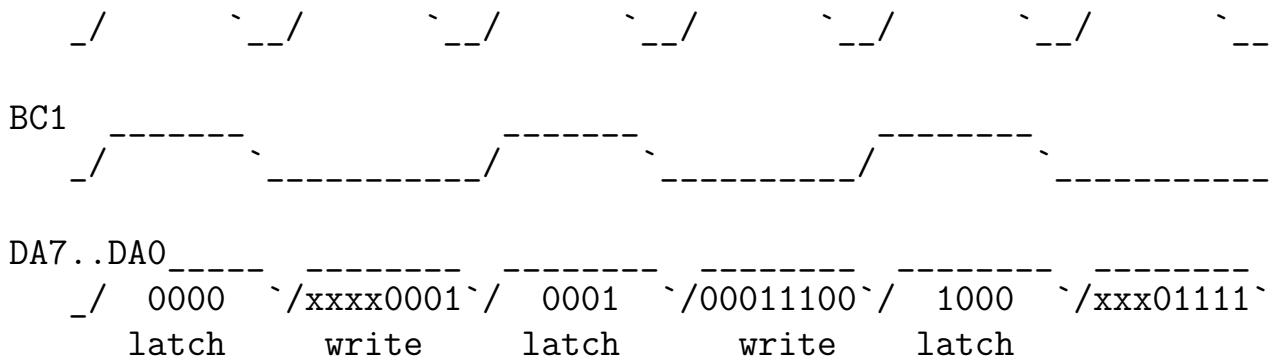
$$toneperiod_{cycles} = 2000000Hz/(16_{cycles} * 440Hz) = 284 = 11C_{hex}$$

An example to play a note at a maximum volume

BDIR	BC1	DA7..DA0	Explanation
1	1	xxxx0000	Latch tone A coarse register address 0 = 0000 _{bin}
1	0	xxxx0001	Write high 4-bits of the 440 Hz note 1 = 0001 _{bin}
1	1	xxxx0001	Latch tone A fine register address 1 _{dec} = 0001 _{bin}
1	0	00011100	Write low 8-bits of the note 1C _{hex} = 00011100 _{bin}
1	1	xxxx1000	Latch channel A volume register address 8 = 1000 _{bin}
1	0	xxx01111	Write maximum volume level 15 _{dec} = 1111 _{bin} with the envelope disabled

Timing diagram





Externally configurable clock divider

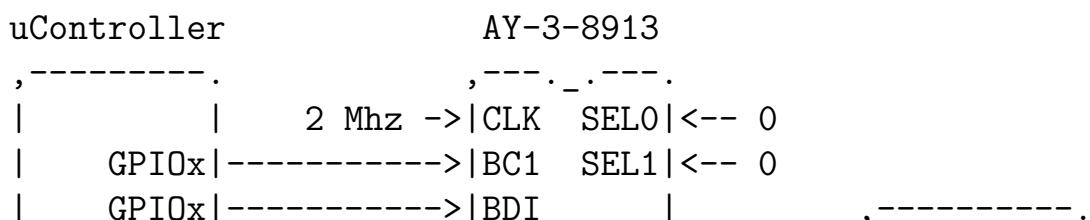
SEL1	SEL0	Description	Clock frequency
0	0	Standard mode, clock divided by 8	1.7 .. 2.0 MHz
1	1	—//—	1.7 .. 2.0 MHz
0	1	New mode for TT05, no clock divider	250 .. 500 kHz
1	0	New mode for TT05, clock div. 128	25 .. 50 MHz

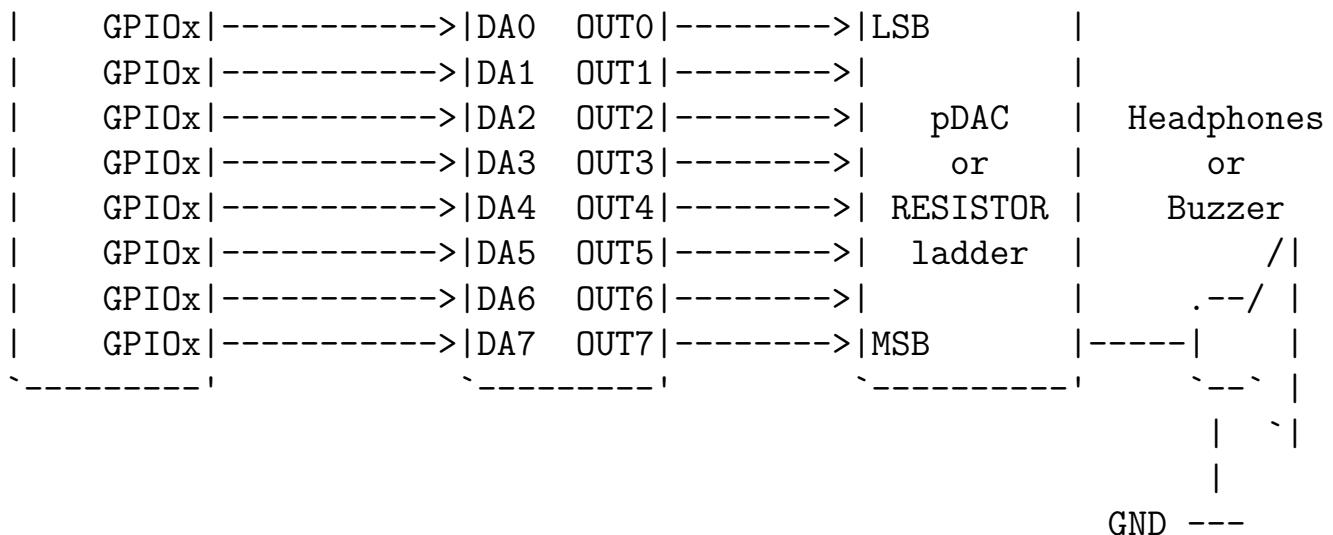
SEL1	SEL0	Formula to calculate the 12-bit tone period value for a note
0	0	$clock_frequency / (16_{cycles} * note_{frequency})$
1	1	—//—
0	1	$clock_frequency / (2_{cycles} * note_{frequency})$
1	0	$clock_frequency / (128_{cycles} * note_{frequency})$

External hardware

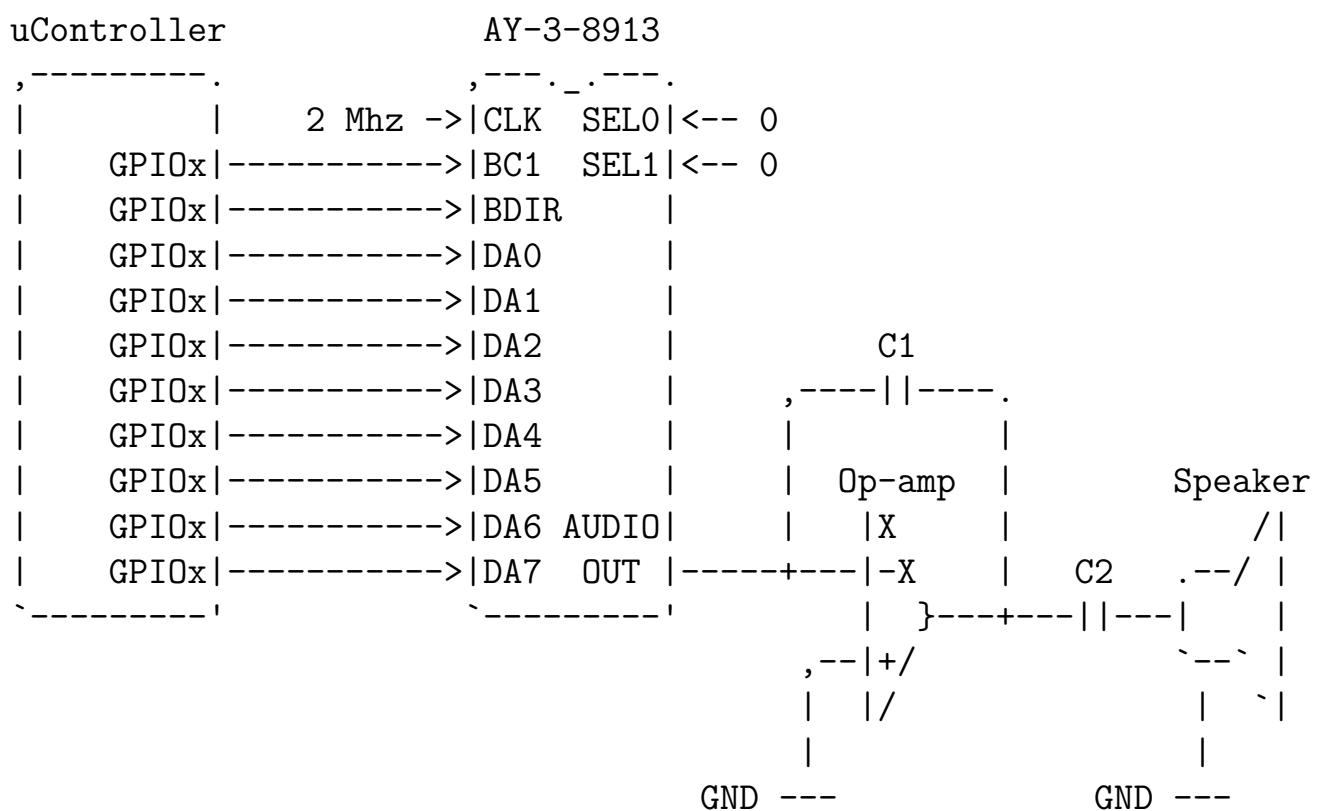
The data bus of the AY-3-8913 chip has to be connected to microcontroller and receive a regular stream of commands. The AY-3-8913 produces audio output and has to be connected to a speaker. There are several ways how the overall schematics can be established.

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

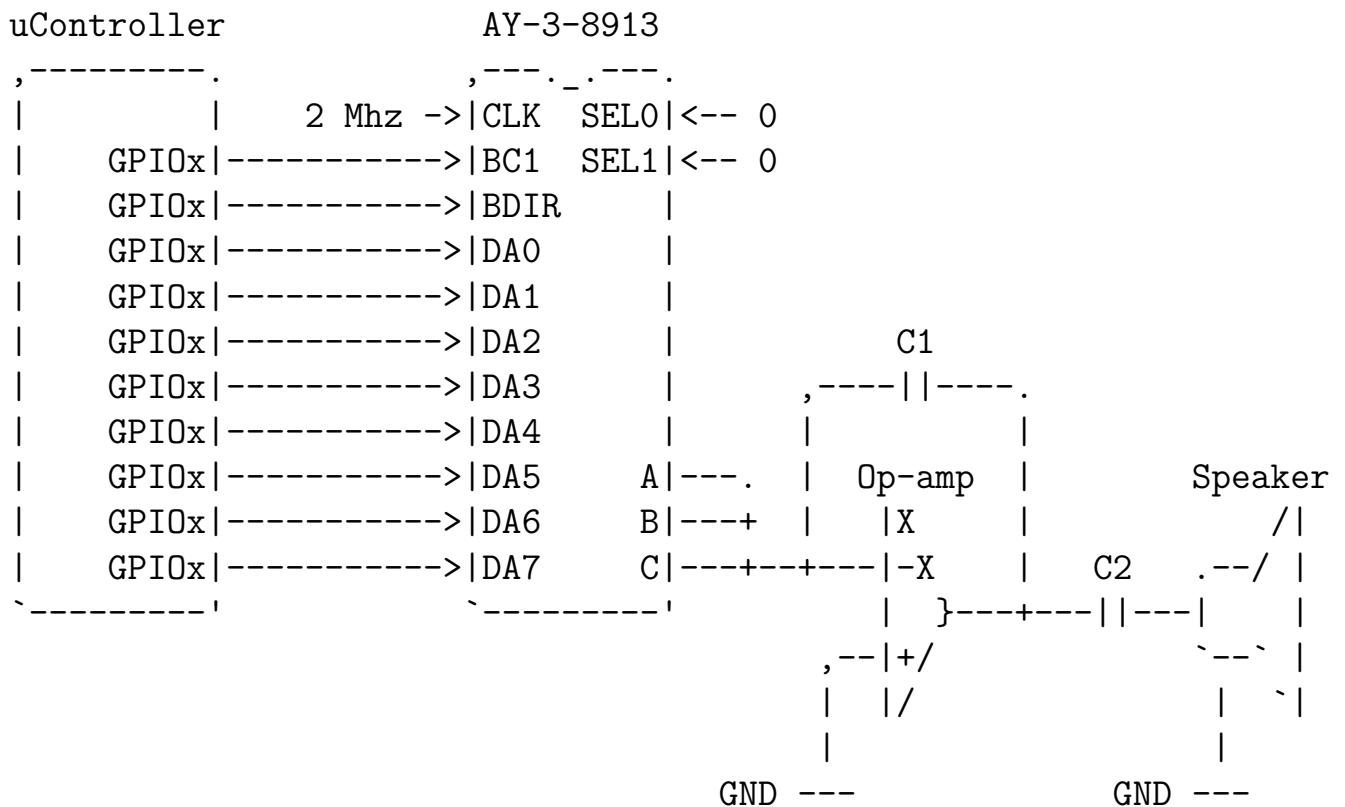




AUDIO OUT through RC filter Another option is to use the Pulse Width Modulated (PWM) AUDIO OUT pin that combines 4 channels with the Resistor-Capacitor based low-pass filter or better the Operation Amplifier (Op-amp) & Capacitor based integrator:



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



Pinout

#	Input	Output	Bidirectional
0	DA0 - multiplexed data/address bus LSB	audio out (PWM)	(in) BC1 bus control
1	DA1 - multiplexed data/address bus	digital audio LSB	(in) BDIR bus direction
2	DA2 - multiplexed data/address bus	digital audio	(in) SEL0 clock divider
3	DA3 - multiplexed data/address bus	digital audio	(in) SEL1 clock divider
4	DA4 - multiplexed data/address bus	digital audio	(out) channel A (PWM)
5	DA5 - multiplexed data/address bus	digital audio	(out) channel B (PWM)
6	DA6 - multiplexed data/address bus	digital audio	(out) channel C (PWM)
7	DA7 - multiplexed data/address bus MSB	digital audio MSB	(out) AUDIO OUT master

Orion Iron Ion [TT10 demo competition] [710]

- Author: Toivo Henningsson
- Description: My contribution to the TT10 demo competition
- GitHub repository
- HDL project
- Mux address: 710
- Extra docs
- Clock: 50400000 Hz

Intro

Curly / Medieval presents



my contribution to the Tiny Tapeout 10 demo competition (which unfortunately got cancelled due to the Efabless shutdown). Code, graphics, and music by Curly (Toivo Henningsson) of Medieval.

The demo was originally written for the Sky130 process (see the sky130 branch), but was updated to the IHP process before submission.

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

How it works

The demo code contains a few different parts, mainly:

- Logo
- Star field
- Floating point unit; used for the twister and spiralling balls
- Synthesizer and sequencer

The video output is produced in VGA mode 640x480 @60 Hz. The logic is clocked at 50.4 MHz, giving two clock cycles per pixel.

Logo Just like in <https://github.com/toivoh/tt08-demo>, the logo is made of big pixels (32x32 this time vs 16x16 before), where each big pixel is split along the diagonal into two triangles. The title is chosen so that the logo should hopefully compress well (and also alludes to the space theme and the echo feeling in the music). The dimensions of the characters are chosen so that patterns should repeat at a scale of 4 pixels, to try to make them more visible to the tool's logic minimizer.

Star field The star field keeps track of the current x and y coordinates with two extra subpixel bits of precision, increasing x or y by up to 4 units in each step. This is to allow the antialiasing effect where stars can move a fraction of a pixel per frame; since the output is RGB222, 2 subpixel position bits are enough. The size of the step to take to the next x or y value is taken from a coarse table that is made so that the resulting x and y curve should be similar to turning in space. The table entries have higher precision, so that they can cause a dithering between e.g. increments of 2 and 3 or 3 and 4.

The distribution of stars is computed using a pseudorandom function that depends on the current x and y value. This is inspired by the pseudorandom function used in the synth (see below). The algorithm can be described as

```
{r0, dr} = bitshuffle({jx, jy}, pattern1)
r = r0
for i=1:2
    r = bitshuffle(r, pattern2) + dr
    r = bitshuffle(r, pattern3)

intensity = r[5:4]
if intensity == 0: intensity = 1
if r[3:0] != 0: intensity = 0
```

where ix, iy are input bits for the current position where we should determine if there is a star, and its brightness. bitshuffle(x, pattern) permutes the bits of x using the fixed pattern. This can be done with only wires, so should be cheap. The combination of addition and bit shuffling means that the effects of input bits propagate in a quite unpredictable way to affect the output bits, creating a pseudorandom behavior. To find good star patterns, I simulated random bit shuffles with a script until I found a pattern that I liked.

The input to the random algorithm is not the full pixel positions ix and iy, but $jx = ix > 2$ and $jy = iy > 1$. Only every fourth

x pixel position and every second y pixel position can hold a star, with black in between. This allows the random algorithm time to converge before the star's intensity is needed.

A typical way to do the antialiasing of the stars would be to calculate numbers px and py that described how well the star aligns with the current x and y position, and use px*py to modulate the intensity. To save on logic, $\min(px, py)$ is used instead. A small lookup table is then used to calculate the 2 bit pixel value from $\min(px, py)$ and the star's intensity.

Floating point unit - twister and spiralling balls effects The demo contains a small floating point unit for approximate floating point calculations, with 5 exponent bits and 11 mantissa bits. The FPU implements addition and subtraction in a similar way to most FPUs. It also supports approximate operations for multiplication and square root (approximate division could be supported as well, but wasn't needed). The approximate operations assume that the concatenation of exponent and mantissa bits are a logarithmic representation of the floating point number: multiply adds {exponent, mantissa}, while square root shifts it right by one. The result is quite inaccurate, but good enough for the computations needed, and contributes some interesting jaggedness to the twister. These cheap approximations of multiply and square root were the motivation to use an FPU in the first place.

Conversion from fixed point to floating point is done by creating a floating point number with a fixed exponent and placing the fixed point number (with its sign bit inverted) in the mantissa, which produces a floating point number representation of `fixed_point_number+bias`. The bias is then subtracted. To convert in the other direction, the bias is added to a floating point number, and the fixed point result is read out from the mantissa.

The FPU code uses the approximation $\cos(0.5\pi*t) = 1 - t^2$, $abs(t) \leq 1$. There didn't seem to be a point in making a more accurate representation given the inaccuracy of the multiplication.

The ALU has a single accumulator register and 5 general purpose registers. One of the inputs to the ALU is always the accumulator, while the other can come from a register, constant, or time varying fixed point value from the outside. The result is always written to the accumulator, and the value in the accumulator can then be written to one of the other registers.

Twister and spiral of balls effects These are implemented by running a short FPU program during horizontal blanking before each scan line starts (different programs for different effects). The program computes up to five x positions for the scan line, which are used to draw horizontal spans of light and dark blue. The x positions share

space with the mantissas in the FPU registers. The programs are written in such a way that they use fewer and fewer FPU registers as they are being overwritten with x positions.

Breaking down FPU instructions into cycles To save area, each FPU instruction is broken into up to 3 single-cycle micro-operations:

- Add/sub:
 - Determine which argument has the largest magnitude
 - Add/Subtract
 - Normalize, calculating the correct exponent and shifting the mantissa to the right position
- Multiply:
 - Calculate carry out from mantissa sum
 - Add {exponent, mantissa}
- Single cycle instructions:
 - Load
 - Square root

The FPU code is stored as instructions rather than micro-operations, which should save some area. One instruction is executed every 4 cycles, which lets the program code be indexed by a running timer.

The logic that represents the program ROM for the FPU has quite high latency. A multicycle timing constraint is used to allow data paths that go through the program ROM to take two cycles. This means that no micro-operation is executed until the second cycle of running each instruction. (The multicycle constraint turned out not to be needed for the IHP version, and was removed.)

Synthesizer The synthesizer produces output samples at 63 kHz, 10 bit resolution. This gives it 800 cycles (half a scan line) per sample, and the usable ouput range of PWM values is 0 - 800. One voice sample is calculated in 64 cycles, which gives time to calculate 12 voices at the same time, plus a little time to update for the next sample. On average, the voices need to have a peak amplitude ≤ 64 steps to fit into the output range; one step per cycle.

The voices are used as follows:

- 4x2 melody/harmony voices: 4 channels with 2 voices per channel with detuning to get a fatter sound

- The frequency is slightly higher for one of the voices in each pair than the other
- Prenoise: the pedal tone with rhythmically changing timbre that runs throughout most of the demo
- Bass drum
- Hihat
- Visualization voice - not heard, used to produce the visible waveforms on screen
 - Can calculate two waveforms per scan line

Aliasing considerations The waveforms are designed to keep aliasing artifacts relatively low:

- The melody/harmony waveforms use piecewise linear sections without a too steep slope, and avoid slopes of less than one unit per sample, which keeps aliasing down.
- The bass drum has a similar approach, using a clipped triangle wave that gradually sinks in frequency.
- The prenoise waveform is kept at a power of 2 frequency, since it would be hard to antialias. The music has been written around this limitation, with the prenoise as a pedal tone/ostinato.
 - Initially, the pedal tone is the tonic note.
 - After the music modulates down by a fifth towards the end, the pedal tone is now the fifth instead.
- The hihat is pure noise and doesn't need any antialiasing considerations.

To gradually reduce the volume of each voice, it is clamped to a decreasing maximum amplitude. This simple method changes the waveform as the volume reduces, but keeps the slopes in their original range. If the volume had been reduced by multiplication, increasing aliasing artifacts would result as the effective range gets reduced.

ALU The synthesizer is based around a small ALU, with a small set of registers

- 11 bit accumulator
- 10 bit output accumulator
- 10 bit output register
- 23 bit oscillator divided into low and high 11 bit parts plus top bit
- two flag bits (predicates)

The oscillator is used to calculate the phase of the waveforms, keep track of time in the demo, index the notes for the music and to know which frame to display.

Calculating voice phases from the shared oscillator There are no registers to keep track of the phases of different synth voices. Instead, for each sample of a voice that needs to be computed, the first 30 cycles are used to compute $\text{phase} = (\text{freq} * \text{osc}) \gg n$ to produce an 11 bit phase in the accumulator. The bits above 11 are truncated, since the waveform repeats after one phase. freq varies between 256 and 511 to choose a note, while n selects the octave.

The product $\text{phase} = (\text{freq} * \text{osc}) \gg n$ is calculated using shifts and adds (at most one of each per cycle), discarding low order bits when they are not needed anymore, and high order bits that will not be needed. To illustrate the method, say that we want to calculate bits 10:3 of the product of an 8 bit and an 11 bit number. The calculation can be visualized as follows:

```

*****
*****
*****
*****
?*****
??*****
???
???
????*
?????**
+ ?????
-----
= ??????*****???

```

We have 11 product terms to add up, each with 8 bits.

- We proceed from the smallest term, adding up terms.
- In the first phase, the bottom bit in the current sum is not needed in the final output, so it can be dropped since no later term will change it (they are all shifted further to the left).
- In the second phase, we are out of bottom bits to ignore, and we can instead start to ignore top bits in the terms, since they are above the range of bits needed in the result.
- By changing the number of steps in the first phase, we can change the shift amount n.

This way, we can use the same number of bits to store each intermediate result as is needed to store the final result. The implementation proceeds by shifting the intermediate result right by one for each step, switching to rotate right when coming to the

second phase (to preserve the bits that are rotated out). When all relevant terms have been added, the result will have been rotated back to the correct position.

Evaluating waveforms using a single accumulator as intermediate storage

The synth ALU has only a single intermediate register to work with, the accumulator (to save area). To evaluate a piecewise linear function (which the melodic/harmonic waveforms are made of), it first evaluates one or several conditions on the current accumulator value to know which piece of the piecewise linear function that it should evaluate, storing the results in the predicates. Then, it can use the predicate values to choose how to transform the accumulator.

Melody/harmony voices There are two waveforms used for the melody/harmony voices: saw like and pulse like. An ideal sawtooth wave includes a sudden jump every period, and an ideal pulse wave contains two. To simulate a gradually closing lowpass filter, these jumps have been changed to ramps. The slope of the ramps is gradually decreased as a note ages.

The pulse like waveform is also uses pulse width modulation by a triangle wave, which is added to the intermediate phase after it itself has been made into a triangle wave as a step in the waveform computation.

Bass drum The bass drum approximates a clamped triangle wave with exponentially decreasing frequency. This was a bit challenging to implement, since there is no register to store the bass drum's phase between samples, instead it has to be recalculated at each sample from the linearly increasing oscillator.

The bass drum uses a variation on the multiplication algorithm, takeing the lower bits of the oscillator and calculating an approximate square. Let x be the relevant oscillator bits in fixed point. As x goes from 0 to one, the function

$$y = (2-x)^2 \bmod 1$$

wraps around 3 times, with the slope at the end being half of the slope at the beginning. Several such quadratic sections are shifted into decreasing octaves to give an approximation of an exponentially decaying frequency. The square approximation is calculated using a variation of the multiplication algorithm described above. It uses only 8 slopes per octave, which seems to be barely enough to give the impression of a continuously descending pitch, but allows the algorithm to use the top 11 bits of the oscillator as one of its inputs.

The bass drum phase is used to evaluate a triangle wave, which is clamped to gradually reduce its volume.

Hihat The hihat is created by noise clamped to a decreasing amplitude. This kind of noise is traditionally created with a Linear Feedback Shift Register (LFSR), but that would have required space for registers to hold the LFSR state. Instead, a new noise value is calculated for each sample based on the oscillator, using the algorithm

```
acc = osc_low
for i=0:3
    acc = bitshuffle(acc, pattern) + osc_high
    acc = acc + (acc >> 1)
```

The pattern used for shuffling is fixed, all it needs is an 11-bit wide multiplexer. After four iterations, the result sounds like noise.

Prenoise pedal tone The prenoise waveform used as a pedal tone is computed using a simplification of the noise algorithm above, with only one iteration:

```
acc = <selected bits from osc>
acc = bitshuffle(acc, pattern)
acc = acc + rotate_right(acc, 1) # could use acc *= 3 instead
```

The permutation used in the bitshuffle step is the same as in the noise case above, and has been chosen for the sonic results it produces in the prenoise case. The result is a waveform with a power of 2 period that changes timbre in a rhythmic manner.

Visualization voice The visualization voice can evaluate the waveform from any of the other voices, using the current scanline's y position instead of the oscillator, to keep the waveform steady from frame to frame. Two waveforms can be evaluated per scan line, one to be shown on the left side of the screen and one on the right side. The synth is synchronized with the display output so that a new visualization waveform sample is computed in the middle of each scan line and one in hblank. There are 3 registers to store the evaluated waveforms, to keep track of two waveforms and have access to the value from the previous scanline when a waveform is displayed.

Sequencer The notes to play are taken from logic that represents a note ROM, and it has quite high latency. Out of the 64 cycles used to compute each voice sample, the first cycle is used to wait for the output from the note ROM to stabilize, and the synthesizer doesn't do anything with the note data until the second cycle. This is accomplished with a multicycle timing constraint. (The multicycle constraint turned out not to be needed for the IHP version, and was removed.)

The note ROM contains data for 4 channels. For each channel and time position pos, it outputs an enable flag, a note value (note and octave), and an age value t0. The actual age is computed as $t = t0 + pos$ (with wraparound). This allows t0 to be piecewise constant in the note ROM. The age t is filled out with low order bits from the oscillator, and used to modulate the waveform and volume of notes as they age. Weaker notes can be achieved by starting them at a higher age.

How to test

Plug in a TinyVGA compatible Pmod on the TT08 demo board's out Pmod. Plug in a Pmod compatible with Mike's audio Pmod on the TT08 demo board's bidir Pmod. Set all inputs to zero to get the default behavior. The demo starts directly after reset.

External hardware

This project needs

- a TinyVGA VGA Pmod.
- Mike's audio Pmod.

Pinout

#	Input	Output	Bidirectional
0		R1	
1		G1	
2		B1	
3		vsync	
4		R0	
5		G0	
6	advance[0]	B0	
7	advance1	hsync	audio_out

My Project [712]

- Author: Asger Wenneberg
- Description: This is a tiny tapeout design
- GitHub repository
- HDL project
- Mux address: 712
- Extra docs
- Clock: 0 Hz

How it works

This is how it works

How to test

This is how to test

External hardware

Nothing yet

Pinout

#	Input	Output	Bidirectional
0	ui_in[0]	uo_out[0]	ui_out[0]
1	ui_in1	uo_out1	ui_out1
2	ui_in2	uo_out2	ui_out2
3	ui_in[3]	uo_out[3]	ui_out[3]
4	ui_in[4]	uo_out[4]	ui_out[4]
5	ui_in[5]	uo_out[5]	ui_out[5]
6	ui_in[6]	uo_out[6]	ui_out[6]
7	ui_in[7]	uo_out[7]	ui_out[7]

simple-viii [714]

- Author: strau
- Description: A simple 8-bit CPU Architecture
- GitHub repository
- HDL project
- Mux address: 714
- Extra docs
- Clock: 10000 Hz

How it works

How to test

External hardware

Pinout

#	Input	Output	Bidirectional
0			cs flash
1			SD0
2			SD1
3			SCK
4			SD2
5			SD3
6			cs ram
7			

ttUART [716]

- Author: Bogdan Tanasa
- Description: UART
- GitHub repository
- HDL project
- Mux address: 716
- Extra docs
- Clock: 50000000 Hz

How it works

This is a UART repeater. It mirrors the Rx on Tx.

How to test

It can be used as a regular UART.

External hardware

No external hardware is needed.

