



Tiny Tapeout CAD 25a

Datasheet

github.com/TinyTapeout/tinytapeout-cad-25a

July 5, 2026

Table of Contents

Chip Renders	1
Shuttle Map	2
GDS	3
Logic Density	4
Projects	5
0000 Chip ROM	6
0001 Tiny Tapeout Factory Test	8
0002 Sea Battle	10
0004 Rounding error	12
0006 VGA Mandelbrot	18
0008 schoolRISCV CPU with Fibonacci program	20
0010 Frequency Counter SSD1306 OLED	22
0032 Patater Demo Kit Wagglng Rainbow on a Chip	23
0033 Orion Iron Ion [TT08 demo competition]	28
0035 Moody-mimosa	29
0037 TinyTapeout Minimal Branch Predictor	35
0039 Ternary 128-element Dot Product	38
0041 GUS16 CPU	39
0043 donut	41
0045 UART	42
0047 FSK Modem +HDLC +UART (PoC)	43
0064 Comm_IC	47
0066 VGA Drop (audio/visual demo)	49
0068 Warp	50
0070 VGA Nyan Cat	52
0072 16 Mic Beamformer	54
0074 PDM Pitch Filter	55

0076	PDM Correlator	56
0078	Bouncy Capsule	57
0097	My Project	58
0099	simple-viii	59
Artwork	MPW-6 "Hack SoC"	60
0101	ttUART	61
0107	UW ASIC - Optimized Dino	62
0109	TT09 SKY130 ROM Test	63
0111	TT09 SKY130 ROM Test (no LVT variant)	65
0128	CMOS design of 4-bit Signed Adder Subtractor	67
0130	Demo by alk0n	69
0134	sky130_as_sc_hs test	71
0138	SPI FPU	72
0142	Space Invaders ASIC	75
0161	Simple 8 Bit ALU	76
0163	cfib Demoscene Entry	81
0165	Munch	82
0167	DemoSiine	85
0169	simon_cipher	93
0171	VGA Screensaver with Zero to ASIC Logo	95
0173	UART Matrix Multiplier	97
0175	TTCAD25a SKY130 RF YOLO Test	101
0194	Zilog Z80	102
0196	"SQUARE-1": VGA/audio demo	106
0198	Sprite Bouncer with Looping Background Options	110
0200	TinyMandelbrot	111
0202	TT08 Pachelbel's Canon demo	113

0204	Wirecube	114
0206	RGBW Color Processor	115
0225	Bandgap	121
0227	LDO BG IREF OSC	122
Artwork	MPW-2 poly layers	123
0229	5MHz Ring Oscillator	124
0231	R-C Oscillators	125
0233	Analog Voltage Controlled Oscillator	127
0235	Voltage Controlled LC-Oscillator	129
0237	Adjustable supply digital counter	133
0239	TT08 Differential Receiver test	135
0257	SONOS Flash Party	136
0258	mulmul	139
0259	Bias Generator	140
0261	Ring Oscillators	141
0263	5-T OTA	143
0264	LaRVa CPU	144
0265	Oscillating Bones	146
0267	2-stage Opamp Designs	148
0269	Dickson Charge Pump	157
0270	Wildcat RISC-V	161
0271	4-Bit Successive Approximation ADC	162
0288	8-Bit Calculator	165
0289	i2c peripherals demonstrating address decoding and i2c reads	166
0290	Tiny PLL	167
0291	TT08 SKY130 ROM 'YOLO' Test	175
0292	nVious Graphics	176

0293	Rotary Encoder WS2812B Control	177
0295	tt08-octal-alu	179
0296	4-bit ALU	181
0299	TSAL_TT	183
Artwork	Detail - Fibonacci design (MPW-2)	185
0301	Divided Ring Oscillator	186
0303	HACK CPU	189
0322	DDC	194
0326	Sequential Shadows Deluxe [TT08 demo competition]	195
0330	AlphaOneSoC	198
0334	2048 sliding tile puzzle game (VGA)	199
0416	Traffic-light-sequence	201
0417	Abacus Lock	202
0418	7 Segment Decode	203
0419	TT08 - experiments with latch-based shift registers	207
0420	DPMU	208
0421	Micro tile container	211
0422	MAC	216
0423	Logic Test	219
0424	pulse_add	220
0425	AES Inverse S-box	221
0426	nyan	222
0427	Super Mario Tune on A Piezo Speaker	223
0428	Brailliance	225
0429	PS2 Decoder	226
0430	Adder with Flow Control	228
0431	TT08 SKY130 Shift Register 'YOLO' Test	231

0454	TinyQV SoC with additional peripherals	232
0460	TinyQV Risc-V SoC	236
0462	VGA Pong with NES Controllers	240
0480	Dempsey-Gorman DAC	242
0481	Obstacle Detection	243
0482	Analog 8 bit 3.3v R2R DAC	244
Artwork	555 render (TT06)	246
0483	Styler	247
0484	R2R DAC	251
0485	resfuzzy	252
0486	Pi Snake	253
0487	CEJMU Beers and Adders	256
0489	I2S to PWM	258
0490	Bandgap Reference	259
0491	DMTD	261
0492	Raw_Transistors	262
0493	PWM generator	264
0494	TT08 Analog Factory Test	266
0495	Simple Stopwatch	267
0514	Current-Mode Bandgap Reference	269
0516	Mixed-Signal	271
0518	Level Shifter	272
0520	TT08 VGA FUN!	275
0522	Sine Wave Synthesizer	278
0526	AICD Playground	280
0544	Clock Divider	282
0546	DPM_Unit	283

0547	6Digit7SegClock	286
0548	Generate VGA output for Color Blindness Test	288
0549	MAC Operation	289
0550	4-bit CLA	290
0551	Xor-Logic	291
0552	SkyKing Demo	292
Artwork	SkullFET render (MPW-4)	293
0553	Simon Says memory game	294
0554	Flame demo	297
0555	SPI 7-segment display	299
0556	Metaballs	300
0557	AtomNPU	301
0558	Micro tile container (group 2)	304
0582	Tiny Neural Network Accelerator	307
0590	SIC-1 8-bit SUBLEQ Single Instruction Computer	310
0608	TinyFPGA resubmit for TT08	314
0609	VSLC - Very Small Logic Controller with a timer and servo control	315
0610	Dummy Counter	324
0611	Gamepad Pmod Demo	325
0612	Why not?	326
0613	VGA Screensaver with Tiny Tapeout Logo	327
0614	Counter	329
0615	SPI test	330
0616	RGB Mixer	331
0617	Huffmann_Coder	332
0618	RGB Mixer demo	333
0619	Vedic multiplier	334

0620	VGA clock	335
0621	NAND	336
0622	synth_simple	337
0623	DaliaProjekt	338
0642	VGA donut	339
0654	raybox-zero TTCAD25a edition	341
0672	Tiny piano	344
Artwork	VGA clock render	346
0673	my_own_chip	347
0674	4 Bit Adder with Overflow Counter	348
0675	Hero on Tape	349
0676	First design	350
0677	digital LIF Neuron	351
0678	example1	352
0679	ANDNOT	353
0680	test/15/02/25	354
0681	Holm's TinyTapeOut 4-bit adder	355
0682	Extremely cool stuff (secret)	356
0683	mp_LIF_neuron	357
0684	Tinytapeout_design_ANP	358
0685	Full Adder	359
0686	TinyTapeout 4 bit ripple carry adder	360
0687	Special code for letter n	361
0710	Neural Net ASIC	362
0718	Zoom Zoom	363
0736	Emil Njor's Design	368
0737	Encoder	369

0738	Asynchronous I2C Registerfile Interface	370
0739	simplePass	371
0740	3-bit register print	372
0741	SSMCI	373
0742	Four basic building blocks	375
0743	TT10_Luke_Clock	376
0744	3 Bit Adder	380
0745	Verilog based clock to 7-segment counter	381
0746	TinyChipDesign	382
Artwork	VGA clock render	383
0747	Simple NAND 2	384
0748	Tappu	385
0749	Adder	386
0750	Enter-Code	387
0751	XOR Cipher	388
0782	LGN hand-written digit classifier (MNIST, 16x16 pixels)	390
0800	zc-sushi-demo	391
0801	Sine Synth	392
0802	Crossyroad	394
0803	DRUM	395
0804	JTAG TAP	396
0805	Tiny Hash Table	398
0806	7-segment with LFSR	400
0807	Asynchronous FIFO	402
0809	Synchronous FIFO	404
0810	KCH CD101 Saw Synth	405
0811	Pulse Width Modulation	407

0812	tt10_zhouzhouthezhou_adder	410
0814	Asynchronous Locking Unit	411
0815	Simple shift Reg	412
0842	TRNG	413
0846	Find The Damn Issue	415
0864	kch cd101	417
0865	Muller C-element	419
0866	SimpleSPIdev	420
0867	Tiny Shader	421
Artwork	TT06 IC - decapped	426
0868	RNG_test	427
0869	Linear Timecode (LTC) generator	428
0870	15bit GCD	430
0871	Priority-encoder	432
0872	XY Spacewar	433
0873	UltraTiny-CPU	434
0874	16-bit Logarithmic Approximate Floating Point Multiplier	437
0875	E-ink display driver	440
0876	TT_spiralPattern	442
0877	I2C EEPROM Project Selection	443
0878	ledtest	445
0879	Overengineered Checkers	446
0897	Dice	447
0899	INTERCAL ALU	448
0901	4-bit minicomputer ALU	453
0903	Hardware UTF Encoder/Decoder	454
0905	RGB Mixer demo5	459

0907	Universal Binary to Segment Decoder	460
0909	Asynchronous Multiplier	469
0910	Supermic	472
0911	VGA Tiny Logo (1 tile)	473
0938	TTCAD25a SKY130 FF YOLO Test	474
0940	Padlock	475
0942	sky130_as_sc_hs dyn test	477
0961	Lab and Lectures SoC	478
0963	asic design is my passion	480
0965	Crispy VGA	481
0967	Calculator	483
Artwork	Oscillating Bones	484
0971	VGA Scroller	485
0973	Glyph Mode	486
0974	KianV uLinux SoC	487
0975	RO	490
Pinout		492
The Tiny Tapeout Multiplexer		493
Overview		493
Operation		493
Pinout		496
Team		498
Using This Datasheet		499
Structure		499
Badges		499
Callouts		500
Figures & Footnotes		500
Updates		500
Where is <u>your</u> design?		501

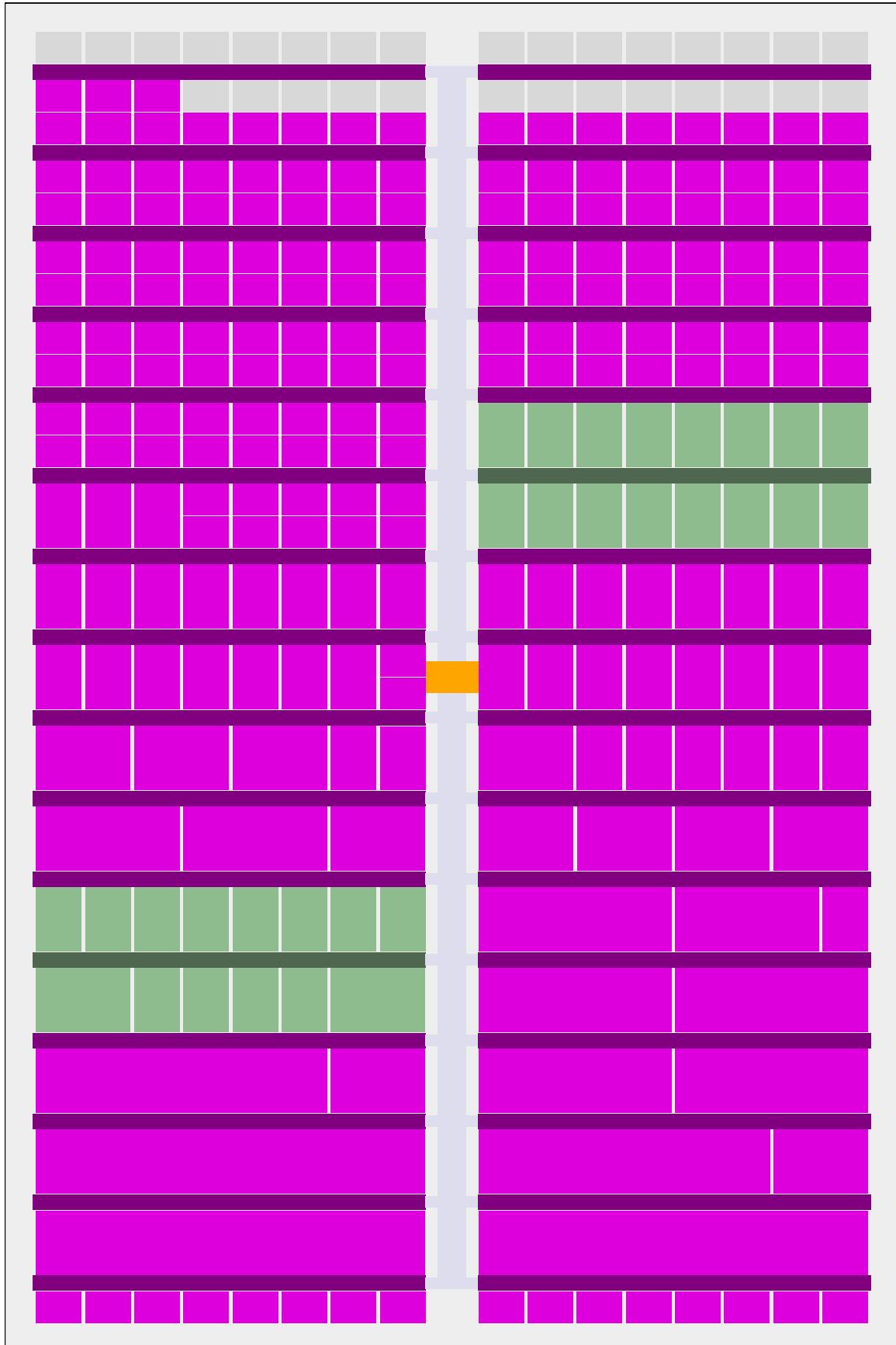
Chip Renders

Online chip viewer

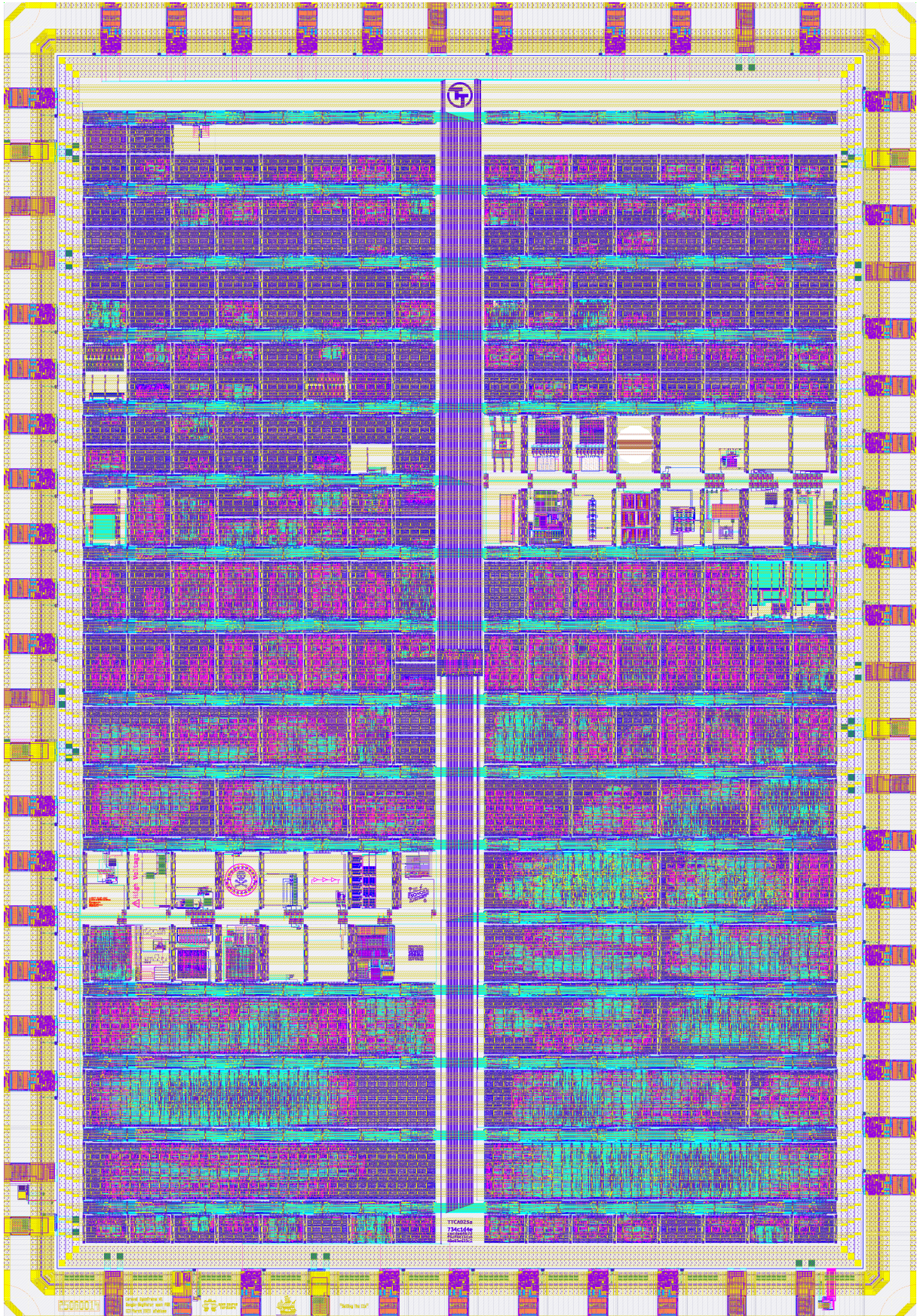


[tinytapeout.com/
decap/ttcad25a](https://tinytapeout.com/decap/ttcad25a)

Shuttle Map

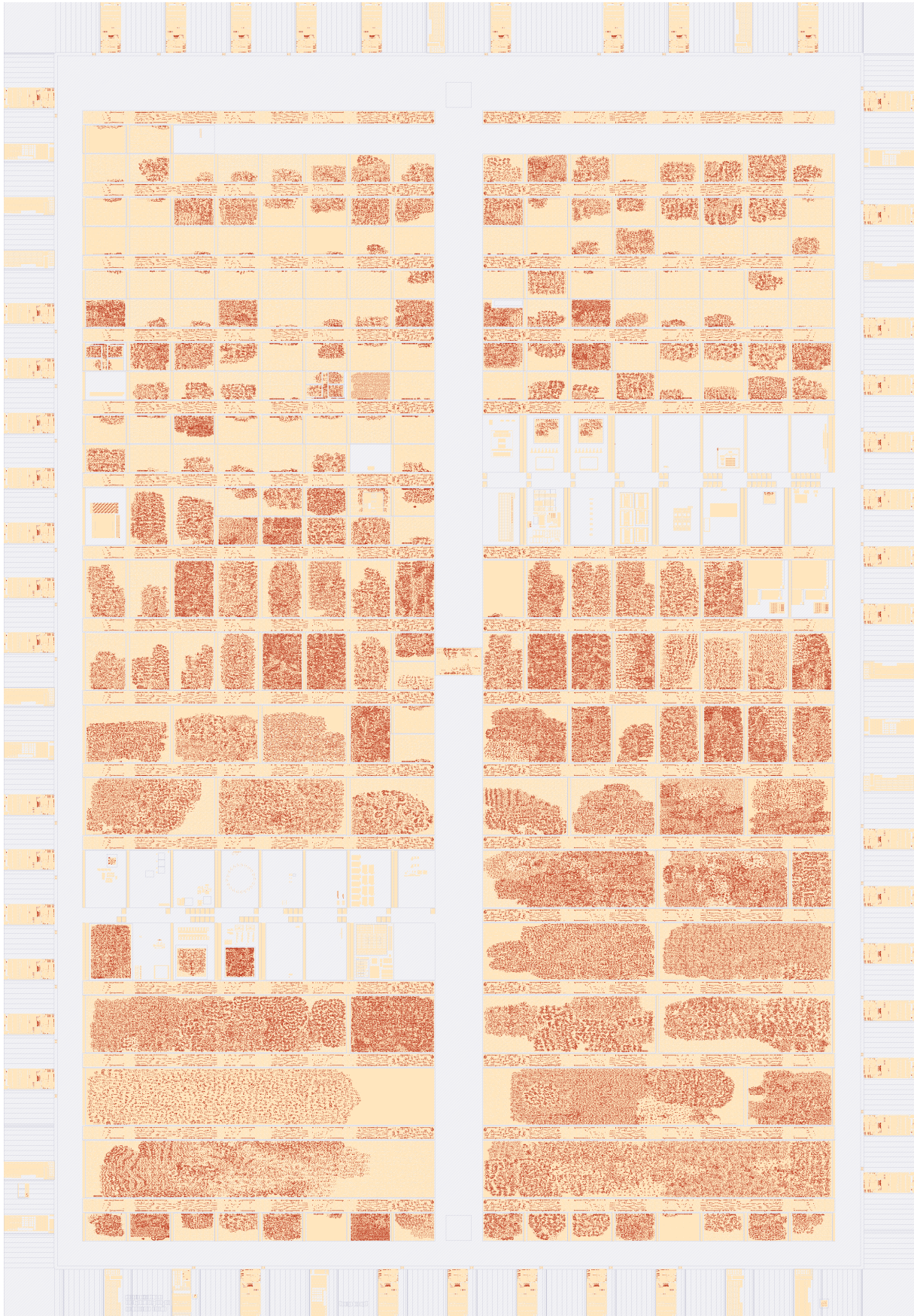


GDS



Logic Density

Local Interconnect Layer



Projects

Chip ROM

by Uri Shaked

0000

HDL Project

github.com/TinyTapeout/tt-chip-rom

"ROM with information about the chip"

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.