Pinout

#	Input	Output	Bidirectional
0	rx_data_in	rx_data_ready	
1		rx_sample_clk	
2		tx_data_out	
3		tx_data_done	
4			
5			
6			
7			

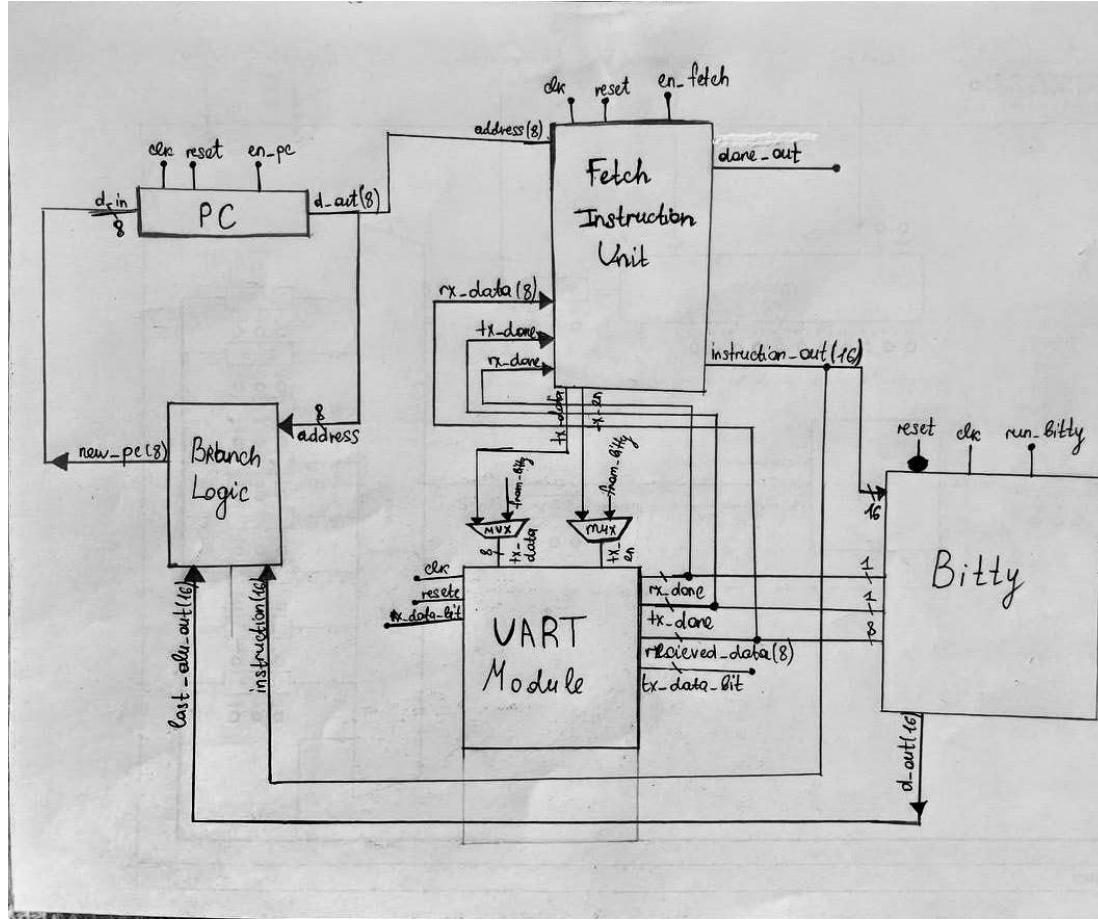
Bitty [718]

- Author: Moldir
- Description: 16-bit simple processor
- GitHub repository
- HDL project
- Mux address: 718
- Extra docs
- Clock: 10 Hz

Bitty System: RTL Design and Verification Framework

This project implements a custom 16-bit processing system, including hardware modules for program counter (PC), instruction fetch, branch logic, UART communication, and integration with the **BittyEmulator** for co-simulation. The provided system allows robust testing of a Verilog-based design using a Python-based cocotb testbench. The testbench orchestrates data transfer via UART, interacts with shared memory, and verifies execution against the emulator.

System Overview



Core Components

1. Program Counter (PC):

- Handles sequential and branch-based instruction execution.
- Interfaces directly with the branch logic for control flow changes.

2. Instruction Fetch Unit:

- Reads and decodes instructions from memory.
- Supplies data to the rest of the system.

3. Branch Logic:

- Evaluates branch conditions and modifies the PC as needed.

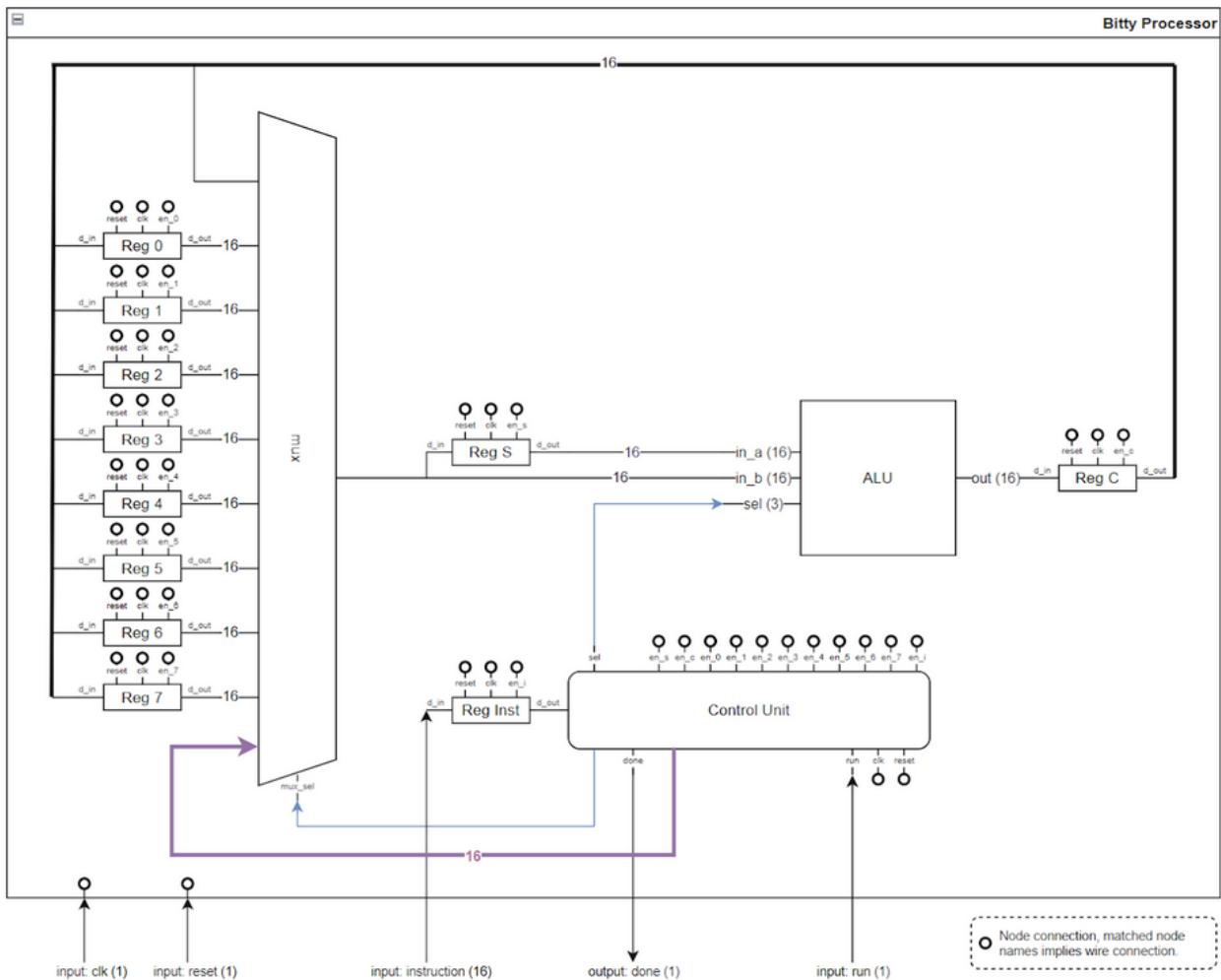
4. UART Module:

- Supports data exchange between the testbench and the DUT.
- Operates with customizable baud rates and clock frequencies.

5. Bitty Emulator:

- Acts as a functional reference model.
- Validates the outputs and internal states of the hardware implementation.

- Includes: Control Unit, registers, ALU, mux



Memory Map

- Shared Memory:

- Synchronizes data between the testbench, the hardware design (DUT), and the emulator.
- Supports up to 256 entries.

- Instruction Set:

- Defined in `instructions_for_em.txt`, loaded by the testbench for execution. Here's the revised version written as a description of a fully implemented system:

Instruction Set Architecture: Fully Implemented 16-bit Processor

Overview This document outlines the complete instruction set architecture (ISA) for a 16-bit processor, detailing its capabilities, operations, and encoding formats. The ISA is designed to deliver robust functionality for arithmetic, logical, control flow, and memory operations while maintaining a simple, efficient structure.

The processor's instruction set enables dynamic memory interactions, conditional branching, and a wide range of data manipulation tasks, providing the foundation for executing complex algorithms and software applications.

Instruction Set

Arithmetic and Logical Operations The processor supports both register-to-register and immediate operations, enabling developers to perform computations efficiently.

Register-to-Register Instructions:

Instruction	ALU Execution Instruction Format															Operation
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Rx	Ry	Reserved				ALU sel		Res.							
add rx, ry	x x x	y y y	r r r	r r r	r r r	r r r	0 0 0	0 0 0	0 0 0	rx = rx + ry						
sub rx, ry	x x x	y y y	r r r	r r r	r r r	r r r	0 0 1	0 0 0	0 0 0	rx = rx - ry						
and rx, ry	x x x	y y y	r r r	r r r	r r r	r r r	0 1 0	0 0 0	0 0 0	rx = bitwise AND of rx and ry						
or rx, ry	x x x	y y y	r r r	r r r	r r r	r r r	0 1 1	0 0 0	0 0 0	rx = bitwise OR of rx and ry						
xor rx, ry	x x x	y y y	r r r	r r r	r r r	r r r	1 0 0	0 0 0	0 0 0	rx = bitwise XOR of rx and ry						
shl rx, ry	x x x	y y y	r r r	r r r	r r r	r r r	1 0 1	0 0 0	0 0 0	rx = rx << ry (shift left rx by ry bits)						
shr rx, ry	x x x	y y y	r r r	r r r	r r r	r r r	1 1 0	0 0 0	0 0 0	rx = rx >> ry (shift right rx by ry bits)						
cmp rx, ry	x x x	y y y	r r r	r r r	r r r	r r r	1 1 1	0 0 0	0 0 0	rx = 0 iff rx == ry or rx = 1 iff rx > ry or rx = 2 iff rx < ry						

1. **add rx, ry**: Adds the value in ry to rx.
 - **Operation**: $rx = rx + ry$
2. **sub rx, ry**: Subtracts the value in ry from rx.
 - **Operation**: $rx = rx - ry$
3. **and rx, ry**: Performs a bitwise AND between rx and ry.
 - **Operation**: $rx = rx \&amp; ry$
4. **or rx, ry**: Performs a bitwise OR between rx and ry.

- **Operation:** $rx = rx \mid ry$

5. **xor rx, ry:** Performs a bitwise XOR between rx and ry .

- **Operation:** $rx = rx \wedge ry$

6. **shl rx, ry:** Shifts the bits in rx left by the number of positions specified in ry .

- **Operation:** $rx = rx \ll ry$

7. **shr rx, ry:** Shifts the bits in rx right by the number of positions specified in ry .

- **Operation:** $rx = rx \gg ry$

8. **cmp rx, ry:** Compares the values in rx and ry .

- **Operation:**

- $rx = 0$ if $rx == ry$
- $rx = 1$ if $rx > ry$
- $rx = 2$ if $rx < ry$

Immediate Instructions:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction	Rx				Immediate								ALU sel	Format		
addi rx, #i	x	x	x	i	i	i	i	i	i	i	i	0	0	0	0	1

1. **addi rx, #i:** Adds the immediate value $#i$ to rx .

- **Operation:** $rx = rx + #i$

2. **subi rx, #i:** Subtracts the immediate value $#i$ from rx .

- **Operation:** $rx = rx - #i$

3. **andi rx, #i:** Performs a bitwise AND between rx and $#i$.

- **Operation:** $rx = rx \& #i$

4. **ori rx, #i:** Performs a bitwise OR between rx and $#i$.

- **Operation:** $rx = rx | #i$

5. **xori rx, #i:** Performs a bitwise XOR between rx and $#i$.

- **Operation:** $rx = rx \wedge #i$

6. **shli rx, #i**: Shifts rx left by $#i$ positions.
 - **Operation:** $rx = rx \ll \#i$
7. **shri rx, #i**: Shifts rx right by $#i$ positions.
 - **Operation:** $rx = rx \gg \#i$
8. **cmpi rx, #i**: Compares the value in rx with the immediate value $#i$.
 - **Operation:**
 - $rx = 0$ if $rx == #i$
 - $rx = 1$ if $rx > #i$
 - $rx = 2$ if $rx < #i$

Memory Operations

	15	14	13	12	11	10	9	8	7	6	5	4	3	
Instruction	Rx			Ry			Reserved							
ld rx, (ry)	x	x	x	y	y	y	r	r	r	r	r	r	r	
st rx, (ry)	x	x	x	y	y	y	r	r	r	r	r	r	r	

Load and Store Instructions:

1. **ld rx, (ry)**: Loads the value from the memory address stored in ry into register rx .
 - **Operation:** $rx = \text{mem}[ry]$
2. **st rx, (ry)**: Stores the value in register rx into the memory address stored in ry .
 - **Operation:** $\text{mem}[ry] = rx$

Encoding Format:

- **Bits 15-13 (Rx)**: Destination register for ld or source register for st.
 - **Bits 12-10 (Ry)**: Register holding the memory address.
 - **Bits 9-3 (Reserved)**: Reserved for future extensions, currently set to zero.
 - **Bit 2 (L/S)**: Load/Store flag (0 for ld, 1 for st).
 - **Bits 1-0**: Instruction format identifier (11 for memory operations).
-

Conditional Branching The processor supports conditional branching with a dedicated encoding format for efficient control flow.

Instruction	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	Immediate												Cond	Format			
bie #i	i	i	i	i	i	i	i	i	i	i	i	i	0	0	1	0	Branch if EQ
big #i	i	i	i	i	i	i	i	i	i	i	i	i	0	1	1	0	Branch if GT
bil #i	i	i	i	i	i	i	i	i	i	i	i	i	1	0	1	0	Branch if LT

Conditional Branch Instructions:

1. **bie addr**: Branch if equal (condition flag EQ is set).
2. **big addr**: Branch if greater (condition flag GT is set).
3. **bil addr**: Branch if less (condition flag LT is set).

Encoding Format:

- **Bits 15-4 (Immediate)**: Encodes the branch target address.
 - **Bits 3-2 (Condition)**:
 - 00: Equal
 - 01: Greater than
 - 10: Less than
 - **Bits 1-0 (Format)**: Instruction format identifier (10 for conditional branching).
-

Here's a detailed step-by-step guide for users to set up and test their system with the assembler and testbench:

How to Use the System Before running the testbench, you must first prepare the assembly instructions or machine code. Here's how:

Step 1: Prepare Instructions

1. **Option 1:** Generate machine code automatically

Run the CIG_run.py script to generate output.txt automatically with pre-defined assembly instructions.

```
python3 CIG_run.py
```

This will create output.txt containing machine code.

2. **Option 2:** Write custom assembly instructions

If you prefer to write your own instructions, directly create or modify the output.txt file. These instructions will later be disassembled for further testing.

Step 2: Disassemble Machine Code

Disassemble the output.txt file (machine code) to generate instructions_for_em.txt (assembly code):

```
./er_tool -d -i output.txt -o instructions_for_em.txt
```

This step ensures that the instructions in instructions_for_em.txt are ready for use in the testbench.

Running the Testbench Once you have the instructions_for_em.txt file ready, navigate to the bitty-tt10/test directory and execute the testbench using make:

```
cd ~/bitty-tt10/test  
make
```

The testbench will:

1. Load the instructions from instructions_for_em.txt.
 2. Simulate UART communication for instruction execution.
 3. Compare the outputs of the DUT (Device Under Test) with expected results.
 4. Log the results, including any discrepancies, into uart_emulator_log.txt.
-

Testbench Overview Assembling Code

To convert `instructions_for_em.txt` into machine code (if needed for testing):

```
./er_tool -a -i instructions_for_em.txt -o output.txt
```

Disassembling Code

To convert machine code (`output.txt`) back into assembly:

```
./er_tool -d -i output.txt -o instructions_for_em.txt
```

Practical Workflow Example

1. Generate Machine Code:

Run `CIG_run.py` to create machine code:

```
python3 CIG_run.py
```

2. Disassemble Code:

Use the `er_tool` to create `instructions_for_em.txt`:

```
./er_tool -d -i output.txt -o instructions_for_em.txt
```

3. Run Testbench:

Navigate to the test directory and run the testbench:

```
cd ~/bitty-tt10/test  
make
```

Key Features of the Testbench

- **Simulated UART Communication:** Generates UART signals and captures DUT transmissions.
 - **Instruction Execution:** Fetches and executes instructions in real-time.
 - **State Validation:** Logs and compares DUT outputs with expected results.
 - **Error Reporting:** Logs any mismatches in `uart_emulator_log.txt`.
-

Following these steps ensures smooth operation from writing or generating instructions to verifying the system's functionality. If you encounter issues, double-check the prepared files or logs for guidance. Let me know if you need further clarification!

How to Use

Setup

1. Prerequisites:

- Install Python and cocotb.
- Ensure Verilog simulation tools (Verilator, Iverilog) are installed.
- Use the following command to install the dependencies:

```
pip install -r requirements.txt
```

2. Input Files:

- Place the instruction file (`instructions_for_em.txt`) in the working directory.
- Modify the file as needed to test specific scenarios.

3. Shared Libraries:

- Ensure `BittyEmulator.py` and `shared_memory.py` are in the project directory.

Running the Test

1. Execute the cocotb testbench:

```
make
```

2. Observe the test results in the terminal and logs:

- Successes and failures are detailed in `uart_emulator_log.txt`.
-

External Hardware This system does not require external hardware. UART communication is emulated within the testbench.

Files Overview

Verilog Files

- `<module_name>.v`: Contains the RTL design files for the system.
- `tb_<module_name>.v`: Top-level Verilog testbench wrapper.

Python Files

- `test_bitty.py`: The cocotb testbench described above.
 - `BittyEmulator.py`: Emulator for reference model validation.
 - `shared_memory.py`: Utility for creating shared memory structures.
-

Limitations and Future Work

1. Hardware Expansion:

- Current implementation is limited to basic arithmetic and control operations.
- Future iterations could incorporate advanced features like pipelining or caching.

2. Error Handling:

- Expand error reporting for unresolved signals during simulation.

3. Scalability:

- Extend memory and instruction sets for larger programs.
-

This project demonstrates a robust framework for RTL verification, combining software co-simulation with hardware modeling for high-fidelity testing and validation.

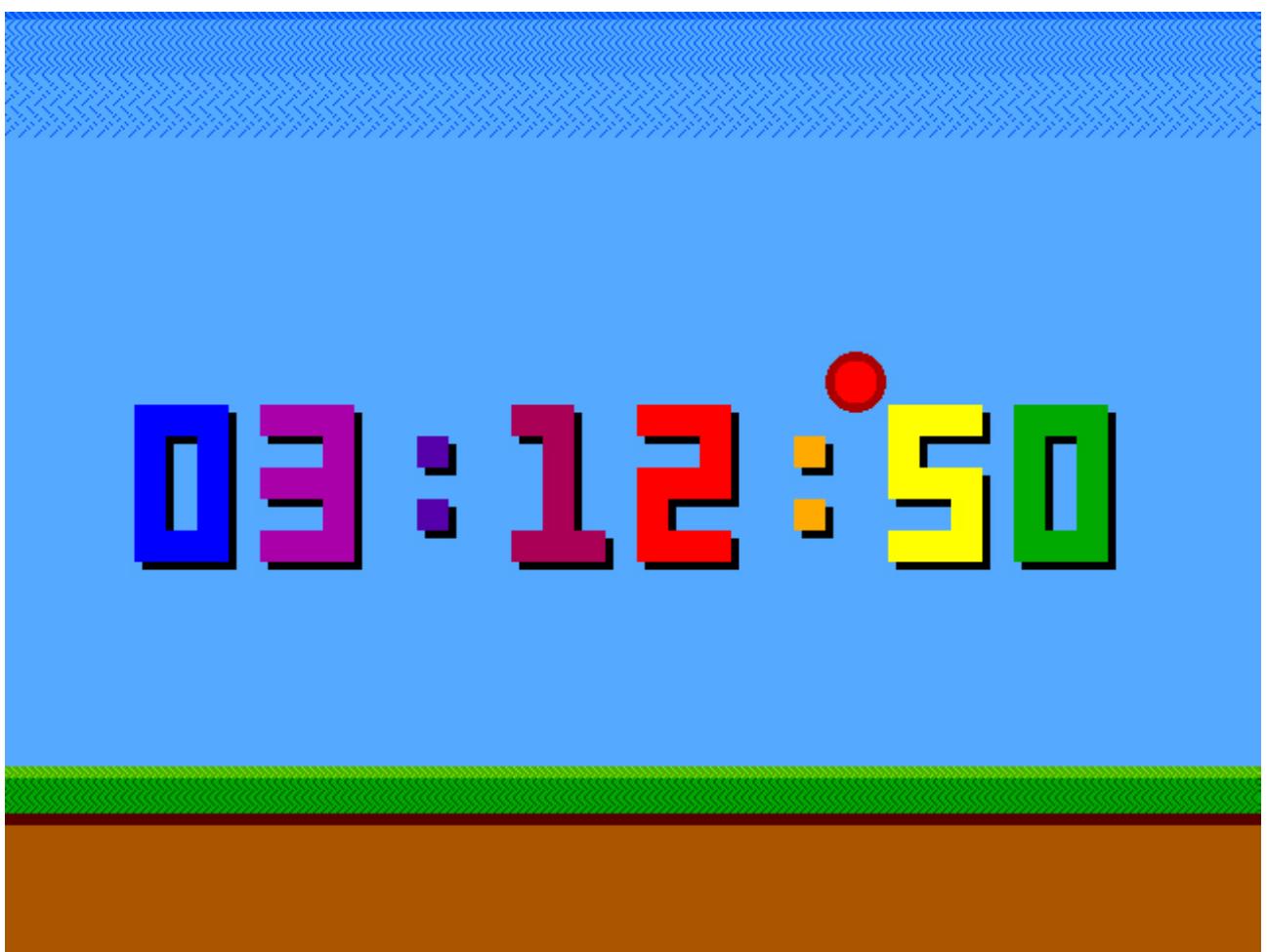
Pinout

#	Input	Output	Bidirectional
0	rx_data_bit	tx_data_bit	
1	sel_baude_rate[0]		
2	sel_baude_rate1		
3			
4			
5			
6			
7			

IHP VGA demo [720]

- Author: algofoogle (Anton Maurovic)
- Description: Simple VGA demo for IHP tapeout (inc. Matt Venn's VGA clock)
- GitHub repository
- HDL project
- Mux address: 720
- Extra docs
- Clock: 25000000 Hz

How it works



Typical Verilog design that generates VGA timing and RGB222 colour outputs compatible with the Tiny VGA PMOD.

It produces a bouncing ball animation over the top of an adaptation of Matt Venn's VGA clock, from here: <https://github.com/mattvenn/tt08-vga-clock>

How to test

- Plug in a VGA monitor via Tiny VGA PMOD.
- Set mode input to 0, i.e. specifying 640x480 60Hz from a 25MHz clock.
- Set show_clock input to 1.
- Set pmod_select input to 0 for Tiny VGA PMOD. Otherwise, 1=Matt's VGA Clock PMOD.
- Supply a 25MHz clock (clock's actual seconds timer assumes exactly 25.000MHz).
- Assert reset.
- Pulse or hold the adj_* inputs to adjust hours, minutes, or seconds.

External hardware

Tiny VGA PMOD and VGA monitor is all you should need externally.

Pinout

#	Input	Output	Bidirectional
0	adj_hrs	r1	hmax
1	adj_min	g1	vmax
2	adj_sec	b1	hblank
3		vsync	vblank
4		r0	visible
5		g0	
6		b0	
7	mode	hsync	

UW ASIC - Optimized Dino [722]

- Author: University of Waterloo ASIC Design Team
- Description: Dino game, but only 2 tiles!
- GitHub repository
- HDL project
- Mux address: 722
- Extra docs
- Clock: 25175000 Hz

How it works

Placeholder

How to test

Placeholder

External hardware

Placeholder

Pinout

#	Input	Output	Bidirectional
0	ui0	uo0	ui0
1	ui1	uo1	ui0
2	ui2	uo2	ui0
3	ui3	uo3	ui0
4	ui4	uo4	ui0
5	ui5	uo5	ui0
6	ui6	uo6	ui0
7	ui7	uo7	ui0

PID Controller [737]

- Author: Kilian Bender
- Description: Hardware implementation of a naive PID Controller
- GitHub repository
- HDL project
- Mux address: 737
- Extra docs
- Clock: 1000000 Hz

How it works

The PID controller module works by continuously adjusting its output based on the difference between the desired value (setpoint) and the measured value (feedback). It does this using three components:

Proportional Term (P): This term corrects the error in proportion to the current difference between the setpoint and the feedback. It applies an immediate response to reduce the error.

Integral Term (I): This term accumulates the past error over time, helping to eliminate any steady-state error that may persist after the proportional correction.

Derivative Term (D): This term predicts future error by observing the rate of change of the current error, thus providing a damping effect to reduce overshooting.

The controller outputs a signal only in the positive direction. That means that we expect a system that naturally tends towards one point. Regarding a application in heating that means that we are not aiming to cool down the system when overshooting or if the setpoint is higher then our feedback but we just output 0 for control.

How to test

Set different values for setpoint and feedback and observe the output in response to it. Change the setpoint to play around.

External hardware

No specific external hardware is required to test the module in a simulation environment. However, for practical deployment, you may need:

Sensor: A sensor to provide the feedback signal, representing the process variable you wish to control.

Actuator: An actuator driven by the control_out signal to affect the process, such as a motor or a heating element.

Pinout

#	Input	Output	Bidirectional
0	setpoint 0	control_signal 0“	feedback 0
1	setpoint 1	control_signal 1	feedback 1
2	setpoint 2	control_signal 2	feedback 2
3	setpoint 3	control_signal 3	feedback 3
4	setpoint 4	control_signal 4	feedback 4
5	setpoint 5	control_signal 5	feedback 5
6	setpoint 6	control_signal 6	feedback 6
7	setpoint 7	control_signal 7	feedback 7

Frequency Counter SSD1306 OLED [739]

- Author: Paweł Sitarz (embelon)
- Description: Simple Frequency Counter displaying result on SSD1306 SPI OLED
- GitHub repository
- HDL project
- Mux address: 739
- Extra docs
- Clock: 1000000 Hz

How it works

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

How to test

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

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

External hardware

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

Pinout

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

#	Input	Output	Bidirectional
7			

Tiny 1-bit AM Radio [741]

- Author: James Ross Sharp
- Description: Synthesizable 1-bit AM radio core
- GitHub repository
- HDL project
- Mux address: 741
- Extra docs
- Clock: 50000000 Hz

How it works

This project is a Software Defined Radio pipeline for AM radio reception written in verilog. It works using an external comparator as a 1-bit ADC frontend. Demodulation is by the digital equivalent of how a crystal radio works, i.e. bandpass filter followed by envelope detector. It is based on this Hackaday Project: <https://hackaday.io/project/170916-fpga-3-r-1-c-mw-and-sw-sdr-receiver> by Alberto Garlassi.

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

This project is different from a previously submitted 3x2 tile tiny tapeout core, which used more conventional SDR techniques. This layout reduces area to 1x2 tiles, with a tradeoff in selectivity.

How to test

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

The core has a SPI interface for setting the demodulation frequency and gain. It consists of a single 24-bit shift register. It has the following format:-

Bit 23	Bits 22 - 20	Bits 19 - 0
Unused	Gain	NCO Phase incr.

The gain can take on the following values:

"Gain" value	Actual Gain
0	x1
1	x2
2	x4
3	x8
4	x16
5 - 7	x32

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

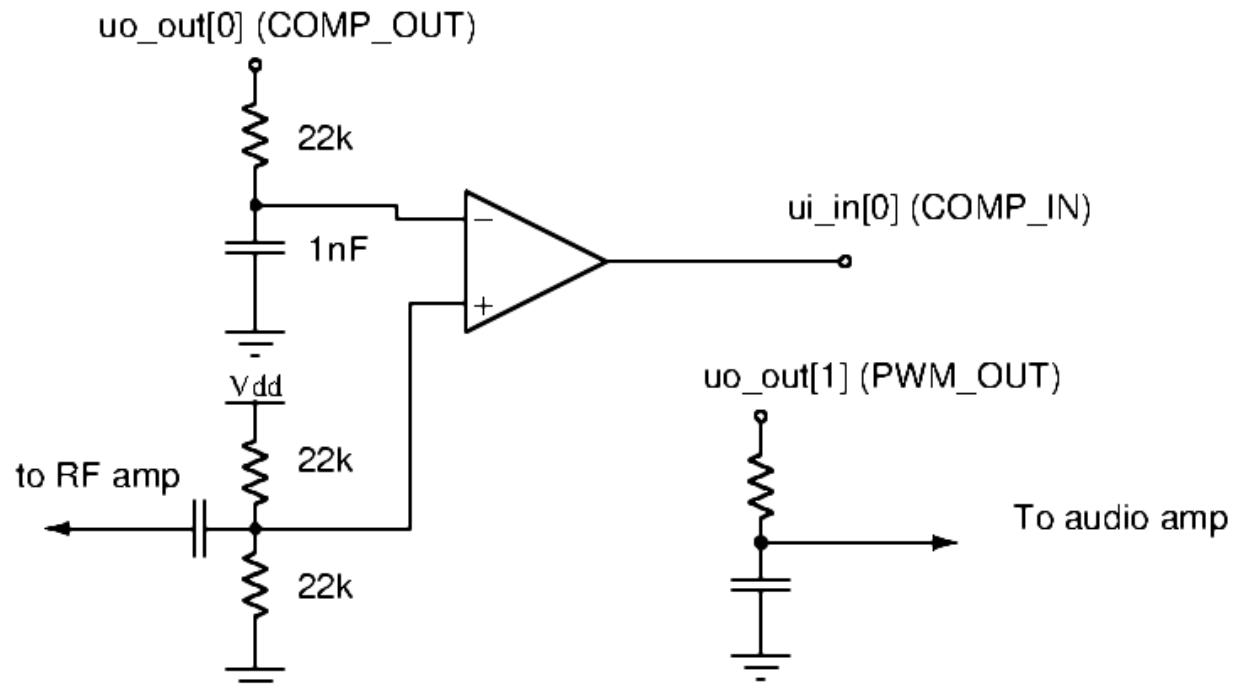
The “NCO Phase increment” is the value that is added to the NCO phase every clock cycle. Use the following python code to calculate the value to write, based on the desired carrier frequency:

```
hex(int((1<<20) * <carrier frequency - (455000/<clock_frequency>*50e6)> /
```

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

```
> hex(int((1<<20) * 936000 / 50000000))
'0x2767'
```

External hardware



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

Pinout

#	Input	Output	Bidirectional
0	COMP_IN	COMP_OUT	
1	SPI_MOSI	PWM	
2	SPI_SCK		
3	SPI_CSb		
4			
5			
6			
7			

FIREngine [743]

- Author: Hao Wang, Andrew Malnicof
- Description: FIR Filter for Audio PMOD
- GitHub repository
- HDL project
- Mux address: 743
- Extra docs
- Clock: 50000000 Hz

How it works

FIREngine is a Digital FIR filter that filters inputs from an I2S2 PMOD ADC and DAC module. The purpose of this design is to filter audio from an I2S2 PMOD device found here: <https://digilent.com/shop/pmod-i2s2-stereo-audio-input-and-output/>. Although the number of taps the filter is not adjustable and must be determined before synthesis, the coefficients of each tap are programmable. This allows for different low, band, and high pass filters to be constructed for multiple audio filtering configurations. It is a parametrizable filter with symmetric or antisymmetric coefficients, odd number of taps. Uses 2s complement and fixed-point data. Coefficients are set via an SDI Interface.

How to test

Use TinyTapeout Demo board to connect PMOD to Tiny Tapeout project, program filter coefficients serially, and experience the results!

External hardware

- I2S2 PMOD device: <https://digilent.com/shop/pmod-i2s2-stereo-audio-input-and-output/>
- Serial programmer

Pinout

#	Input	Output	Bidirectional
0	SPI CS		DAC MCLK
1	SPI MOSI		DAC LRCK

#	Input	Output	Bidirectional
2			DAC SCLK
3	SPI SCLK		DAC Data
4			ADC MCLK
5			ADC LRCK
6			ADC SCLK
7			ADC Data

znah_vga_ca [745]

- Author: Alexander Mordvintsev
- Description: Simple VGA 1D cellular automata generator
- GitHub repository
- HDL project
- Mux address: 745
- Extra docs
- Clock: 25175000 Hz

How it works

VGA signal generator iterates through a number of 1D elementary cellular automata

How to test

Plug and play

External hardware

VGA PMOD

Pinout

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

TRNG [746]

- Author: Muhammad Bilal
- Description: Generate Raw and Hashed Random Numbers
- GitHub repository
- HDL project
- Mux address: 746
- Extra docs
- Clock: 50000000 Hz

How it works

This True Random Number Generator (TRNG) operates by leveraging a noise source sampled by a digital circuit. The sampled data conditioning using SHA-256 to ensure cryptographic-quality randomness. A state machine controls data collection, processing, and output transmission via UART. The TRNG supports two modes: raw entropy output for analysis and hashed output for secure applications. Built-in health tests, such as the Repetition Count Test, verify entropy quality.

How to test

1. Connect a UART terminal to receive random number outputs.
2. Select between raw entropy mode or hashed output mode via control signal. (default is 0 for Hashed output)
3. Monitor the output stream for randomness analysis or cryptographic use.
4. Run statistical tests to validate entropy quality. (Visit [github @ sp 800-90b](#))

External hardware

- ZCU102 FPGA Board
- UART-to-USB Adapter (for serial communication)
- Oscilloscope (for debugging noise source if needed)

Pinout

#	Input	Output	Bidirectional
0	TRNG_Enable	failure	
1	ctrl_mode	hash_rdy	

#	Input	Output	Bidirectional
2		UART_Tx	
3			
4			
5			
6			
7			

CORA-16 [747]

- Author: Andrew Dona-Couch
- Description: Simply 16-bit CPU
- GitHub repository
- HDL project
- Mux address: 747
- 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	Trap execution	----- ----- Drop
		0000 0010		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 ----- ----- 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	----- --##

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
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

T3 (Tiny Ternary Tapeout) CSA [749]

- Author: Arnav Sacheti & Jack Adiletta
- Description: Ternary Matmul Processor using CSA
- GitHub repository
- HDL project
- Mux address: 749
- Extra docs
- Clock: 50000000 Hz

Tiny Ternary Tapeout Project Documentation

Inspiration The inspiration for this Tiny Tapeout project comes from the “Scalable MatMul-free Language Modeling” paper, which explores a novel approach to language modeling that bypasses traditional matrix multiplication (MatMul) operations. Standard neural network models, especially those used for language processing, rely heavily on matrix multiplications to handle complex data transformations. However, these operations can be computationally expensive and power-intensive, especially at large scales.

The key insight of this research is to leverage alternative mathematical structures and sparse representations, reducing the need for resource-heavy MatMul operations while still enabling efficient language processing. By reimagining the model architecture to avoid these multiplications, it opens up possibilities for more energy-efficient, scalable models, particularly in hardware-constrained environments like microchips. This Tiny Tapeout project aims to implement and experiment with these principles on a small scale, designing circuitry that emulates the core ideas of this MatMul-free approach. This can pave the way for more efficient and compact language models in embedded systems, potentially transforming real-time, on-device language processing applications.

How it works The `tt_um_tiny_ternary_tapeout_csa.v` module is designed to perform matrix multiplication using a pipelined architecture. Here's a step-by-step explanation of how it works:

Loading the Weights (`tt_um_load.v`):

The module starts by loading the weights for the matrix. These weights are stored in an internal register array and are used for the matrix multiplication operations.

Matrix Multiplication (`tt_um_mult.v`):

The module performs matrix multiplication by iterating over the columns of the weight matrix and calculating the temporary output values based on the weights and input vectors. For each column, the module multiplies the input vector elements by the corresponding weights and sums the results to produce the output values.

Pipelined Architecture:

The module is pipelined, meaning that it can continuously accept new input vectors while performing computations on the previous inputs. As new inputs are driven into the module, the current computations are completed, and the results are stored in a pipeline register. During the next clock cycle, the outputs are produced as 8-bit integers, allowing for continuous data processing without interruption.

Outputting Results:

After driving all the inputs, the outputs are produced as 8-bit integers. These outputs represent the result of the matrix multiplication operation. By leveraging a pipelined architecture, the `tt_um_mult.v` module ensures efficient and continuous data processing, allowing for high-throughput matrix multiplication operations.

Example: Using a Ternary Array for Efficient Computation In this example, we'll create a 4x2 ternary array and demonstrate how it can be used to process a 1x4 input vector.

Step 1: Define a Ternary Array

A ternary array is one where each element can take on one of three possible values, commonly -1 , 0 , or $+1$. These values simplify calculations because instead of performing complex multiplications, you can use additions, subtractions, or ignore the zero entries altogether.

Let's create a sample 4x2 ternary array:

$$\text{Array} = [+1 \ 0 \ -1 \ +1 \ 0 \ -1 \ +1 \ +1]$$

Step 2: Define the Input Vector

Let's assume we have a 1x4 input vector:

$$\text{Input} = [2 \ -1 \ 3 \ 0]$$

Step 3: Compute the Output without Matrix Multiplication

Instead of performing a matrix multiplication, we'll calculate the output using simpler operations based on the ternary values.

For each column in the ternary array:

- Multiply +1 entries by the corresponding input values.
- Subtract the values for -1 entries.
- Ignore the 0 entries.

Step 4: Calculate Each Column's Output

Let's compute each column separately:

- **Column 1 Calculation:**

- Row 1: ($+1 \times 2 = 2$)
- Row 2: ($-1 \times -1 = +1$)
- Row 3: ($0 \times 3 = 0$)
- Row 4: ($+1 \times 0 = 0$)

$$\text{Sum of Column 1: } (2 + 1 + 0 + 0 = 3)$$

- **Column 2 Calculation:**

- Row 1: ($0 \times 2 = 0$)
- Row 2: ($+1 \times -1 = -1$)
- Row 3: ($-1 \times 3 = -3$)
- Row 4: ($+1 \times 0 = 0$)

$$\text{Sum of Column 2: } (0 - 1 - 3 + 0 = -4)$$

Final Output Vector

Combining the results from each column, we get the final output vector:

$$\text{Output} = [3 \ -4]$$

How to test To test the Matrix Multiplier with an external MCU like a Raspberry Pi, follow these steps:

1. Setup:

- Connect the Raspberry Pi to the Matrix Multiplier hardware using appropriate GPIO pins.
- Ensure that the Raspberry Pi has the necessary libraries installed for GPIO manipulation.

Pinout

#	Input	Output	Bidirectional
0	A1	Q1	B1
1	A2	Q2	B2
2	A3	Q3	B3
3	A4	Q4	B4
4	A5	Q5	B5
5	A6	Q6	B6
6	A7	Q7	B7
7	A8	Q8	B8

Basic Oszilloscope and Signal Generator [751]

- Author: Pascal Gesell
- Description: Basic oscilloscope & signal generator on an ASIC
- GitHub repository
- HDL project
- Mux address: 751
- Extra docs
- Clock: 25000000 Hz

Authors: Pascal Gesell, Dr. Torsten Maehne, Dr. Theo Kluter

How it works

This is a basic oscilloscope design using the experimental VHDL template. It samples the input signal from channel 1 of an ADC Pmod (Digilent PmodAD1) and buffers the samples on an external FRAM. The captured signal is output on screen via a BlackMesa HDMI Pmod. Test signals are generated using Direct Digital Synthesis and are output on channel 1 of the DAC Pmod (Digilent PmodDA2). Four buttons and two switches allow to control the oscilloscope and choose the test signal to generate.

When the trigger button is pressed, a single-shot measurement is taken when the trigger criteria is met. The trigger criteria can be the vertical and horizontal position as well as the trigger level (positive edge or negative edge). The data is buffered onto the external FRAM, with the goal to contain 32k samples before the trigger event and 32k samples after the trigger event. After the data is collected, the data is displayed on the HDMI screen.

Since an external FRAM memory is used with no buffers on the chip, the displayed oscilloscope screen is actually rotated by 90° to the right. Thus only one sample needs to be read from the FRAM per output video line. A Python script is provided for convenience to read the video frames captured by an USB HDMI video grabber, rotate them by 90° to the left and display them on the screen.

The signal generator supports a few basic waveforms: sine, square, triangle and saw-tooth. The frequency and amplitude can be adjusted using the buttons and switches. The signal generator is also used to test the trigger functionality and the display of the oscilloscope.

The scope settings are continuously output via UART at 9600 baud (8N1) on `uo_out(3)`.



How to test

Connect the various Pmods to the TinyTapeout 4+ demo board or FPGA board according to the pinout description in the info.yaml file. Connect the output of the DAC to the input of the ADC and connect the HDMI Pmod to a screen or HDMI capture card. Run the trigger to capture a single-shot measurement and display the data on the screen.

External hardware

To test and use this project, you will need the following hardware:

- 1 BlackMesaLabs 3-bit HDMI Pmod : A 3-bit HDMI Pmod
- 1 Digilent PmodAD1 : A 12-bit ADC Pmod
- 1 Digilent PmodDA2 : A 12-bit DAC Pmod
- 1 FM25W256G : 32k x 8 FRAM Pmod
- 1 Digilent PmodBTN : A 4 Buttons Pmod
- 1 Digilent PmodSWT : A 4 Switches Pmod, of which only 2 are used

- Optionally, an HDMI capture card to display the HDMI output on a computer screen

Attention: The above Pmods cannot be directly connected to the TinyTapeout 4+ demo board! The Pmods' pins need to be individually connected to the right Pmod pins of the TinyTapeout 4+ demo board, as documented in the IO section.

FPGA Implementation

The design has been implemented and tested on a Sipeed Tang Nano 9k FPGA board using my own base PCB to have enough available Pmods to test the design.

Acknowledgements

This work was realized under the supervision of Dr. Torsten Maehne and Dr. Theo Kluter as part of my project work in the 5th term of my Bachelor studies of electrical engineering and information technology at Berner Fachhochschule (BFH), Biel/Bienne, Switzerland.

Pinout

#	Input	Output	Bidirectional
0	FRAM MISO	ADC CS	HDMI Pmod Green
1	Button 1	DAC MOSI	HDMI Pmod Clock
2	Button 3	ADC SCLK	HDMI Pmod HSYNC
3	Switch 1	FRAM SCLK	UART_TX Settings Info (9600bps, 8N1)
4	ADC MISO	DAC CS	HDMI Pmod Red
5	Button 2	DAC SCLK	HDMI Pmod Blue
6	Button 4	FRAM CS	HDMI Pmod DE
7	Switch 2	FRAM MOSI	HDMI Pmod VSYNC

1bit_am_sdr [752]

- Author: James Sharp
- Description: 1bit AM software defined radio
- GitHub repository
- HDL project
- Mux address: 752
- Extra docs
- Clock: 50000000 Hz

How it works

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

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

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

How to test

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

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

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

The gain can take on the following values:

“Gain” value	Actual Gain
0	x1

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

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

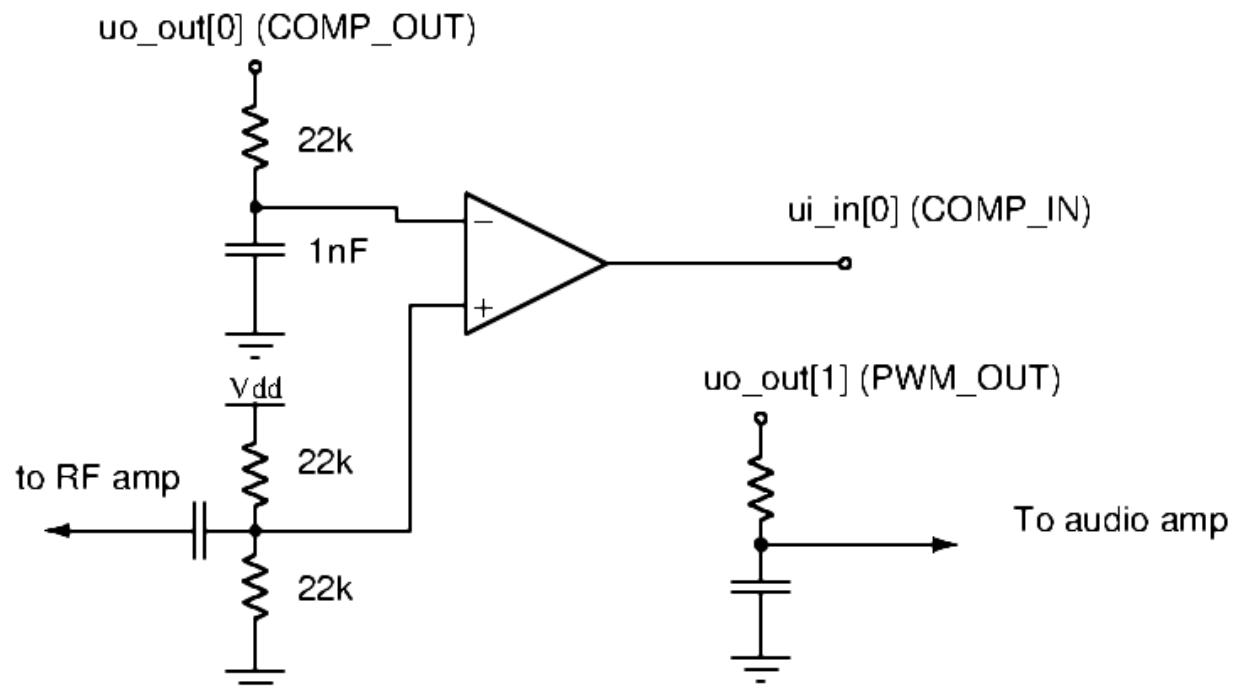
The “NCO Phase increment” is the value that is added to the NCO phase every clock cycle. Use the following python code to calculate the value to write, based on the desired carrier frequency:

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

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

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

External hardware



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

Pinout

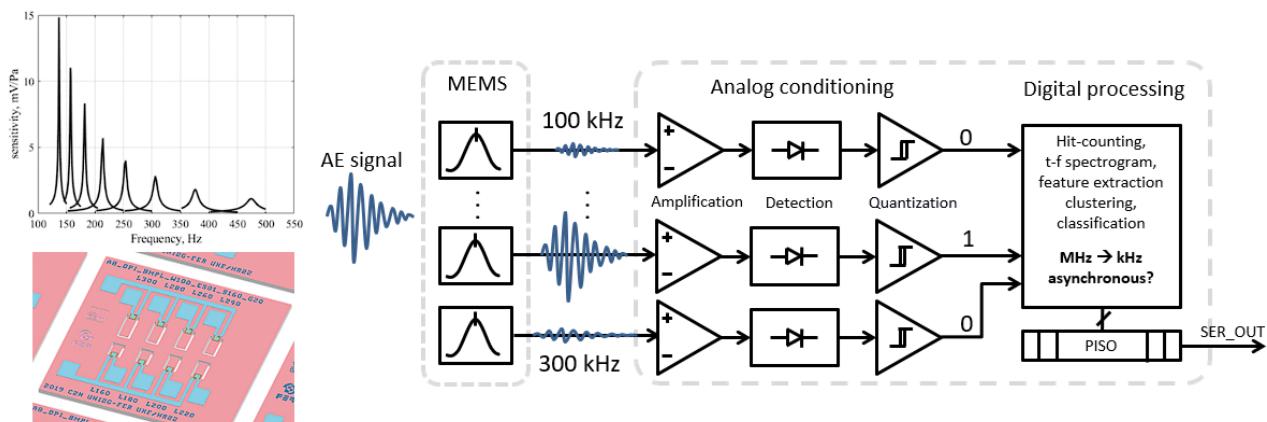
#	Input	Output	Bidirectional
0	COMP_IN	COMP_OUT	
1	SPI_MOSI	PWM	
2	SPI_SCK		
3	SPI_CSb		
4			
5			
6			
7			

15 channels emission counter [753]

- Author: Coline Chehense, Dinko Oletic
- Description: Counts the number of pulses received on each of 15 input channel and returns periodically a serial output of these values.
- GitHub repository
- HDL project
- Mux address: 753
- Extra docs
- Clock: 12000000 Hz

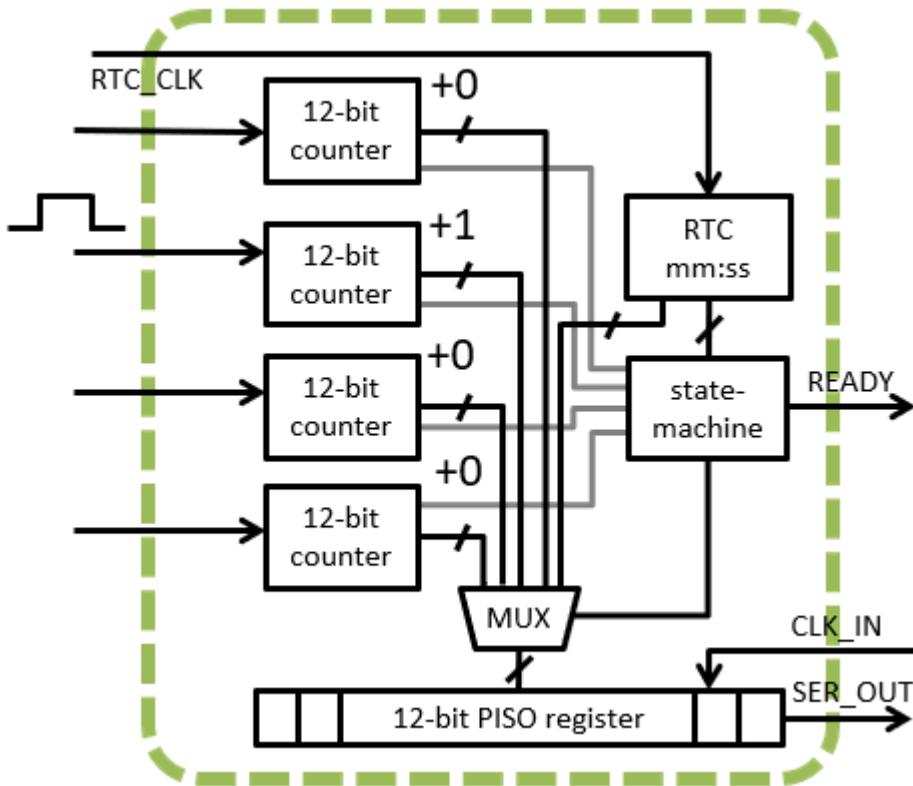
How it works

This is an early work-in progress test implementation of a digital readout, part of a low-power mixed-signal multichannel sensor interface for acoustic emission detection. The sensor interface is developed to support a passive, micromechanically-implemented ultrasonic signal frequency decomposition MEMS device, based on an array of piezo-electric micro-resonators: <https://ieeexplore.ieee.org/document/9139151>.



The digital readout part implemented here, tracks cumulative number of ultrasonic acoustic events/emissions occurrences at each channel i.e. at the specific ultrasonic frequency, over a longer time interval. It is assumed that each acoustic emission event is represented by a short single digital input pulse. An analog conditioning circuit for pulse-shaping of input signals is not implemented here. The digital design consists of fifteen 12-bit channel-counters with overflow detection, a mm:ss real-time clock (RTC), a parallel-input-serial output (PISO) readout register, controlled by a readout state-machine. The counters store number of intermittently-occurring short digital input pulses, accumulated within the RTC's time-measurement interval 00:00 - 59:59, at each of the four input channels. Periodically, after every RTC overflow (1 h with assumed 1 Hz RTC input clock signal), the state-machine performs sequential serial readout of the RTC time and all channels, and resets all channel counters. Additionally,

readout and individual channel reset is initiated by overflow at any of individual input channel counter.



This design is part of research activities <https://www.fer.unizg.hr/liss/aemems>. The design is generally applicable for low-power wake-up sensor interfaces, acoustic event detection, non-destructive testing, particle-counters, or as a generic pulse-counting digital building block. This is the second TinyTapeout submission of the design. The first version was submitted to TT04, it featured 4 channels, and had timing issues during serial readout.

How to test

Input signals are short rising-edge digital pulses, connected to input pins “channel 1” to “channel 15”. Output data becomes ready for serial readout at the output pin “serial_out” when overflow is signalled via the output “ready” pin ovf_global. Output bits are serially clocked-out using the input pin “clk”. Specifically, RTC overflow is signalled via output pin “ovf_RTC_out”, and overflow at an individual channel via the pin “ovf_ch_out”. The rest of output pins are used for debugging of the state-machine’s internal states.

External hardware

Logics analyzer will come handy.

Pinout

#	Input	Output	Bidirectional
0	RTC	serial_out	Channel 8
1	Channel 1	ovf_global	Channel 9
2	Channel 2	ovf_RTC	Channel 10
3	Channel 3	a0	Channel 11
4	Channel 4	a1	Channel 12
5	Channel 5	a2	Channel 13
6	Channel 6	a3	Channel 14
7	Channel 7	SL_out	Channel 15

VGA Pong with NES Controllers [754]

- Author: Brandon S. Ramos
- Description: Pong using 2 NES Controllers with a VGA display
- GitHub repository
- HDL project
- Mux address: 754
- Extra docs
- Clock: 25175000 Hz

How it works

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

How to test

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

Bidirectional:

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

Output:

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

5. b

6.

7.

8.

External hardware

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

Pinout

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

Tiny RAM DFF 2r1w [755]

- Author: Darryl Miles
- Description: RAM made from DFF 2r1w (32x8)
- GitHub repository
- HDL project
- Mux address: 755
- Extra docs
- Clock: 10000000 Hz

How it works

This is a really bad implementation of RAM that uses standard verilog to implement a dual-port-read single-port-write RAM using D-type flipflops.

- DO_A Data Out Port-A
- DI_A Data In Port-A
- DO_B Data Out Port-B
- AD_A Address Port-A
- AD_B Address Port-B
- LOHI_A Nibble (4bit) select Port-A
- LOHI_B Nibble (4bit) select Port-B
- W_EN Write Enable (Port-A implied)

2 pages of 16 bytes (8-bits) is the total storage. The high 1-bit of address are set via RST_N configuration, see below. the low 4-bits of address are supplied on the signal lines.

How to test

The external ports are as you would expect for a RAM module, similar to other ram modules based on the pin descriptions.

Memory reads occur all the time, there is no read-enable. Only port-A can be used to write into. The RST_N does not change the contents of the RAM storage area.

The RST_N release (posedge) is used to latch some additional configuration bits, so the following values are significant and can only be changed by clocking RST_N with a posedge which causes capture:

- uio_in[0] ADDRHI 1-bit to change the RAM page that can be accessed. This is a way to fill out the TT 1x1 tile space a little and allow the upper storage area to be accessible.
- uio_in[3:1] unused
- uio_in[4] READ_BUFFERED_A enable this will enable a synchronous output buffer register on the PORT-A to be enable, so the read value becomes available at the next cycle (pipelined) and held between cycles. If this works as expected this makes the port output asynchronous or synchronous.
- uio_in[5] READ_BUFFERED_B enable, same as above but for PORT-B.
- uio_in[6] WRITE_THROUGH this will activate a MUX bypass that has the effect of implementing a READ_AFTER_WRITE policy, so the currently written value is also the value found at the output port. When inactive (set logic 0) a READ_BEFORE_WRITE policy should be in effect. TODO check this works as expected when READ_BUFFERED_A is active.
- uio_in[7] unused, due to it also being the WRITE-ENABLE bit when in normal operations so allowing the CLK to run freely across reset and an unwanted write occurring.

External hardware

The standard PCB and RP2040 can be used to access. I expect a Micro Python interface to follow in an update.

Future areas to explore

Write a custom placement and wiring router to perform better placement and congestion architecture so that RAM size and WORD WIDTH. This would perform placement into the standard cell track layout so it can be run as a first pass to pack a solution into a design. Ideally leaving the external signals accessible at the edges of the area.

This might allow packing of any width, any depth, single/dual port (as options into the placement process) allowing for consistent size estimations to be made.

It seems when standard placement is left to solve this problem you don't get a result that scales with increased area usage. TODO some research into exactly what occurs in those scenarios, it is expected this maybe due to wiring congestion problem of cells just being in the wrong place / locality and requiring a lot of wiring to get a solution.

I pick dual-port-read support as that should provide a harder problem to solve as a single-port-read needs less wiring.

NOTES

PL_TARGET_DENSITY_PCT=95%
PL_RESIZER_HOLD_SLACK_MARGIN=0.08
GRT_RESIZER_HOLD_SLACK_MARGIN=0.03
CLOCK_PERIOD=100 (10MHz)

- 32x8 3 slew, 26 fanout vio, +106 buffers, resized 646, +16 tie, +238 hold buffers, No room for 156 instances.
- 28x8 1 slew, 36 fanout vio, +124 buffers, resized 763, +16 tie, +201 hold buffers, No room for 23 instances.
- 26x8 1 slew, 28 fanout vio, +105 buffers, resized 646, +16 tie, +191 hold buffers,
10091 vio, 6289 vio after 6th, did not get much better, 6H to 4025 (incomplete pass)
- 24x8 0 slew, 26 fanout vio, +97 buffers, resized 632, +16 tie, +176 hold buffers,
9084 vio, 5305 vio after 6th, best 2134 vio after 24th
- 22x8 0 slew, 20 fanout vio, +82 buffers, resized 555, +16 tie, +171 hold buffers,
7079 vio, 3583 vio after 6th, best 428 vio after 29th, 6H to 427
- 20x8 4 slew, +101 buffers, +101 buffers, resized 649, +16 tie, +151 hold buffers,
6622 vio, 3390 vio after 6th, best 991 vio after 24th
- 18x8 2 slew, 23 fanout vio, +72 buffers, resized 496, +16 tie, +138 hold buffers,
4379 vio, 1299 vio after 6th, 0 vio after 43rd,
SUCCESS
- 16x8 0 slew, 24 fanout vio, +76 buffers, resized 506, +16 tie, +120 hold buffers,
4802 vio, 1631 vio after 6th, 1 vio after 56th, 6H to 64th

Pinout

#	Input	Output	Bidirectional
0	DI_A[0]	DO_A[0]	AD_B[0] (in)
1	DI_A1	DO_A1	AD_B1 (in)
2	DI_A2	DO_A2	AD_B2 (in)
3	DI_A[3]	DO_A[3]	AD_B[3] (in)
4	AD_A[0]	DO_B[0]	LOHI_A (in)
5	AD_A1	DO_B1	LOHI_B (in)

#	Input	Output	Bidirectional
6	AD_A2	DO_B2	
7	AD_A[3]	DO_B[3]	W_EN (in)

Sprite Bouncer with Looping Background Options [768]

- Author: Jacob Mack
- Description: Sprite bouncer hardware that supports multiple background options and sprites.
- GitHub repository
- HDL project
- Mux address: 768
- Extra docs
- Clock: 25000000 Hz

How it works

Sprite ROM, background control registers, and audio ROM are configured using SPI

How to test

Configure background, sprite image, and sfx on bounce using SPI

External hardware

Audio Pmod and Tiny VGA Pmod

Pinout

#	Input	Output	Bidirectional
0	vga_control[0]	R1	
1	vga_control1	G1	
2	vga_control2	B1	
3	vga_control[3]	VSYNC	
4	vga_control[4]	R0	
5	vga_control[5]	G0	
6	vga_control[6]	B0	
7	vga_control[7]	HSYNC	

Glyph Mode [769]

- Author: James Ross
- Description: Submission for VGA Demoscene
- GitHub repository
- HDL project
- Mux address: 769
- Extra docs
- Clock: 25175000 Hz

How it works

This is a standalone VGA demo that runs with or without input. It will accept two pins `ui_io[0]` and `ui_io[1]` for palette color selection:

<code>ui_io[1:0]</code>	Palette
0	Green (default)
1	Red
2	Blue
3	Pride

How to test

Plug into a VGA monitor and select this circuit to test

External hardware

Requires the TinyVGA PMOD

Pinout

#	Input	Output	Bidirectional
0	Palette 0	R1	
1	Palette 1	G1	
2		B1	
3		VSync	
4		R0	

#	Input	Output	Bidirectional
5		G0	
6		B0	
7		HSync	

VGA Scroller [771]

- Author: FavoritoHJS
- Description: Scrolls across a very pixelated cityscape
- GitHub repository
- HDL project
- Mux address: 771
- Extra docs
- Clock: 25000000 Hz

How it works

The terrain is based on an LFSR, using the deterministic randomness of one to generate each layer of the city.

How to test

Set Clock to 25.18MHz, and use a Tiny VGA carrier board for video.

External hardware

This project requires a Tiny VGA carrier board to display video.

Pinout

#	Input	Output	Bidirectional
0	Rh		
1	Gh		
2	Bh		
3	vsync		
4	RI		
5	GI		
6	BI		
7	hsync		

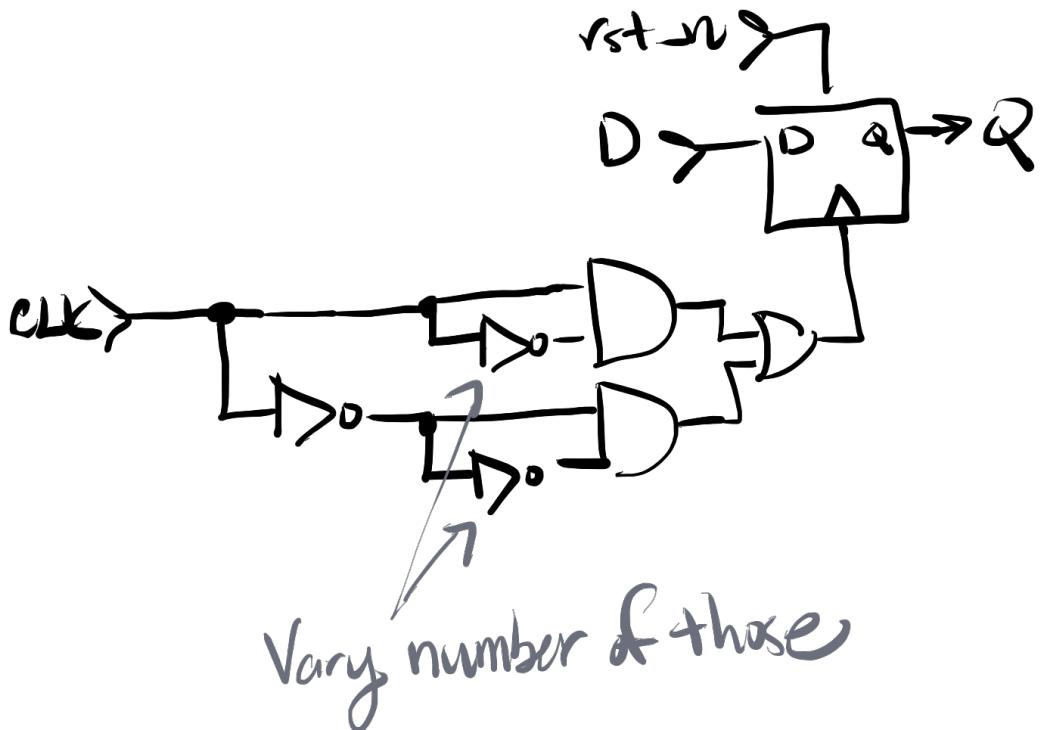
DDR throughput and flop aperature test [773]

- Author: Eric Smith
- Description: Grab data on every edge of clock with varying pos pulse width
- GitHub repository
- HDL project
- Mux address: 773
- Extra docs
- Clock: 0 Hz

How it works

Badly probably.