Project Pinout

Digital Pins

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

Tiny Tapeout Factory Test

by Tiny Tapeout

0001

HDL Project

github.com/TinyTapeout/tt10-factory-test

“Factory test module”

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 and `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	uo_out value	uio pins
0	X	Input mirror	<code>ui_in</code>	High-Z
1	0	Bidirectional mirror	<code>uio_in</code>	High-Z
1	1	Counter	<code>counter</code>	<code>counter</code>

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

How to test

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

Project Pinout

Digital Pins

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

Sea Battle

by Yuri Panchul

0002

23 MHz

HDL Project

github.com/yuri-panchul/tt08-sea-battle-vga-game

“Sea Battle is a VGA game with sprites for the Tiny Tapeout Demoscene competition.”

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 testbench for simulation. We just hope it is going to work on ASIC silicon as is.

External hardware

Buttons and a Tiny VGA connector.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Key right	VGA red [1]	—
1	Key left	VGA green [1]	—
2	—	VGA blue [1]	—
3	—	VGA vsync	—
4	—	VGA red [0]	—
5	—	VGA green [0]	—
6	—	VGA blue [0]	—

#	Input	Output	Bidirectional
7	—	VGA hsync	—

Rounding error

by Edwin Török

0004

25.25 MHz

HDL Project

github.com/edwintorok/roundingerror

"Competition entry"

How it works

Idea

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

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

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

HardCaml

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

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

VGA signal generation

ModeLine

VGA signal timing is described in "3. DMT Video Timing Parameter Definitions" in "VESA Display Monitor Timing Standard Version 1.0, Rev. 13", and is implemented in `src/generator/modeline.m1`. Examples on how to implement them on an FPGA are available in [several places](#).

The code supports several resolutions, however to conserve area for the demo I've chosen only [640x480@59.94Hz](#), which has negative hsync/vsync polarities. This resolution would need a 25.175 MHz pixel clock, however that can't be produced exactly by the TT08 board, it can only approximate it using a PWM. Therefore, the design is configured to run at the nearest frequency that can be exactly generated: 25.25 MHz, which should be within the 0.5% acceptable by the standard. The [ModeLine](#) implemented is: `ModeLine "640x480_59.94" 25.175 640 656 752 800 480 490 492 525 -hsync -vsync.`

(This has 59.94 refresh rate and not 60Hz due to the standard preferring NTSC and its 1.001 adjustment).

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

Counters

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

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

R, G, B colours

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

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

Generating the colours

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

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

Ray marching

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

CORDIC

Fixed point arithmetic with 9 bits of precision is used in the HDL, with the exponent tracked by the generator code to reduce register width (though this is not as good as tracking it in hardware, but that'd require more area). Vector normalization is implemented using the [CORDIC](#) implementation

provided by HardCaml, configured to use 10 bits, and a limited number of iterations (4) to fit into the desired area. This works by rotating the vector until its angle is 0, and then rotating a second unit vector to match the rotation of the original. Or equivalently transform the original from rectangular to polar coordinates, overwrite the length with 1, and convert back from polar to rectangular. CORDIC is defined for 2D in the library, and I define a 3D wrapper based on [rectangular to spheric coordinate conversions](#), although there would be ways to directly compute a 3D version of CORDIC, that is not implemented here.

This is implemented in `src/vecmath`.

GLSL ES “emulation”

The low level operations are wrapped by a higher level embedded DSL that allows writing code quite similar to [GLSL ES](#), with a very small number of operators: arithmetic (+, -, *, /), comparison (==, < >), abs, min, max, clamp, length, distance, dot, normalize, reflect.

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

OpenLane configuration

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

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

Lack of audio

Audio is enabled, but is only a very simple test signal based on `hsync` and `vsync`.

Simulating

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

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

How to test

Configuration

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

Main demo

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

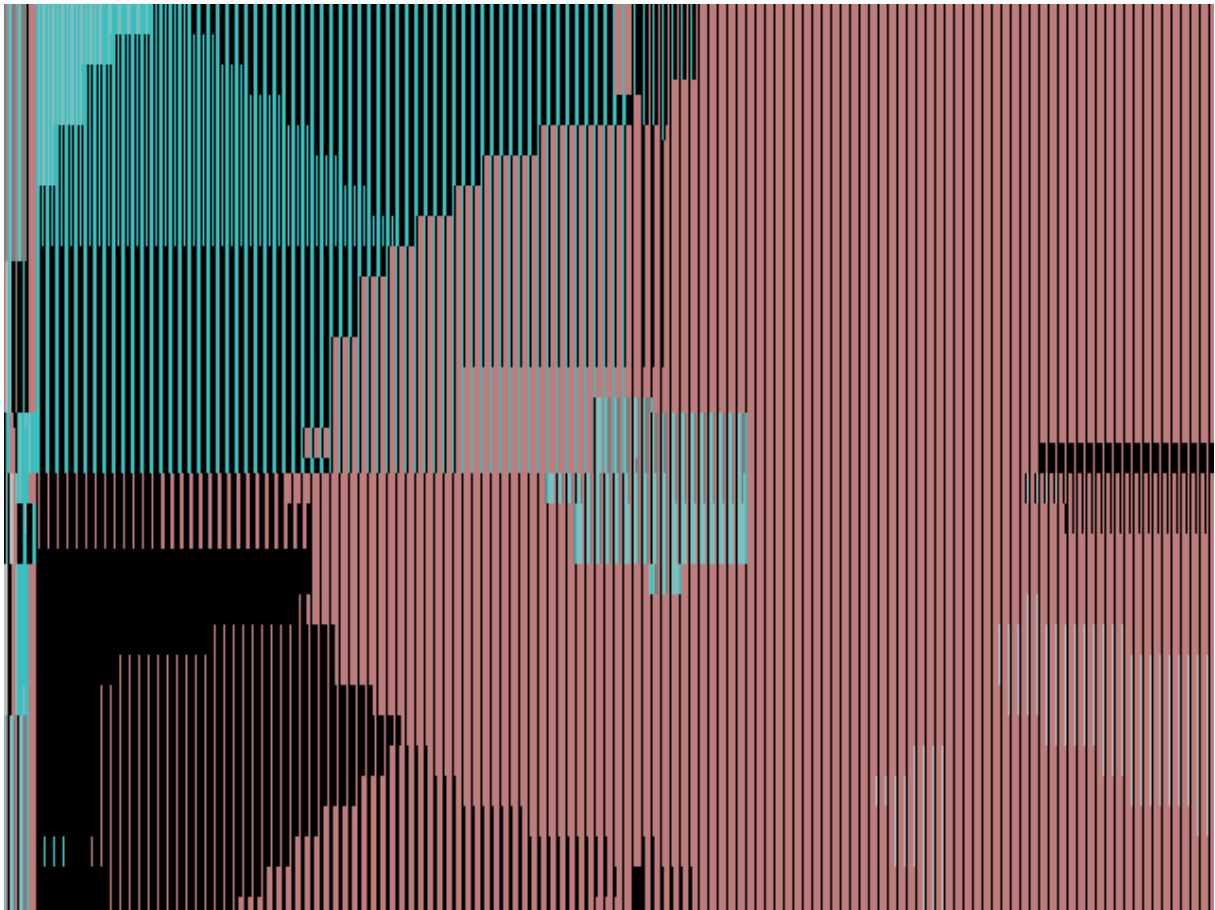
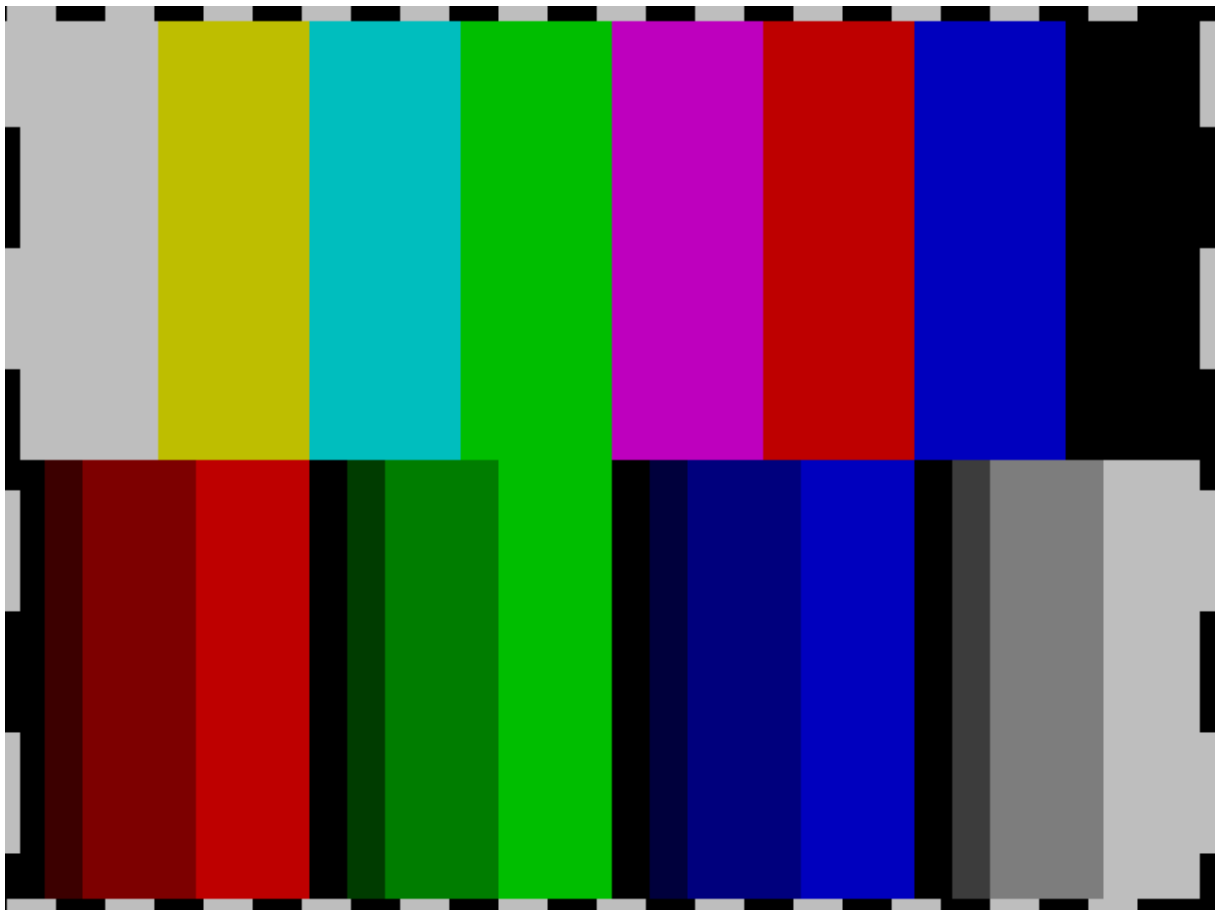


Figure 4.1: circles

Test mode

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



The “audio” out is connected, but is not expected to result in anything audible.

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 `uio[7]`

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	test mode (0=no, 1=yes)	r1	—
1	—	g1	—
2	—	b1	—
3	—	vsync	—
4	—	r0	—

#	Input	Output	Bidirectional
5	—	g0	—
6	—	b0	—
7	—	hsync	PWM output

VGA Mandelbrot

by **Mike Bell**

0006

100 MHz

HDL Project

github.com/MichaelBell/tt08-mandelbrot

“Mandelbrot on VGA, racing the beam”

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:

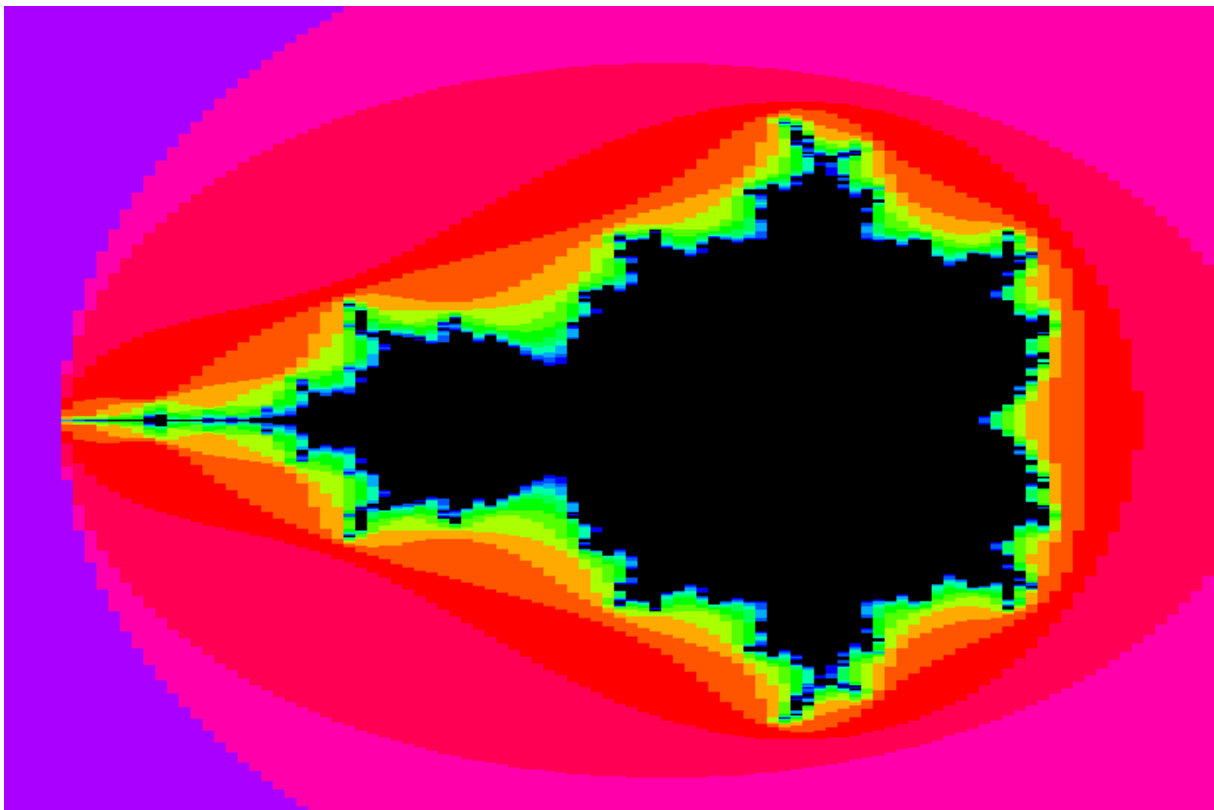


Figure 6.1: The Mandelbrot set

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

The output image is at a 720x480 resolution (103x480 Mandelbrot pixels).

How to test

Provide a 100MHz clock.

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

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

These values should only be updated during vsync.

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

Note there are 103 columns and 480 rows displayed.

External hardware

[Tiny VGA Pmod](#) in the output socket.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Ctrl 0	R[1]	Input 5
1	Ctrl 1	G[1]	Input 6
2	Ctrl 2	B[1]	Input 7
3	Input 0	vsync	Input 8
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

schoolRISCV CPU with Fibonacci program

by **Stanislav Zhelnio, Alexander Romanov, Yuri Panchul & Mike Kuskov**

0008

50 MHz

HDL Project

github.com/ddvca/tt08-schoolriscv-cpu-with-fibonacci-program

"A minimalistic SoC with a schoolRISCV educational CPU and a ROM memory with a program that computes the Fibonacci numbers."