Use a positive edge detector on the clock and its compliment. Or together those detectors to get 2 positive pulses per period or a 2x clock. Vary clk 2x pos pulse width by varying number of inv per detect.



How to test

Carefully.

External hardware

Analog Discovery 3

Pinout

#	Input	Output	Bidirectional
0	pulse = 1 inv	q for pulse = 1 inv	
1	pulse = 3 inv	q for pulse = 3 inv	
2	pulse = 5 inv	q for pulse = 5 inv	
3	pulse = 7 inv	q for pulse = 7 inv	
4		q for normal flop	
5		1	
6		1	
7		clk	

Wildcat RISC-V [774]

- Author: Martin Schoeberl
- Description: Wildcat: a 3-stage RISC-V implementation
- GitHub repository
- HDL project
- Mux address: 774
- Extra docs
- Clock: 50000000 Hz

How it works

It is a RISC-V CPU

How to test

TBD

External hardware

Flash and RAM PMOD (not really)

Pinout

#	Input	Output	Bidirectional
0	in0	out0	inout0
1	in1	out1	inout1
2	in2	out2	inout2
3	in3	out3	inout3
4	in4	out4	inout4
5	in5	out5	inout5
6	in6	out6	inout6
7	in7	out7	inout7

Calculator [775]

- Author: JING Shuangyu
- Description: A calculator do basic calculation
- GitHub repository
- HDL project
- Mux address: 775
- Extra docs
- Clock: 10000000 Hz

How it works

the calculator can support addition, subtraction, multiplication and division on positive integer number.

How to test

The project can be tested by enter input through the keypad and then check whether the display shows the desire output.

External hardware

The calculator need a 4x4 matrix keypad for input and a 3-digit seven segment display to show the calculated result.

Pinout

#	Input	Output	Bidirectional
0	ROW_1	sseg_A	0
1	ROW_2	sseg_B	E_1
2	ROW_3	sseg_C	E_2
3	ROW_4	sseg_D	E_3
4		sseg_E	COL_1
5		sseg_F	COL_2
6		sseg_G	COL_3
7		sseg_dp	COL_4

Crispy VGA [777]

- Author: James Meech
- Description: The scrolling VGA example from the vga playground but as you set more inputs high it gets successively more crispy
- GitHub repository
- HDL project
- Mux address: 777
- Extra docs
- Clock: 0 Hz

How it works

This project “Crispy VGA” takes as input the output of a standard tiny tapeout VGA project. Crispy VGA then adds a programmable amount of random noise to the VGA signal and passes it through to the output. The `ui0_in[0]` input sets the noise on the `hsync` signal. The `ui0_in1` input sets the noise on the `B` signal. The `ui0_in2` input sets the noise on the `G` signal. The `ui0_in[3]` input sets the noise on the `R` signal. The `ui0_in[4]` input sets the noise on the `vsync`. The `ui0_in[5]` signal sets the noise level applied to the `R`, `G`, and `B` wires to high or low. The `ui0_in[0:5]` inputs set the successively increasing noise levels on the audio signal.

How to test

Plug an existing tiny tapeout VGA project into the input of this design. Plug the output of this design into a standard VGA input monitor. Power up both tiny tapeout boards and select the appropriate control bits for the level of noise that you want to see on the output VGA signal.

External hardware

You will need a VGA input monitor and a computer that can output a VGA signal or a second tiny tapeout ASIC with a working VGA design that follows the standard pinout. You will also need two tiny tapeout VGA adapters and two VGA cables.

Pinout

#	Input	Output	Bidirectional
0	R1 vga input	R1 vga input	Crispy input bit 0 that toggles the noise on the hsync signal
1	G1 vga input	G1 vga input	Crispy input bit 1 toggles the noise on the B signal on output
2	B1 vga input	B1 vga input	Crispy input bit 2 toggles the noise on the G signal on output
3	vsync vga input	vsync vga input	Crispy input bit 3 toggles the noise on the R signal on output
4	R[0] vga input	R[0] vga input	Crispy input bit 4 that toggles the noise on the vsync signal
5	G[0] vga input	G[0] vga input	Crispy input bit 5 that sets the noise level applied to the output
6	B[0] vga input	B[0] vga input	Audio input bit
7	hsync vga input	hsync vga input	Audio output bit

asic design is my passion [779]

- Author: Nicholas Junker
- Description: baby's first asic - cheeky little text meme
- GitHub repository
- HDL project
- Mux address: 779
- Extra docs
- Clock: 25175000 Hz

How it works

Real, real bad graphic design & fun shapes bouncing around on the screen.

How to test

Hook up to VGA monitor using the TinyTapeout VGA module.

External hardware

Tiny VGA Pmod peripheral!

Pinout

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

TinyQV Risc-V SoC [780]

- Author: Michael Bell
- Description: A Risc-V SoC for Tiny Tapeout
- GitHub repository
- HDL project
- Mux address: 780
- Extra docs
- Clock: 64000000 Hz

How it works

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

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

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

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

The SoC includes a UART and an SPI controller.

Address map

Address range	Device
0x0000000 - 0x0FFFFFF	Flash
0x1000000 - 0x17FFFFFF	RAM A
0x1800000 - 0x1FFFFFF	RAM B
0x7FFF00 - 0x7FFFFFF	Internal RAM (32 bytes, wrapped)
0x8000000 - 0x8000007	GPIO
0x8000010 - 0x800001F	UART
0x8000020 - 0x8000027	SPI
0x8000028 - 0x800002B	PWM
0x8000030 - 0x8000033	DEBUG
0x8000034 - 0x800003B	TIME
0x8000040 - 0x8000047	GAME

Address range	Device

GPIO

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

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

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

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
DATA	0x8000020 (R)	Reads the last received byte
CONFIG	0x8000024 (W)	The low 4 bits set the clock divisor for the SPI clock to 2*(value +
STATUS	0x8000024 (R)	Bit 0 indicates whether the SPI is busy, bytes should not be written

PWM

Register	Address	Description
LEVEL	0x8000028 (W)	Set the PWM output level (0-255)

DEBUG See debug docs

TIME

Register	Address	Description
MTIME	0x8000034 (RW)	Get/set the 1MHz time count
MTIMECMP	0x8000038 (RW)	Get/set the time to trigger the timer interrupt

GAME

Register	Address	Description
Controller 1	0x80000040 (R)	Controller 1 state
Controller 2	0x80000044 (R)	Controller 2 state

Controller state is in the low 12 bits of the register, in order (MSB to LSB): b, y, select, start, up, down, left, right, a, x, l, r

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 SD1: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	1 MHz clock for time	SPI MOSI	SCK
4	Game controller latch	SPI CS	SD2
5	Game controller clock	SPI SCK	SD3
6	Game controller data	Debug UART TX	RAM A CS
7	UART RX	Debug signal / PWM	RAM B CS / PWM

Dice [781]

- Author: ZHU QUANHAO
- Description: after you press the button the system will generate a random number from 0-F
- GitHub repository
- HDL project
- Mux address: 781
- Extra docs
- Clock: 50000000 Hz

How it works

It generate number by inverter ring

How to test

press the button to capture number

External hardware

2 LED display

Pinout

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

4-bit minicomputer ALU [783]

- Author: Mike McCann
- Description: this design provides basic arithmetic and logic functions
- GitHub repository
- HDL project
- Mux address: 783
- Extra docs
- Clock: 0 Hz

How it works

The project is a 4-bit ALU section that is useful in mini and micro computer CPUs.

How to test

This device can be tested by inputting data on the two input ports (A/B), a function code (F0, F1, F2) and observing the output on pins d0, d1, d2, d3.

External hardware

This project was tested using an Altera FPGA (EP2C20F484C7).

Pinout

#	Input	Output	Bidirectional
0	da0	d0	NEG_ZERO
1	da1	d1	ci_left
2	da2	d2	ci_right
3	da3	d3	COM
4	db0	co_left	F0
5	db1	co_right	F1
6	db2	EQU	F2
7	db3	ZERO	

RGB Mixer demo5 [785]

- Author: Matt Venn
- Description: Zero to ASIC demo project
- GitHub repository
- HDL project
- Mux address: 785
- 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	debug bit 0		
7	debug bit 1		

AlphaOneSoC [786]

- Author: Abhiram Gopal Dasika
- Description: A 32-bit RISC-V SoC, based on TinyQV by Michael Bell
- GitHub repository
- HDL project
- Mux address: 786
- Extra docs
- Clock: 64000000 Hz

How it works

A simple 32-bit RISC-V SoC on the RV32EC ISA, the project works by flashing the instruction code into the memory and observing the outputs over your desired method, via GPIO, UART or SPI. The TT10-IHP submission is based entirely on MichaelBell's TinyQV from TT-06.

How to test

Flash the PMOD with instructions (somehow) and boot up the processor. The processor will start executing the instructions

External hardware

- QSPI + Flash PMOD

Pinout

#	Input	Output	Bidirectional
0	Interrupt 0	UART TX	Flash CS
1	Interrupt 1	UART RTS	SD0
2	SPI MISO	SPI DC	SD1
3	GPI3	SPI MOSI	SCK
4	GPI4	SPI CS	SD2
5	GPI5	SPI SCK	SD3
6	GPI6	Debug UART TX	RAM A CS
7	UART RX	Debug Signal	RAM B CS

Asynchronous Multiplier [787]

- Author: Tommy Thorn
- Description: An asynchronous multiplier
- GitHub repository
- HDL project
- Mux address: 787
- Extra docs
- Clock: 50000000 Hz

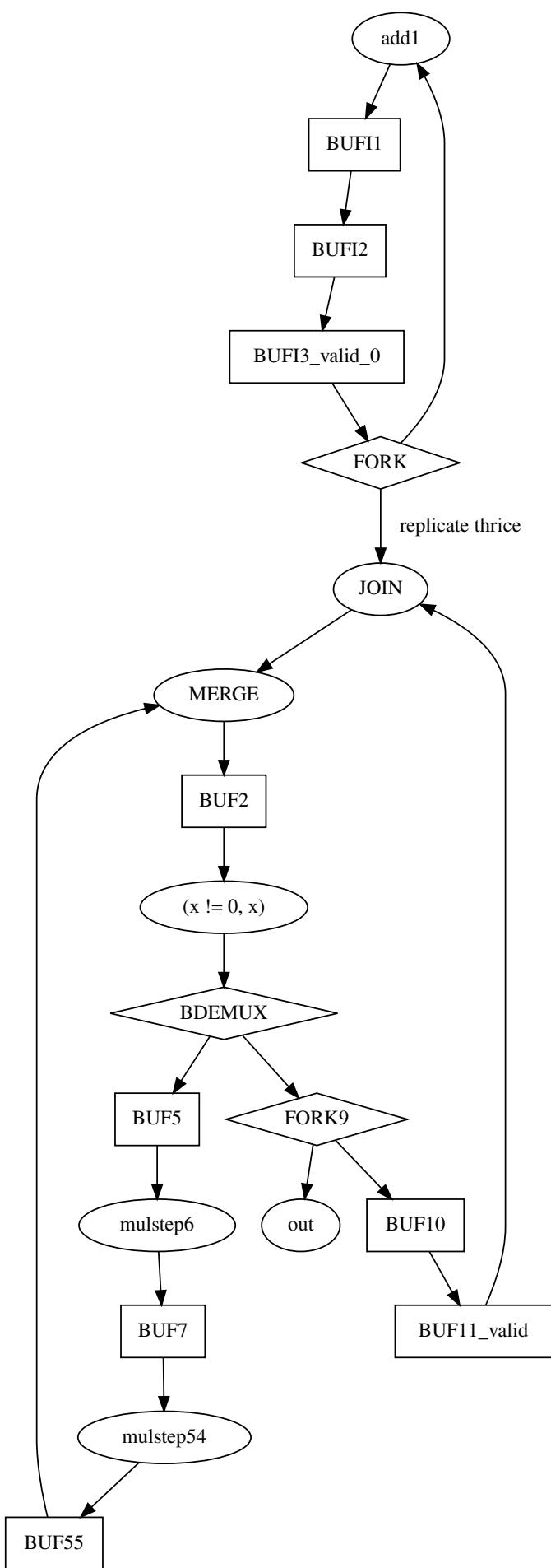
How it works

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

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

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

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



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

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

How to test

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

External hardware

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

Pinout

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

Hamming Code (7,4) [801]

- Author: Sebastien Paradis
- Description: (7,4) Hamming Encoder/Decoder
- GitHub repository
- HDL project
- Mux address: 801
- Extra docs
- Clock: 0 Hz

How it works

This implementation of the (7,4) Hamming Code allows for the same input to be used for encoding and decoding, with dynamic selection of the mode using the MSB of the input.

Hamming Encoder (7,4) Overview The Hamming (7,4) encoder is a linear error-correcting code that encodes 4 data bits into 7 bits by adding 3 parity bits, which can detect and correct a single-bit error.

Parity Format

{p1 p2 p3}

Data Format

{d1 d2 d3 d4}

Input An 8-bit input “ui” with the following format (note the form is {7 6 5 4 3 2 1 0})

Input Pins

- ui[0] - Bit 0 for 4-bit data input, d4
- ui1 - Bit 1 for 4-bit data input, d3
- ui2 - Bit 2 for 4-bit data input, d2
- ui[3] - Bit 3 for 4-bit data input, d1
- ui[4] - X
- ui[5] - X
- ui[6] - X
- ui[7] - Mode Selector (0 => Encode, uses ui[3:0]; 1 => Decode, uses ui[6:0])

Output An 8-bit output “uo” with the following format (note the form is {7 6 5 4 3 2 1 0})

Output Pins

- $uo[0]$ - Bit 0 for 7-bit encoded output, d4
- $uo[1]$ - Bit 1 for 7-bit encoded output, d3
- $uo[2]$ - Bit 2 for 7-bit encoded output, d2
- $uo[3]$ - Bit 3 for 7-bit encoded output, p3
- $uo[4]$ - Bit 4 for 7-bit encoded output, d1
- $uo[5]$ - Bit 5 for 7-bit encoded output, p2
- $uo[6]$ - Bit 6 for 7-bit encoded output, p1
- $uo[7]$ - X

Encode Mode

- Encode Mode is selected by setting the MSB of the input (bit 7) LOW (0).
- If encode mode is chosen, the encoder will use bits 3:0 as the four data bits to be encoded, and produce a 7-bit encoded output.
- Bit 6:4 are not involved in any encoding.

Encode Mode Input Format

{selector, X, X, X, d1, d2, d3, d4}

Encode Mode Output Format

{p1, p2, d1, p3, d2, d3, d4}

Parity Bit Calculations

1. p1 covers bits d1, d2, and d4.
 - $p1 = d1 \text{ XOR } d2 \text{ XOR } d4$
2. p2 covers bits d1, d3, and d4.
 - $p2 = d1 \text{ XOR } d3 \text{ XOR } d4$
3. p3 covers bits d2, d3, and d4.
 - $p3 = d2 \text{ XOR } d3 \text{ XOR } d4$

Expected Outputs of Encode Mode

- 0XXXd1d2d3d4 -> 0p1p2d1p3d2d3d4
- 00000000 -> 00000000
- 00000010 -> 00101010
- 00000001 -> 01101001
- 00000011 -> 01000011
- 00000100 -> 01001100
- 00000101 -> 00100101
- 00000110 -> 01100110
- 00000111 -> 00001111
- 00001000 -> 01110000
- 00001001 -> 00011001
- 00001010 -> 01011010
- 00001011 -> 00110011
- 00001100 -> 00111100
- 00001101 -> 01010101
- 00001110 -> 00010110
- 00001111 -> 01111111

Hamming Decoder (7,4) Overview The decoder checks the received 7-bit word for errors and corrects a single-bit error if detected. The process involves recalculating the parity bits and comparing them with the received parity.

Decode Mode

- Decode Mode is selected by setting the MSB of the input (bit 7) HIGH (1).
- If decode mode is chosen, the decoder will use bits 7:0, both the data and parity bits, and produce a 7-bit decoded output. The decoded output will be the originally encoded input as long as there were less than 2 flipped bits between encoder output and decoder input.

Decode Mode Input Format

{p1, p2, d1, p3, d2, d3, d4}

Decode Mode Output Format

{p1, p2, d1, p3, d2, d3, d4}

- a maximum of 1 bit could be flipped at position {S2, S1, S0}.

Syndrome Calculation The syndrome indicates the position of an error (if any):

1. S_0 is recalculated using the same bits used to calculate p_1 during encoding:
 - $S_0 = p_1' \text{ XOR } d_1 \text{ XOR } d_2 \text{ XOR } d_4$
2. S_1 recalculates p_2 :
 - $S_1 = p_2' \text{ XOR } d_1 \text{ XOR } d_3 \text{ XOR } d_4$
3. S_2 recalculates p_3 :
 - $S_2 = p_3' \text{ XOR } d_2 \text{ XOR } d_3 \text{ XOR } d_4$

Error Correction The syndrome $\{S_2, S_1, S_0\}$ gives the error location:

- If the syndrome is 000, no error is detected.
- If the syndrome is non-zero, the position of the error corresponds to the syndrome value (1 for the least significant bit, 7 for the most significant bit).
- E.g. if syndrome is 010, then. Our error bit is at bit 4
- If an error is detected, flip the bit at the position indicated by the syndrome.

How to test

Testing can be done by applying known data inputs with LOW as the value of the 7th bit (encode mode), and ensuring that the output is the expected encoding value (see table of expected outputs in encode mode).

Similarly, known encoded values can be used as input, with the 7th bit as HIGH (decode mode), and we can ensure that the output is the exact same as the original encoded value, even if we flip 1 bit. This should be done for each of the 7 bits for all encoded values

External hardware

TBD based on implementation.

#	Input	Output
---	-------	--------

Pinout

#	Input	Output
0	LSB/Bit 0 for 4-bit Encoder Input OR LSB/Bit 0 for 7-bit Decoder Input	LSB/Bit 0 for 7-bit Decoder Input
1	Bit 1 for 4-bit Encoder Input OR Bit 1 for 7-bit Decoder Input	Bit 1 for 7-bit Decoder Input
2	Bit 2 for 4-bit Encoder Input OR Bit 2 for 7-bit Decoder Input	Bit 2 for 7-bit Decoder Input
3	MSB/Bit 3 for 4-bit Encoder Input OR Bit 3 for 7-bit Decoder Input	Bit 3 for 7-bit Decoder Input
4	Bit 4 for 7-bit Decoder Input	Bit 4 for 7-bit Decoder Input
5	Bit 5 for 7-bit Decoder Input	Bit 5 for 7-bit Decoder Input
6	MSB/Bit 6 for 7-bit Decoder Input	MSB/Bit 6 for 7-bit Decoder Input
7	Mode Selector (0 => Encode, uses ui[3:0]; 1 => Decode, uses ui[6:0])	Mode Selector (ui[6:0])

Space Detective Maze Explorer [803]

- Author: Esteban Oman Mendoza
- Description: A maze explorer game, output uses qty5 7segment displays or LED equivalent
- GitHub repository
- HDL project
- Mux address: 803
- Extra docs
- Clock: 50000 Hz

How it works

This is a maze running on hardware. 3 user inputs are used. `user_input[0]` is used to walk forward on low, and `user_input[2:1]` is used as direction select where `2'b00 = N`, `2'b01 = East`, `2'b10 = South`, and `2'b11 = West`“ bit 2 is the most significant bit

How to test

You will need to wire up qty 5 7-segment displays or led equivalent. seg 0 is the right most or least significant segment, and seg4 being the left most of Most significant segment. Hook up all the common pins. for example pin 1 from seg0 connects to all other pin1 on the other 4 segments, they are then connected to the corresponding output pins `uo[7:0]`

Outputs

`uo[0]: " a` `uo[0] = a` “ `uo1: " —` `uo1 = b` “ `uo2: " f | g | b` `uo2 = c` “
`uo[3]: " | |` `uo[3] = d` “ `uo[4]: " —` `uo[4] = e` “ `uo[5]: " | |` `uo[5] = f` “ `uo[6]: " e | d |` `c` `uo[6] = g` “ `uo[7]: " | |` `uo[7] = dp` “ — `dp` `uo([7:0]` is the decoded segment signals to display the game output.

using 5 pnp transistors with `Vcc` (`I` used. `3.3V`) at the emmiter, and the common anode (`I` used <http://www.xlitx.com/datasheet/5161AS.pdf>) connected to the collector, make a connection to `ui0[4:0]` to represent seg4-seg0. example `ui0 5'b011111` would turn on seg 4 (low = on) each segment is mapped to `ui0[0]: "state LSB"` `ui0[1]: "state MSB"` `ui0[2]: "Direction LSB"` `ui0[3]: "Direction MSB"` `ui0[4]: "Top half of segment used for wall representation. 0-0, 1-1,...,5-5.`

External hardware

qty 5 7-segment display or LED equivalent to visualize the game
qty 5 current limitting resistors for the 7 segments
qty 5 current limitting resistors to manage current through the output pins. these are connected to the base breadboard and enough wiring to make all the connections

Pinout

#	Input
0	user_input[0] Move forward (move one step in selected direction)
1	user_input1 (considered least significant bit used in selection direction) Used to select facing
2	user_input2 (considered least most significant bit used in selection direction) Used to select
3	not used
4	not used
5	not used
6	not used
7	not used

Senol Gulgongul tt09 [805]

- Author: Senol Gulgongul
- Description: Display the letters of SEnOLGULGONUL on 7-Seg using internal oscillator
- GitHub repository
- HDL project
- Mux address: 805
- Extra docs
- Clock: 0 Hz

How it works

Displays letters of SEnOLGULGONUL on 7-Seg display by using internal three gate oscillator

How to test

Connect external R1, R2 and C for three gate oscillator and clk input and watch letters on 7-seg

External hardware

output pins a,b,c,d,e,f,g,dp are connected to a 7-Seg display, two inout and two bioutput for oscillator

Pinout

#	Input	Output	Bidirectional
0	inv3_in	a	inv3_out
1	inv1_in	b	inv2_out
2		c	
3		d	
4		e	
5		f	
6		g	
7		dp	

4 bit ALU [807]

- Author: Gabriela Alfaro
- Description: A simple design of an Arithmetic Logic Unit capable of basic operations: addition, subtraction , multiplication, division and some logic operations.
- GitHub repository
- HDL project
- Mux address: 807
- Extra docs
- Clock: 0 Hz

How it works?

The 4-bit ALU (Arithmetic Logic Unit) is designed to perform a range of arithmetic and logical operations on two 4-bit inputs, A and B. The operation is determined by a 3-bit control signal, Opcode, which specifies the function to execute, such as addition, subtraction, multiplication, division, and bitwise operations (AND, OR, NOT, XOR).

When an arithmetic operation like addition is selected, the ALU outputs an 8-bit result, ALU_Result, to accommodate larger sums or products, and it sets a Carry flag if there's an overflow. For logical operations like AND or OR, the ALU applies the operation bit-by-bit between A and B. The Zero flag is activated when the result is zero, providing a useful condition for further logic. This flexibility allows the ALU to handle various computational tasks, making it a crucial part of digital systems that require multi-functional data processing.

How to test?

To test the design, the operation codes are:

- Addition (000)
- Subtraction (001)
- Multiplication (010)
- Division (011)
- Logic AND (100)
- Logic OR (101)
- Logic NOT (110)
- Logic XOR (111)

Pinout

#	Input	Output	Bidirectional
0	A[0]	ALU_Out[0]	ZeroFlag
1	A1	ALU_Out1	CarryOut
2	A2	ALU_Out2	
3	A[3]	ALU_Out[3]	
4	B[0]	ALU_Out[4]	
5	B1	ALU_Out[5]	
6	B2	ALU_Out[6]	
7	B[3]	ALU_Out[7]	

Elevator Design [809]

- Author: Jocelyn Zhu
- Description: Simulation of an elevator design on a digital clock display.
- GitHub repository
- HDL project
- Mux address: 809
- Extra docs
- Clock: 0 Hz

How it works

The project implements an elevator interface on a digital clock based on user floor selection. The user selects a floor using the board switches, and the display increments/decrements floor numbers according to the elevator's state (moving up, moving down, or idle). Once the elevator reaches the selected floor, the display shows the user-selected floor number until a different floor is chosen or the switches are reset. When the switches are reset, the display decrements back to the default floor.

How to test

Use board switches 0-7 to select the desired floor.

External hardware

A LED display is used to show elevator operation and the selected floor number.

Pinout

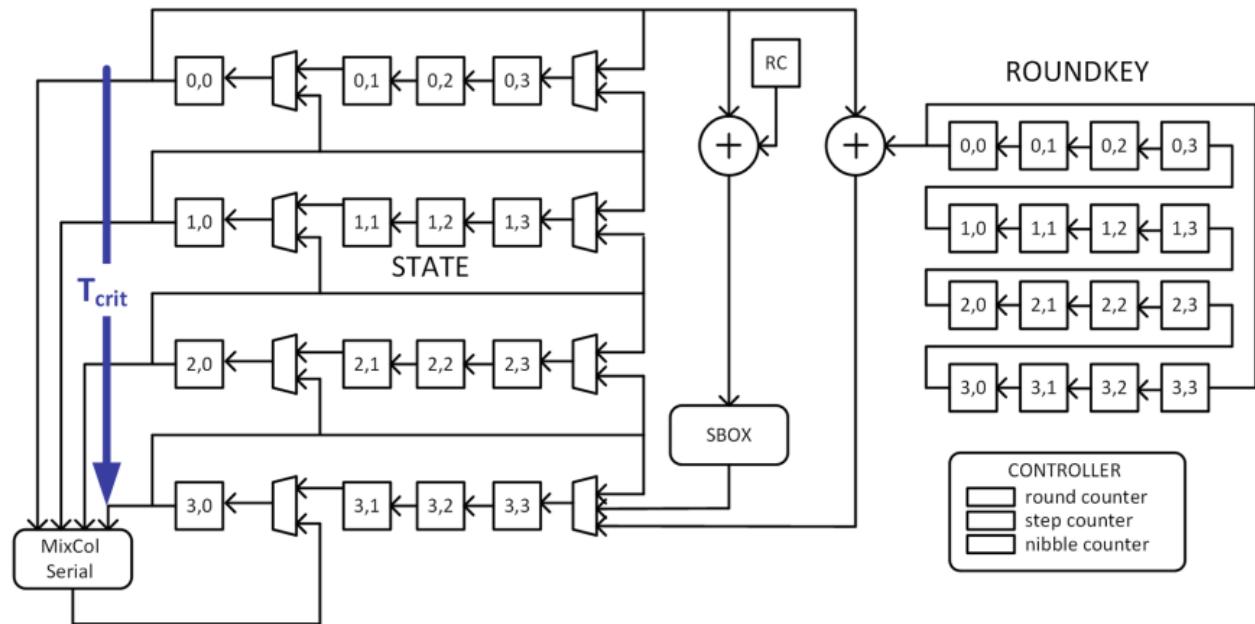
#	Input	Output	Bidirectional
0	Switch 0	Segment A	
1	Switch 1	Segment B	
2	Switch 2	Segment C	
3	Switch 3	Segment D	
4	Switch 4	Segment E	
5	Switch 5	Segment F	
6	Switch 6	Segment G	
7	Switch 7	A dot that appears during the IDLE state	

LED Bitserial Cipher [811]

- Author: simon cipher
- Description: A bitserial implementation of the LED cipher
- GitHub repository
- HDL project
- Mux address: 811
- Extra docs
- Clock: 0 Hz

How it works

tt09-led-serial is a nibble-serial implementation of the LED block cipher, proposed in 2012 and defined in The LED Block Cipher by J. Guo et. al. The cipher encrypts a 64-bit block of plaintext with a 128-bit key into a 64-bit block of ciphertext. The nibble-serial implementation enables a very compact implementation as most of the datapath logic can be reused over each nibble. The downside is that such nibble-serial implementations have a much larger latency. The nibble-serial architecture shown below was presented and analyzed earlier in Differential Fault Intensity Analysis on PRESENT and LED Block Ciphers by N. F. Galathy et. al.



To further reduce the I/O pinout constraints, this design also serializes the data-input (64 bit plaintext and 128 bit key) as well as the data-output (64 bit ciphertext).

Activity	Cycles
Load Plaintext	64
Load Key	128
Read Ciphertext	64
Encrypt	2045

The module is controlled through the bits of the input word `ui_in`. The serial data format is MSB to LSB. That is, given a block of plaintext `0x0123...`, the bits would be shift in as in the bitstring `0b0000000100100011...`.

Bit	Name	Function
7-6	unused	NA
5	start	Assert to start encryption
4	getct	Assert to shift out ciphertext bit
3	loadkey	Assert to shift in key bit
2	loadpt	Assert to shift in plaintext bit
1	keyi	Key input bit
0	datai	Plaintext input bit

The results are generated on the output word `uo_out`.

Bit	Name	Function
7-2	unused	NA
1	done	1 indicates encryption complete
0	dataq	Ciphertext output bit

LIMITATIONS

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

How to test

This block could be tested with some integration on a Raspberry PI to control `ui_in` and `uo_out`. The typical sequence of operation is as follows.

1. Wait until done == 1, which indicates that the cipher is idle
2. Assert loadkey, and shift in key bits. Repeat 128 times. De-assert loadkey.
3. Assert loadpt, and shift in plaintext bits. Repeat 64 times. De-assert loadpt.
4. Assert start for one clock cycle.
5. Wait until done == 1.
6. Assert getct and shift out ciphertext bits. Repeat 64 times. De-assert getct.

Here are two three sample test vectors. Consult the testbench for additional test vectors.

Plaintext	Key	Ciphertext
0000000000000000	00000000000000000000000000000000	3decb2a0850cdba1
0123456789abcdef	00000000000000000000000000000000	da261393c73be9ce
12153524c0895e81	00000000000000000000000000000000	29db5fe262572f4e

External hardware

You will need external hardware to use the block cipher.

Pinout

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

freqSweep [813]

- Author: Jesus Minguillon
- Description: Frequency sweeper
- GitHub repository
- HDL project
- Mux address: 813
- Extra docs
- Clock: 5000000 Hz

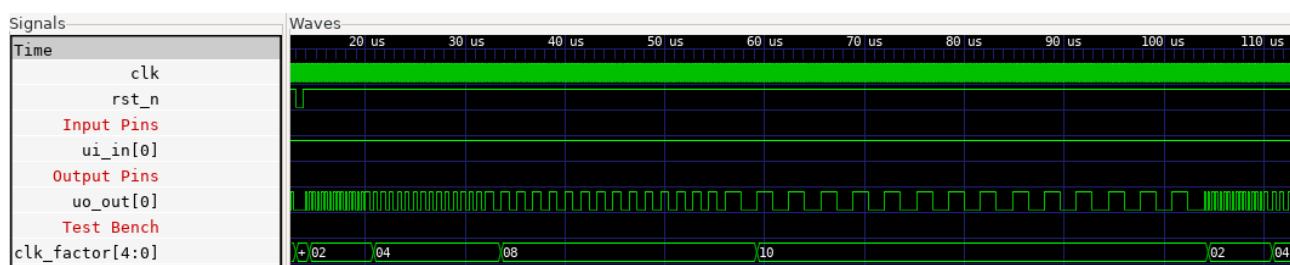
How it works

The project (src/project.v) implements a clock frequency sweeper in Verilog. It uses the Tiny Tapeout clock as input to generate a clock signal at the output $uo[0]$ whose frequency is divided by 2 every 15 clock cycles. It goes from 1/2 to 1/16 of the input clock frequency and starts again. Input $ui[0]$ is used as internal enable (active high).

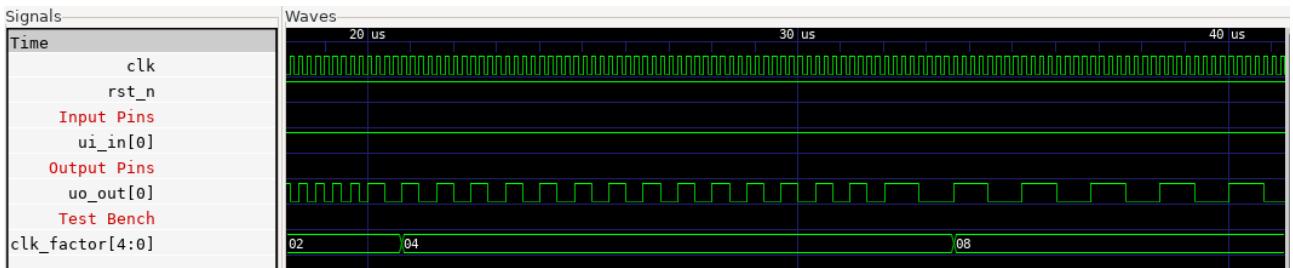
How to test

For simulation, use the test bench (test/tb.v), which includes a module (src/periodCount.v) for measuring the frequency (or period) ratio between input and output clocks. Use the Python script (test/test.py) if you want to perform unit tests using cocotb. For hardware testing, make sure the internal enable (input $ui[0]$) is high and check the output clock at output $uo[0]$ using an oscilloscope.

Gate level simulation (5 MHz input clock) The frequency (or period) ratio between the input clock (clk) and the output clock ($uo_out[0]$) is given by clk_factor register of periodCount module:



Zoom in:



Unit tests:

```
(venv) jesus@ws131571:~/tt09-verilog-freqSweep/test$ make -B GATES=yes
rm -f results.xml
"make" -f Makefile results.xml
make[1]: Entering directory '/home/jesus/tt09-verilog-freqSweep/test'
mkdir -p sim_build/gl
/usr/bin/iverilog -o sim_build/gl/sim.vvp -D COCOTB_SIM=1 -s tb -g2012 -D
rm -f results.xml
MODULE=test TESTCASE= TOPLEVEL=tb TOPLEVEL_LANG=verilog \
    /usr/bin/vvp -M /home/jesus/ttsetup/venv/lib/python3.12/site-pac
    -.--ns INFO      gpi                               ..mbed/gpi_embed.
    -.--ns INFO      gpi                               ../gpi/GpiCommon.
0.00ns INFO      cocotb                            Running on Icarus
0.00ns INFO      cocotb                            Running tests with
0.00ns INFO      cocotb                            Seeding Python ran
0.00ns INFO      cocotb.regression                 Found test test.t
0.00ns INFO      cocotb.regression                 running test_star
0.00ns INFO      cocotb.tb                         Startup test with
0.00ns INFO      cocotb.tb                         rst_n = 0, ui_in[0]
VCD info: dumpfile tb.vcd opened for output.
2.00ns INFO      cocotb.tb                         uo_out[0] = 0
2000.00ns INFO    cocotb.tb                         rst_n = 1, ui_in[0]
2202.00ns INFO    cocotb.tb                         uo_out[0] = 0
4000.00ns INFO    cocotb.tb                         rst_n = 1, ui_in[0]
4202.00ns INFO    cocotb.tb                         uo_out[0] = 1
4402.00ns INFO    cocotb.tb                         uo_out[0] = 0
4402.00ns INFO    cocotb.tb                         Wait until 6 us
6000.00ns INFO    cocotb.tb                         End of startup te
6000.00ns INFO    cocotb.regression                test_startup pass
6000.00ns INFO    cocotb.regression                running test_rese
6000.00ns INFO    cocotb.tb                         Reset test
```

```

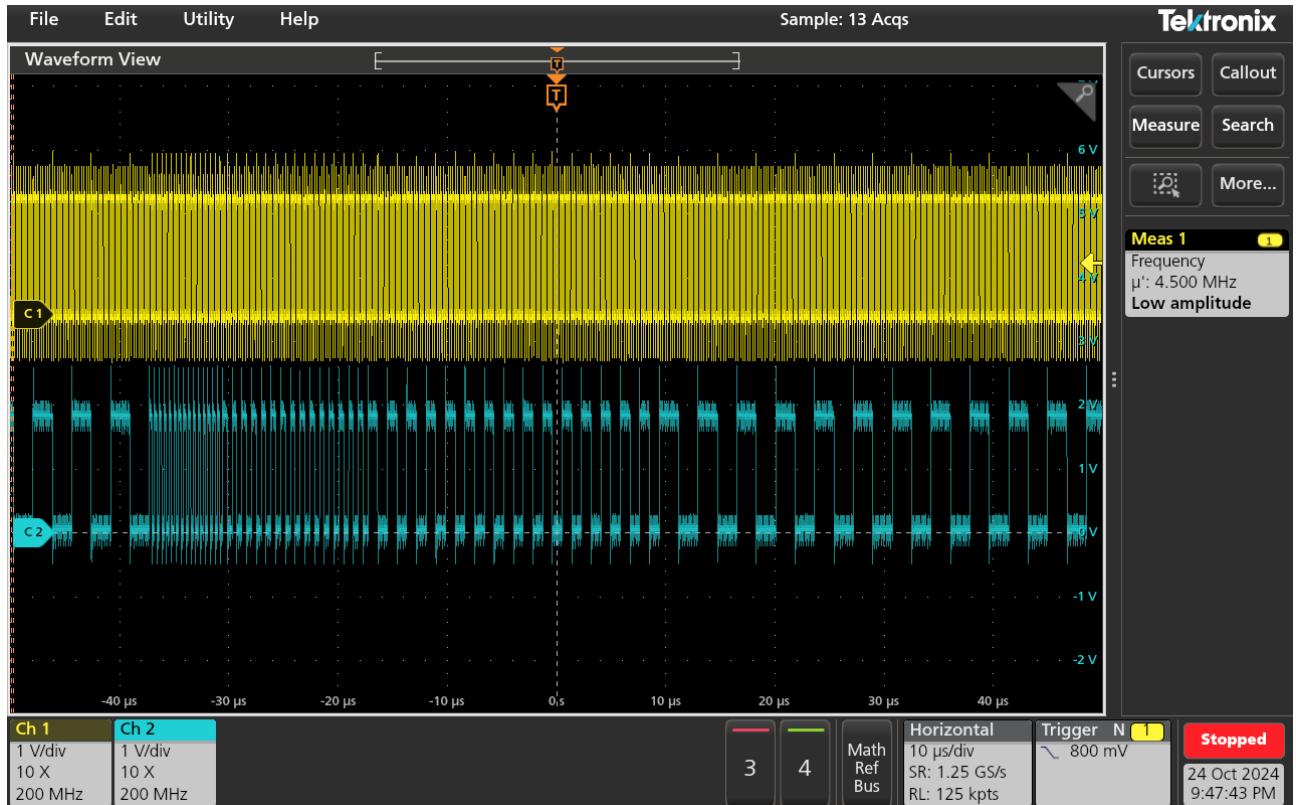
6000.00ns INFO cocotb.tb rst_n = 0, ui_in[0] = 0
6002.00ns INFO cocotb.tb uo_out[0] = 0
6002.00ns INFO cocotb.tb clk_factor = 1
6202.00ns INFO cocotb.tb uo_out[0] = 0
6202.00ns INFO cocotb.tb clk_factor = 1
8000.00ns INFO cocotb.tb uo_out[0] = 0
8000.00ns INFO cocotb.tb clk_factor = 1
8000.00ns INFO cocotb.tb rst_n = 1, ui_in[0] = 1
8202.00ns INFO cocotb.tb uo_out[0] = 1
8202.00ns INFO cocotb.tb clk_factor = 1
8402.00ns INFO cocotb.tb uo_out[0] = 0
8402.00ns INFO cocotb.tb clk_factor = 1
8402.00ns INFO cocotb.tb Wait until 10 us
10000.00ns INFO cocotb.tb End of reset test
10000.00ns INFO cocotb.regression test_reset passed
10000.00ns INFO cocotb.regression running test_inter
10000.00ns INFO cocotb.tb Internal enable test
10000.00ns INFO cocotb.tb rst_n = 1, ui_in[0] = 0
11600.00ns INFO cocotb.tb rst_n = 1, ui_in[0] = 1
11802.00ns INFO cocotb.tb uo_out[0] = 1
11802.00ns INFO cocotb.tb clk_factor = 2
11802.00ns INFO cocotb.tb Wait until 13 us
13000.00ns INFO cocotb.tb Reset for 800 ns
14000.00ns INFO cocotb.tb End of internal enable test
14000.00ns INFO cocotb.regression test_internal_enable
14000.00ns INFO cocotb.regression running test_period_count
14000.00ns INFO cocotb.tb Period count test
14402.00ns INFO cocotb.tb clk_factor = 2
20802.00ns INFO cocotb.tb clk_factor = 4
33602.00ns INFO cocotb.tb clk_factor = 8
59202.00ns INFO cocotb.tb clk_factor = 16
59202.00ns INFO cocotb.tb Wait until 200 us
200000.00ns INFO cocotb.tb End of period count test
200000.00ns INFO cocotb.regression test_period_count
200000.00ns INFO cocotb.regression **** TEST ****
***** test.test_start ****
** test.test_start ****
** test.test_rese ****
** test.test_inter ****
** test.test_peri ****
***** test.test_stop ****

```

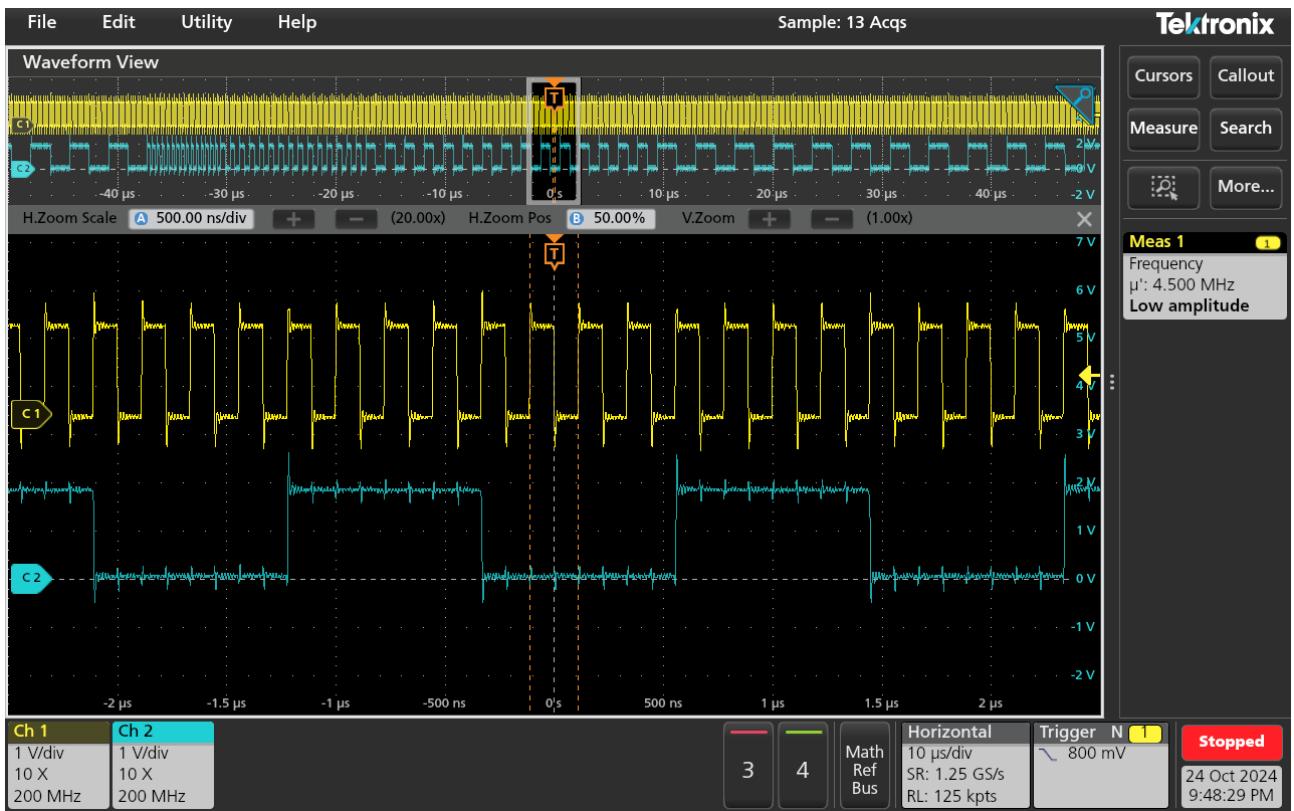
** TESTS=4 PASS=4

make[1]: Leaving directory '/home/jesus/tt09-verilog-freqSweep/test'

Test using Tang Nano 9K FPGA (4.5 MHz input clock) Input clock signal (yellow) and output clock signal (blue) acquired with an oscilloscope (analog inputs and passive probes):



Zoom in:



External hardware

No external hardware is needed.

Pinout

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

Simple PWM Module [815]

- Author: Tobi McKellar
- Description: PWM for LED control.
- GitHub repository
- HDL project
- Mux address: 815
- Extra docs
- Clock: 0 Hz

How it works

A basic PWM controller. ui[5:0] control the reference. When ui[6] is low, this reference is used to set the PWM duty cycle. when ui[6] is high, functionality changes from manual reference to a triangular reference generated internally. In this mode, ui[5:0] control the frequency of the triangular reference. ui[7] enables pwm output when high. PWM is output on uo[7].

How to test

Set ui[7] to high. Measure the output on uo[7]. Flick the other input switches and see what happens!

External hardware

None, but an LED and resistor would be nice.

Pinout

#	Input	Output	Bidirectional
0	Manual mode PWM reference control	PWM output	
1	Manual mode PWM reference control		
2	Manual mode PWM reference control		
3	Manual mode PWM reference control		
4	Manual mode PWM reference control		
5	Manual mode PWM reference control		
6	Toggle PWM breathe or manual mode		
7	Enable PWM output		

INTERCAL ALU [817]

- Author: Rebecca G. Bettencourt
- Description: An ALU for the five operators of the INTERCAL programming language.
- GitHub repository
- HDL project
- Mux address: 817
- Extra docs
- Clock: 0 Hz

How it works

As an educational project, it is inevitable that Tiny Tapeout would attract various pedagogical examples of common logic circuits, such as ALUs. While ALUs for common operations such as addition, subtraction, and binary bitwise logic are surprisingly common, it is much rarer to encounter one that can calculate the five operations of the INTERCAL programming language. Due to either the cost-prohibitive nature of Warmenhoian logic gates or general lack of interest, such a feat has never been performed until now. With chip production finally within reach of the average person, all it takes is one person who has more dollars than sense to design the fabled INTERCAL ALU (Arrhythmic Logic Unit).

The pin assignments for this design are roughly as follows. The /OE (output enable) and /WE (write enable) signals are active low, so should be set HIGH by default.

#	Dedicated Input	Dedicated Output	Bidirectional I/O
0	A0 (address)	D0 (output only)	D0 (input and output only)
1	A1 (address)	D1 (output only)	D1 (input and output only)
2	S0 (selector)	D2 (output only)	D2 (input and output only)
3	S1 (selector)	D3 (output only)	D3 (input and output only)
4	S2 (selector)	D4 (output only)	D4 (input and output only)
5	S3 (selector)	D5 (output only)	D5 (input and output only)
6	/OE (output enable)	D6 (output only)	D6 (input and output only)
7	/WE (write enable)	D7 (output only)	D7 (input and output only)

This ALU has two 32-bit registers, B and A (in no particular order). (These may also be thought of as four 16-bit registers, AL, AH, BL, and BH.) To write a byte to a register, set A0 and A1 to the byte address, set S0 LOW for the A register or HIGH for the B register, set S1 through S3 LOW, set the bidirectional I/O pins to the byte

value, set /WE LOW, then set /WE HIGH again. (Do not set S1 through S3 HIGH when writing, or else something unpredictable will happen, most likely nothing.)

To read a register or result, set A0 and A1 to the byte address, set S0 through S3 to the desired operation, set /OE LOW, read the byte value from the bidirectional I/O pins, then set /OE HIGH. Results can also be read from the dedicated outputs; the dedicated outputs are not affected by the /OE signal, as they do not need to care about your feelings.

The operations supported are listed below. An attempt was made to make it understandable.

						Address				
						A	3	2	1	0
						A1	1	1	0	0
S	S3	S2	S1	S0	Operation	A0	1	0	1	0
0	0	0	0	0	A		AH		AL	
1	0	0	0	1	B		BH		BL	
2	0	0	1	0	AND16		& AH		& AL	
3	0	0	1	1	AND32		& A			
4	0	1	0	0	OR16		V AH		V AL	
5	0	1	0	1	OR32		VA			
6	0	1	1	0	XOR16		? AH		? AL	
7	0	1	1	1	XOR32		? A			
8	1	0	0	0	MINGLE16L		AL \$ BL			
9	1	0	0	1	MINGLE16H		AH \$ BH			
10	1	0	1	0	SELECT16		AH~BH		AL~BL	
11	1	0	1	1	SELECT32		A ~ B			

Operations 0 and 1 simply return the current value of the A or B register, respectively. This corresponds with the values of S0 through S3 used in write mode. This is not unintentional. This might also explain why S1 through S3 must be LOW in write mode.

Operations 2 through 7 correspond to INTERCAL's unary AND, unary OR, and unary XOR operators, represented by ampersand (&), book (V), and what (?), respectively. From the INTERCAL manual:

Operations 2, 4, and 6 work on the 16-bit halves of the A register independently, while operations 3, 5, and 7 work on the 32-bit whole of the A register.

Operations 8 and 9 correspond to INTERCAL's *interleave* (also called *mingle*) operator, represented by big money (\$). From the INTERCAL manual:

Operation 8 returns the interleave of the lower halves of A and B, while operation 9 returns the interleave of the upper halves of A and B. (Should the chip fabrication process allow for it, operation 8½ will, of course, return the interleave of the middle halves of A and B.)

Operations 10 and 11 correspond to INTERCAL's *select* operator, represented by sqiggle (~). From the INTERCAL manual:

To help understand the select operator, the INTERCAL manual also provides a helpful circuitous diagram.

Use of operations 12 and above is not recommended, unless undefined behavior is required.

How to test

The following example calculations found in the INTERCAL manual should be particularly illuminating.

S	A	B	F
MINGLE16L (8)	0	256	65536
MINGLE16L (8)	65535	0	2863311530
MINGLE16L (8)	0	65535	1431655765
MINGLE16L (8)	255	255	65535
SELECT16 (10)	51	21	5 *
SELECT16 (10)	179	201	9
SELECT16 (10)	201	179	17
SELECT16 (10)	179	179	31
SELECT16 (10)	201	201	15
AND16 (2)	77		4
OR16 (4)	77		32879
XOR16 (6)	77		32875

These test cases are included in the (unfortunately Python and not INTERCAL) test.py file. As these are likely more INTERCAL operations than any sensible person will ever perform, they should be sufficient for testing purposes. However, for curiosity's sake, an extensive set of additional test cases have also been included.

- Not found in the INTERCAL manual.

External hardware

The ALU may be used without external hardware, although seeing the output values may present a challenge. Instead, it is recommended to use a microcontroller of some sort to drive the inputs and read the outputs, as microcontrollers are designed to do. The implementation of the rest of the INTERCAL language is left as an exercise for the reader.

Further reading

The INTERCAL Programming Language Revised Reference Manual by Donald R. Woods and James M. Lyon with revisions by Louis Howell and Eric S. Raymond (can recommend highly enough)

Pinout

#	Input	Output	Bidirectional
0	A0 (address)	D0	D0
1	A1 (address)	D1	D1
2	S0 (selector)	D2	D2
3	S1 (selector)	D3	D3
4	S2 (selector)	D4	D4
5	S3 (selector)	D5	D5
6	/OE (output enable)	D6	D6
7	/WE (write enable)	D7	D7

Universal Binary to Segment Decoder [819]

- Author: Rebecca G. Bettencourt
- Description: Decodes various binary codes to various segmented displays.
- GitHub repository
- HDL project
- Mux address: 819
- Extra docs
- Clock: 0 Hz

How it works

This project is composed of four modules:

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

BCD to seven segment decoder

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

0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
0	:	2	3	4	5	6	7	8	9

1010 1011 1100 1101 1110 1111						
V0=0						
V1=0						
V2=0						
V0=1						
V1=0	c	d	u	e	t	
V2=0						
V0=0			-	-	-	
V1=1	o	o	-	-	-	
V2=0						
V0=1						
V1=1	0	:	2	3	4	5
V2=0						

1010 1011 1100 1101 1110 1111						
V0=0						
V1=0		=	=	=	-	
V2=1	-	-	-	-	-	
V0=1						
V1=0	-	L	C	F	E	
V2=1	-	-	-	-	-	
V0=0						
V1=1	-	E	H	L	P	
V2=1	-	-	-	-	-	
V0=1						
V1=1	A	b	C	d	E	F
V2=1						

The signals used in this mode are:

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

The pin assignments in this mode are:

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

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

ASCII to seven segment decoder

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

FS=0:

0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111

D6=0 D5=1 D4=0	-	'	''	!!	5	'	e	-	C	3	o	4	J	-	-	P
D6=0 D5=1 D4=1	0	1	2	3	4	5	6	7	8	9	-	-	-	=	F	P
D6=1 D5=0 D4=0	E	A	b	C	d	E	F	G	H	I	J	K	L	O	N	O
D6=1 D5=0 D4=1	P	9	r	S	7	U	v	8	E	Y	Z	C	4	3	n	-
D6=1 D5=1 D4=0	-	8	b	c	d	E	F	g	h	7	J	F	I	n	o	
D6=1 D5=1 D4=1	P	9	r	S	t	U	v	8	=	Y	Z	2	4	I	-	

FS=1:

0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111

D6=0 D5=1 D4=0	-	'	''	!!	e	-	c	o	q	u	-	.	r
D6=0 D5=1 D4=1	0	1	2	3	4	5	6	7	8	9	-	-	c	-	p
D6=1 D5=0 D4=0	E	A	b	C	d	E	F	G	H	-	U	A	L	O	N
D6=1 D5=0 D4=1	P	Q	R	S	T	U	V	W	X	=	Y	Z	E	H	S
D6=1 D5=1 D4=0	-	-	b	c	d	U	R	Q	H	-	J	A	U	N	O
D6=1 D5=1 D4=1	P	Q	R	S	T	U	V	W	X	=	Y	Z	E	I	Z

The signals used in this mode are:

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

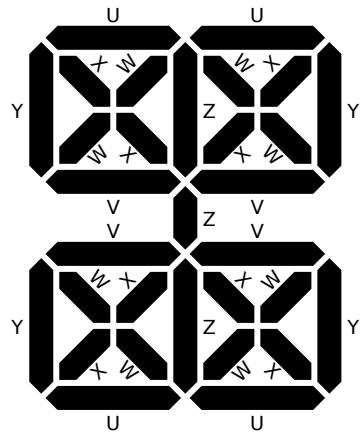
The pin assignments in this mode are:

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

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

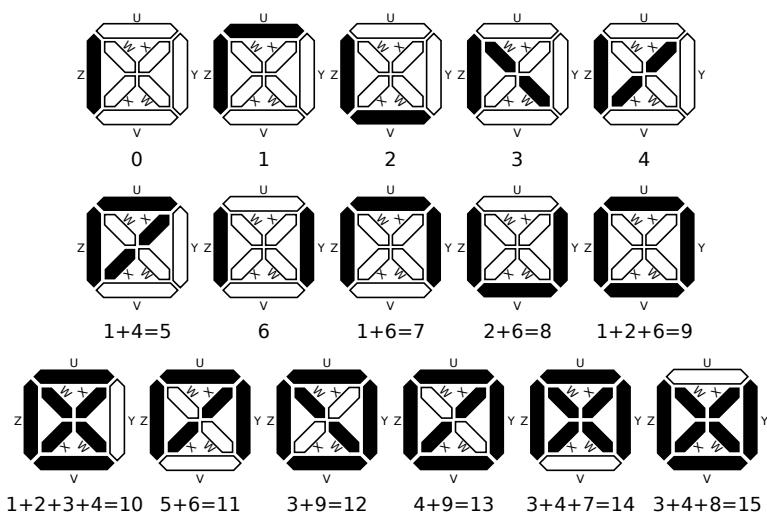
Dual BCD to Cistercian numeral decoder

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



The patterns produced for each input value are shown below.

Patterns as seen in top right (units) position:



The signals used in this mode are:

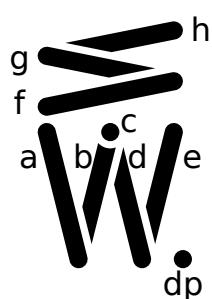
- **/AL** - Active low. If HIGH, outputs will be HIGH when lit. If LOW, outputs will be LOW when lit.
- **/BI** - Blanking input. If LOW, all segments will be blank regardless of other inputs, including **/LT1** and **/LT2**.
- **/LT1** - Lamp test for digit 1. When **/BI** is HIGH and **/LT1** is LOW, all segments for digit 1 will be lit.
- **/LT2** - Lamp test for digit 2. When **/BI** is HIGH and **/LT2** is LOW, all segments for digit 2 will be lit.
- **A1, B1, C1, D1** - BCD input for digit 1 from least significant bit **A1** to most significant bit **D1**.
- **A2, B2, C2, D2** - BCD input for digit 2 from least significant bit **A2** to most significant bit **D2**.
- **U1, V1, W1, X1, Y1** - Outputs for digit 1 on a Cistercian segmented display.
- **U2, V2, W2, X2, Y2** - Outputs for digit 2 on a Cistercian segmented display.

The pin assignments in this mode are:

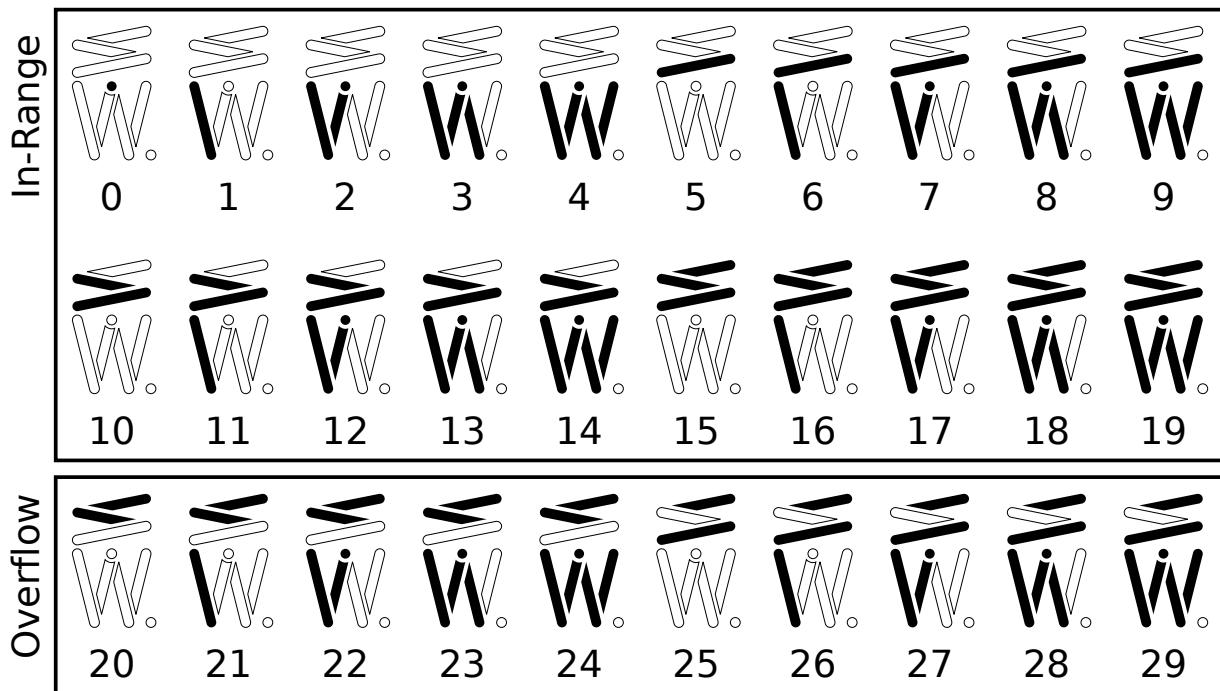
	Dedicated Input	Dedicated Output	Bidirectional
0	A1	Segment U1	Output - Y1
1	B1	Segment U2	Output - Y2
2	C1	Segment V1	Input - /LT1
3	D1	Segment V2	Input - /LT2
4	A2	Segment W1	Input - /BI
5	B2	Segment W2	Input - /AL
6	C2	Segment X1	Input - LOW
7	D2	Segment X2	Input - HIGH

BCV to Kaktovik numeral decoder

This mode converts a *vigesimal* (base 20) digit in BCV (binary-coded vigesimal) to its representation on the segmented display for Kaktovik numerals shown below.



The patterns produced for each input value are shown below.



The signals used in this mode are:

- **/AL** - Active low. If HIGH, outputs will be HIGH when lit. If LOW, outputs will be LOW when lit.
- **/BI** - Blanking input. If LOW, all segments will be blank regardless of other inputs, including **/LT**.
- **/LT** - Lamp test. When **/BI** is HIGH and **/LT** is LOW, all segments will be lit.
- **/RBI** - Ripple blanking input. If the BCV value is zero and **/RBI** is LOW, all segments will be blank.
- **/VBI** - Overflow blanking input. If the BCV value is out of range and **/VBI** is LOW, all segments will be blank.
- **A, B, C, D, E** - BCV input from least significant bit **A** to most significant bit **E**.
- **a, b, c, d, e, f, g, h** - Outputs for a Kaktovik segmented display.
- **/RBO** - Ripple blanking output. HIGH when BCV value is nonzero or **/RBI** is HIGH.
- **V** - Overflow. HIGH when BCV value is out of range (greater than or equal to 20).