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 *uio_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.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	Test pass	CPU reg a0[0]
1	—	Test fail	a0[1]
2	—	—	a0[2]
3	—	—	a0[3]
4	—	—	Program Counter pc[0]
5	—	—	pc[1]
6	—	—	pc[2]
7	—	—	pc[3]

Frequency Counter SSD1306 OLED

by Pawel Sitarz (embelon)

0010

1 MHz

HDL Project

github.com/embelon/tt08-frequency-counter-oled

"Simple Frequency Counter displaying result on SSD1306 SPI OLED"

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](#)

Project Pinout

Digital Pins

#	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	—
7	—	—	—

Patater Demo Kit Wagglng Rainbow on a Chip

by Jaeden Amero

0032

25.175 MHz

HDL Project

github.com/Patater/tt08-demokit

"A 6-bit Wagglng Rainbow demo"

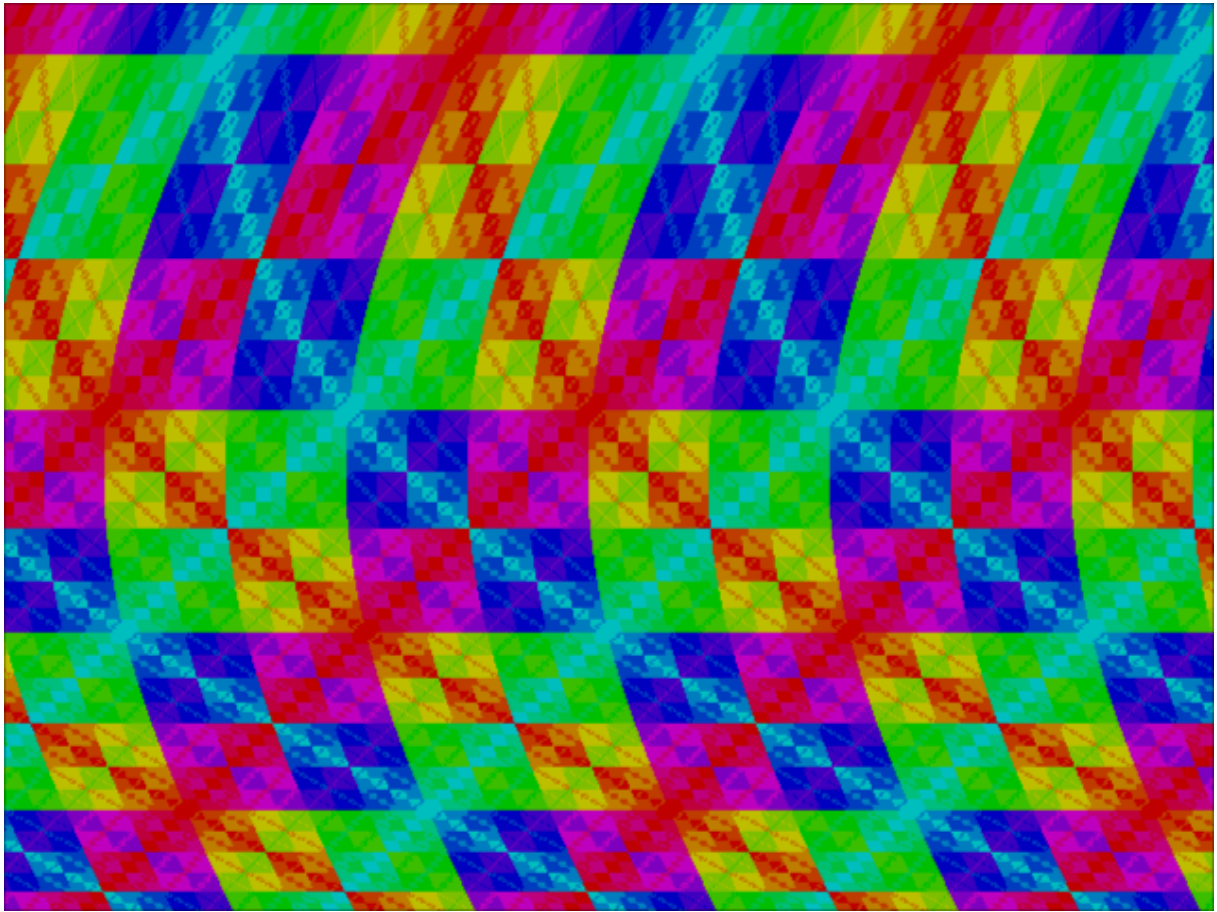
How it works

This design outputs a wagglng 6-bit rainbow demo on VGA.

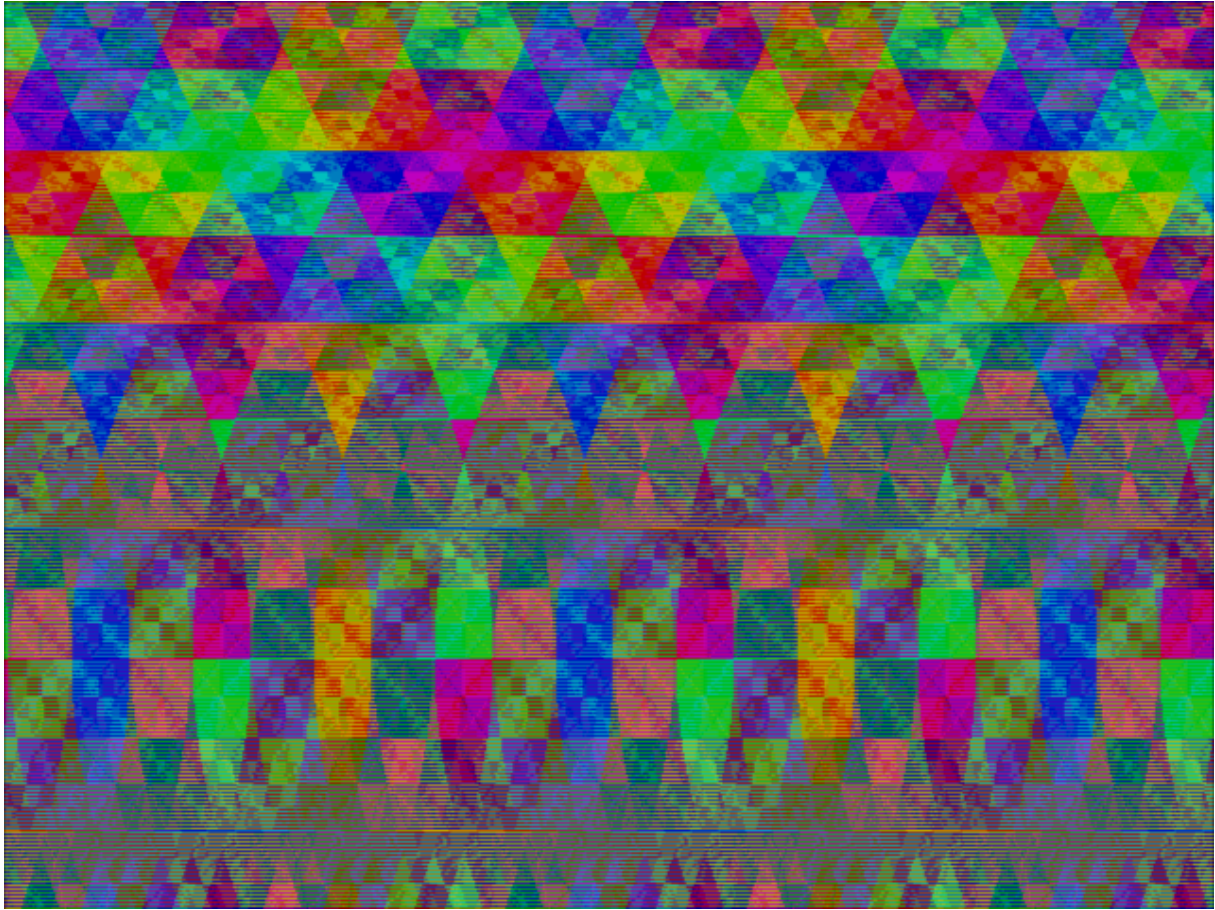
The demo will change effect based on inputs on `ui_in`.

Pin	Pin Name	Setting	Effect
<code>ui_in[0]</code>	DUAL_EN	Dual mode	Horizontally flips the image each scan line
<code>ui_in[1]</code>	HWAVE_EN	H wave	Enables use of horizontal waves
<code>ui_in[4:2]</code>	P0_OFF_{2,1,0}	P0 offset	Controls the speed of the H wave
<code>ui_in[7:5]</code>	P1_OFF_{2,1,0}	P1 offset	Controls the speed of the V wave

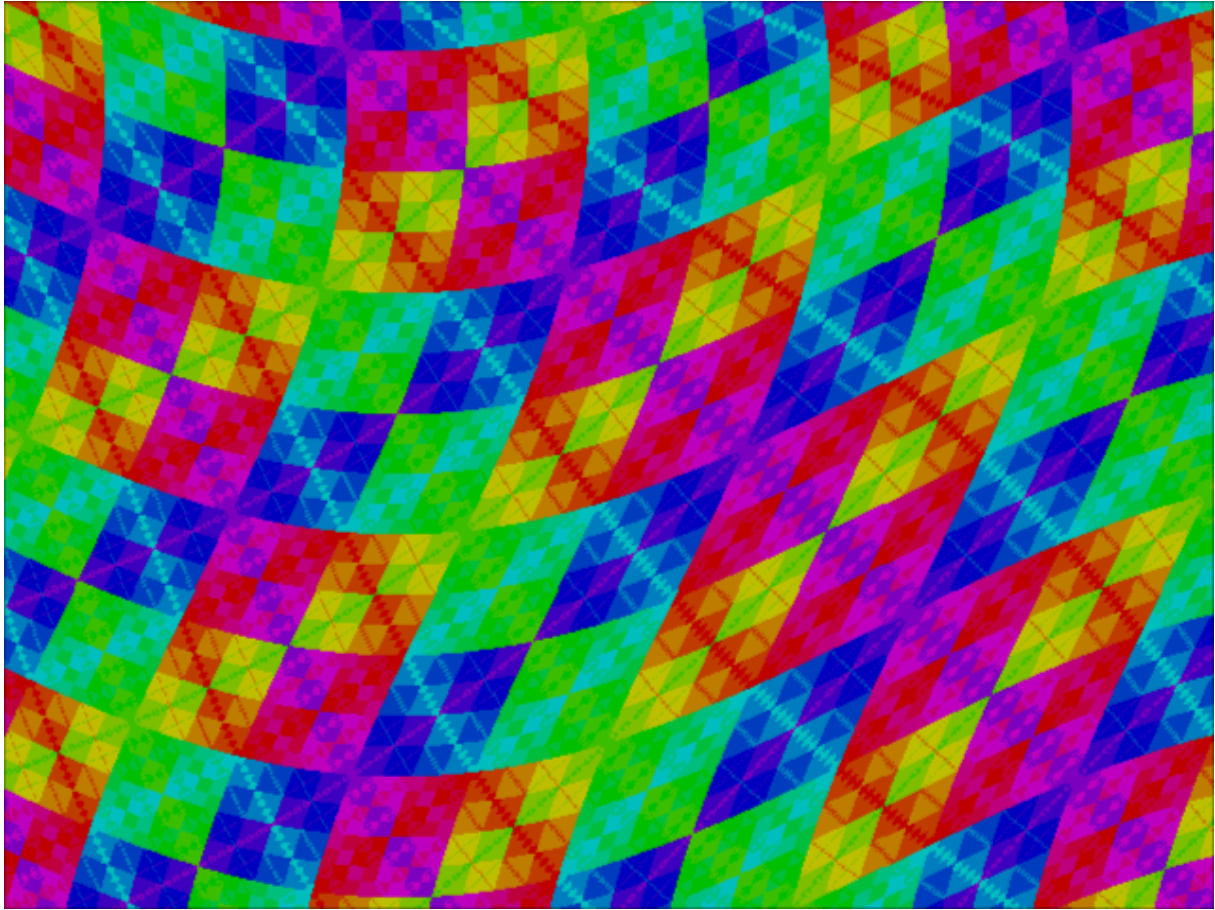
Screenshots



Default



Dual Mode



H Wave

Video

A video of a different (software rather than hardware) implementation, of the waggling 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](#)

Project Pinout

Digital Pins

#	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	—

Orion Iron Ion [TT08 demo competition]

by **Toivo Henningsson**

0033

50.4 MHz

HDL Project

github.com/toivoh/tt10-demo

"My contribution to the TT10 demo competition"

How it works

TODO

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](#).

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	R1	—
1	—	G1	—
2	—	B1	—
3	—	vsync	—
4	—	R0	—
5	—	G0	—
6	advance[0]	B0	—
7	advance[1]	hsync	audio_out

Moody-mimosa

by **D. Levante-Schmidiger**

0035

1 MHz

HDL Project

github.com/schda1/tt10-moody-mimosa

“Moody ASIC reacting to external stimuli”

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 code and the additional information required for hardening.
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 Alchitry board. For this part, I use the vivado command line utilities and the alchitry loader.
simulation	A simulation based on Pyverilator and PyQt6 with a graphical interface for optimizing the model (fine-

	tuning, feedback loops) and debugging the ASIC. Pyverilator uses the actual verilog sources, compiles them into a C/C++ application and allows to access it from within python.
<code>misc/mimosa_logger</code>	An STM32 application for an STM32G474 microcontroller to debug the ASIC. This logs all inputs and outputs, returns them via UART/FTDI, and allows for reconstructing the complete history using the simulation. I also added a CppUTest unit test for illustration.

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

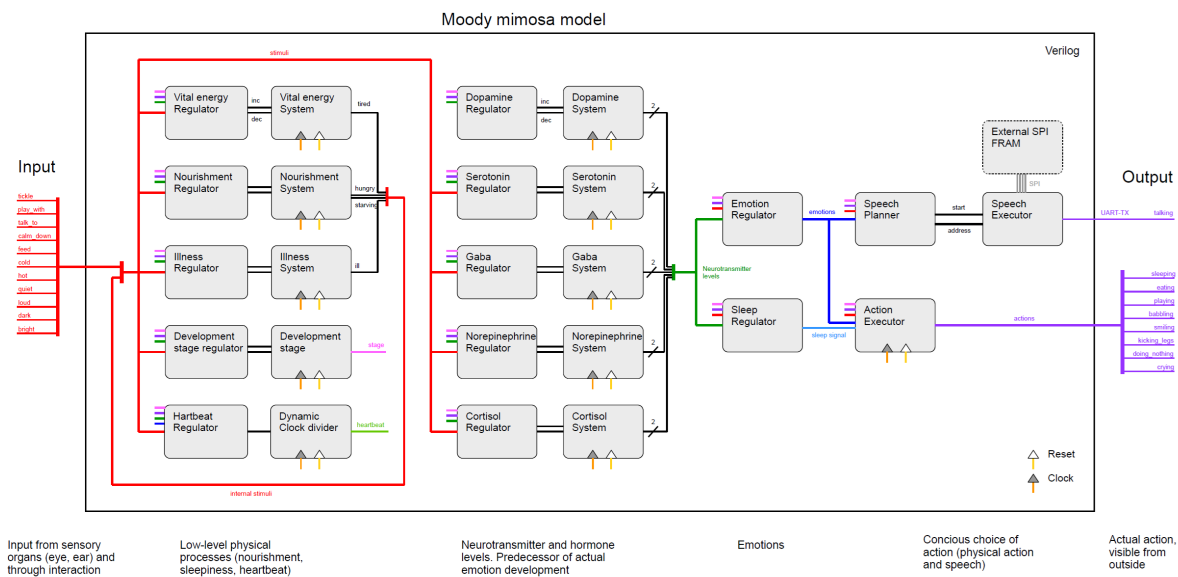
The moody mimosa model depends on several layers of abstraction.

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 individuum (tickle, play with, calm down, talk to, feed).
4. Basic resources:
 - Neurotransmitters: Dopamine, Serotonin, Gamma-aminobutyric acid (Gaba), Norepinephrine
 - Hormons: 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_in[1]	stimulus_1	Interaction: Play with
ui_in[2]	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
uio_in[0]	0	stimulus_8	Environment: Loud
uio_in[1]	0	stimulus_9	Environment: Dark
uio_in[2]	0	stimulus_10	Environment: Bright
uio_out[3]	-	-	-
uio_out[4]	-	-	-
uio_out[5]	-	-	-
uio_out[6]	-	-	-
uio_out[7]	-	-	-

Pin	Name	Function
uo_out[0]	action_0	Action: Sleeping

uo_out[1]	action_1	Action: Eating
uo_out[2]	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

Project Pinout

Digital Pins

#	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_MOSI

#	Input	Output	Bidirectional
6	HOT	DOING_NOTHING	UART_TX
7	QUIET	CRYING	CLK_MODEL

TinyTapeout Minimal Branch Predictor

by **Tristan Robitaille**

0037

1 MHz

HDL Project

github.com/TristanRobitaille/tt_branch_pred

“A minimal perceptron-based branch predictor”

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 prediction 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.ld -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

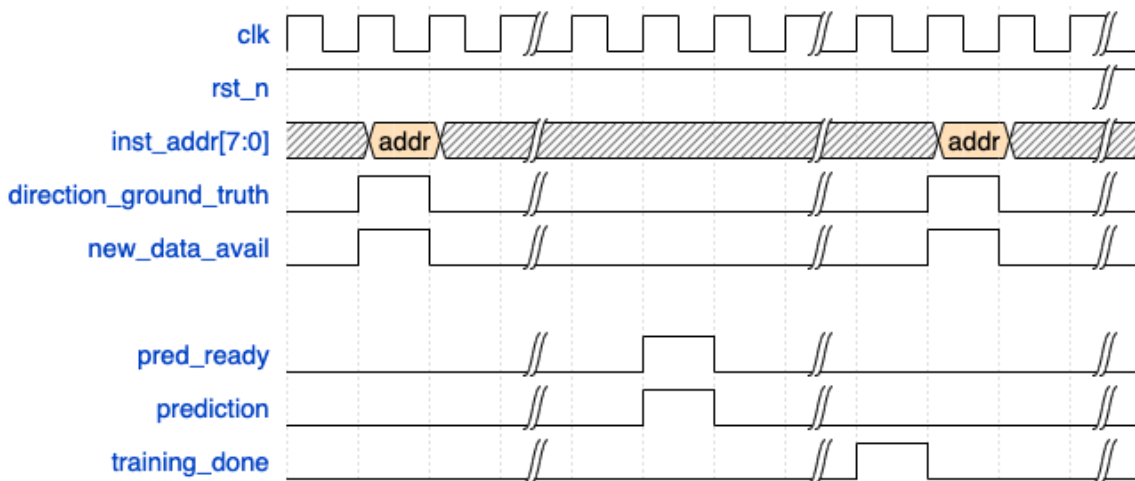
The branch predictor reads the lowest 8b of a 32b RISC-V address on pins `inst_addr`, the branch direction on pin `uio[1]` and a pulse notifying that new data is available on pin `new_data_avail`. Once the prediction is ready, it pulses pin `pred_ready`, pushes the prediction on pin `prediction`. Once 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.

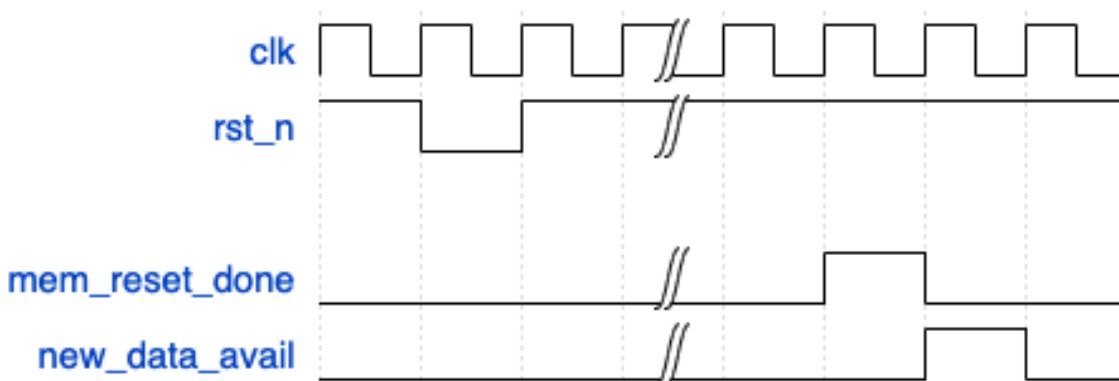
Notes:

- 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.)

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	inst_addr[0]	pred_ready	new_data_avail
1	inst_addr[1]	prediction	direction_ground_truth
2	inst_addr[2]	training_done	DEBUG_perceptron_index[0]
3	inst_addr[3]	mem_reset_done	DEBUG_perceptron_index[1]
4	inst_addr[4]	DEBUG_new_data_avail_posedge	DEBUG_perceptron_index[2]
5	inst_addr[5]	DEBUG_state_pred[0]	DEBUG_wr_en
6	inst_addr[6]	DEBUG_state_pred[1]	DEBUG_history_buffer_output
7	inst_addr[7]	DEBUG_state_rst_mem	history_buffer_request

Ternary 128-element Dot Product

by ReJ aka Renaldas Zioma

0039

HDL Project

github.com/rejunity/tt10-ternary-dot-product

*"sum($A * B$) where A is a binary vector, B is a ternary vector"*

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.