The pin assignments in this mode are:

	Dedicated Input	Dedicated Output	Bidirectional
0	A	Segment a	Output - h
1	B	Segment b	Output - V
2	C	Segment c	

	Dedicated Input	Dedicated Output	Bidirectional
3	D	Segment d	Input - /LT
4	E	Segment e	Input - /BI
5		Segment f	Input - /AL
6	/VBI	Segment g	Input - HIGH
7	/RBI	/RBO	Input - HIGH

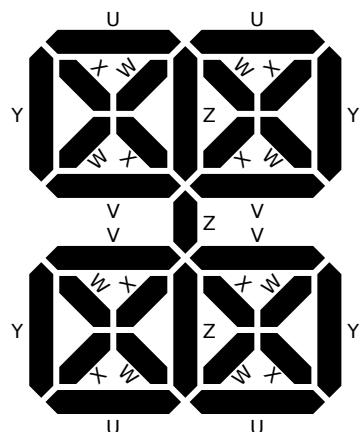
How to test

The test directory includes extensive tests for each of the four modules.

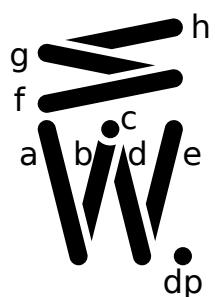
External hardware

For the BCD and ASCII modes, a standard seven-segment display is used.

For the Cistercian mode, a segmented display like the one below is used. There are design files for such a display here.



For the Kaktovik mode, a segmented display like the one below is used. There are design files for such a display here.



Pinout

#	Input	Output	Bidirectional
0	A; D0; A1; A	Segment a; U1; a	X6; X6; Y1; h
1	B; D1; B1; B	Segment b; U2; b	X7; X7; Y2; V
2	C; D2; C1; C	Segment c; V1; c	X9; X9; /LT1; -
3	D; D3; D1; D	Segment d; V2; d	/LT; FS; /LT2; /LT
4	V0; D4; A2; E	Segment e; W1; e	/BI (blanking input)
5	V1; D5; B2; -	Segment f; W2; f	/AL (active low)
6	V2; D6; C2; /VBI	Segment g; X1; g	M0 (mode select)
7	/RBI; LC; D2; /RBI	/RBO; /LTR; X2; /RBO	M1 (mode select)

RO [833]

- Author: Arna Roy
- Description: Implementation of simple RO
- GitHub repository
- HDL project
- Mux address: 833
- Extra docs
- Clock: 20000000 Hz

How it works

The tt_um_roy1707018 module integrates two essential components:

Ring Oscillator-Based Buffer System which essentially a True Random Number Generator or TRNG (ro_buffer_counter) S-Box Cryptographic Component (ascon_sbox) Ring Oscillator-Based Buffer System (ro_buffer_counter) This module contains a buffer driven by two control signals: ro_activate_1: Controls the first set of ring oscillators (bit 0 of ui_in). ro_activate_2: Controls the second set of ring oscillators (bit 1 of ui_in). It also includes a 3-bit signal (bits 2 to 4 of ui_in) that selects a specific output from the buffer. The module comprises a total of 16 ring oscillators, split into two sets of 8. A 64-bit shift register within the submodule stores the last 64 bits of these oscillators' outputs. The selection bits determine which specific set of 8 values from the shift register is presented as the 8-bit output, which is then processed and connected to uo_out.

S-Box Cryptographic Component (ascon_sbox) The second submodule implements an S-Box, a crucial non-linear substitution step used in cryptographic algorithms like ASCON. This S-Box is activated by bit 7 of ui_in and receives bits 2 to 6 of ui_in as input, producing a 5-bit output.

Final Output The final output, uo_out, is the result of a bitwise XOR operation between TRNG and the S-Box. This combination effectively merges the functionalities of both components into a single output signal.

How to test

In the simulation level, from the testbench we sent different values to the input to see if the ring oscillators or SBOX are working correctly or not.

External hardware

No external hardware is needed for this design.

Pinout

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

CMOS design of 4-bit Signed Adder Subtractor [835]

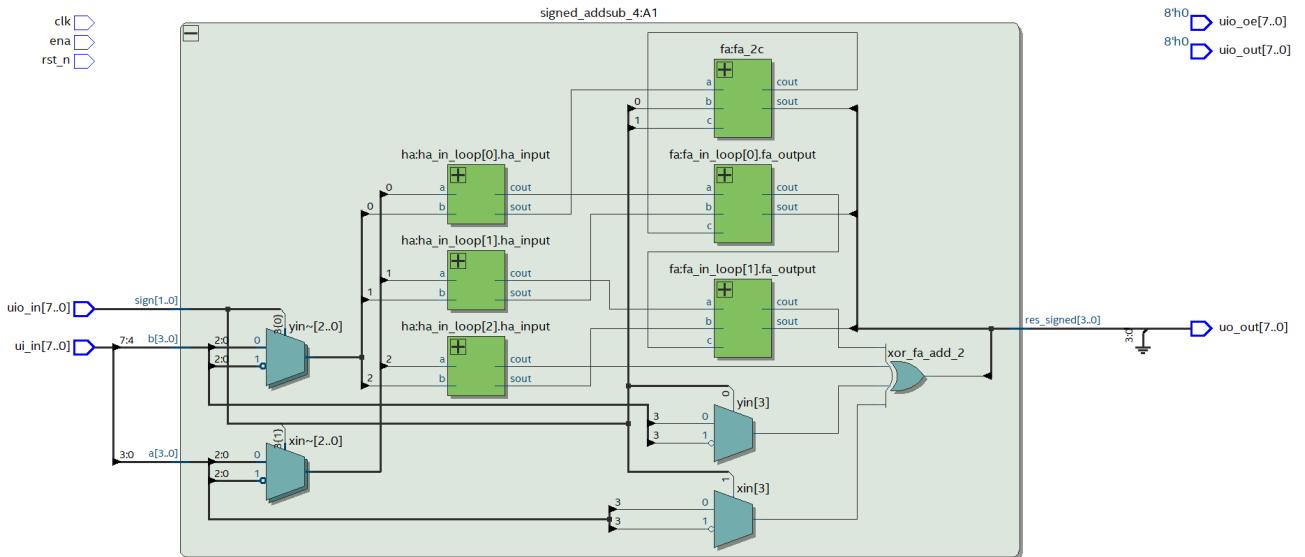
- Author: Vivek Chiranjit
- Description: The project is a signed binary 4-bit adder-subtractor module.
- GitHub repository
- HDL project
- Mux address: 835
- Extra docs
- Clock: 0 Hz

How it works

The project is a signed binary 4-bit adder-subtractor module. The module is constructed using muxes, half adders and full adders.

Depending on the sign[1:0] bits, the circuit can perform the following operations:

sign[1:0]	Operation
00	A + B
01	-A + B
10	A - B
11	-A - B



How to test

The signed_addsub_tb testbench includes extensive test cases for the 4-bit Signed adder-subtractor circuit. The design has been tested using QuestaSim.

External hardware

None

Pinout

#	Input	Output	Bidirectional
0	a0	s0	sign0
1	a1	s1	sign1
2	a2	s2	
3	a3	s3	
4	b0		
5	b1		
6	b2		
7	b3		

LaRVa CPU [836]

- Author: J. Arias
- Description: RISC-V CPU design
- GitHub repository
- HDL project
- Mux address: 836
- Extra docs
- Clock: 24000000 Hz

How it works

This project includes a RISC-V CPU (LaRVa) with a serial port and a few more peripherals. Memory has to be provided externally. An included bootloader allows the execution of programs loaded through the serial port. (See TinyTlaRVa.pdf file) As a last addition a JTAG interface is also included. (See jtag_LaRVaTT.pdf file)

How to test

Connect a serial port 8-bit, no parity, 115200 bps, and send an 'L'. The bootloader code should reply with another 'L'. For more complete tests an external board with SRAM memory and address latches has to be attached to the PMOD ports of the prototype board. Also, some testing could be carried out using the JTAG port.

External hardware

A memory board has to be attached to user PMOD connectors.

More docs

<https://www.ele.uva.es/~jesus/larva.pdf>

https://www.ele.uva.es/~jesus/larva_perif.pdf

#	Input	Output	Bidirectional
---	-------	--------	---------------

Pinout

#	Input	Output	Bidirectional
0	tck	xbh	xd[0]
1	tms	xlal	xd1
2	tdi	xlal	xd2
3	rxd	pwmout_tdo	xd[3]
4	gpi[0]	txd	xd[4]
5	gpi1	xhh	xd[5]
6	gpi2	xoeb	xd[6]
7	gpi[3]	xweb	xd[7]

Patater Demo Kit Waggling Rainbow on a Chip [837]

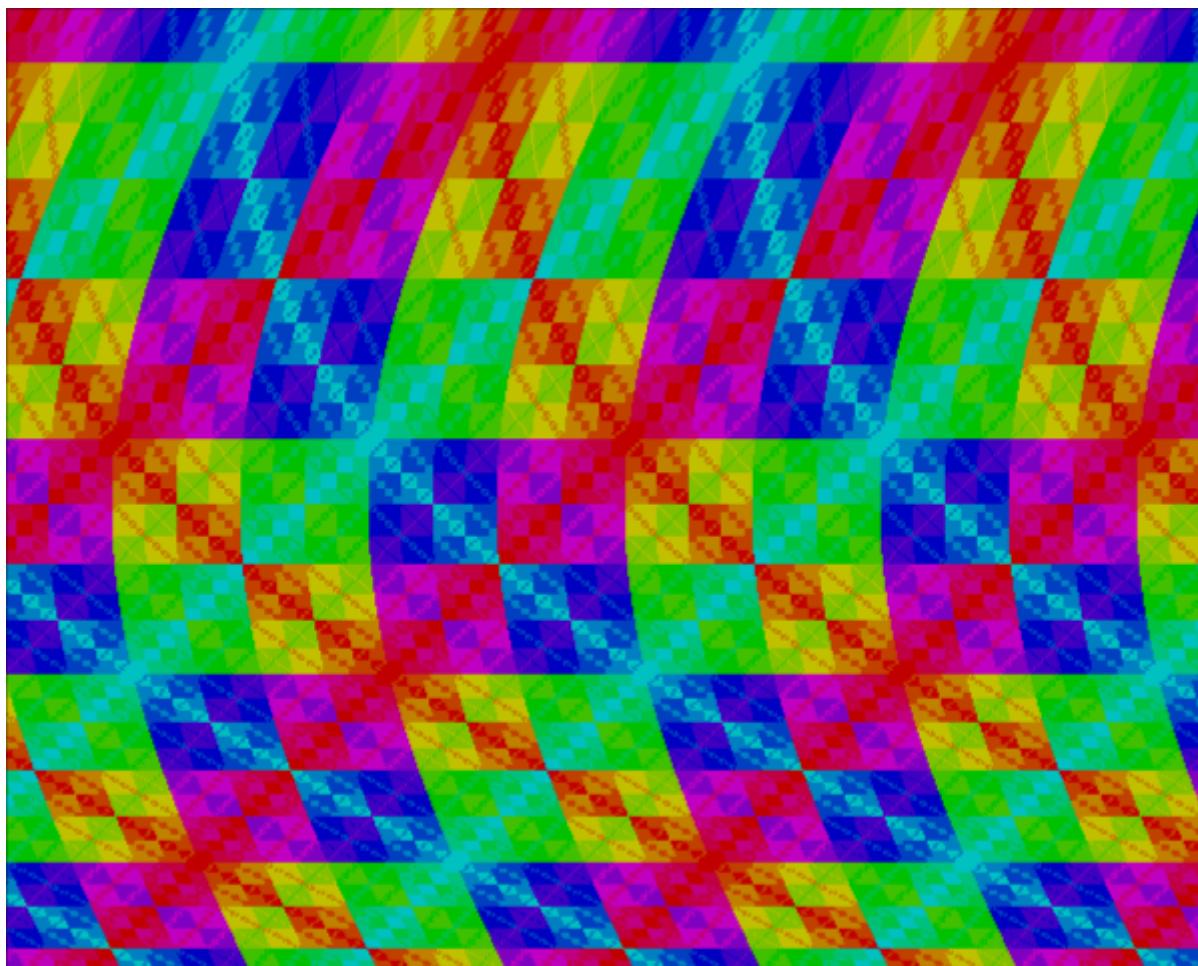
- Author: Jaeden Amero
- Description: A 6-bit Waggling Rainbow demo
- GitHub repository
- HDL project
- Mux address: 837
- Extra docs
- Clock: 25175000 Hz

How it works

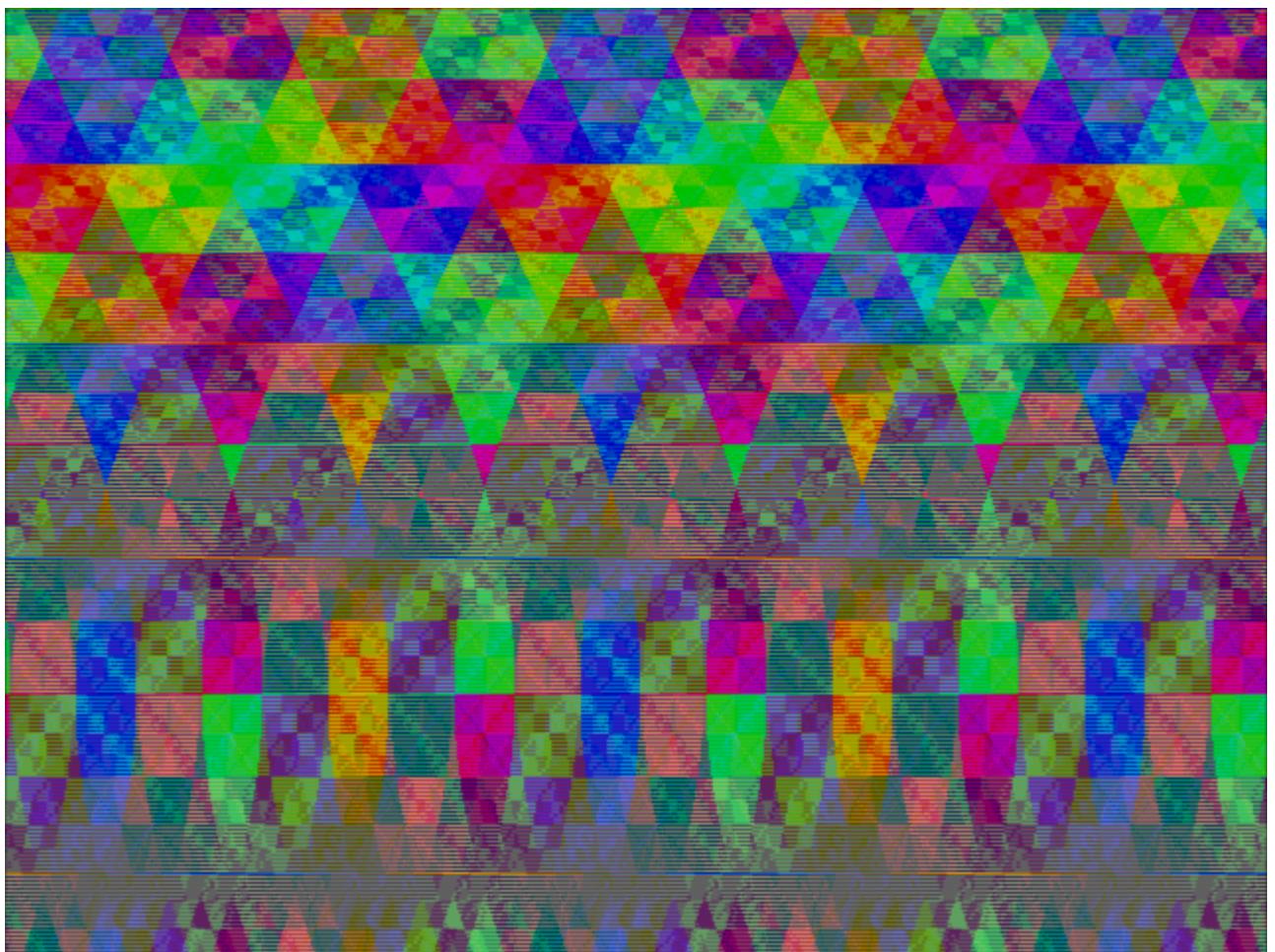
This design outputs a waggling 6-bit rainbow demo on VGA.

The demo will change effect based on inputs on ui_in.

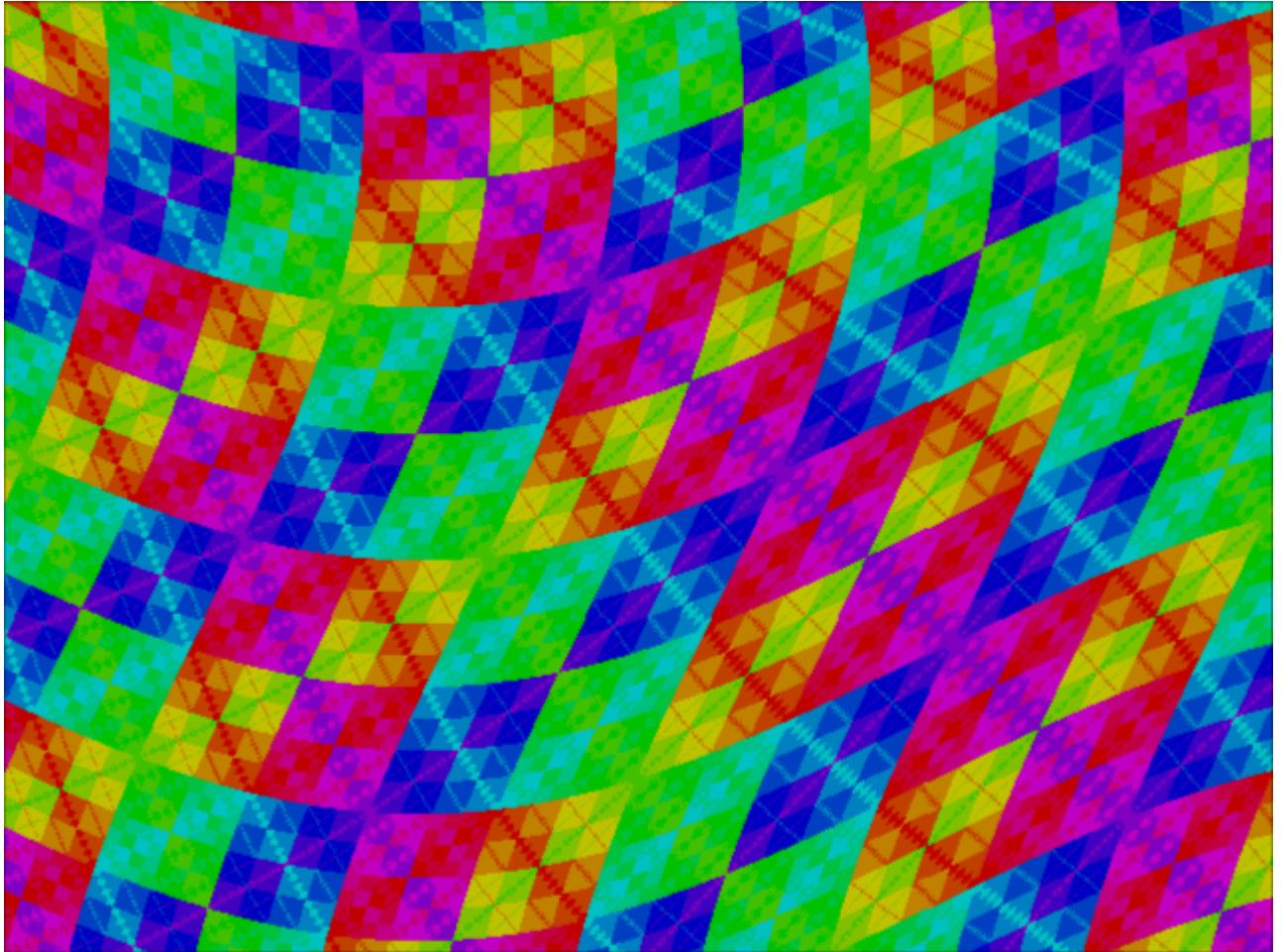
Pin	Pin Name	Setting	Effect
ui_in[0]	DUAL_EN	Dual mode	Horizontally flips the image each scan line
ui_in[1]	HWAVE_EN	H wave	Enables use of horizontal waves
ui_in[4:2]	P0_OFF_{2,1,0}	P0 offset	Controls the speed of the H wave
ui_in[7:5]	P1_OFF_{2,1,0}	P1 offset	Controls the speed of the V wave



Screenshots
Default



Dual Mode



H Wave

Video A video of a different (software rather than hardware) implementation, of the wagging rainbow effect can be found at https://www.youtube.com/watch?v=AxT45_7WZUQ.

How to test

If wanting to test without hardware, use the VGA playground. Copy and paste the contents of the entire `src/project.v` file into the playground's text editor, replacing all previous content. Then, change the name of the module from `tt_um_patater_demokit` to `tt_um_vga_example` and the simulator will start running in your browser.

If testing with hardware, use a TinyVGA PMOD. Clock the design with 25.175 MHz as described in `info.yaml` (25.157 MHz is standard for 60 Hz 640x480 VGA video).

If testing with lower level simulation tools, an incomplete cocotb test bench (`test/test.py`) is provided. Passing the tests in the cocotb bench is no guarantee that the design will work.

External hardware

External hardware required:

- TinyVGA PMOD

Pinout

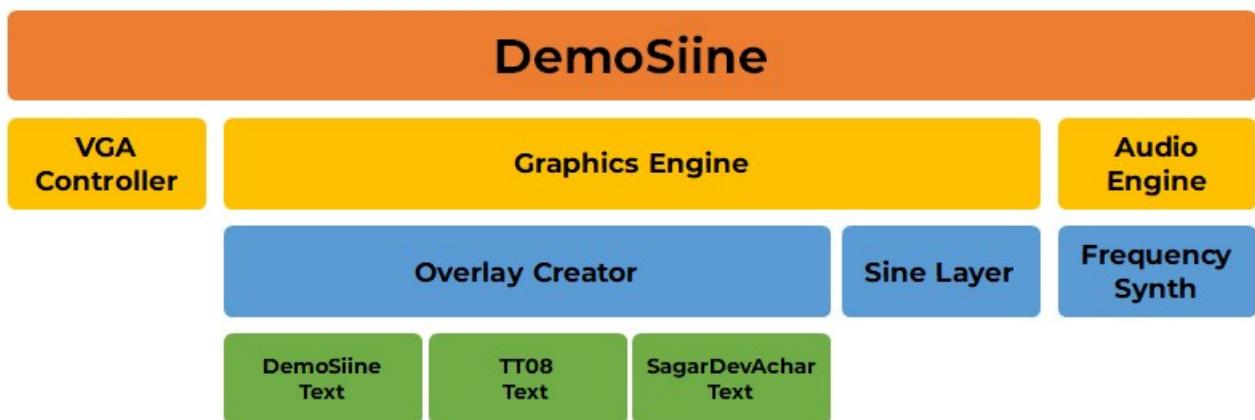
#	Input	Output	Bidirectional
0	DUAL_EN	R1	
1	HWAVE_EN	G1	
2	P0_OFF_0	B1	
3	P0_OFF_1	VSync	
4	P0_OFF_2	R0	
5	P1_OFF_0	G0	
6	P1_OFF_1	B0	
7	P1_OFF_2	HSync	

DemoSiine [839]

- Author: SagarDevAchar
- Description: A Wavy and Rainbowy TT08 Demoscene Submission
- GitHub repository
- HDL project
- Mux address: 839
- Extra docs
- Clock: 25000000 Hz

How it works

The project structure is as shown below:



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

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

External hardware

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

How to test

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

Input Configurations

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

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

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

Previews

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



INPUT = xx000000 (Default)



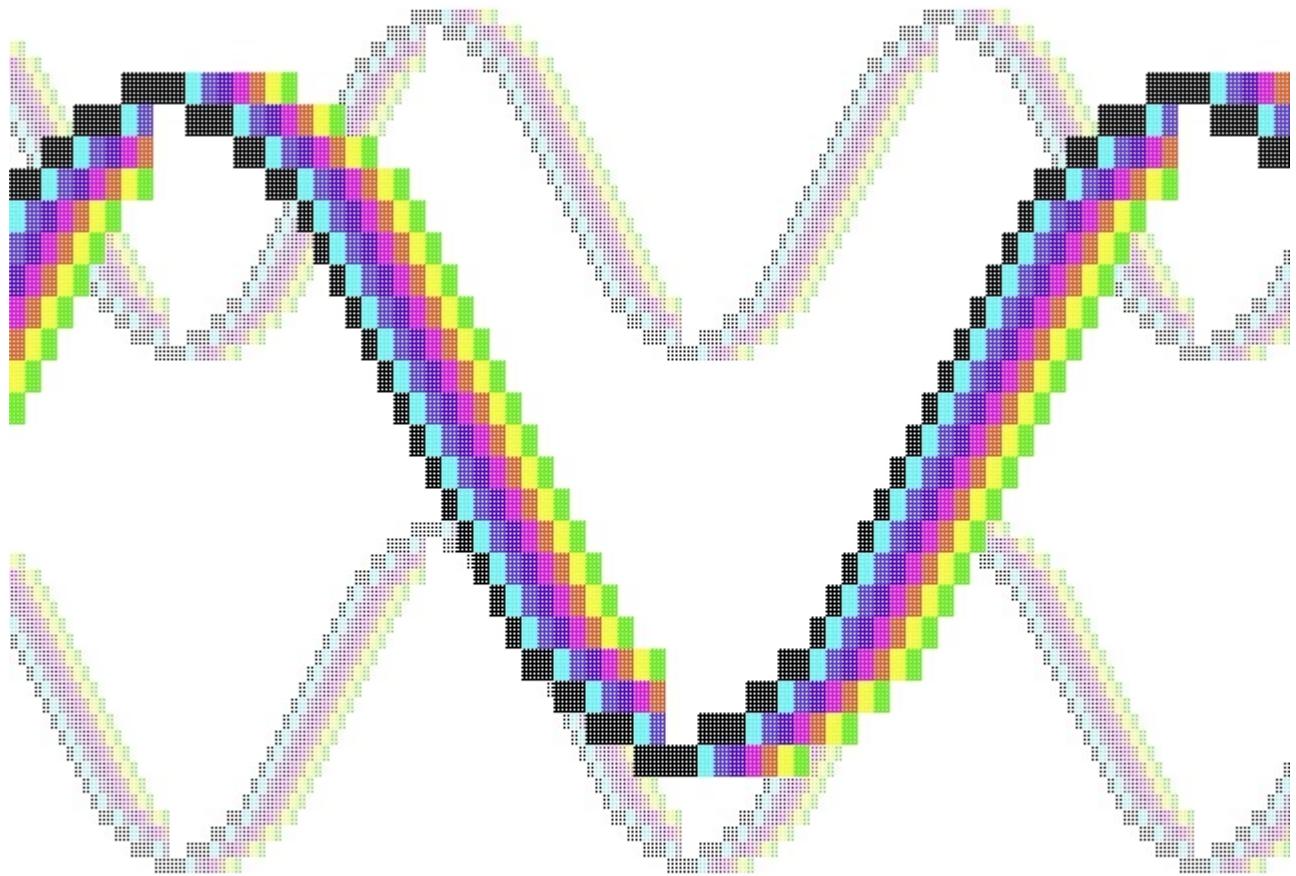
INPUT = $x \times 1 \times 1000$



INPUT = xx100001



INPUT = xx110110



INPUT = xx0x1001



INPUT = xx010110

Pinout

#	Input	Output	Bidirectional
0	Frame Positive / Negative	Video Red MSB	
1	Enable / Disable Little Sine Layer	Video Green MSB	
2	Enable / Disable Big Sine Layer	Video Blue MSB	
3	Enable / Disable Overlay	Video V-Sync	
4	Toggle Overlay Style	Video Red LSB	
5	Toggle Background Style	Video Green LSB	
6	Run / Stop Animation	Video Blue LSB	
7	Play / Pause Audio	Video H-Sync	Audio Output

"SQUARE-1": VGA/audio demo [840]

- Author: Zachary Catlin
- Description: On video: munching squares. On audio: the logistic map.
- GitHub repository
- HDL project
- Mux address: 840
- Extra docs
- Clock: 25200000 Hz

How to test

Assuming the ASIC is connected to the TT demo board and suitable interface electronics have been connected (see “External hardware”), select the `tt_um_zec_square1` project to get started. If `rst_n` is not automatically set to logic high upon selection, you’ll need to manually disable the reset. Enable the reset again when you’re done.

If not using the demo board, you’ll need to supply the ASIC with a 25.175 MHz or 25.200 MHz clock, do the appropriate interactions with the project-selection logic to select `tt_um_zec_square1`, and use the pinout to connect to video and audio output devices. Note: `y1` and `y0` are the high-order and low-order bits (respectively) of color component `y`.

The video part of the demo repeats with a cycle time of ~8.5 seconds, while the audio part repeats with a cycle time of just under 2 minutes.

External hardware

Assuming the ASIC is connected to the TT demo board, VGA output is obtained by connecting a Tiny VGA Pmod or compatible module to the OUTPUT Pmod connector, and audio output is obtained by connecting a Tiny Tapeout Audio Pmod to the BIDIR Pmod connector.

How it works

SQUARE-1 contains a VGA-compatible video demo and an independent audio demo, described separately below.

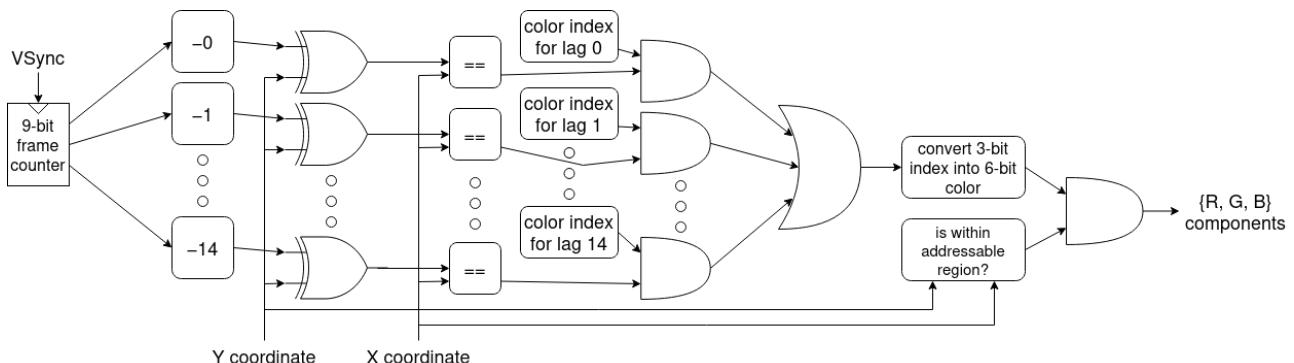
Video While the demoscene dates to the mid-1980s, people have been making aesthetically-interesting graphics with a tiny amount of code for much longer. One of the earliest “display hacks” is munching squares, first implemented c. 1962 on MIT’s PDP-1 (hence this demo’s name). The original version has feedback and user-configurability (see Norbert Landsteiner’s write-up for more details and a PDP-1 emulator), but a simple variant requires only two N -bit variables— t , a frame counter, and y , a row counter, used thus:

```
t ← 0
loop
    wait for end of frame
    t ← (t + 1) mod  $2^N$ 
    for y ← 0 to  $2^N - 1$ 
        plot (t XOR y, y)
```

As the algorithm has so little state and involves simple operations, a “racing the beam” implementation requires little silicon area. SQUARE-1 uses $N = 9$ and accepts that the bottom bit of the square gets lost off the 640×480 screen.

However, a simple implementation of the algorithm would not *look* much like the original version! PDP-1 munching squares uses a Type 30 point display, which was built around a radar-scope CRT using P7 phosphor. P7 is actually a combination of two substances—a bright, short-persistence (decay constant ~ 20 microseconds) far-blue phosphor excited by the electron beam, and a dimmer, long-persistence (main decay constant ~ 100 milliseconds, but with a long tail lasting several seconds) yellow phosphor excited by the light from the blue phosphor. As a result, the plotted points have a white or blue-white appearance, then become yellow and visibly fade away.

Fortunately, since each frame only has one point in each line, and said point is different in each frame, it’s easy to parallelize an emulation of persistence, which is done in `src/project.v`, which conceptually works like this:



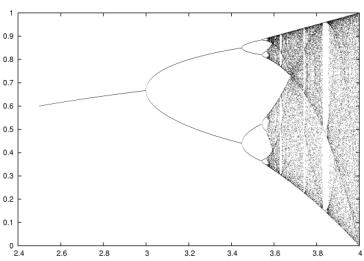
Apart from the VSync/HSync/coordinate-generating module, it’s almost entirely combinational logic. SQUARE-1 simulates 14 frames ($\sim 1/4$ second) of persistence prior to

the current frame—not quite a Type 30, but enough to get the feel of the thing on modern displays.

Audio The audio demo is a sonification of the logistic map. To give a quick overview, the following iteration:

$$x_{i+1} \leftarrow rx_i(1 - x_i)$$

maps values of $x \in (0, 1)$ to values in $(0, 1)$ when $r \in (1, 4)$. When $r \in (1, 3]$, the sequence of x_i values converges to a single value (the *attractor*), but much more interesting behavior happens when $r \in (3, 4)$:

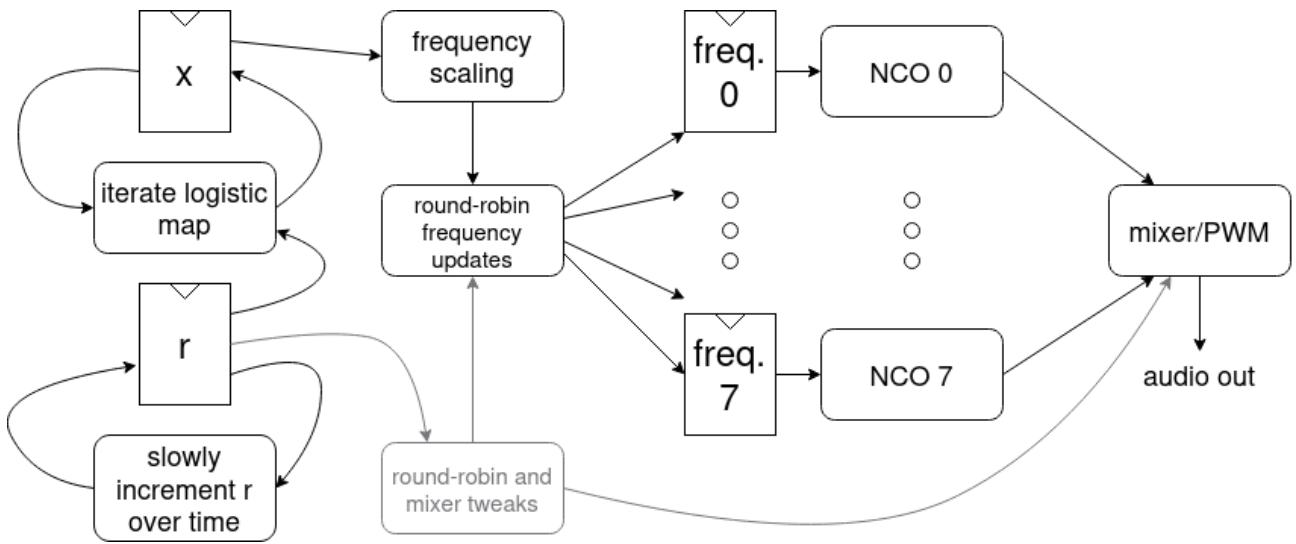


Credit: Ap on en.wikipedia.org

First, the attractor becomes a period-2 cycle, then period-4, -8, -16... and then it exhibits chaotic behavior. That iteratively applying a quadratic polynomial would result in such behavior came as quite a surprise back in the 1960s, and to this day the logistic map is a popular demonstration of mathematical chaos in a simple system.

So, what does it mean to turn the logistic map into a sound? The way SQUARE-1 does it, values of x_i at a given r are scaled from $(0, 1)$ to approximately $(200, 1200)$ Hz, which are then used as the frequencies of an ensemble of 8 square-wave generators. The square waves are then added together and used as the input to a PWM generator, the last providing the sound output. r is varied to cover the range $[\frac{17}{16}, 4)$ over a period of ~2 minutes, varying faster over $r < 3$ to get to the good stuff sooner.

Finally, over a few small portions of the chaotic region, we change the number of square-wave generators that get frequency updates and get mixed together. The reason is that within the chaotic region, there are islands of periodicity, the largest of which have attractors of period 3, 5, and 6. Tweaking the number of active generators to be a multiple of the period leads to better-sounding results within the islands.



Greetz

Eh, I'm not *that* social...

...Hi, Mom! Hi, Dad!

Well, also, thanks to the organizers of the TT08 demoscene competition for finally inspiring me to get off my rear and go sculpt some silicon. Thanks as well to the open source EDA and silicon communities for making all this feasible.

Pinout

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

Munch [841]

- Author: bytex64
- Description: Displays munching squares through VGA PMOD
- GitHub repository
- HDL project
- Mux address: 841
- Extra docs
- Clock: 25175000 Hz

How it works

This generates VGA output for a munching squares animation plus some other stuff, and some simple music. It uses the VGA and audio PMODs listed below.

Clock generation A clock generator module divides the main clock `clk` and `vsync` into two derived clocks - a 393kHz PWM clock for audio output, and a 10Hz “audio tick” clock that drives the pattern sequencer. The clock generator also provides counters for PWM, volume modulation, and audio pattern sequencing.

LFSR An 11-bit LFSR provides a noise source. It is an XNOR type, shifting down towards the LSB and inserting the new bit at MSB. XNOR taps are on bits 0 and 2. Bits 0-5 of the LFSR register are used to provide a noise channel and randomized video noise dithering.

Video The video output is the standard 640x480 @ 60Hz, using a 25.175MHz pixel clock and negative polarity HSYNC/VSYNC. Timing is implemented with a simple two-counter design shamelessly stolen from the Tiny Tapeout VGA playground.

A fixed palette of eight colors is used, and eight brightness levels are created by mixing random bits with the 2-bit per channel brightness levels. Video is output on three layers - the background and layers 0 and 1. A non-black pixel overrides a pixel on any lower layer.

Audio Audio comes from a basic PSG inspired by the SN76489. There are four channels of sound based on 12-bit timers - three pulse channels and one noise channel. The only real difference between a pulse channel and a noise channel is that the pulse channel flips state when the timer counts down, and the noise channel takes a random state from the LFSR. Each channel also has a two-bit volume level.

The 25.175MHz clock is divided by a 6-bit counter to create a 393KHz PWM output. 6-bits gives 64 possible levels. The PWM high period is a simple sum of the four channels' volumes at any given instant (multiplied by two with the low bit dithered from the LFSR). This does mean the PWM will glitch if volume levels change in the middle of a PWM cycle, but that's fine in practice since it's all low-pass filtered anyway.

The four channels are programmed through a sequencer that provides note and volume data to the PSG. The sequencer is clocked by dividing VSYNC by 6, so the sequencer moves through pattern rows at 10Hz, or 600 ticks per minute. Each pattern of 16 ticks represents one measure, four beats, which means the music proceeds at 150 BPM.

The sequencer cycles through pre-programmed patterns of notes. Note timer data is read from an indirection list of notes, then connected directly to the PSG reload values. This does mean the oscillators are not synchronized at note start. Volumes are modulated through a single repeating pattern per channel, indexed from the top two bits of the sequencer div-by-6 clock divider. This means the volume is a three-step pattern cycling at the start of each pattern tick.

Text Generator On-screen text uses a segmented approach, where each segment is defined by a mathematical description of a line segment. Each character is then defined by which segments are off or on, like a multi-segment LED display. So text is generated at full resolution despite its large size; each character is 50x100 pixels.

The text generator is just a sequencer over an input bit stream, indexed by the horizontal and vertical position. In this implementation the input is at most six characters long. The text can be positioned arbitrarily, but for this demo it is fixed.

Stage Sequencer A slower stage clock is derived from the pattern clock. It ticks once every pattern cycle, and drives an overarching "stage sequencer". Each stage counts down for a pre-programmed number of patterns, then switches to the next. The stage number is used in various logic to change the text and colors over time.

Extra outputs In addition to the audio and video, the three highest bits of the internal pattern counter are output on `ui0_out[6:4]`. The two highest bits count out the four beats in a pattern, and bit 1 has a negative edge at the beginning of each beat. This could be used for beat synchronization with external systems - I just used it for debugging.

How to test

Set the input clock for 25.175MHz. The Pico/RP2040 can output 25.177MHz on GPOUT0 with a 125MHz main clock and a divider of 4 [integer part] and 247 [fractional part]. This worked on my TV.

Reset, and enjoy. :)

External hardware

- Leo's VGA PMOD
- Tiny Tapeout Audio Pmod

Pinout

#	Input	Output	Bidirectional
0		R1	
1		G1	
2		B1	
3		VSYNC	
4	R0		beat clock bit 1 (output)
5	G0		beat clock bit 2 (output)
6	B0		beat clock bit 3 (output)
7	Hsync		audio (output)

cfib Demoscene Entry [843]

- Author: Christian Fibich
- Description: Generates VGA video and PWM audio
- GitHub repository
- HDL project
- Mux address: 843
- Extra docs
- Clock: 50000000 Hz

How it works

My entry to the Tinytapeout Demoscene Competition.

It (pseudo-randomly) generates a soundtrack via PWM and displays a waveform via VGA.

How to test

Connect VGA and PWM Pmod.

Then just apply clock and (asynchronous) reset.

External hardware

The project uses:

- Tiny VGA Pmod via `uo_out[7:0]` (<https://github.com/mole99/tiny-vga>)
- Mike's audio Pmod via `ui0_out[7]` (<https://github.com/MichaelBell/tt-audio-pmod>)

Pinout

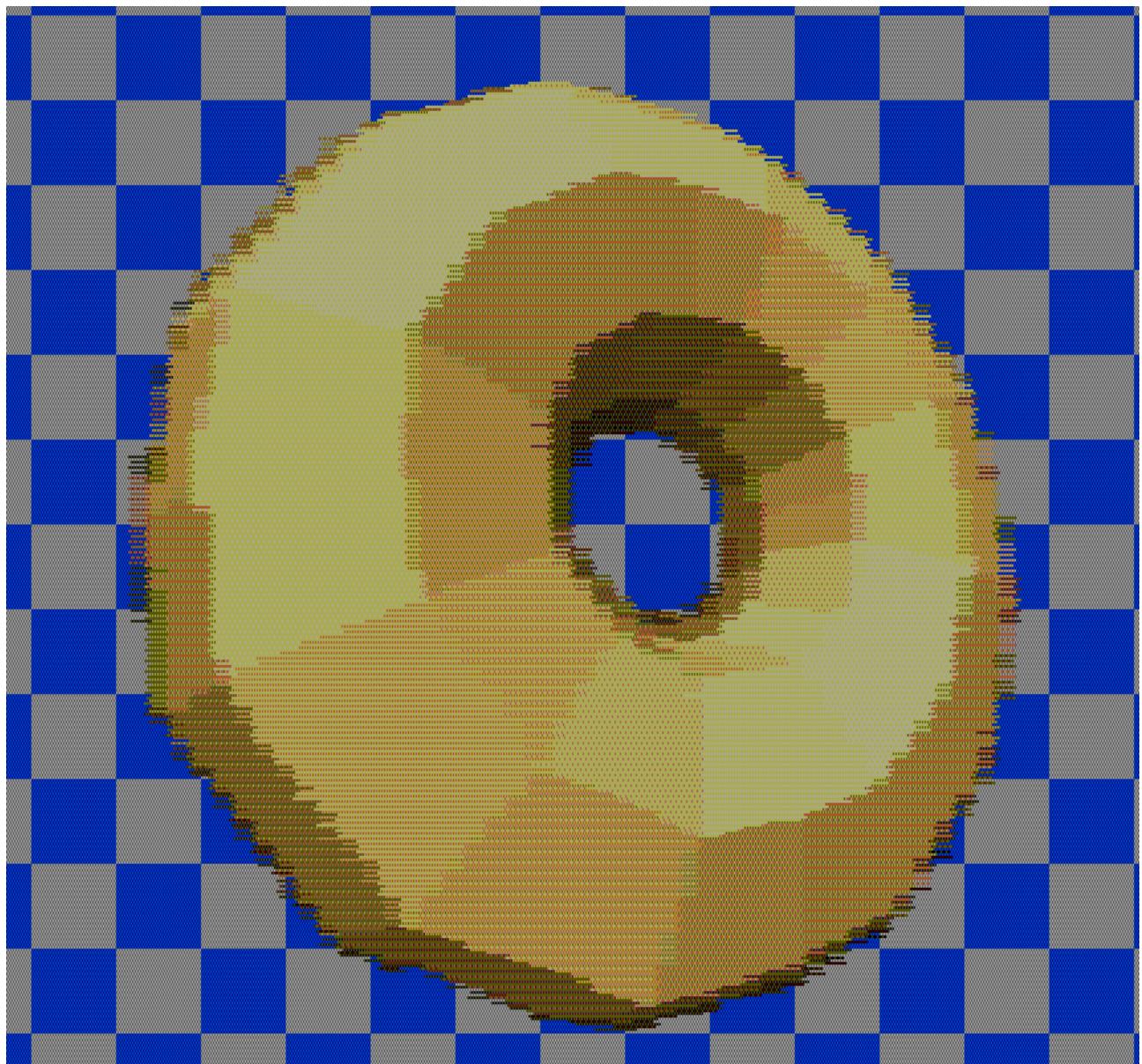
#	Input	Output	Bidirectional
0		r1	
1		g1	
2		b1	
3		vsync	
4		r[0]	

#	Input	Output	Bidirectional
5		g[0]	
6		b[0]	
7		hsync	pwm

VGA donut [844]

- Author: Andy Sloane
- Description: Renders a 3D torus on a VGA display
- GitHub repository
- HDL project
- Mux address: 844
- Extra docs
- Clock: 48000000 Hz

VGA Donut



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

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

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

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

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

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

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

Pinout

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

4-bit ALU [845]

- Author: Richard Xu, Louis Barbosa, Hallie Ho, Emmy Xu, Gia Bhatia, Emily Chen
- Description: The 4-bit ALU is designed to perform basic arithmetic and logical operations on 4-bit binary numbers
- GitHub repository
- HDL project
- Mux address: 845
- Extra docs
- Clock: 0 Hz

Project Datasheet: 4-Bit ALU

Overview The 4-Bit ALU module is a digital system designed to perform various arithmetic and logical operations on 4-bit binary numbers. It supports operations such as addition, subtraction, multiplication, division, and several logical operations. Additionally, it includes an encryption function that can be used to encrypt 4-bit inputs using an 8-bit key.

How it Works The module accepts two 4-bit binary numbers, a and b , and a 4-bit operation code (opcode) that determines the operation to be performed. The results are then output through the `uo_out` wire, while additional status information, such as carry out and overflow, is output through the `ui0_out` wire. The `ui0_oe` wire controls the enable or disable functionality for the `ui0_in` and `ui0_out` wires.

Operations

- **ADD:** Adds a and b , producing a 4-bit result and a carry out.
- **SUB:** Subtracts b from a , producing a 4-bit result and a borrow indication.
- **MUL:** Multiplies a and b , producing an 8-bit result.
- **DIV:** Divides a by b , producing a 4-bit quotient and remainder. Division by zero is handled by returning a zero result.
- **AND:** Performs a logical AND operation on a and b .
- **OR:** Performs a logical OR operation on a and b .
- **XOR:** Performs a logical XOR operation on a and b .
- **NOT:** Performs a logical NOT operation on a , with b being ignored.

Encryption Function

- **ENC:** Encrypts the inputs a and b using an 8-bit key derived from concatenating a and b (treated as an 8-bit value). The encryption function applies an XOR operation between this 8-bit concatenated value and a fixed 8-bit key (KEY). The result is an 8-bit encrypted value.

Encryption Details:

- **Key Generation:** The key for the encryption function is a fixed 8-bit value.
- **Encryption Operation:** The concatenated value of a and b (forming an 8-bit value) is XORed with the fixed 8-bit key.

How to Test

To test the 4-Bit ALU module, follow these steps:

1. Connect Inputs:

- Connect the `ui_in` wire to the 4-bit inputs a and b .
- Connect the `uo_in` wire to the 4-bit opcode.

2. Connect Outputs:

- Connect the `uo_out` wire to an 8-bit output display or register to observe the operation result.
- Connect the `uo_out` wire to observe the carry out and overflow status.

3. Signal Management:

- Ensure the `ena` signal is active (high).
- Provide a clock signal to the `clk` input.
- Optionally, use the `rst_n` signal to reset the module by pulling it low.

4. Operation Testing:

- Cycle through various opcode values and corresponding a and b inputs to verify the correct operation of the module.
- For encryption, ensure a and b are combined and XORed with the fixed key. Verify that the result matches expected encrypted values.

Pinout

#	Input	Output	Bidirectional
0	a[0]	result[0]	opcode[0]
1	a1	result1	opcode1
2	a2	result2	opcode2
3	a[3]	result[3]	opcode[3]
4	b[0]	result[4]	
5	b1	result[5]	
6	b2	result[6]	carry_out
7	b[3]	result[7]	overflow

Morse Code Keyer [847]

- Author: Brady Etz
- Description: Convert a keyed CW input to morse tones and 7-segment character output
- GitHub repository
- HDL project
- Mux address: 847
- Extra docs
- Clock: 12000000 Hz

How it works

Morse Keyer takes a paddle-type dit/dah signal ($\text{io}[0:1]$) and converts it to an auxilliary Morse code output ($\text{io}[4]$) and a buzzer tone ($\text{io}[5]$). The design outputs auxilliary dit/dah signals ($\text{io}[2:3]$) to send to other gear, like a radio. Additionally, it outputs to the demonstration board's seven-segment display ($\text{out}[7:0]$) to reveal the character you just completed as you key.

To use a straight-key input (or press a single button to key Morse code) set $\text{in}[0]$ HIGH. For Iambic paddles, set $\text{in}[0]$ LOW. To use Iambic keying type A, set in1 LOW. For Iambic-B, set in1 HIGH. (<https://ag6qr.net/index.php/2017/01/06/iambic-a-or-b-or-does-it-matter/>) WPM control is set between 7 WPM and ~100 WPM with $\text{in}[7:4]$ via the demo board dip switches. The timing element in this system divides the system clock first with a 512x prescaler, then feeds it into the variable delay below:

Control [7:4]	WPM	Clocks	Timer Preset
4'b0000	110.3	255	12'b000011111111
4'b0001	55.0	511	12'b000111111111
4'b0010	36.7	767	12'b001011111111
4'b0011	27.5	1023	12'b001111111111
4'b0100	22.0	1279	12'b010011111111
4'b0101	18.3	1535	12'b010111111111
4'b0110	15.7	1791	12'b011011111111
4'b0111	13.7	2047	12'b011111111111
4'b1000	12.2	2303	12'b100011111111
4'b1001	11.0	2559	12'b100111111111
4'b1010	10.0	2815	12'b101011111111
4'b1011	9.2	3071	12'b101111111111
4'b1100	8.5	3327	12'b110011111111
4'b1101	7.9	3583	12'b110111111111

Control [7:4]	WPM	Clocks	Timer Preset
4'b1110	7.3	3839	12'b111011111111
4'b1111	6.9	4095	12'b111111111111

WARNING: The auxilliary Morse output MUST NOT be used as a raw radio TX control for a homemade radio. This is because the keying interface must control the transmitted wave shape to maintain acceptable RF bandwidth. Be a good RF neighbor. Always use the provided keyer inputs for your radio. These are typically provided with a 3.5mm TRS jack, with Sleeve = GND, Ring = Dah, and Tip = Dit/Straight.

Most radio systems use active-low / open-collector signaling to protect systems operating at various supply voltages. Please see the External Hardware section for recommendations.

How to test

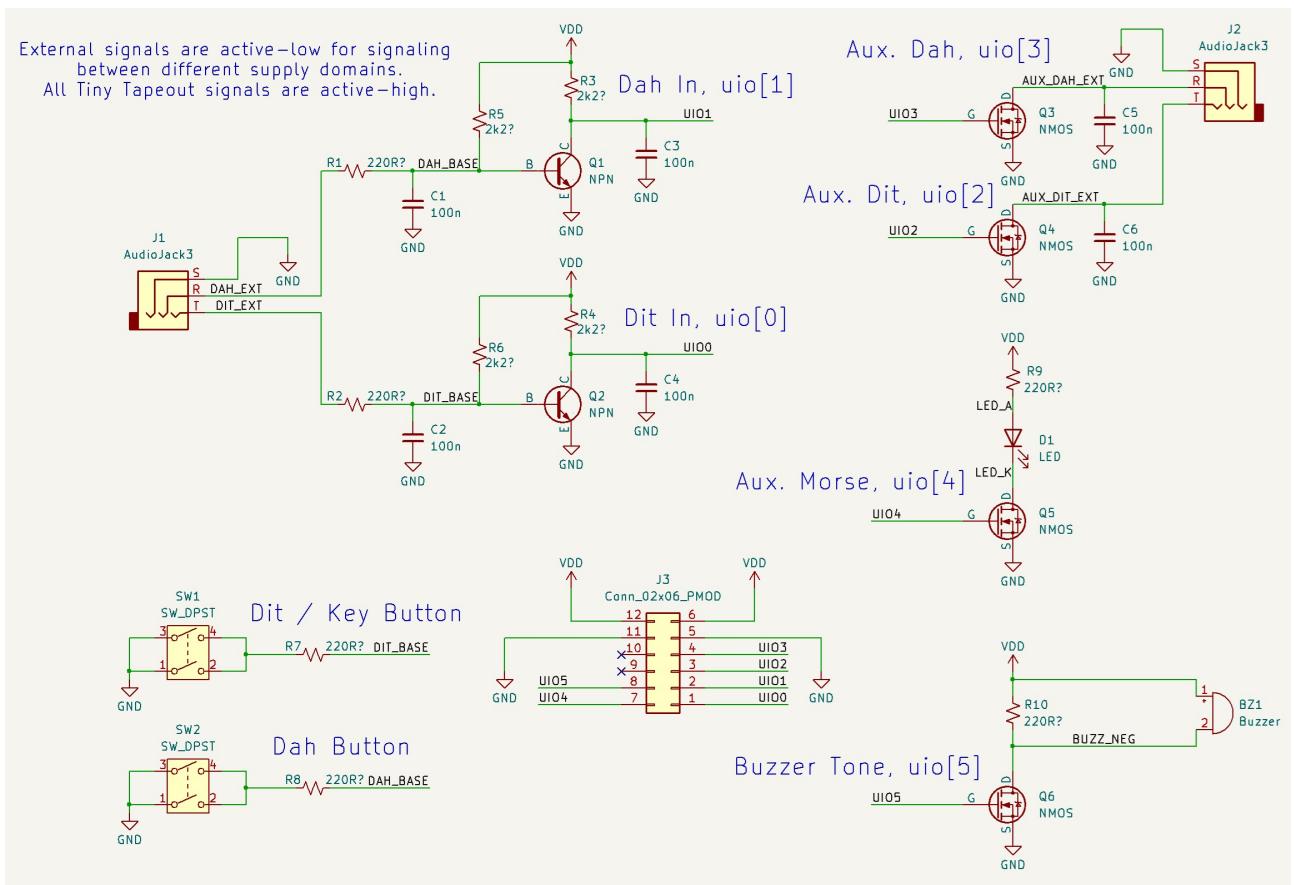
Set the input clock frequency to 12 MHz. Set the dip switches (in[7:0]) on the dev board to the desired paddle setup and WPM rate. Attach hardware like that shown in the External Hardware section to use.

External hardware

For the best experience, and to use custom radio hardware and paddles, I recommend assembling a companion PCB affixed the bidirectional PMOD.

A /pcb/ directory accompanies the standard Tiny Tapeout directories with the KiCad files.

Please see the schematic below for a screenshot of the recommended application schematic:



Pinout

#	Input	Output	Bidirectional
0	Paddle Selection (1 = Iambic)	7-Seg. Display A	External Dit / Straight In (active-low)
1	Iambic-A/B Type Selection (1 = B)	7-Seg. Display B	External Dah In (active-high)
2		7-Seg. Display C	Aux. Dit Paddle Out (active-high)
3		7-Seg. Display D	Aux. Dah Paddle Out (active-high)
4	WPM Select [0] (LSB)	7-Seg. Display E	Aux. Morse Out (active-high)
5	WPM Select 1	7-Seg. Display F	Buzzer Tone Out
6	WPM Select 2	7-Seg. Display G	
7	WPM Select [3] (MSB)	7-Seg. Display .	

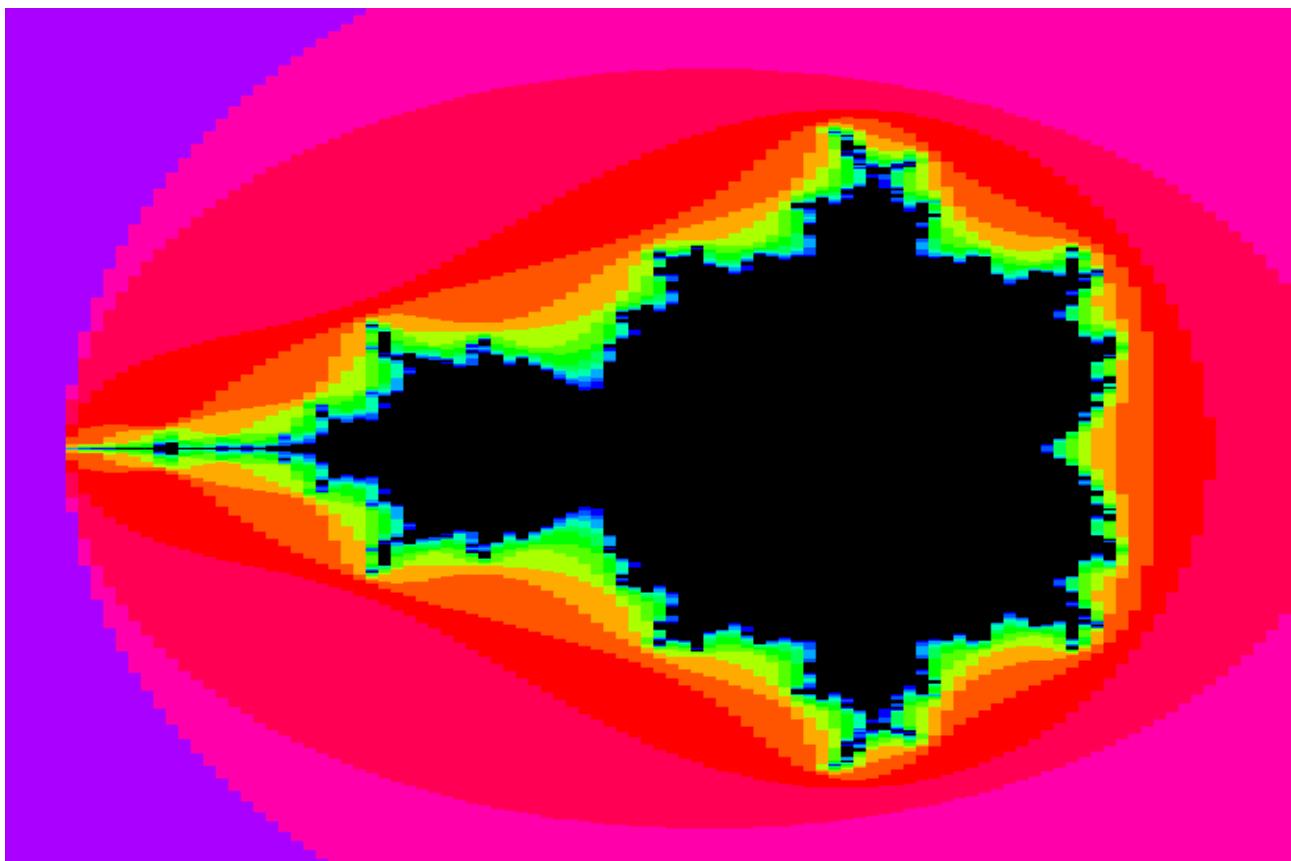
VGA Mandelbrot [848]

- Author: Mike Bell
- Description: Mandelbrot on VGA, racing the beam
- GitHub repository
- HDL project
- Mux address: 848
- Extra docs
- Clock: 100000000 Hz

How it works

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

One iteration of the computation is done over two clock cycles, and a maximum iteration depth of 14 iterations is used. The design is clocked at 100MHz, allowing four clock cycles per 25MHz pixel clock. This means one value is computed every 7 pixels, giving a result like this:



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

The output image is at a 720x480 resolution (~103x480 Mandelbrot pixels).

How to test

Provide a 100MHz clock.

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

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

These values should only be updated during vsync.

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

Note there are 103 columns and 480 rows displayed.

External hardware

Tiny VGA Pmod in the output socket.

Pinout

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

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

nVious Graphics [849]

- Author: James Ross
- Description: Submission for VGA Demoscene
- GitHub repository
- HDL project
- Mux address: 849
- Extra docs
- Clock: 25175000 Hz

How it works

This is a VGA demo that runs without input, but also accepts 8-bit input on the ui_io[7:0] pins to display a virtual 7-segment LED display (with decimal).

How to test

Basic Functionality Plug into a VGA monitor, select this circuit to test, and reset.