Project Pinout

Digital Pins

#	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

by J. Arias

0041

24 MHz

HDL Project

github.com/jesari-git/tt10-gus16-jarias

"16-bit CPU design"

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-IIs in FPGAs)

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	gpi[0]	xbh	xd[0]
1	gpi[1]	xlal	xd[1]
2	gpi[2]	xlal	xd[2]
3	rx	pwmout	xd[3]
4	gpi[3]	tx	xd[4]

#	Input	Output	Bidirectional
5	gpi[4]	gpo	xd[5]
6	gpi[5]	xoeb	xd[6]
7	gpi[6]	xweb	xd[7]

donut

by **Daniel Endraws**

0043

50.35 MHz

HDL Project

github.com/daniel-endraws/tt08-donut

"Showing a Donut"

How it works

Each ellipse is hand crafted to create a donut.

How to test

Connect the PMOD VGA.

External hardware

TinyVGA PMOD

Project Pinout

Digital Pins

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

UART

by **Darryl Miles**

0045

HDL Project

github.com/dlmiles/tt08-poc-uart

“UART”

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)

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	altclk	—	busData0
1	busMode0	—	busData1
2	busMode1	—	busData2
3	rxid	dtr	busData3
4	dtr	txd	busData4
5	dcd	rts	busData5
6	cts	intTx	busData6
7	ri	intRx	busData7

FSK Modem +HDLC +UART (PoC)

by Darryl Miles

0047

HDL Project

github.com/dlmiles/tt06-poc-fskmodem-hdlctrx

"FSK Modem w/ HDLC transceiver + UART (PoC digital side)"

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, the shifts the expectation of the result.

I also need to validate the DAC 8bit loading scheme prevents any chirping (visibly 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 incorpotates a full-duplex HDLC frame processor attached to a UART (ttl interface), the UART process encodes the frame in format similar to KISS format used by TNCs, with a few modifications.

Lower Digital (included here)

This manages the receiver clock recovery PLL circuit and interface, the original designs used EPROM lookup tables with 12bit address (which has visibility on at least the previous encoded bit) and provides an 8bit data output.

The data outputs are then fed into a respective 8bit DAC

The receiver has a PLL lock detector which is used to provide DCD (Data Carrier Detect) signal. While the hardware design is capable of full-duplex operation it is often used in Amateur Radio situations in a half-duplex situation with a carrier sense channel sharing algorithm.

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-commercial radio trancievers. This may have been a comparator circuit (unsure at this time), fed into a DFF to synchronise the incoming data to the x16 (of datarate) clock recovery timing
- 2 x opamp to provide PLL lock detection (unsure how this works atm), I would guess it can detect when the signal is being centered and has been centered for some number of samples, maybe via slow capacitance charge up when the UP/DOWN line is managing to meet an approximate 50%/50% duty cycle per x16 clock recovery tick.
- 2 x opamp to provide zero-crossing detection, this is used to provide the PLL its feedback mechanism (the UP/DOWN line) to advance or retard the edge alightment.

It is hoped all items can be incorporated into the same design using the analogue GDS facility with TT and connected to the respective lower digital signal.

At this time we bring out the interconnection points (between analogue and lower digital) to the external interface of TT and we provide a configuration mechnism to be externally or internally driven/internally sourced. This should allow for a significant level of simulation and experimentation by users of the project to understand and explore FSK/PLL theory by picking a testing configuration combintation, being full-duplex it should be able to loop-back at various levels to understand each part better. While also providing those

with a Ham Radio license to try out on air communicating with their local users or AMSAT.

Have fun... 73s de G7LED

How to test

When the final design is completed, there should be a number of visible and testable aspects available to observe the working of various functions.

I am not expecting this PoV project to yeild good result due to the limited time spent on it just before submission deadlines for TT06.

Check back with the repo for a testing regime.

External hardware

At this PoC stage, testing with RP2040 and FPGA external boards to validate the electrical interface acrhetecture makes sense and provided the most options.

Project Pinout

Digital Pins

#	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	TableAddr[1]
4	TableData[1]	Tx Error	TableAddr[2]
5	TableData[2]	Sending	TableAddr[3]
6	TableData[3]	—	TableAddr[4]
7	UART RX	Tx Clock Stobe	TableAddr[5]

Comm_IC

by Bhavuk

0064

20 GHz

HDL Project

github.com/Bhavuk-HDL/tty8_comm_ic

"Communication protocols: UART, SPI, I2C"

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 Aynchronous 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 e receive 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

Project Pinout

Digital Pins

#	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_out[1]
4	—	SCL	data_out[2]
5	—	busy_uart	data_out[3]
6	—	busy_spi	error_i2c
7	—	busy_i2c	—

VGA Drop (audio/visual demo)

by **ReJ** aka Renaldas Zioma, **eriQue** aka Erik Hemming & **Matthias Kampa**

0066

25.2 MHz

HDL Project

github.com/rejunity/tt08-vga-drop

"Tiny 8 part Megademo! TBL^Nesnausk^SonikClique"

How it works

VGA signal generator

How to test

We are learning how VGA and Sky130 works here

External hardware

VGA PMOD

Project Pinout

Digital Pins

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

Warp

by **sylefeb**

0068

25 MHz

HDL Project

github.com/sylefeb/tt08-compo-entry

“Demo on TinyTapeout? Let's do something!”

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 lense distortion, achieving the final look.

Audio

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

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

How to test

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

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

External hardware

- VGA PMOD
- Audio PMOD

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

Project Pinout

Digital Pins

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

VGA Nyan Cat

by **Andy Sloane**

0070

25.175 MHz

HDL Project

github.com/a1k0n/tt08-nyan

"Displays the classic nyan.cat animation"

VGA nyan cat



Figure 70.1: nyancat preview

How it works

Outputs nyancat on VGA with music!

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

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

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

How to test

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

External hardware

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

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

Project Pinout

Digital Pins

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

16 Mic Beamformer

by Armaan Gomes

0072

HDL Project

github.com/arghunter/CIC-Filter-Verilog

"A 0 delays fixed delay and sum beamformer that can utilize up to 16 input microphones"

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

Project Pinout

Digital Pins

#	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

by Armaan Gomes

0074

HDL Project

github.com/arghunter/Customizable-PDM-Pitch-Filter-ASIC

"It uses a moving average filter and decimator to filter out a specific frequency"

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

Project Pinout

Digital Pins

#	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	—

PDM Correlator

by Armaan Gomes

0076

HDL Project

github.com/arghunter/Customizable-PDM-Cross-Correlator-ASIC

"A chip that performs either cross or auto correlation on PDM microphone inputs"

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

Microphones, clock generator, spi port List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

#	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	—

Bouncy Capsule

by **htfab**

0078

25 MHz

HDL Project

github.com/htfab/bouncy-capsule

“Demoscene project featuring... well, a bouncy capsule”

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](#))

Project Pinout

Digital Pins

#	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

My Project

by **Asger Wenneberg**

0097

HDL Project

github.com/AsgerWenneb/trt10-verilog

"This is a tiny tapeout design"

How it works

This is how it works

How to test

This is how to test

External hardware

Nothing yet

Project Pinout

Digital Pins

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

simple-viii

by **strau**

0099

10 kHz

HDL Project

github.com/strau0106/tt-simple-viii

"A simple 8-bit CPU Architecture"

How it works

How to test

External hardware

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	—	cs flash
1	—	—	SD0
2	—	—	SD1
3	—	—	SCK
4	—	—	SD2
5	—	—	SD3
6	—	—	cs ram
7	—	—	—



ttUART

by **Bogdan Tanasa**

0101

50 MHz

HDL Project

github.com/bgdtanasa/ttUART

"UART"

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.

Project Pinout

Digital Pins

#	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	—	—	—

UW ASIC - Optimized Dino

by University of Waterloo ASIC Design Team

0107

25.175 MHz

HDL Project

github.com/UW-ASIC/Dino

"Dino game, but only 2 tiles!"

How it works

Placeholder

How to test

Placeholder

External hardware

Placeholder

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	ui0	uo0	uio0
1	ui1	uo1	uio1
2	ui2	uo2	uio2
3	ui3	uo3	uio3
4	ui4	uo4	uio4
5	ui5	uo5	uio5
6	ui6	uo6	uio6
7	ui7	uo7	uio7

TT09 SKY130 ROM Test

by Sylvain Munaut

0109

HDL Project

github.com/smunaut/tt09-rom-test

“Test of some prototype ROM macros”

How it works

Just some registers in front of a few ROM macros to be able to send the address and capture data at specific intervals.

How to test

Set `clk` and `rst_n` to select one of the 4 possible testing mode.

Load a test address to read setting half the bits on `ui[6:0]` and then using both `ui[7]` and `uio[7]` to load the internal preload register.

Then apply a clock edge on `uio[6]` to clock the address register which will transfer the address from the pre-load register to the actual address register and send the address to the ROMs.

After some delay, apply a clock edge on `uio[5]` which will capture the output of the ROM.

External hardware

To do any meaningful timing testing you'll need some FPGA hardware to drive the various control signal in sequence with precise timings.

The exact testing platform is still TBD.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	<code>addr_in[0]</code>	<code>data[0]</code>	<code>data[5]</code>
1	<code>addr_in[1]</code>	<code>data[1]</code>	<code>data[6]</code>
2	<code>addr_in[2]</code>	<code>data[2]</code>	<code>data[7]</code>
3	<code>addr_in[3]</code>	<code>data[3]</code>	<code>data[8]</code>
4	<code>addr_in[4]</code>	<code>data[4]</code>	<code>data[9]</code>
5	<code>addr_in[5]</code>	—	clk of data register
6	<code>addr_in[6]</code>	—	clk of addr register

#	Input	Output	Bidirectional
7	clk of addr_ld[13:7] register	—	clk of addr_ld[6:0] register

TT09 SKY130 ROM Test (no LVT variant)

by Sylvain Munaut

0111

HDL Project

github.com/smunaut/tt09-rom-nolvt-test

"Test of some prototype ROM macros modified to remove LVT implant on bitcell"

How it works

Just some registers in front of a few ROM macros to be able to send the address and capture data at specific intervals.

How to test

Set `clk` and `rst_n` to select one of the 4 possible testing mode.

Load a test address to read setting half the bits on `ui[6:0]` and then using both `ui[7]` and `uio[7]` to load the internal preload register.

Then apply a clock edge on `uio[6]` to clock the address register which will transfer the address from the pre-load register to the actual address register and send the address to the ROMs.

After some delay, apply a clock edge on `uio[5]` which will capture the output of the ROM.

External hardware

To do any meaningful timing testing you'll need some FPGA hardware to drive the various control signal in sequence with precise timings.

The exact testing platform is still TBD.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	<code>addr_in[0]</code>	<code>data[0]</code>	<code>data[5]</code>
1	<code>addr_in[1]</code>	<code>data[1]</code>	<code>data[6]</code>
2	<code>addr_in[2]</code>	<code>data[2]</code>	<code>data[7]</code>
3	<code>addr_in[3]</code>	<code>data[3]</code>	<code>data[8]</code>
4	<code>addr_in[4]</code>	<code>data[4]</code>	<code>data[9]</code>
5	<code>addr_in[5]</code>	—	clk of data register

#	Input	Output	Bidirectional
6	addr_in[6]	—	clk of addr register
7	clk of addr_ld[13:7] register	—	clk of addr_ld[6:0] register

CMOS design of 4-bit Signed Adder Subtractor

by Vivek Chiranjit

0128

HDL Project

github.com/ChiranjitPatel/tt08_SignedAddSub_HDL

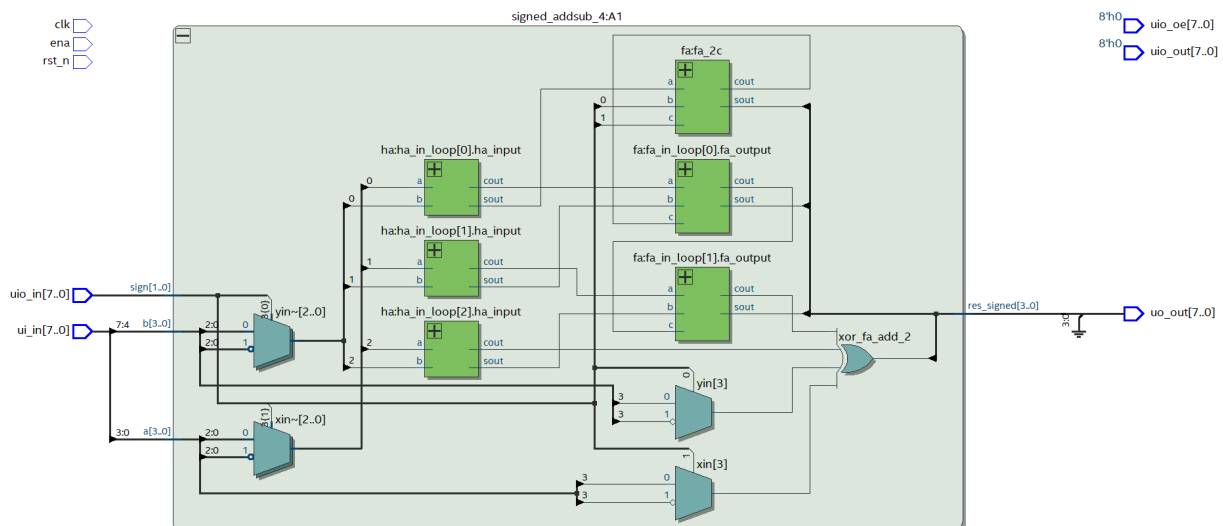
"The project is a signed binary 4-bit adder-subtractor module."

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 Ques-taSim.

External hardware

None

Project Pinout

Digital Pins

#	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	—	—

Demo by a1k0n

by Andy Sloane

0130

48 MHz

HDL Project

github.com/a1k0n/tt08-vgademo

"Tiny Tapeout demo competition entry"

a1k0n's tinytapeout08 demo compo entry



Figure 130.1: preview

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.

Project Pinout

Digital Pins

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

sky130_as_sc_hs test

by Tholin

0134

10 MHz

HDL Project

github.com/AvalonSemiconductors/tt_sky130_as_sc_hs_test

"Testing sky130_as_sc_hs SCL"

How it works

A simple 6-bit MCU, but built with the sky130_as_sc_hs SCL. Check [here](#) for documentation on the CPU architecture.

How to test

Wire up a 25Q32 (or compatible) spiflash with valid bytecode using the SDO, SDI, SCLK and CSb pins, pull INT low and HS low. A test program is included in this repo under test/asm/ which counts up on the PA lines.

External hardware

25Q32 (or compatible) spiflash to hold CPU bytecode. Anything you want on the GPIO lines.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	PB[0]	PC[0]	PA[0]
1	PB[1]	PC[1]	PA[1]
2	PB[2]	PC[2]	PA[2]
3	PB[3]	PC[3]	PA[3]
4	PB[4]	PC[4]	PA[4]
5	PB[5]	PC[5]	PA[5]
6	INT	SDO	HS
7	SDI	SCLK	CSb

SPI FPU

by **Sebastian Pfeiler**

0138

50 MHz

HDL Project

github.com/Qwendu/tt_spi_fpu

"A 32bit floating point adder accessible over spi"

How it works

This is a floating point unit accesible over SPI.

```
SCLK = ui[0]
NCS  = ui[1]
COP1 = ui[2]
CIP0 = 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 COP1) 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_integration_test`.

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.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	SPI_CLK	SPI_OUT	Unused
1	SPI_NCS	Unused	Unused

#	Input	Output	Bidirectional
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

Space Invaders ASIC

by **Lukas Krupp** & **Adam Gebregziabher**

0142

25 MHz

HDL Project

github.com/ppurkl/tt10-chip-invaders

"HDL implementation of the retro game Space Invaders"

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input1	out1	—
1	input2	out2	—
2	input3	out3	—
3	—	out4	—
4	—	out5	—
5	—	out6	—
6	—	out7	—
7	—	out8	—

Simple 8 Bit ALU

by Joseph Johnson

0161

1 kHz

Wokwi Project

github.com/jwjbader/tt08-8bit-alu

wokwi.com/projects/407852791999030273

"a very simple ALU"

How it works

The project currently works by including a 8-bit shift register connected to the bus. The SI pin can be used to shift data into the register, and the ESO pin determines whether the shift-register will output to the bus. The bus is a conjunction of OR-gates making it one directional (as simply combining the wires seems less possible). There is a tri-state buffer that connects to two registers (register A and B) allowing them to interact with the bus. The direction is controlled using DIRA and DIRB. If the DIRA/DIRB sets the direction to be using the bus as an input, LA/LB determines whether the register should load what is on the bus. Register B is connected to the ALU and may be inverted to form a 2's Complement subtraction using the AS pin. The ALU will thus perform addition or subtraction depending on the state of AS. It will take Register A as the other input and output to the bus if the EAO pin is pulled high.

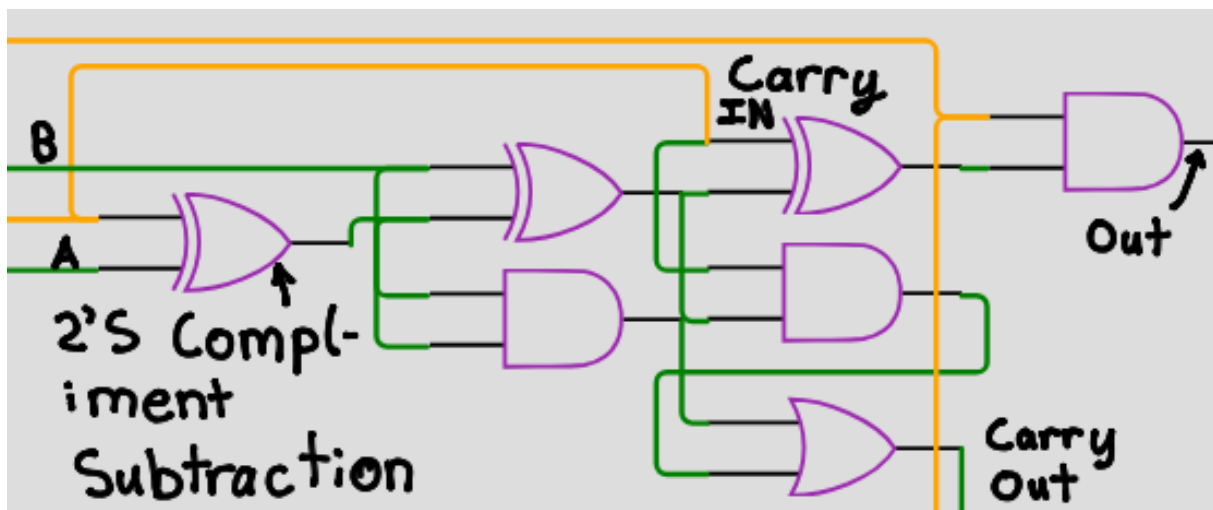


Figure 161.1: Annotated diagram of ALU

You can see here an annotated diagram of a single portion of the ALU. The wires labeled A and B represent the arbitrarily named inputs (in this case, the input labeled A actually comes from the B register and vice versa). The original xor gate allows for 2's compliment subtraction with input A. The orange wire labeled carry in comes from the AS pin and determines whether

A is being added or subtracted. 2's complement subtraction requires one be added to the number so we carry one in (this is the LSB, so nothing would be carried in typically). The output on the right is the single digit output. The carry out will be hooked up to the carry in of the next segment to add numbers greater than 1 bit.

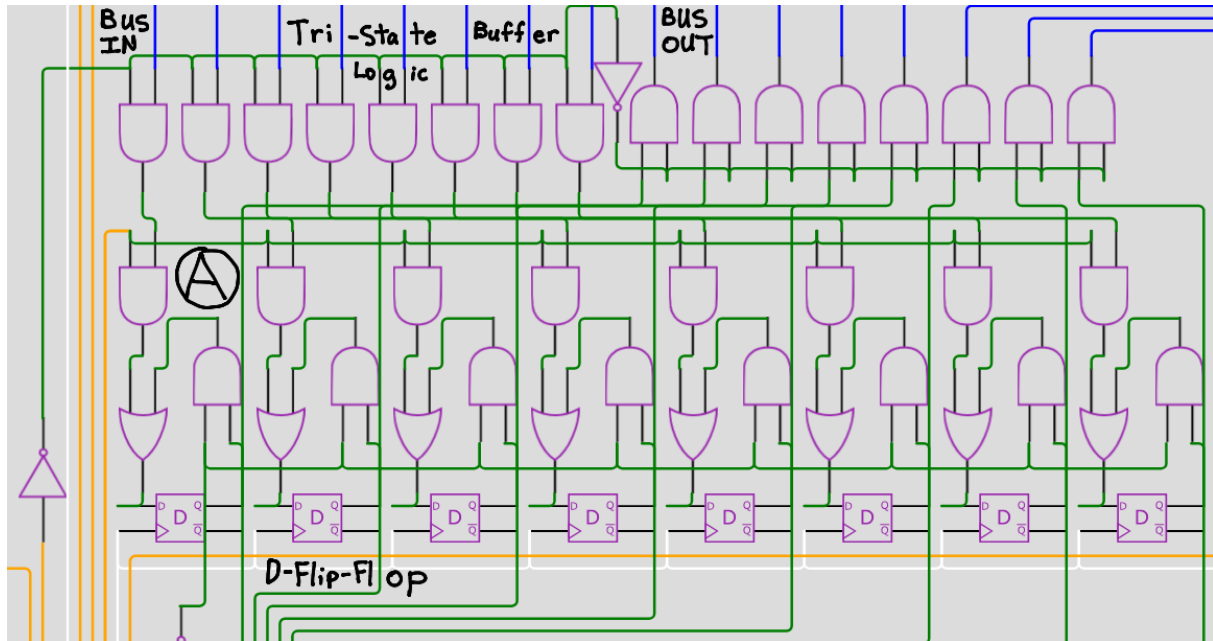


Figure 161.2: Annotated diagram of register

You should be able to see here an annotated diagram of a single register. The 8 blue wires to the right of “Bus IN” represent the bus input and will only be active if the DIRA pin is set low (because of the inverter towards the bottom left). This is inverted to determine whether the bus should be considered an output and the bus output appears on the top right. The section labeled “A” is a special circuit that does not load any data if the LA pin is not high. Because the D-Flip-Flop circuit will complain if its clock input contains any logic when being converted, a more difficult method to prevent unintended data modification must be used. If the LA pin is not high, the Q pin is piped back to the D pin on each of the flip-flops, retaining the data.

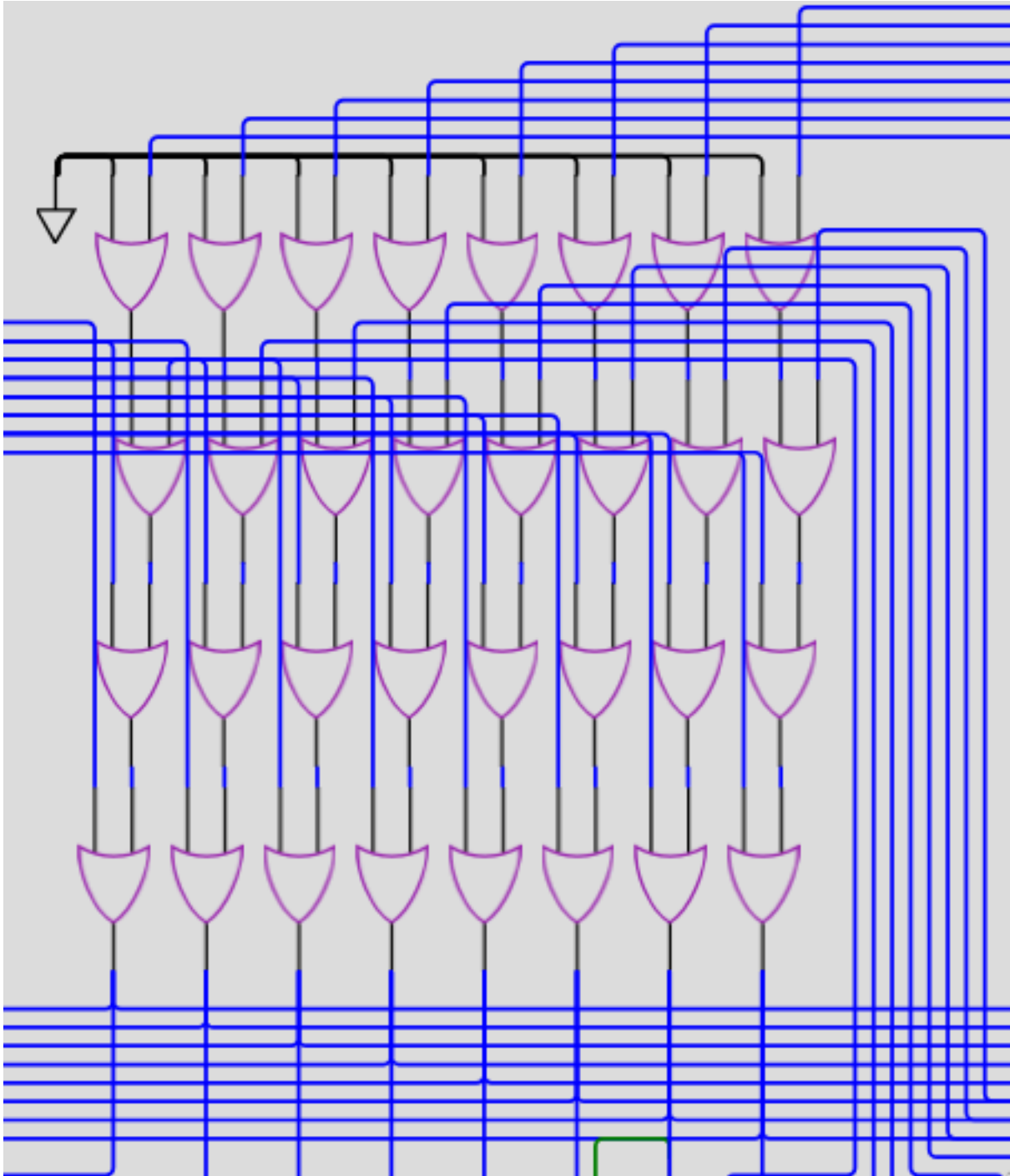


Figure 161.3: Diagram of Bus

This picture demonstrates the data bus. It uses logical OR gates to connect a series of wires together which are then used to produce output. The bus output is shown as the output of the entire circuit, and can be used to send data between the ALU and registers.

How to test

Input your two numbers MSB first (using the SI pin), pulsing the clock after inputting each individual bit. Set the ESO pin high to output the contents of

the shift-register to the bus. Leave DIRA and DIRB on their default to allow information to stream in from the bus (inverting them will output the contents of register A or B to the bus). Set LA or LB appropriately depending on where you would like to load information. Pulse the clock to load information from the bus to any listening registers. Reset LA/LB to prevent accidentally overwriting data from the bus. Whenever Register A and B are set correctly, set the AS pin depending on whether you desire to add or subtract register B. Set the EAO pin high to output the contents of the ALU to the bus. The output pins will mirror the bus and be produced with OUT0 indicating the LSB and OUT7 indicating the MSB.

You will likely wish to shift a few zeros into the shift-register first to ensure it is not full of corrupted data (it likely will be when the chip is initially started).

One fun thing to do is load the value 1 into register A and 0 into register B then set the contents of the ALU to output to the bus and set LB high. This will cause 1 to be added to register B every clock pulse and you should be able to see the output count up. You can do the same thing but backwards if you set the AS pin high.

Pin Location	Pin Name	Expanded Name	Description
0	SI	Serial Input	user may input data serially using this pin
1	ESO	Enable Serial Output	outputs the shift-register to the bus if high
2	DIRA	A Direction	allows register A to act as an input if low and vice versa
3	DIRB	B Direction	allows register B to act as an input if low and vice versa
4	EAO	Enable ALU Output	outputs the contents of the ALU to the bus if high
5	LA	Load A	loads information from the bus to register A if high
6	LB	Load B	loads information from the bus to register B if high

7	AS	Add Subtract	determines whether B is added (low) or subtracted (high)
---	----	--------------	--

External hardware

A sequence of 8 LEDs is hooked up to the outputs to display the final number in binary.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	SI	OUT0	—
1	ESO	OUT1	—
2	DIRA	OUT2	—
3	DIRB	OUT3	—
4	EAO	OUT4	—
5	LA	OUT5	—
6	LB	OUT6	—
7	AS	OUT7	—

cfib Demoscene Entry

by **Christian Fibich**

0163

50 MHz

HDL Project

github.com/cfib/tt08-cfib-demo

“Generates VGA video and PWM audio”

How it works

My entry to the Tintapeout Demoscene Competition.

It (pseudo-randomly) generates a soundtrack via PWM and displays a waveform via VGA.

How to test

Connect VGA and PWM Pmod.

Then just apply clock and (asynchronous) reset.

External hardware

The project uses:

- Tiny VGA Pmod via `uo_out[7:0]` (<https://github.com/mole99/tiny-vga>)
- Mike’s audio Pmod via `uio_out[7]` (<https://github.com/MichaelBell/tt-audio-pmod>)

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	r[1]	—
1	—	g[1]	—
2	—	b[1]	—
3	—	vsync	—
4	—	r[0]	—
5	—	g[0]	—
6	—	b[0]	—
7	—	hsync	pwm

Munch

by **bytex64**

0165

25.175 MHz

HDL Project

github.com/bytex64/tt-munch

"Displays munching squares through VGA PMOD"

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 indirected 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 `uio_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](#)

Project Pinout

Digital Pins

#	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)

DemoSiine

by SagarDevAchar

0167

25 MHz

HDL Project

github.com/SagarDevAchar/tt08-demosiine

“A Wavy and Rainbowy TT08 Demoscene Submission”

How it works

The project structure is as shown below:

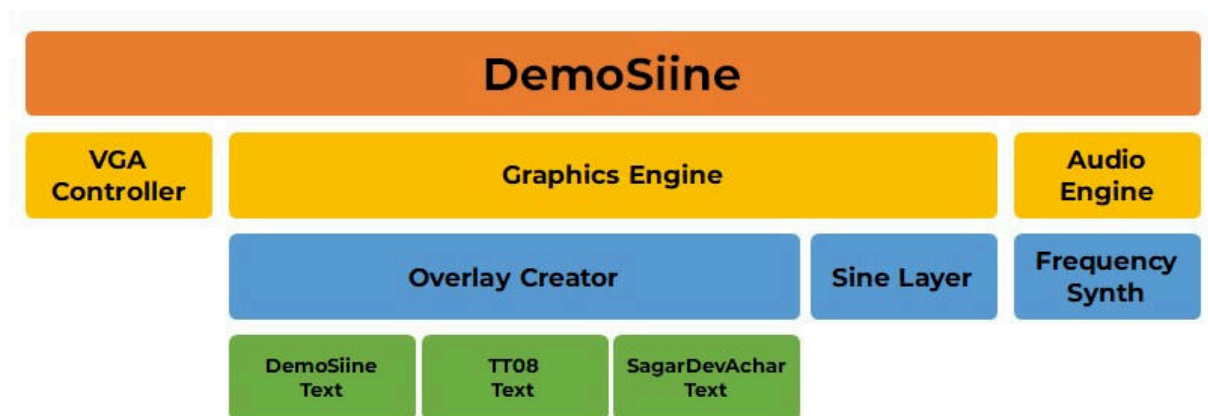


Figure 167.1: DemoSiine Project Structure

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

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

External hardware

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

How to test

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

Input Configurations

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

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

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

Previews

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



INPUT = xx000000 (Default)

Figure 167.2: DemoSiine Default Video Output Preview



INPUT = xx1x1000

Figure 167.3: DemoSiine Video Output Preview 2



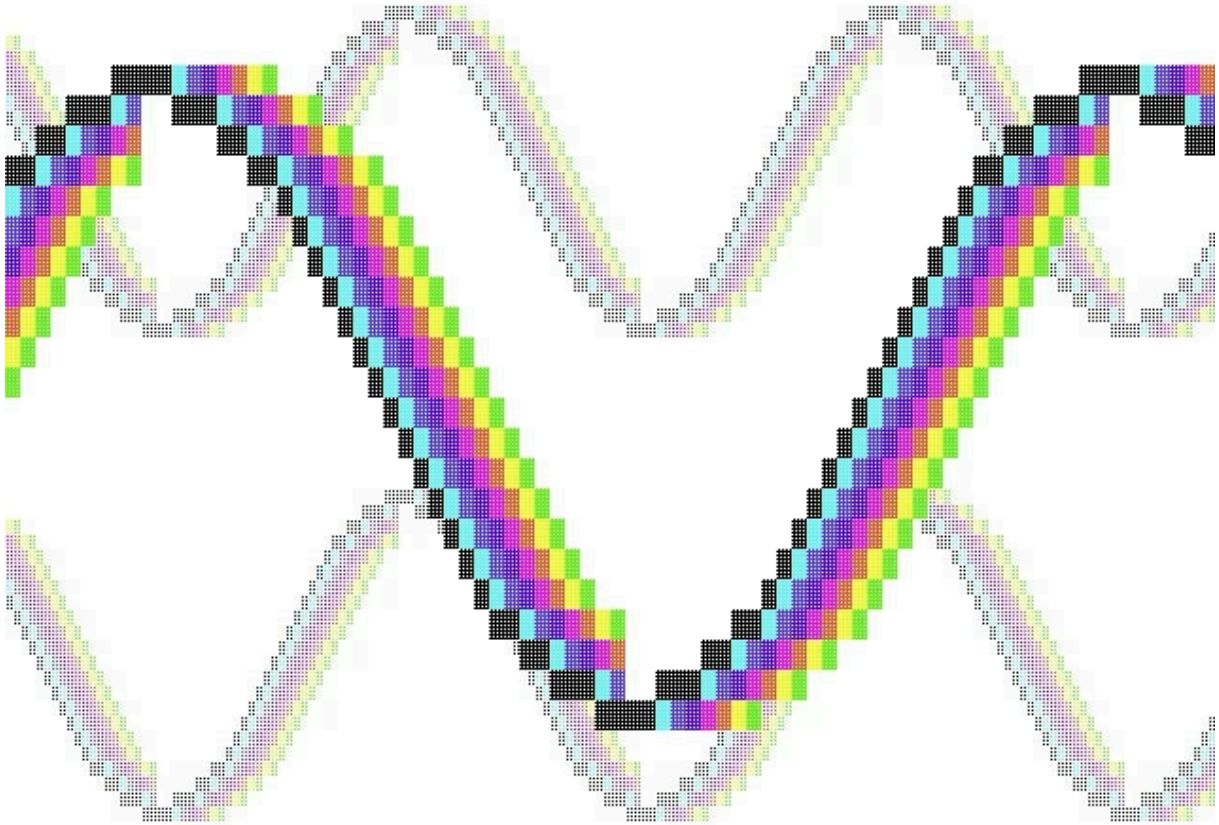
INPUT = xx100001