External Input To test the user input functionality, connect the ui_io[7:0] pins. The idea is that this would be a possibly useful graphical extension to the dozens of existing projects that utilize the 7-segment LED display to show results.

External hardware

Requires the TinyVGA PMOD

Pinout

#	Input	Output	Bidirectional
0	Segment A	R1	
1	Segment B	G1	
2	Segment C	B1	
3	Segment D	VSync	
4	Segment E	R0	
5	Segment F	G0	
6	Segment G	B0	

#	Input	Output	Bidirectional
7	Segment H	HSync	

TinyMandelbrot [850]

- Author: Gerrit Grutzeck
- Description: A mandelbrot generator
- GitHub repository
- HDL project
- Mux address: 850
- Extra docs
- Clock: 0 Hz

How it works

The project has two parts, first a module to generate a Mandelbrot. Second, a VGA driver, which fetches the data from a framebuffer, which is emulated by the RP2040.

How to test

RP2040 Mode For this the mode pin has to be selected. Then the configuration should be shifted into the project. Finally the render can be started and the data received via a logic analyzer or the RP2040.

VGA Mode For this the RP2040 has to be programmed with a special firmware to emulate the framebuffer. The the VGA mode has to be selected. Then the configuration should be shifted into the project. Finally the render can be started. After the rendering is finished, the Mandelbrot should be displayed via VGA.

External hardware

To the output Pmod connector the TinyVGA Pmod should be connected, if the VGA mode is used.

Pinout

#	Input	Output	Bidirectional
0	serial enable	R1 or ctr[0]	write data[0]
1	serial data	G1 or ctr[0]	write data1
2	serial clock	B1 or ctr[0]	write data2
3	output select	vsync or ctr[0]	write data[3]

#	Input	Output	Bidirectional
4	frame data[0]	R[0] or new counter	reset write pointer
5	frame data1	G[0]	write data
6	frame data2	B[0]	reset read pointer
7	frame data[3]	hsync	read

8-Bit Calculator [851]

- Author: Randy Zhu
- Description: ChipCraft Page 157 Lab ID: C-EQUALS
- GitHub repository
- HDL project
- Mux address: 851
- Extra docs
- Clock: 0 Hz

How it works

8-Bit Calculator from ChipCraft Lab ID: C-EQUALS

How to test

Tested with Makerchip simulation.

External hardware

None.

Pinout

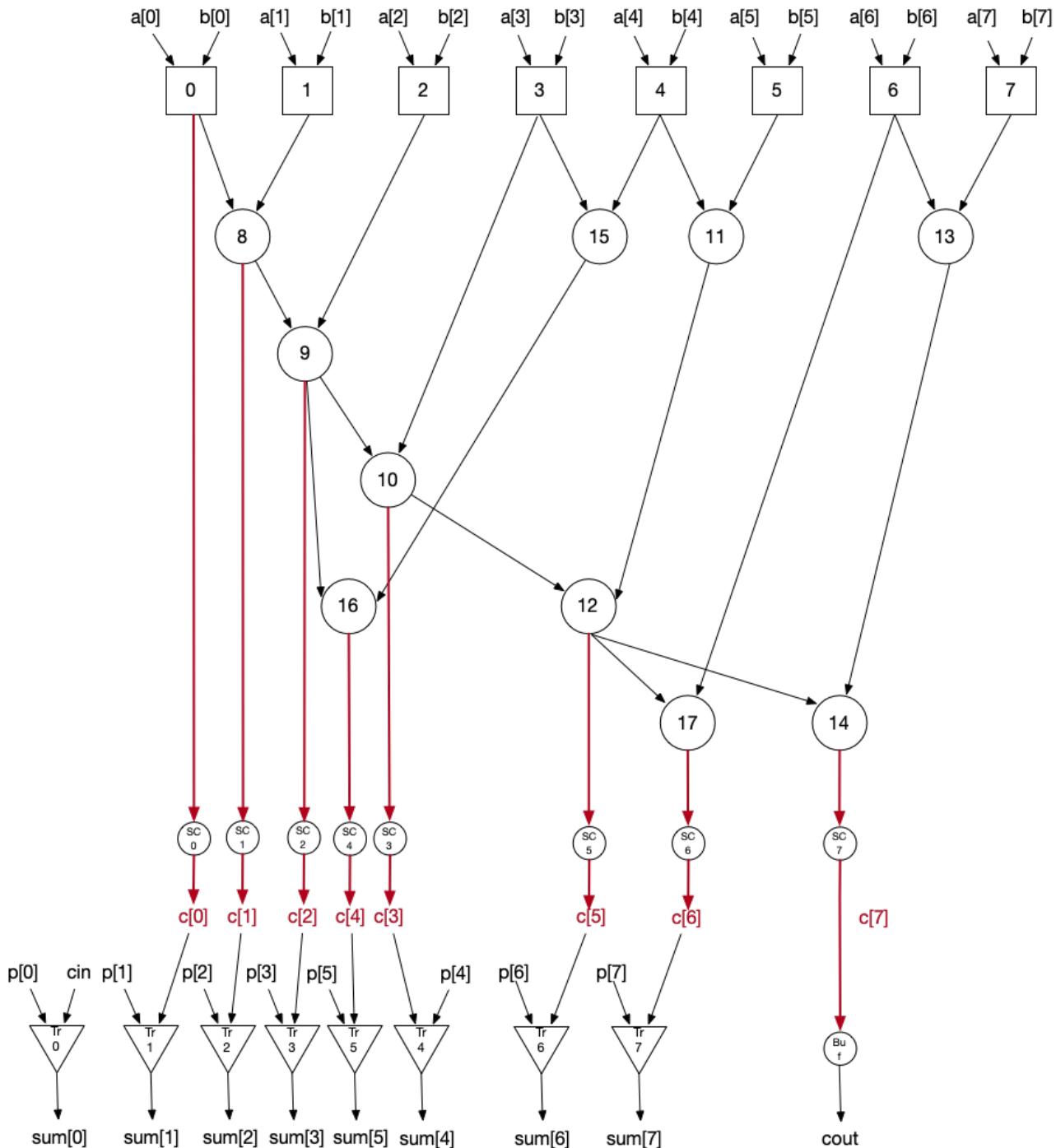
#	Input	Output	Bidirectional
0	Unused	Unused	Unused
1	Unused	Unused	Unused
2	Unused	Unused	Unused
3	Unused	Unused	Unused
4	Unused	Unused	Unused
5	Unused	Unused	Unused
6	Unused	Unused	Unused
7	Unused	Unused	Unused

tiny-tapeout-8bit-GPTPrefixCircuit [865]

- Author: Weihua Xiao
- Description: In this project, we use large language model to automatically create a totally-new prefix network-based high speed adder, for getting a good trade-off between PPA (power performance and area).
- GitHub repository
- HDL project
- Mux address: 865
- Extra docs
- Clock: 0 Hz

How it works

LLM-aided design of a totally-new 8-bit prefix network-based high speed adder:



In this figure, the squares represent the Square module in project.v, the circles represent the BigCircle module in project.v, the small circles represent the SmallCircle in project.v, and the triangles represent the Triangle module in project.v. Each carry signal ($c[i]$) is generated by circles and each sum signal ($sum[i]$) is generated by triangles.

How to test

This test systematically applies all combinations of 8-bit values to dut.a and dut.b, verifies the resulting sum dut.sum against the expected 8-bit result ((dut.a + dut.b) & 0xFF), and asserts that the dut behaves correctly.

External hardware

No external hardware

Pinout

#	Input	Output	Bidirectional
0	a[0]	sum[0]	b[0]
1	a1	sum1	b1
2	a2	sum2	b2
3	a[3]	sum[3]	b[3]
4	a[4]	sum[4]	b[4]
5	a[5]	sum[5]	b[5]
6	a[6]	sum[6]	b[6]
7	a[7]	sum[7]	b[7]

LIF on a Ring Topology [867]

- Author: Taylor Kergan
- Description: LIF neurons connected in a ring that displays different firing patterns.
- GitHub repository
- HDL project
- Mux address: 867
- Extra docs
- Clock: 0 Hz

How it works

This project implements eight leaky integrate-and-fire (LIF) neurons that are connected in a ring topology. Each neuron:

1. Integrates input current over time
2. Leaks voltage according to a decay constant
3. Fires when voltage reaches threshold
4. Influences its neighbors through coupling currents

The system supports multiple firing patterns:

- Independent: Neurons fire based only on their input current
- Wave: Activity propagates around the ring
- Synchronous: All neurons tend to fire together
- Clustered: Neurons form synchronized pairs
- Burst: Strong neighbor coupling creates burst patterns

How to test

The system can be tested through several inputs:

1. Base current ($ui_in[7:3]$): Controls the fundamental firing rate
2. Pattern select ($ui_in[2:0]$): Chooses the firing pattern
3. Coupling strength ($uo_in[7:0]$): Sets the strength of inter-neuron connections

To observe behavior:

1. Monitor spike outputs ($uo_out[7:0]$): Each bit represents one neuron's spikes
2. Watch voltage state ($uo_out[7:0]$): Shows membrane potential of first neuron
3. Run different patterns to see:

- Wave propagation
- Synchronization
- Burst patterns
- Clustering effects

Test sequence:

1. Apply reset (rst_n)
2. Enable system (ena)
3. Set desired pattern and current
4. Monitor outputs for expected behavior

External hardware

N/A.

Pinout

#	Input	Output	Bidirectional
0	Pattern select bit 0 (LSB)	Spike output from neuron 0	Coupling strength bit 0 (L)
1	Pattern select bit 1	Spike output from neuron 1	Coupling strength bit 1
2	Pattern select bit 2 (MSB)	Spike output from neuron 2	Coupling strength bit 2
3	Base current scaling bit 0 (LSB)	Spike output from neuron 3	Coupling strength bit 3
4	Base current scaling bit 1	Spike output from neuron 4	Coupling strength bit 4
5	Base current scaling bit 2	Spike output from neuron 5	Coupling strength bit 5
6	Base current scaling bit 3	Spike output from neuron 6	Coupling strength bit 6
7	Base current scaling bit 4 (MSB)	Spike output from neuron 7	Coupling strength bit 7 (M)

Delta-Sigma ADC Decimation Filter [869]

- Author: Alexander Sheldon
- Description: Decimation filter for output of a delta-sigma ADC.
- GitHub repository
- HDL project
- Mux address: 869
- Extra docs
- Clock: 50000000 Hz

How it works

Digital low pass and decimation filter for use at the output of a delta-sigma ADC. Analog will hopefully be included on the next shuttle.

How to test

Input 1 bit data on ui_in[0] at 50MHz representing the output of a delta-sigma modulator Will generate 16 bit data on the GPIOs at 50MHz/64=781.25kHz

External hardware

TBD

Pinout

#	Input	Output	Bidirectional
0	dec_in	mux_out[0]	div_clk8x
1		mux_out1	
2		mux_out2	div_clk
3		mux_out[3]	
4		mux_out[4]	
5		mux_out[5]	
6		mux_out[6]	
7		mux_out[7]	

an lfsr with synaptic neurons (excitatory or inhibitory) [871]

- Author: kai juarez-jimenez
- Description: each bit edge in the LFSR will mimic synaptic input that either excites / inhibits the next “neuron”, showing behaviors similar to how synapses manage signal in nns.
- GitHub repository
- HDL project
- Mux address: 871
- Extra docs
- Clock: 0 Hz

How it works

this project implements a neuromorphic-inspired Linear Feedback Shift Register (LFSR) with “synaptic neurons” that simulate excitatory/inhibitory responses. each bit in the LFSR behaves like a neuron, where transitions (rising/falling edges) from 0 to 1 or 1 to 0 generate excitatory or inhibitory signals, simulating synaptic inputs in neural networks. these signals modify the LFSR’s feedback path, resulting in pseudo-random output sequences that mimic synaptic interactions by either enhancing (excitatory) or suppressing (inhibitory) activity.

additionally, this design allows for customizable seed inputs, set through external input pins, enabling users to initialize the LFSR with a specific seed to observe varying sequence outputs. this feature provides added flexibility and control over the pseudo-random behavior.

How to test

1. clock initialization: Run a clock signal to provide timing for the LFSR operation.
2. reset: hold the reset pin active (low) to initialize the LFSR state with the selected seed.
3. seed testing: configure the seed by setting the ui_in input pins, then observe the LFSR output sequence through uo_out.
4. cycle observation: monitor the output sequence over multiple clock cycles to verify pseudo-random behavior, and repeat for different seed values for varied sequences.

External hardware

N/A

Pinout

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

Perceptron [873]

- Author: Clarence Chan
- Description: Hardware implementation of a single layer perceptron
- GitHub repository
- HDL project
- Mux address: 873
- Extra docs
- Clock: 0 Hz

How it works

Given 8 bits of inputs and 8 bits of weights, the single layer perceptron will classify the inputs as class 0 or 1.

How to test

Initialize inputs and expected output in `test.py`, then run `make` in the `test` subdirectory.

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	Input bit 1	Perceptron class output	Weight bit 1
1	Input bit 2		Weight bit 2
2	Input bit 3		Weight bit 3
3	Input bit 4		Weight bit 4
4	Input bit 5		Weight bit 5
5	Input bit 6		Weight bit 6
6	Input bit 7		Weight bit 7
7	Input bit 8		Weight bit 8

Matmul System [875]

- Author: Abarajithan
- Description: Matmul System
- GitHub repository
- HDL project
- Mux address: 875
- Extra docs
- Clock: 0 Hz

How it works

This is a simple system that performs matrix-vector multiplication. The matrix $K[R,C]$ and vector $X[R]$ is sent from outside through UART. They are decoded by a UART RX module, and sent into the matrix-vector multiplication core as AXI-Stream. The core performs the multiplication and outputs the result as AXI-Stream. The result is then packed into UART format by the UART TX module and sent outside.

How to test

```
iverilog -g2012 -o compiled src/mvm_uart_system.v src/uart_rx.v src/uart_
```

External hardware

None

Pinout

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

Verilog ring oscillator [877]

- Author: algofoogle (Anton Maurovic)
- Description: Simple ring oscillator by instantiating a sky130 inv_2 inverter ring
- GitHub repository
- HDL project
- Mux address: 877
- Extra docs
- Clock: 0 Hz

What is this?

Everyone has done a ring oscillator using inverter cells. Now it's my turn!

This simple example uses Verilog to instantiate a ring of (an odd number of) sky130_fd_sc_hd__inv_2 cells – **UPDATE:** Actually, since this is targeting IHP instead, there is a polyfill that somebody else wrote to map sky130 cells to generic cells (that OpenLane will then map to IHP cells).

It produces its output on `uo_out[0]`.

Assuming each inverter introduces a delay of ~70ps, and there are 1001 of them, then hopefully this will oscillate at ~14MHz?

Pinout

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

Delta RNN and Leaky Integrate-and-Fire Nueron Circuit [879]

- Author: Katherine Rogacheva
- Description: A physical representation of a delta recurrent neural network (Delta RNN) and a leaky integrate-and-fire (LIF) neuron, that creates an artificial spike when the difference in the previous and current state is greater than a set delta threshold.
- GitHub repository
- HDL project
- Mux address: 879
- Extra docs
- Clock: 0 Hz

How it works

Takes input voltages and treats that as the input current injection into the LIF neuron

How to test

N/A

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	Input current bit [0]	State variable bit [0]	Difference in states bit 1
1	Input current bit 1	State variable bit 1	Difference in states bit 2
2	Input current bit 2	State variable bit 2	Difference in states bit [3]
3	Input current bit [3]	State variable bit [3]	Difference in states bit [4]
4	Input current bit [4]	State variable bit [4]	Difference in states bit [5]
5	Input current bit [5]	State variable bit [5]	Difference in states bit [6]
6	Input current bit [6]	State variable bit [6]	Difference in states bit [7]
7	Input current bit [7]	State variable bit [7]	Difference in states bit [8]

Generador PWM multipropósito con frecuencia y ciclo de trabajo modulable [881]

- Author: Marco Vázquez, Paúl González, Abimael Jimenez, UACJ
- Description: A PWM generator with a 6-bit input that allows the user to enter a denominator that divides the frequency. Using a pair of control inputs, we can increase or decrease the duty cycle of the modulated output by 10%.
- GitHub repository
- HDL project
- Mux address: 881
- Extra docs
- Clock: 5000 Hz

How it works

Overall, the module converts a high-speed clock signal into a PWM signal with adjustable frequency and duty cycle. The user receives a high-frequency clock signal and, through a frequency divider, generates a lower-frequency clock. Then, they control the high duration of the PWM signal using buttons that increase or decrease the duty cycle value.

A 5kHz signal is received; the 6-bit divider only accepts numbers from 2 to 63 (decimal). The possible output frequencies for the PWM range from 2500Hz (5kHz/2) to 79Hz (5kHz/63), which can be used in different electronic components such as RGB LEDs, servomotors, stepper motors, sensors, and other circuits.

How to test

- Connect the clock signal: Assign a high-frequency clock.
- Apply the reset signal: Initially set the reset to high to restart the module. This will reset all counters and the duty cycle to their initial values.
- Set the frequency divider: Define the frequency divider value to adjust the speed of the clock used. This value controls the PWM signal frequency. A higher divider value will result in a lower PWM frequency, and vice versa.
- Duty cycle adjustment buttons: When activating the increment button, the duty cycle will increase by 10%. When activating the decrement button, the duty cycle will decrease by 10%.

Recommendation: Use the PWM signal only as a control signal; the power supply for the devices it is applied to should come from an external power source.

External hardware

The PWM output should go to a PMOD to have that control signal available on a device.

Pinout

#	Input	Output	Bidirectional
0	increase_duty	pwm_out0	
1	decrease_duty	pwm_out1	
2	divisor[0]	pwm_out2	
3	divisor1	pwm_out3	
4	divisor2	pwm_out4	
5	divisor[3]	pwm_out5	
6	divisor[4]	pwm_out6	
7	divisor[5]	pwm_out7	

Linear Feedback Shift Register [883]

- Author: Steve Jenson <stevej@gmail.com>
- Description: An implementation of a Linear Feedback Shift Register for TT09
- GitHub repository
- HDL project
- Mux address: 883
- Extra docs
- Clock: 0 Hz

How it works

Read the `ui_out` pins, each read should be different than the last. To reset the shift register, reset the chip, or set the ‘`write_enable`’ pin high after offering a value on `ui_in` as a seed.

How to test

Read several bytes from `ui_in`, they should each be different.

External hardware

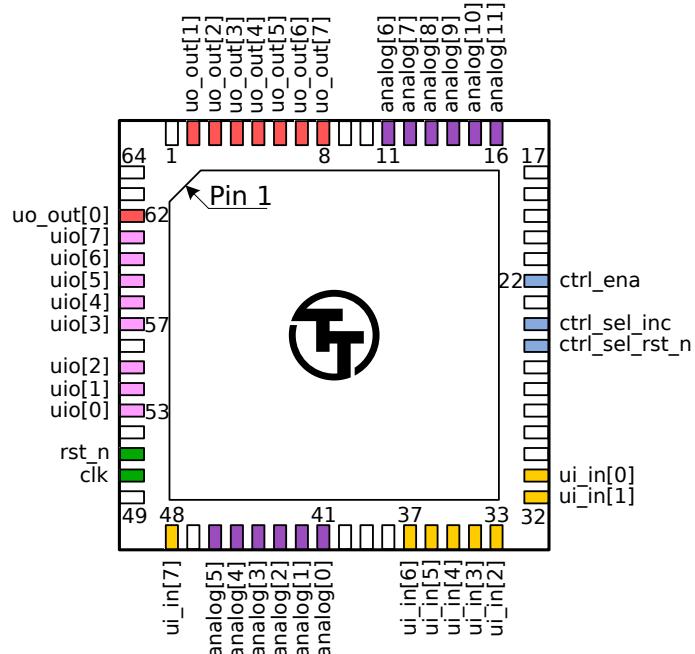
No external hardware needed other than to read the pins.

Pinout

#	Input	Output	Bidirectional
0	Seed Bit 1	LFSR Bit 1	Write Enable
1	Seed Bit 2	LFSR Bit 2	
2	Seed Bit 3	LFSR Bit 3	
3	Seed Bit 4	LFSR Bit 4	
4	Seed Bit 5	LFSR Bit 5	
5	Seed Bit 6	LFSR Bit 6	
6	Seed Bit 7	LFSR Bit 7	
7	Seed Bit 8	LFSR Bit 8	

Pinout

The chip is packaged in a 64-pin QFN package. The pinout is shown below.



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.

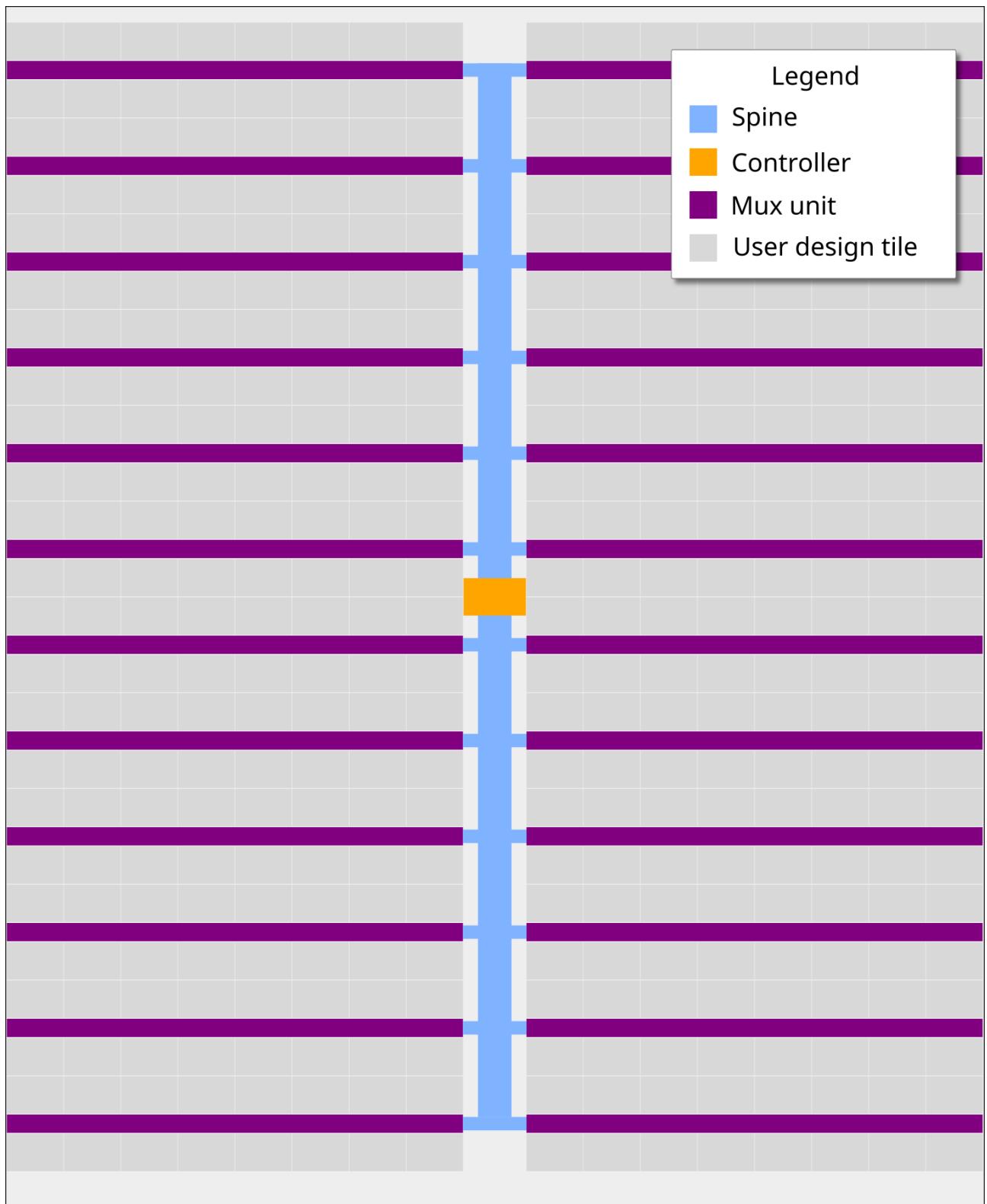
It has the following features:

- 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 called a tile, and each design can occupy up to 16 tiles.

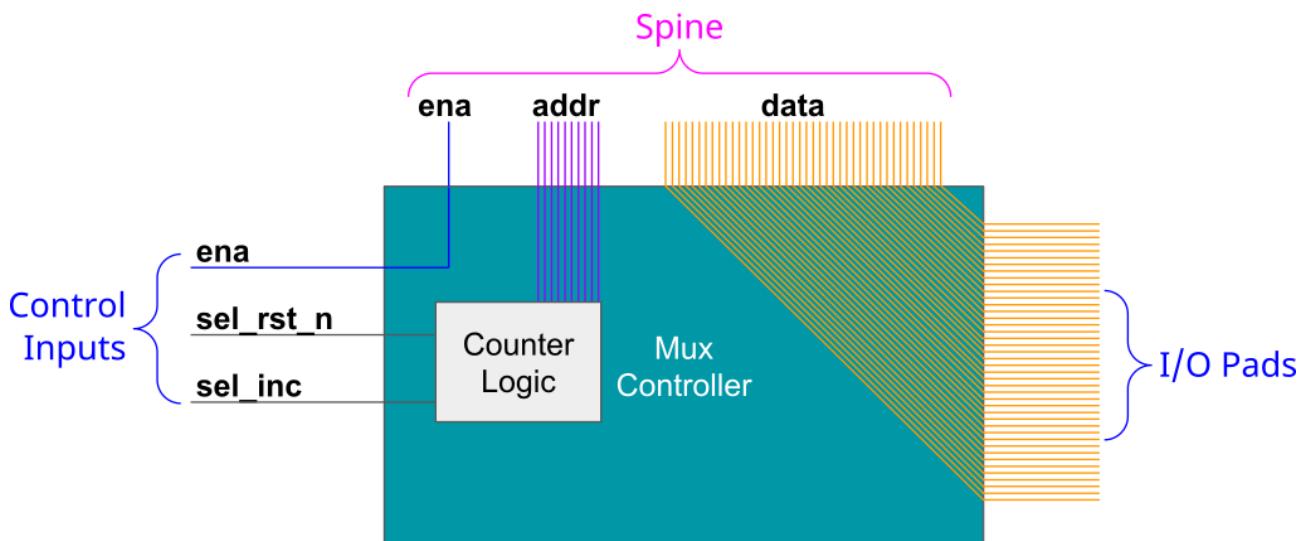
Operation

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

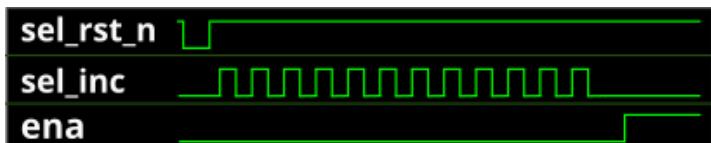


The mux controller has 3 inputs lines:

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:



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/364347807664031745>. It contains an Arduino Nano that decodes the currently selected mux address and displays it on a 7-segment display. Click on the button labeled `RST_N` to reset the counter, and click on the button labeled `INC` to increment the counter.

The Spine

The controller and all the muxes are connected together through the spine. The spine has the following signals going on it:

From controller to mux:

- `si_ena` - the ena input
- `si_sel` - selected design address (10 bits)
- `ui_in` - user clock, user `rst_n`, user inputs (10 bits)
- `uio_in` - bidirectional I/O inputs (8 bits)

From mux to controller:

- `uo_out` - User outputs (8 bits)
- `uio_oe` - Bidirectional I/O output enable (8 bits)
- `uio_out` - Bidirectional I/O outputs (8 bits)

The only signal which is actually generated by the controller is `si_sel` (using `sel_rst_n` and `sel_inc`, as explained above). The other signals are just going through from/to the chip IO pads.

The Multiplexer (The Mux)

Each mux branch is connected to up to 16 designs. It also has 5 bits of hard-coded address (each unit gets assigned a different address, based on its position on the die).

The mux implements the following logic:

If `si_ena` is 1, and `si_sel` matches the mux address, we know the mux is active. Then, it activates the specific user design port that matches the remaining bits of `si_sel`.

For the active design:

- `clk`, `rst_n`, `ui_in`, `uio_in` are connected to the respective pins coming from the spine (through a buffer)
- `uo_out`, `uio_oe`, `uio_out` are connected to the respective pins going out to the spine (through a tristate buffer)

For all others, inactive designs (including all designs in inactive muxes):

- `clk`, `rst_n`, `ui_in`, `uio_in` are all tied to zero
- `uo_out`, `uio_oe`, `uio_out` are disconnected from the spine (the tristate buffer output enable is disabled)

Pinout

QFN64 pin	Function	Signal
1	Mux Control	ctrl_ena
2	Mux Control	ctrl_sel_inc
3	Mux Control	ctrl_sel_RST_n
4	Reserved	(none)
5	Reserved	(none)
6	Reserved	(none)
7	Reserved	(none)
8	Reserved	(none)
9	Output	uo_out[0]
10	Output	uo_out1
11	Output	uo_out2
12	Output	uo_out[3]
13	Output	uo_out[4]
14	Output	uo_out[5]
15	Output	uo_out[6]
16	Output	uo_out[7]
17	Power	VDD IO
18	Ground	GND IO
19	Analog	analog[0]
20	Analog	analog1
21	Analog	analog2
22	Analog	analog[3]
23	Power	VAA Analog
24	Ground	GND Analog
25	Analog	analog[4]
26	Analog	analog[5]
27	Analog	analog[6]
28	Analog	analog[7]
29	Ground	GND Core
30	Power	VDD Core
31	Ground	GND IO
32	Power	VDD IO
33	Bidirectional	ui0[0]
34	Bidirectional	ui01
35	Bidirectional	ui02
36	Bidirectional	ui0[3]
37	Bidirectional	ui0[4]
38	Bidirectional	ui0[5]

QFN64 pin	Function	Signal
39	Bidirectional	ui[6]
40	Bidirectional	ui[7]
41	Input	ui_in[0]
42	Input	ui_in1
43	Input	ui_in2
44	Input	ui_in[3]
45	Input	ui_in[4]
46	Input	ui_in[5]
47	Input	ui_in[6]
48	Input	ui_in[7]
49	Input	rst_n †
50	Input	clk †
51	Ground	GND IO
52	Power	VDD IO
53	Analog	analog[8]
54	Analog	analog[9]
55	Analog	analog[10]
56	Analog	analog[11]
57	Ground	GND Analog
58	Power	VDD Analog
59	Analog	analog[12]
60	Analog	analog[13]
61	Analog	analog[14]
62	Analog	analog[15]
63	Ground	GND Core
64	Power	VDD Core

† Internally, there's no difference between `clk`, `rst_n`, and `ui_in` pins. They are all just bits in the `pad_ui_in` bus. However, we use different names to make it easier to understand the purpose of each signal.

Funding

IHP PDK support for Tiny Tapeout was funded by The SwissChips Initiative.

The manufacturing of Tiny Tapeout IHP 0p2 silicon was funded by the German BMBF project FMD-QNC (16ME0831).

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