Figure 167.4: DemoSiine Video Output Preview 3



INPUT = xx110110

Figure 167.5: DemoSiine Video Output Preview 4



INPUT = xx0x1001

Figure 167.6: DemoSiine Video Output Preview 5



INPUT = xx010110

Figure 167.7: DemoSiine Video Output Preview 6

Project Pinout

Digital Pins

#	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

simon_cipher

by Simon Cipher

0169

HDL Project

github.com/Secure-Embedded-Systems/tt08-simon

"Bitserial implementation of Simon-128"

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.

Project Pinout

Digital Pins

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

VGA Screensaver with Zero to ASIC Logo

by **Matt Venn**

0171

25.175 MHz

HDL Project

github.com/mattvenn/Sky130-Z2A-Screensaver

“Zero to ASIC Logo bouncing around the screen (640x480, TinyVGA Pmod)”

How it works

Displays a bouncing Zero to ASIC logo on the screen, with animated color gradient.



Figure 171.1: Tiny Tapeout screensaver

How to test

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

- `tile` (`ui_in[0]`) to repeat the logo and tile it across the screen,
- `solid_color` (`ui_in[1]`) 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](#)

Project Pinout

Digital Pins

#	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	—

UART Matrix Multiplier

by Agneya Tharun

0173

50 MHz

HDL Project

github.com/RTimothyEdwards/tt10-verilog-template-agneya

"Multiplies two 2x2 matrices received via UART and transmits the result matrix back via UART"

How it works

This project implements a 2x2 matrix multiplier that communicates via UART. The design receives two 2x2 matrices through UART serial communication, performs matrix multiplication, and transmits the result back via UART.

Matrix Multiplication Process:

- Input: Two 2x2 matrices A and B
- Matrix A = $[[A0, A1], [A2, A3]]$
- Matrix B = $[[B0, B1], [B2, B3]]$
- Result C = $A \times B = [[C00, C01], [C10, C11]]$

Where:

- $C00 = A0 \times B0 + A1 \times B2$
- $C01 = A0 \times B1 + A1 \times B3$
- $C10 = A2 \times B0 + A3 \times B2$
- $C11 = A2 \times B1 + A3 \times B3$

Communication Protocol:

- UART configuration: 115,200 baud, 8N1 (8 data bits, no parity, 1 stop bit)
- Input: 8 bytes sent sequentially (A0, A1, A2, A3, B0, B1, B2, B3)
- Output: 4 bytes sent sequentially (C00, C01, C10, C11) - lower 8 bits of each result
- Parallel output: Result matrix elements are also available on dedicated output pins

State Machine:

1. Wait for UART input (8 bytes)
2. Compute matrix multiplication
3. Transmit results via UART (4 bytes)
4. Return to waiting state

How to test

Required Setup:

- UART terminal or microcontroller capable of serial communication
- Connection to ui[0] (UART RX) and uio[0] (UART TX)
- 100MHz clock input
- Active-low reset

Testing Procedure:

1. Connect UART interface:

- Connect UART TX from your device to ui[0] (chip's RX input)
- Connect UART RX from your device to uio[0] (chip's TX output)
- Set terminal to 120,000 baud, 8N1

2. Send test matrices: Send 8 bytes in sequence representing matrices A and B:

Example: $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, $B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$

Send bytes: 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08

3. Receive results: The chip will respond with 4 bytes representing the result matrix C:

Expected result: $C = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$

Received bytes: 0x13, 0x16, 0x2B, 0x32

4. Monitor parallel outputs:

- uo[0]: C00 (lower 8 bits)
- uo[1]: C01 (lower 8 bits)
- uo[2]: C10 (lower 8 bits)
- uo[3]: C11 (lower 8 bits)

Test Cases:

- Identity matrix: $A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, $B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} \rightarrow C = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$
- Zero matrix: $A = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$, $B = \text{any matrix} \rightarrow C = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$
- Small values: Test with single-digit numbers to verify correctness
- Edge cases: Test with values that might cause overflow (results > 255)

External hardware

Required:

- USB cable (USB-A to Micro-USB or USB-C, depending on your development board. Testing was performed using a Micro-USB cable.)
- No additional UART adapter needed - uses built-in USB-UART bridge

Optional:

- Logic analyzer or oscilloscope for debugging UART communication

- LED indicators connected to parallel output pins (uo[0:3]) to visualize results
- Push button for manual reset control

Connection Setup:

- **USB Connection:** Connect USB cable directly to development board
- **UART Interface:** Automatically available through built-in FTDI USB-UART bridge
 - ui[0]: Connected to FTDI TX (receives data from computer)
 - uio[0]: Connected to FTDI RX (sends data to computer)
- **Parallel Outputs:** uo[0:3] available for monitoring or LED indicators
- **Clock/Reset:** Handled by board's built-in clock and reset circuitry

Computer Setup:

- Install FTDI drivers if not automatically detected
- The board will appear as a COM port (Windows) or /dev/ttyUSB* (Linux/Mac)
- You may use the mmult_comm.py file at the base of this repository to test the chip.

UART Settings:

- Baud rate: 120,000
- Data bits: 8
- Parity: None
- Stop bits: 1
- Flow control: None

Usage:

1. Connect USB cable to development board
2. Open serial terminal and connect to the board's COM port
3. Send 8 bytes representing your two 2x2 matrices
4. Receive 4 bytes back representing the result matrix
5. Optionally monitor parallel outputs on uo[0:3] pins

Credits

- Special thanks to Mr. Tim Edwards for his assistance and mentorship throughout this project
- Mr. Zach Kingman for providing the opportunity to complete this project during school hours.
- Tiny Tapeout team

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	UART RX	Matrix Result C00[7:0]	UART TX
1	—	Matrix Result C01[7:0]	—
2	—	Matrix Result C10[7:0]	—
3	—	Matrix Result C11[7:0]	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

TTCAD25a SKY130 RF YOLO Test

by Sylvain Munaut

0175

HDL Project

github.com/smunaut/ttcad25a-rf-yolo-test

“Test structures for a future register file macro”

How it works

See design data for information

How to test

No test procedure has been written yet and it's not meant to be tested by the “demo board” but instead on a dedicated test platform.

External hardware

To do any meaningful timing testing you'll need some FPGA hardware to drive the various control signal in sequence with precise timings.

The exact testing platform is still TBD.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	reb_ena	rlb[14]	rla[10]
1	reb_data	rlb[15]	rla[11]
2	we_ena	rlb[16]	rla[12]
3	we_data	rlb[17]	rla[13]
4	rea_ena	rlb[18]	rla[14]
5	rea_data	rlb[19]	rla[15]
6	in_data	rlb[20]	rla[16]
7	in_clk	rlb[21]	rla[17]

Zilog Z80

by **ReJ aka Renaldas Zioma**

0194

16 MHz

HDL Project

github.com/rejunity/z80-open-silicon

"Z80 open-source silicon. Goal is to become a silicon proven, pin compatible, open-source replacement for classic Z80."

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 testbench
- 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

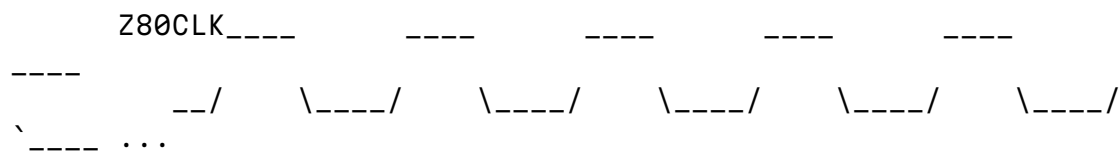
			,-----.---.-----.		
<--	A11	1		40	A10 -->
<--	A12	2		39	A9 -->
<--	A13	3	Z80 CPU	38	A8 -->
<--	A14	4		37	A7 -->
<--	A15	5		36	A6 -->
-->	CLK	6		35	A5 -->
<-->	D4	7		34	A4 -->
<-->	D3	8		33	A3 -->
<-->	D5	9		32	A2 -->
<-->	D6	10		31	A1 -->
	VCC	11		30	A0 -->
<-->	D2	12		29	GND
<-->	D7	13		28	/RFSH -->
<-->	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 -->
			\-----'		

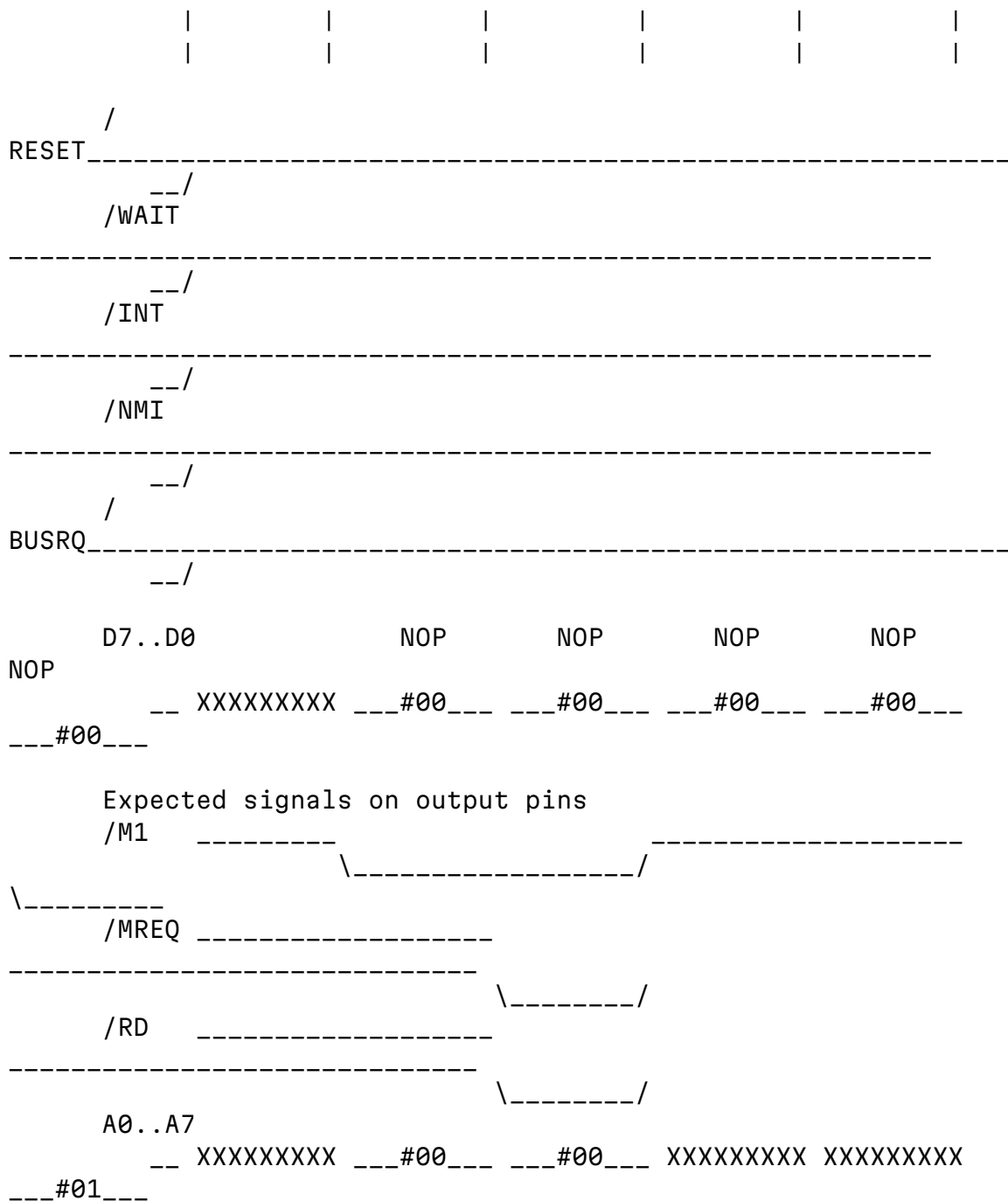
How to test

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





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.

Project Pinout

Digital Pins

#	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	CONFIG -- 00: short MREQ IORQ RD; 01: short MREQ IORQ RD starting early	/WR, A4, A12	D4
5	CONFIG -- 10: long MREQ IORQ; 11: long MREQ IORQ RD starting early	/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	/BUSAK, A7, A15	D7

"SQUARE-1": VGA/audio demo

by Zachary Catlin

0196

25.2 MHz

HDL Project

github.com/zec/tt08-square1

"On video: munching squares. On audio: the logistic map."

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:

```

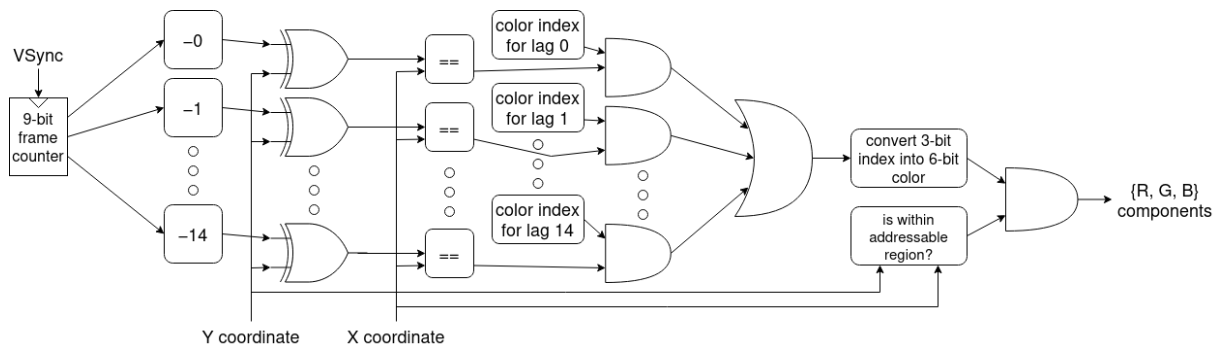
t ← 0
loop
  wait for end of frame
  t ← (t + 1) mod 2N
  for y ← 0 to 2N-1
    plot (t XOR y, y)

```

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:



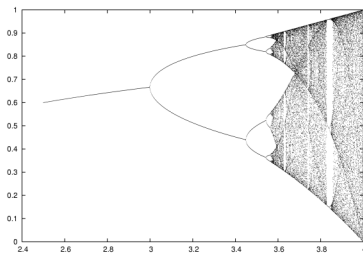
Apart from the VSync/HSync/coordinate-generating module, it’s almost entirely combinational logic. SQUARE-1 simulates 14 frames (1/4 second) of persistence prior to the current frame—not quite a Type 30, but enough to get the feel of the thing on modern displays.

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 $r \in (3, 4)$:

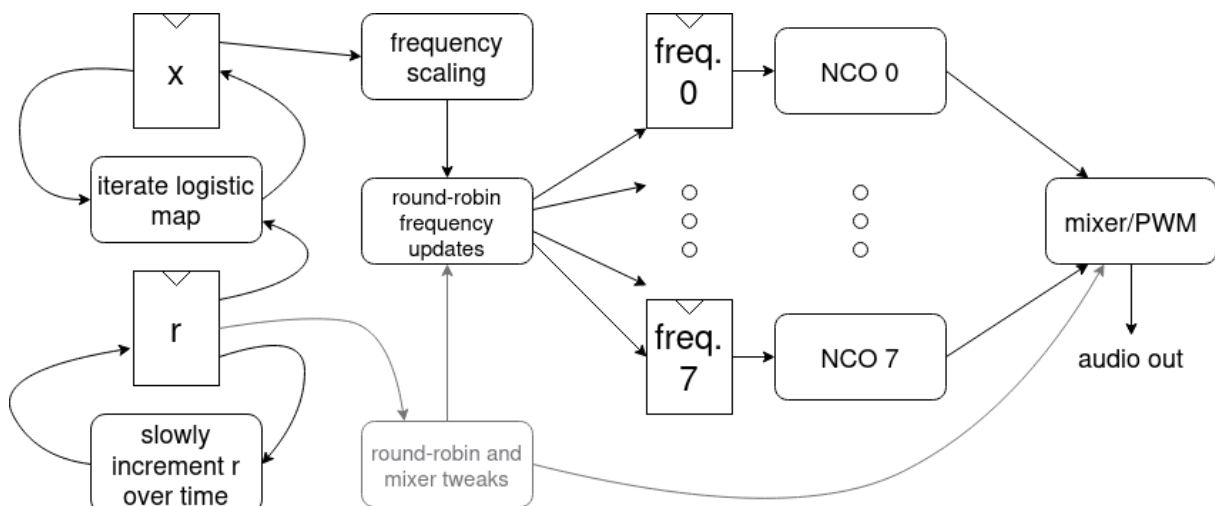


Credit: Ap on en.wikipedia.org

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.

Project Pinout

Digital Pins

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

Sprite Bouncer with Looping Background Options

by **Jacob Mack**

0198

25 MHz

HDL Project

github.com/jmack2201/tt08-jmack2201-demoscene

"Sprite bouncer hardware that supports multiple background options and sprites."

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	vga_control[0]	R1	—
1	vga_control[1]	G1	—
2	vga_control[2]	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	—

TinyMandelbrot

by **Gerrit Grutzeck**

0200

HDL Project

github.com/gfg-development/tt08-tiny-mandelbrot

"A mandelbrot generator"

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.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	serial enable	R[1] or ctr[0]	write data[0]
1	serial data	G[1] or ctr[0]	write data[1]
2	serial clock	B[1] or ctr[0]	write data[2]
3	output select	vsync or ctr[0]	write data[3]
4	frame data[0]	R[0] or new counter	reset write pointer
5	frame data[1]	G[0]	write data

#	Input	Output	Bidirectional
6	frame data[2]	B[0]	reset read pointer
7	frame data[3]	hsync	read

TT08 Pachelbel's Canon demo

by **Mike Bell**

0202

36 MHz

HDL Project

github.com/MichaelBell/tt08-canon

"Tiny Tapeout visuals with the classic Canon soundtrack"

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

Project Pinout

Digital Pins

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

Wirecube

by **Leo Moser**

0204

50.35 MHz

HDL Project

github.com/mole99/tt08-wirecube

"A demo for the Tiny Tapeout demoscene competition - see for yourself!"

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](#)

Project Pinout

Digital Pins

#	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	—

RGBW Color Processor

by Enrico Sanino

206

66 MHz

HDL Project

github.com/thexeno/tt08-rgbw-controller

"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."

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:

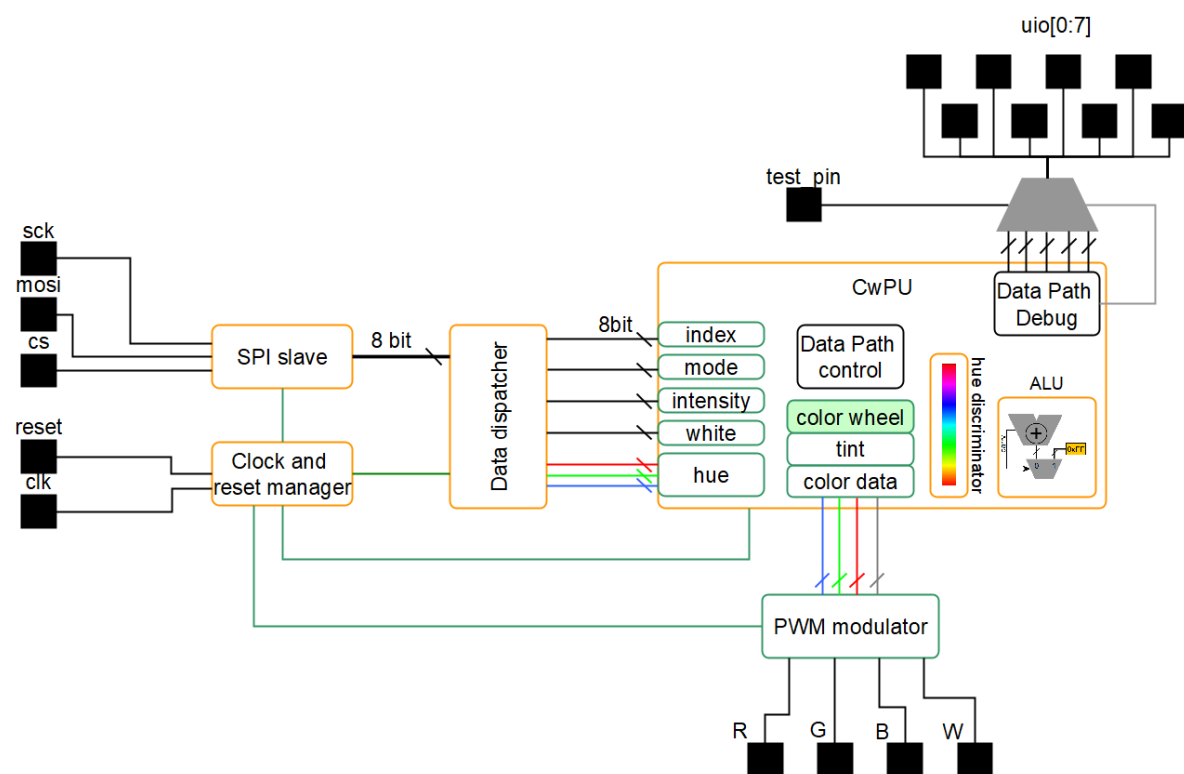


Figure 206.1: block diagram image

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 output 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 multiplier, 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:

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:

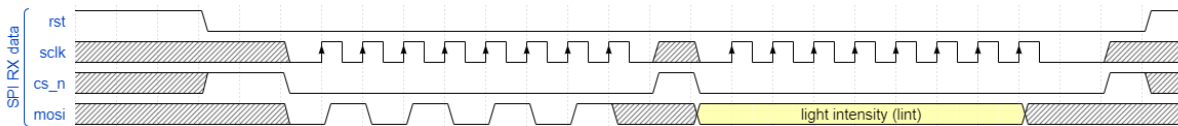


Figure 206.3: SPI transaction image, bit detail

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

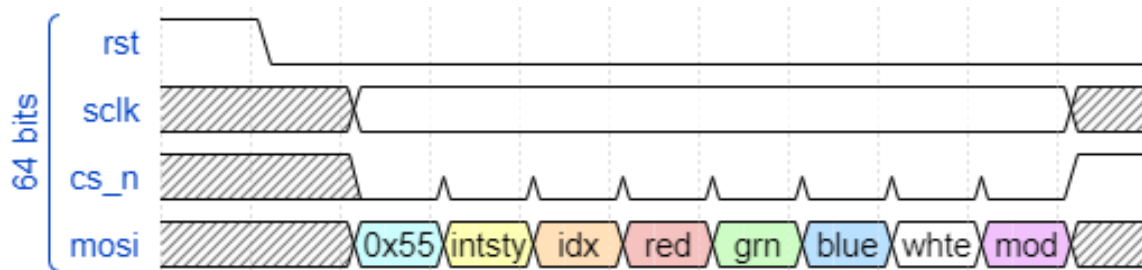


Figure 206.4: SPI transaction image, whole packet structure

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 a 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_out[1] -> Green LED

uo_out[2] -> 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 $RGBW = ((r,g,b)+w)intensity$, 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 $RGBW = spi(red, green, blue, white)$ 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](#).

Project Pinout

Digital Pins

#	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

Bandgap

by ZHU QUANHAO

0225

Analog Project

github.com/DanielZhu123/tt10-analog-bandgap

"constant 1.2V output"

How it works

input 3.3V output 1.2V

How to test

put in the oven and test it in different temperature

External hardware

no

Project Pinout

Digital Pins

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

Analog Pins

ua#	analog#	Description
0	2	0 bit

LDO BG IREF OSC

by Lukas Baumann

0227

Analog Project

github.com/devstdin/tt08-ldo-bg-ref-osc

"Analog building-blocks for Tiny Tapeout 8"

Analog Building Blocks for Tiny Tapeout 8

- [Bandgap](#)
 - [OTA](#)
- [Ref. Current Source](#)
- [LDO](#)
 - [OPAMP](#)
- [Ring Oscillator](#)

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	OSC_EN	OSC_OUT	—
1	OSC_RESET	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Analog Pins

ua#	analog#	Description
0	0	LDO_OUT



5MHz Ring Oscillator

by Alex Jaeger

0229

Analog Project

github.com/AlexJaeger/tt_um_alexjaeger_ringoscillator

"A _very_ simple 5MHz ring oscillator"

How it works

This is a single-pin analog design project. The design presents a 5MHz clock signal at the output pin ua[0].

How to test

The design will be automatically enabled when selected on the evaluation board.

External hardware

No extra hardware required.

Project Pinout

Digital Pins

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

Analog Pins

ua#	analog#	Description
0	1	Oscillator Output

R-C Oscillators

by R. Timothy Edwards

0231

Analog Project

github.com/RTimothyEdwards/tt10_rc_oscillators

“16MHz and 500kHz oscillators, three of each”

How it works

This project consists of two R-C oscillators, originally designed for the Efab-less Frigate project’s timing subsystem. Both run on a 3.3V supply and produce an output in the 1.8V (digital) domain. Both are enabled by a single “enable” input in the 1.8V (digital) domain. One runs at 500kHz, the other at 16MHz. Each oscillator has three copies, to help measure cross-coupling and frequency variation across factors like position on the wafer.

How to test

The project has size oscillators with the enable pins at “ui_in[0]” to “ui_in[5]”. Four oscillators (2 each 500kHz and 16MHz) have outputs connected to “uo_out[0]” to “uo_out[3]”. The remaining two oscillators (one each 500kHz and 16MHz) have outputs connected to analog outputs ua[0] and ua[1].

External hardware

Note that the two analog outputs will swing only to 1.8V. The outputs can be read directly by an oscilloscope or frequency counter.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	ena0	dout0	—
1	ena1	dout1	—
2	ena2	dout2	—
3	ena3	dout3	—
4	ena4	—	—
5	ena5	—	—
6	—	—	—
7	—	—	—

Analog Pins

ua#	analog#	Description
0	3	dout4
1	2	dout5

Analog Voltage Controlled Oscillator

by Georg Boecherer

0233

Analog Project

github.com/gbsha/tt08-analog-vco

“A voltage controlled oscillator with 4 differential stages.”

How it works

A voltage controlled oscillator (VCO) using 4 differential stages and a bias controller. At each stage, the outputs are low-pass filtered by nfet transistors connected as capacitors to ground.

The first analog pin `ua[0]` can be used as input to control the frequency by changing the gate voltages of nfet current sources connected to ground. The second analog pin `ua[1]` can be used to monitor the gate voltage generated by the bias controller of the pfet current sources connected to VDD. Alternatively, the second analog pin `ua[1]` can be used as input pin to overwrite the bias controller.

The 4 differential stages provide in total 8 tap signals with phase differences of $360/8$. The 8 tap signals are converted to digital signals by buffers and output at the 8 digital output pins `uo[0]` to `uo[7]`. *NB:* the 8 tap signals are ordered by wiring convenience, not by phase.

For further information, see github.com/gbsha/tt08-analog-vco for updated information.

How to test

Connect a control voltage or current to the analog input pin `ua[0]` and observe the oscillating signals at the digital output pins.

External hardware

DC power supply for the control voltage/current, oscilloscope for monitoring the outputs.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	tap 0 signal	—
1	—	tap 1 signal	—

#	Input	Output	Bidirectional
2	—	tap 2 signal	—
3	—	tap 3 signal	—
4	—	tap 4 signal	—
5	—	tap 5 signal	—
6	—	tap 6 signal	—
7	—	tap 7 signal	—

Analog Pins

ua#	analog#	Description
0	0	bias control signal
1	5	optional bias control signal

Voltage Controlled LC-Oscillator

by Noritsuna Imamura

0235

Analog Project

github.com/noritsuna/tt08-analog-Vctrl_LC_oscillator

“Voltage Controlled LC-Oscillator of 2GHz for Skywater130nm”

This project is “Voltage Controlled LC-Oscillator of 2GHz for Skywater130nm”.

Circuit

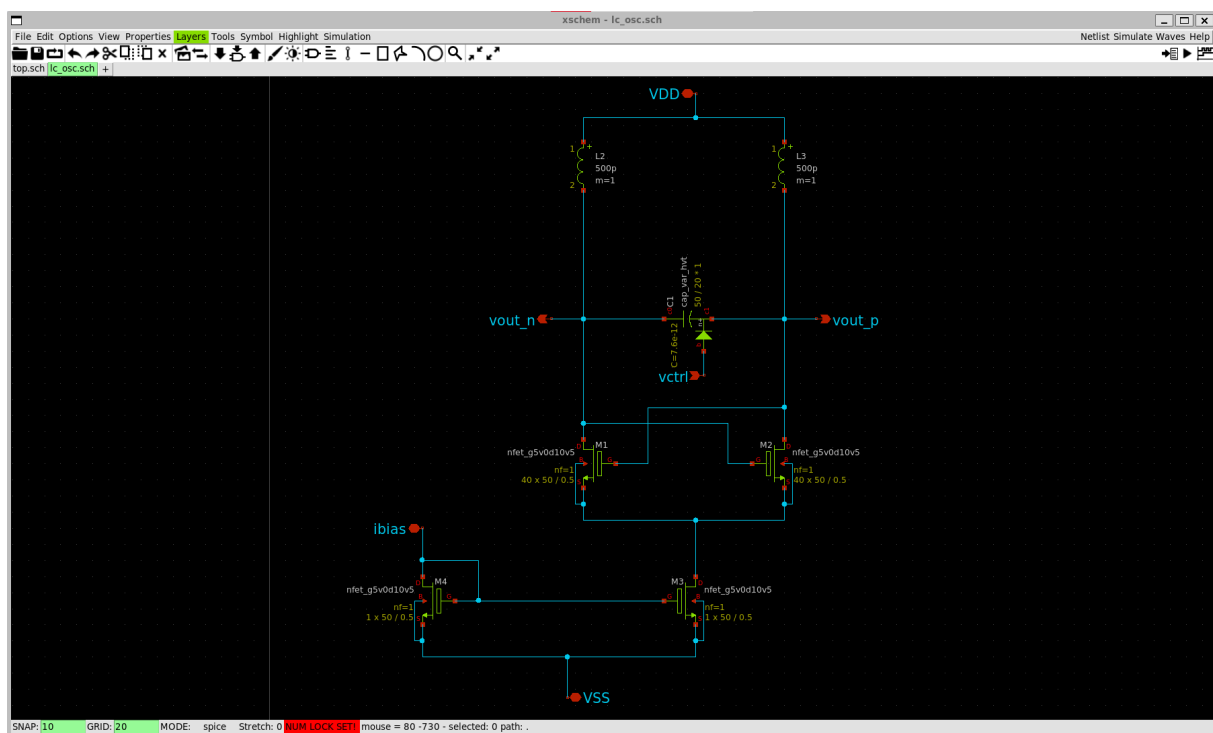


Figure 235.1: circuit

Test Bench

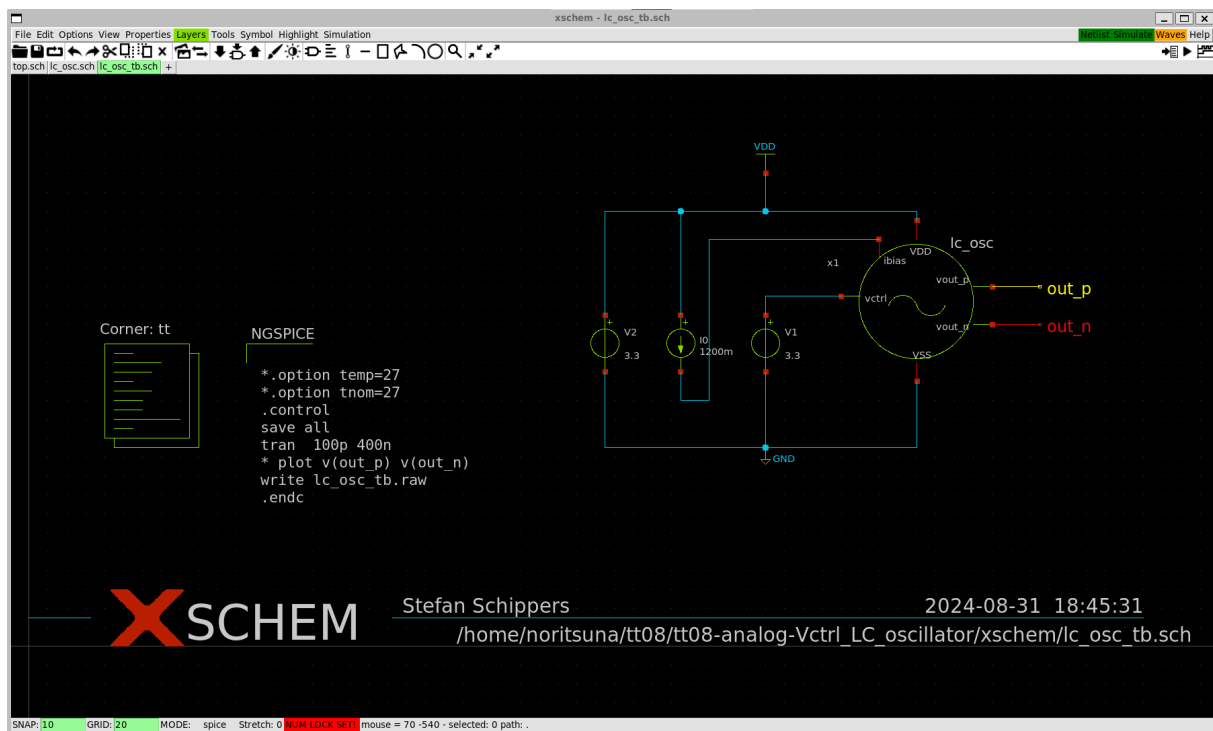
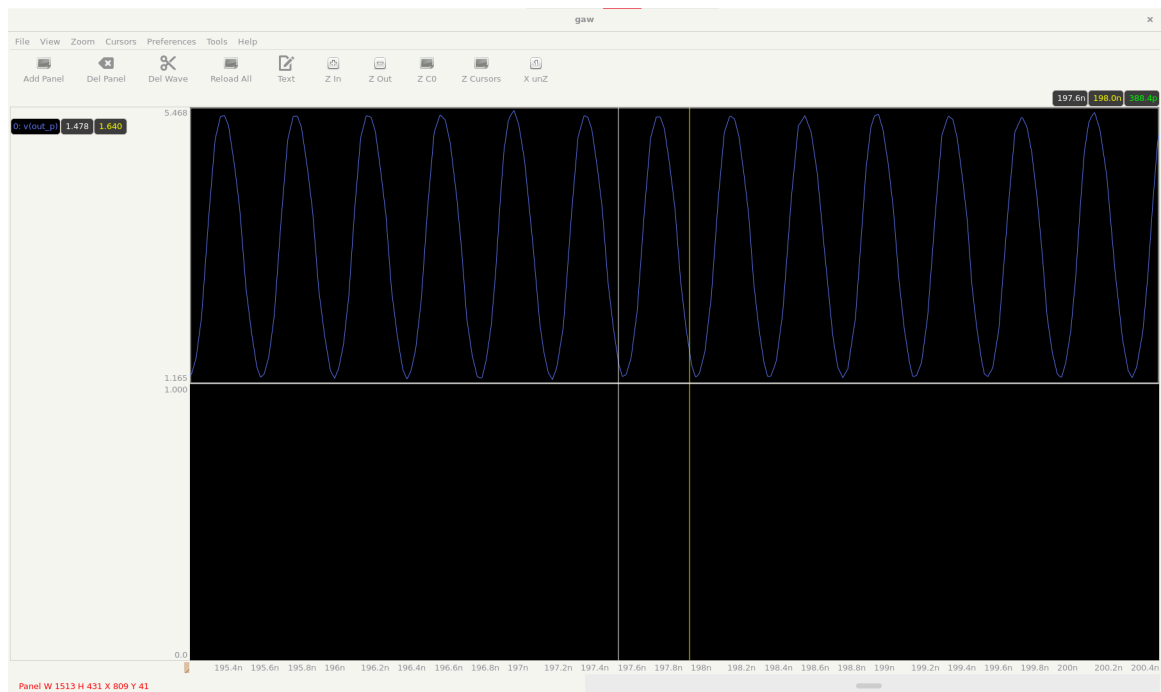


Figure 235.2: test bench

Result

- 388pSec = 2.6GHz



Layout

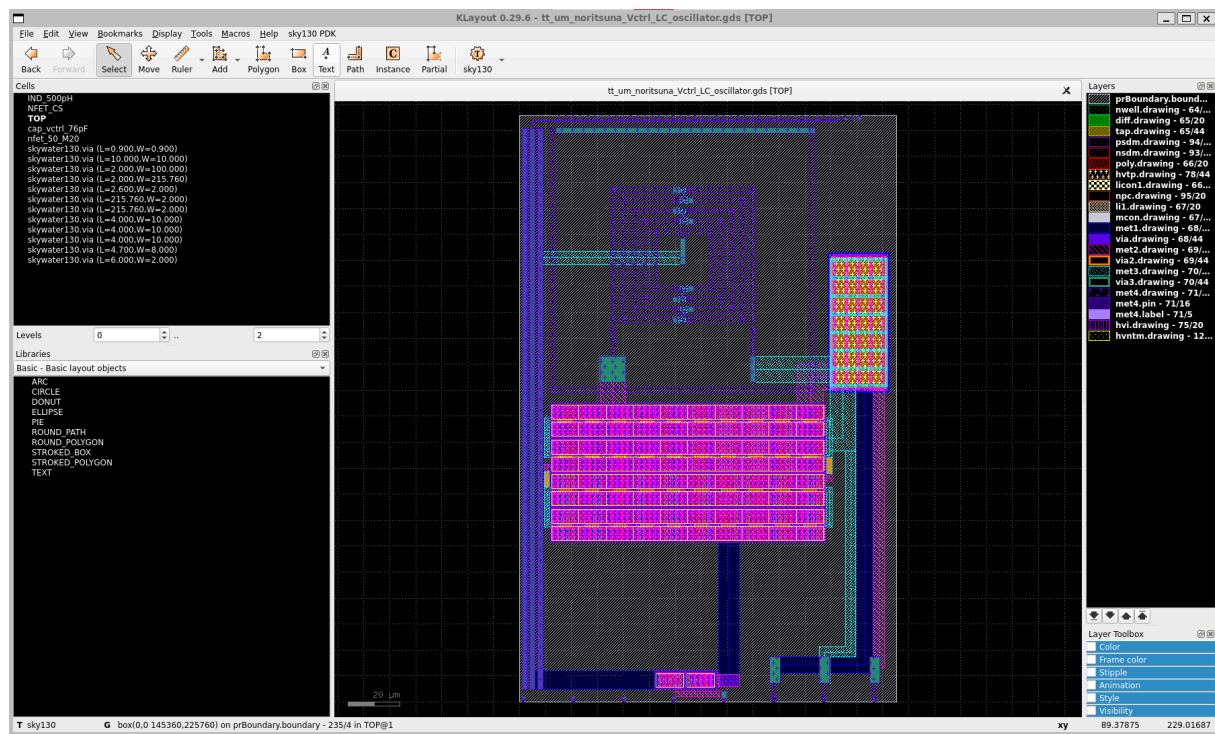


Figure 235.3: layout

Tapeout

This project made by [Tiny Tapeout](#).

Project Pinout

Digital Pins

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

Analog Pins

ua#	analog#	Description
0	4	vout_p
1	1	vout_n

ua#	analog#	Description
2	3	vctrl
3	2	ibias

Adjustable supply digital counter

by Matt Venn

0237

Analog Project

github.com/mattvenn/adjustable-psu-digital-counter

“Allow analog supply to supply a digital counter”

How it works

Danger! Self destructing design! Remove analog supply resistor before enabling or you will fry the digital design.

How to test

Connect a variable PSU to the disconnected analog supply and see what happens.

External hardware

PSU

Project Pinout

Digital Pins

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

Analog Pins

ua#	analog#	Description
0	5	n_rst
1	0	clk
2	4	count[3]
3	1	count[2]
4	3	count[1]

ua#	analog#	Description
5	2	count[0]

TT08 Differential Receiver test

by Sylvain Munaut

0239

Analog Project

github.com/smunaut/tt08-diff-rx

"Small test module to test functionality of a differential input receiver"

How it works

FIXME

How to test

FIXME

External hardware

FIXME

Project Pinout

Digital Pins

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

Analog Pins

ua#	analog#	Description
0	5	clk_n
1	0	clk_p
2	4	data_n
3	1	data_p
4	3	ibias
5	2	out

SONOS Flash Party

by Uri Shaked & Aidan Medcalf

0257

Analog Project

github.com/urish/tt10-sonos-flash-party

“Experimental SONOS Flash controller for Tiny Tapeout”

How it works

An experiment in creating a functional SONOS flash storage device for Tiny Tapeout (SkyWater 130 nm process).

We generate two internal voltages using charge pumps, and use them to program and erase the flash:

- `vprgpos` is at 7.4V, generated using a two-stage charge pump from the VAPWR (3.3V supply).
- `vprgneg` is at -4.5V, generated using a two-stage charge pump from the VAPWR (3.3V supply).

The `prog_en` and `erase_en` inputs control how we apply these voltages to the SONOS flash cell:

<code>prog_en</code>	<code>erase_en</code>	SONOS gate	SONOS body	Description
1	0	7.4V	-4.5V	Programming
0	1	-4.5V	7.4V	Erase
0	0	0V	0V	Read / No operation
1	1	7.4V	7.4V	Invalid state

In summary, we apply about 11.9V to the SONOS gate and body when programming, and -11.9V when erasing (we’ll get a more accurate value once we have a post-layout simulation).

The `read_en` connects the source of the SONOS gate to the ground, and weakly pulls the SONOS drain to the VDPWR (1.8V supply). When the SONOS is erased, the drain is pulled towards 0V. When the SONOS is programmed, the drain is pulled towards 1.8V. We buffer the drain with an inverter to get a clean digital output, which is available on the `data_out` pin. The following table summarizes the behavior of the `data_out` pin:

State	<code>read_en</code>	Drain voltage	<code>data_out</code>
Erased	1	0.51V	1
Programmed	1	1.8V	0

any	0	undefined	undefined
-----	---	-----------	-----------

Note: all of the voltages above were obtained from pre-layout simulations, and are approximate.

The charge pump voltage is divided by about 3.5 and available on the `vprogmon` analog output (should be around 2.11V, according to simulation).

How to test

Clock the design with a 2 MHz clock. Observe the generated voltage on the `vprogmon` (`ua[0]`) analog output. If you see about 2.11V, the charge pump is working correctly.

To program the flash (set the data to 0), pulse `prog_en` high for about 10 milliseconds. To erase the flash (set the data to 1), pulse `erase_en` high for about 15 milliseconds. In both cases, the charge pump needs to be clocked with a 2 MHz clock.

Read the flash by pulsing `read_en` high for about 100ns, and observe the `data_out` digital output. It is not required to clock the charge pump while reading the flash, but it is not harmful either.

Note: the pulse width and clock frequency above are just rough estimates. More accurate values will be determined in the future, once we get the silicon back.

External hardware

None

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	<code>prog_en</code>	<code>data_out</code>	—
1	<code>erase_en</code>	—	—
2	<code>read_en</code>	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Analog Pins

ua#	analog#	Description
0	7	vprogmon

mulmul

by JJ Wong

0258

HDL Project

github.com/jayjaywong12/tt08

"Small 4-bit vector multiplication engine"

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.

Project Pinout

Digital Pins

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

Bias Generator

by Rod Burt

0259

Analog Project

github.com/rburt16/tt08-analog-bias-generator

"A test chip for a simple CMOS beta multiplier current source."

How it works

A test chip to evaluate mismatch of a simple CMOS beta multiplier 1uA current source. Eight identical cells are oriented in one direction and another eight cells are rotated 90 degrees for evaluation.

How to test

All current outputs (i_out) are wired together and brought out on ua[0]. Disable individual cells 0 -> 7 with ui[0] -> ui[7] and cells 8 -> 15 with uio[0] -> uio[7].

External hardware

None

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	disable 0	—	disable 8
1	disable 1	—	disable 9
2	disable 2	—	disable 10
3	disable 3	—	disable 11
4	disable 4	—	disable 12
5	disable 5	—	disable 13
6	disable 6	—	disable 14
7	disable 7	—	disable 15

Analog Pins

ua#	analog#	Description
0	11	i out

Ring Oscillators

by **Matt Venn**

0261

Analog Project

github.com/mattvenn/tt08-analog-ring-osc

"Ring Oscillators using analog output pins"

How it works

Aiming to create 2 ring oscillators at around 600MHz and 300MHz. The output will be quite attenuated due to the pad.

- Ring oscillator 1 is made of 18 inverters and a NAND gate for enable.
- Ring oscillator 2 is made of 36 inverters and a NAND gate for enable.

To get a good output current, a 2 stage inverter is used with large drive transistors.

- [Ring oscillator 1](#)
- [Ring oscillator 2](#)
- [Driver](#)

The output waveform of the 600MHz is expected to be as shown in the cyan trace (out_parax). The ring_out_parax and pre_drive_parax are internal signals. See the xschem test bench for more details.

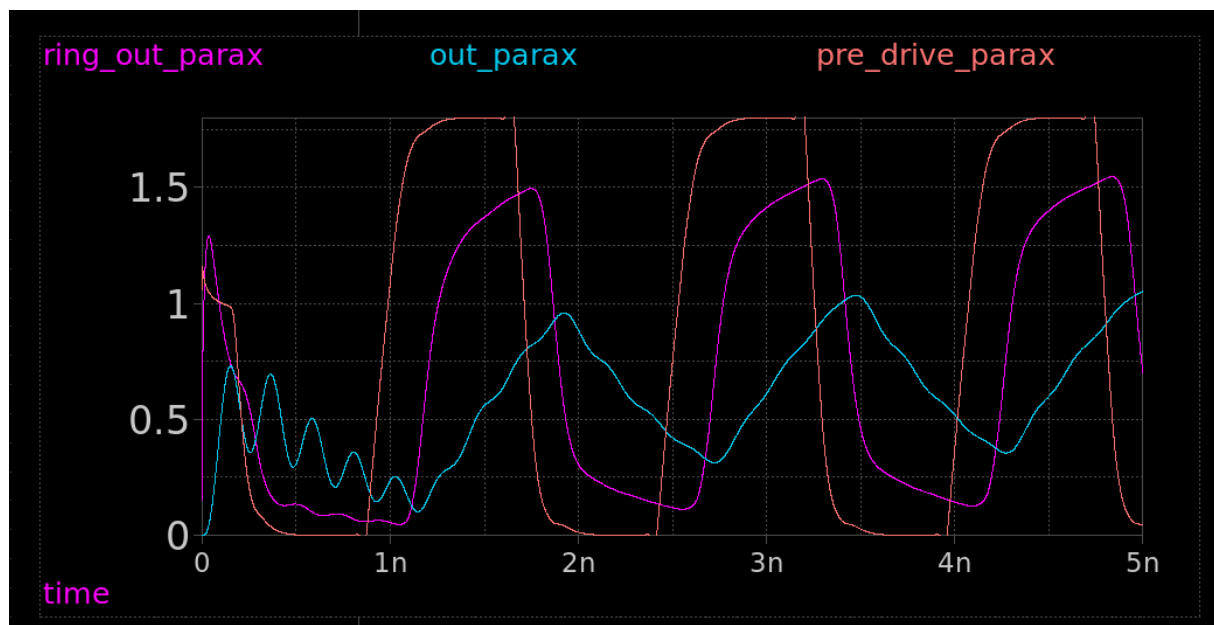


Figure 261.1: output waveform

How to test

- Enable 600 MHz oscillator 1 by setting user input pin 0 high and measure the signal at analog output 0.
- Enable 300 MHz oscillator 2 by setting user input pin 1 high and measure the signal at analog output 1.

External hardware

Oscilloscope.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Enable ring 1	ring_oscillator1	—
1	Enable ring 2	ring_oscillator2	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Analog Pins

ua#	analog#	Description
0	11	—
1	6	—

5-T OTA

by S Mishra

0263

Analog Project

github.com/Sud-ana/TT08

"A simple 5-Transistor operational amplifier with external bias current"

How it works

Apply a VDD of 1.8V and VSS ground. Then apply a 1uA current to BIAS pin. Then apply 0.9V to MINUS pin, while applying a voltage of 0-1.8V on PLUS pin in 10mV steps. Observe the DIFFOUT pin and check that DIFFOUT shoots up to 1.8V when PLUS is greater than MINUS by 10mV. ## How to test

Observe the DIFFOUT pin and check that DIFFOUT shoots up to 1.8V when PLUS is greater than MINUS by 10mV.

External hardware

Power Supply, Sourcemeter for currents, Multimeter to measure voltage.

Project Pinout

Digital Pins

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

Analog Pins

ua#	analog#	Description
0	11	DIFFOUT
1	7	BIAS
2	9	MINUS
3	8	PLUS

LaRVa CPU

by J. Arias

0264

24 MHz

HDL Project

github.com/jesari-git/tt10_larva-jarias

"RISC-V CPU design"

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 boot-loader 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 boot-loader 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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	tck	xbh	xd[0]
1	tms	xlal	xd[1]
2	tdi	xlal	xd[2]
3	rx	pwmout_tdo	xd[3]
4	gpi[0]	txd	xd[4]
5	gpi[1]	xhh	xd[5]

#	Input	Output	Bidirectional
6	gpi[2]	xoeb	xd[6]
7	gpi[3]	xweb	xd[7]

Oscillating Bones

by Uri Shaked

0265

Analog Project

github.com/urish/tt10-oscillating-bones

"A stylish ring oscillator built from SkullFET transistors"

How it works

A simple yet stylish ring oscillator that uses a chain of 21 SkullFET inverters to generate a square wave output. Based on simulation, the oscillator should have a frequency of around 169.2 MHz.

Pin	Expected frequency
osc_out	169.2 MHz
osc_div_2	84.6 MHz
osc_div_4	42.3 MHz
osc_div_8	21.1 MHz
osc_out_3v3	169.2 MHz

How to test

Connect an oscilloscope to one of `osc_out_3v3` or `osc_div_8` (uo_out pin 3) pin and enjoy the show. The remaining output pins, `osc_out`, `osc_div_2`, and `osc_div_4`, are likely to be changing too fast for GPIO pins to handle (the GPIO pins are limited to around 33 MHz), so you will probably not be able to observe them directly.

Simulation results

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

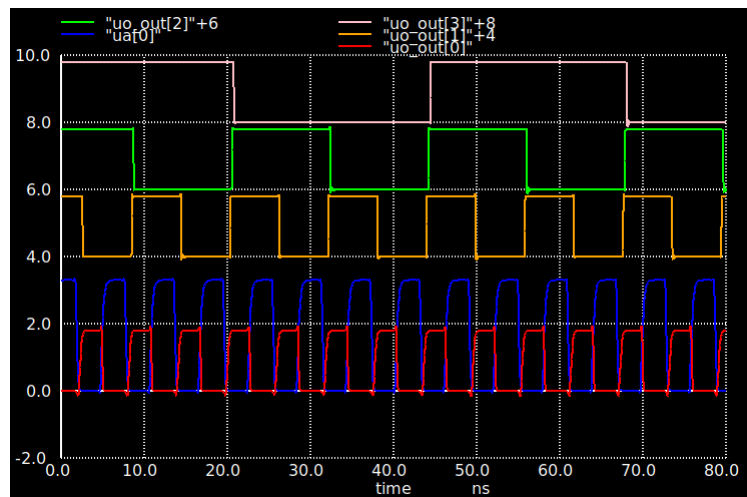


Figure 265.1: Simulation results

The outputs are shifted by 2 volts to make them easier to see in the graph. “uo_out[0]” is the main output of the oscillator, “ua[0]” is the 3v3 analog output, and “uo_out[1]”/“uo_out[2]”/“uo_out[3]” are the divided outputs.

Please note that the simulation results do not account for all parasitics, only the primary ones. Consequently, the actual frequency of the oscillator is likely to be lower than the simulated value.

Project Pinout

Digital Pins

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

Analog Pins

ua#	analog#	Description
0	8	osc_out_3v3

2-stage Opamp Designs

by Vipul Sharma

0267

Analog Project

github.com/engrvip123/tt08-ssp-opamp

"This project contain opamp circuits designed by participants under SSP training program"

How it works

This project contains in total 3 circuits: 2 nos. of 2-stage opamp and 1 POR (Power on reset) circuit. These circuits are designed by participants of analog IC design training conducted by SSP (Saudi Semiconductor Program).

2-stage Opamp

A 2-stage Miller operational amplifier (op-amp) circuit comprises two amplification stages, with a Miller capacitor connected between the output of the first stage and the input of the second stage. This arrangement improves stability and bandwidth, making it ideal for high-gain, high-frequency applications. The 2-stage Miller op-amp is frequently utilized in precision analog signal processing, active filters, and high-impedance buffer circuits.

First Stage: An NMOS differential input pair, selected for its high transconductance and speed, amplifies the difference between the input signals.

Second Stage: A common-source amplifier further increases the gain.

Miller Compensation involves using a capacitor between the output of the first stage and the input of the second stage to stabilize the op-amp by introducing a dominant low-frequency pole, which ensures stability in feedback systems.

In the op-amp design submissions, **Amr Abdelrahman** and **Majid Sami** each presented their designs. The details of their submissions are as follows.

(I) Design by Amr Abdelrahman

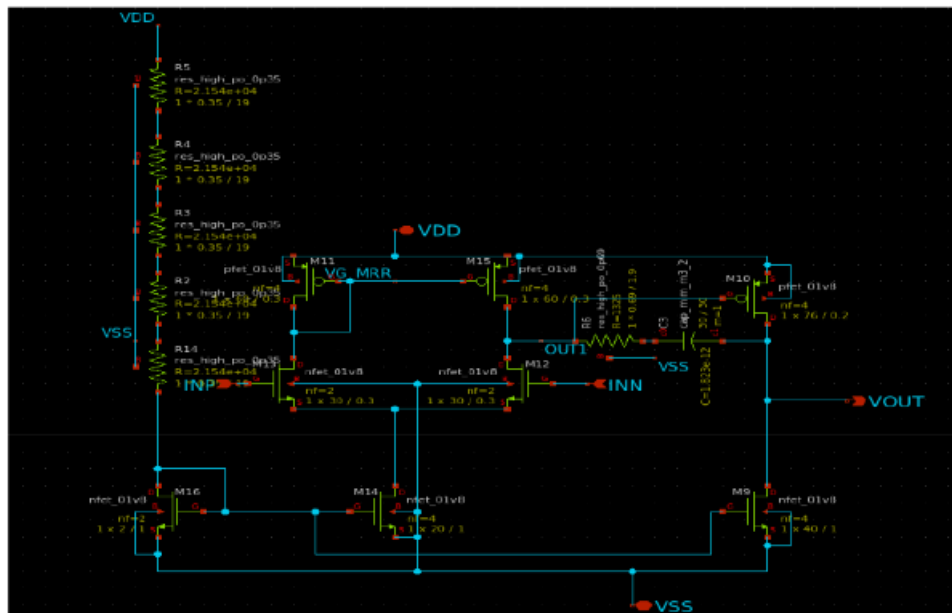


Figure 267.1: Opamp Schematic

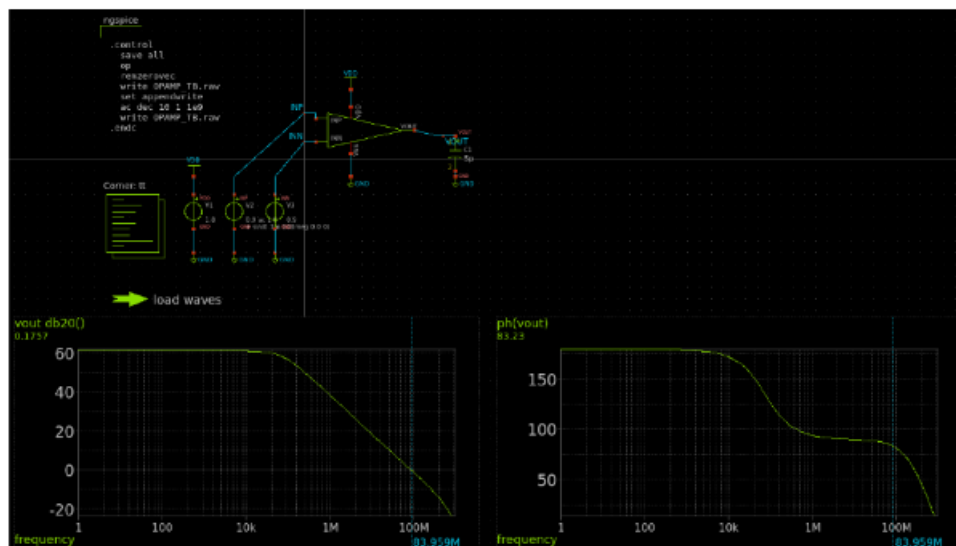


Figure 267.2: Opamp Testbench

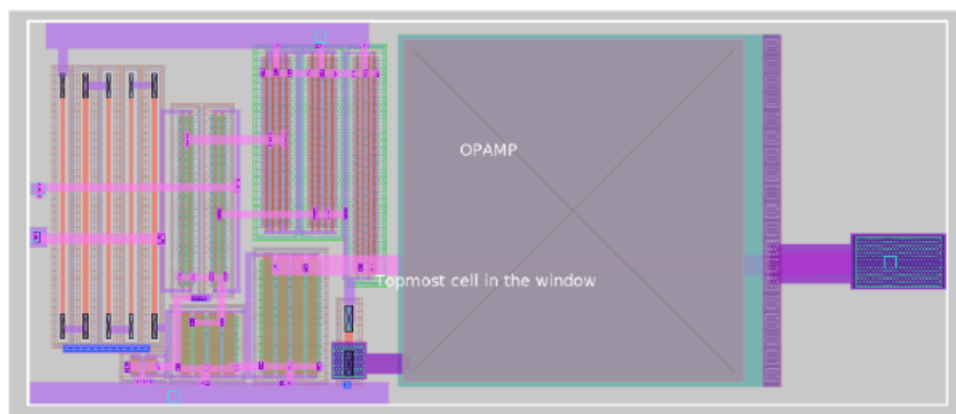


Figure 267.3: Opamp Layout

Specification	Target Value	Achieved Value	Unit
Power Supply (V_{DD})	1.8	1.8	V
Negative Supply (V_{SS})	0	0	V
Load Capacitance	5	5	pF
Gain (A_v)	60	61.2	dB
Unity Gain Bandwidth (UGB)	5	82.5	MHz
Slew Rate	10	42	V/ μ s
Output Range	0.3-1.5	0.21-1.55	V
Input Range	0.8-1.5	0.9-1.5	V
Power Dissipation	5	0.655	mW
Phase Margin (PM)	-	80	degrees

Figure 267.4: Opamp Simulation Specifications

(2) Design by Majid Sami

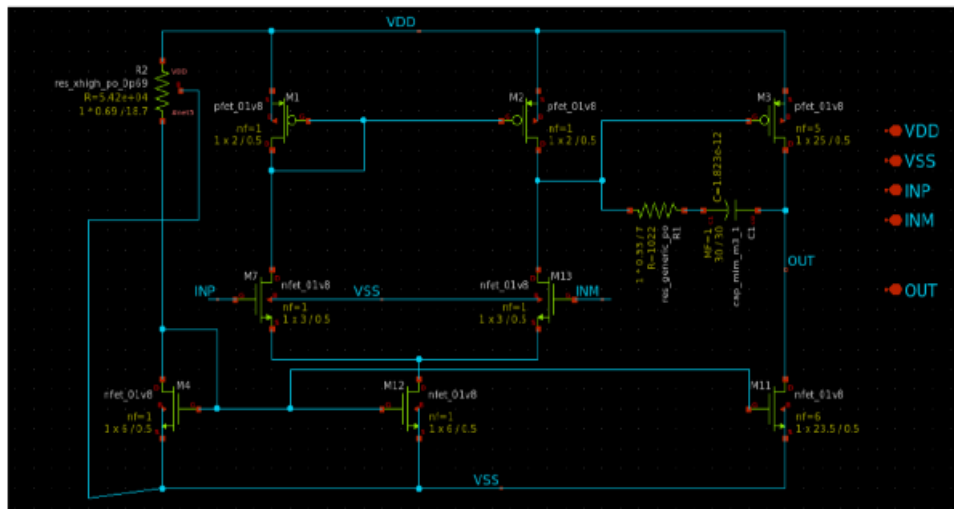


Figure 267.5: Opamp Schematic

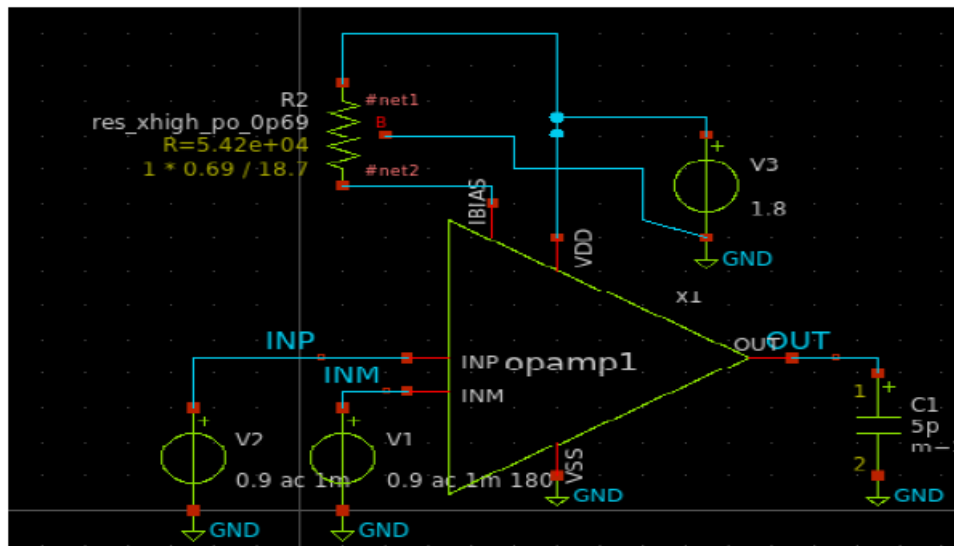


Figure 267.6: Opamp Testbench

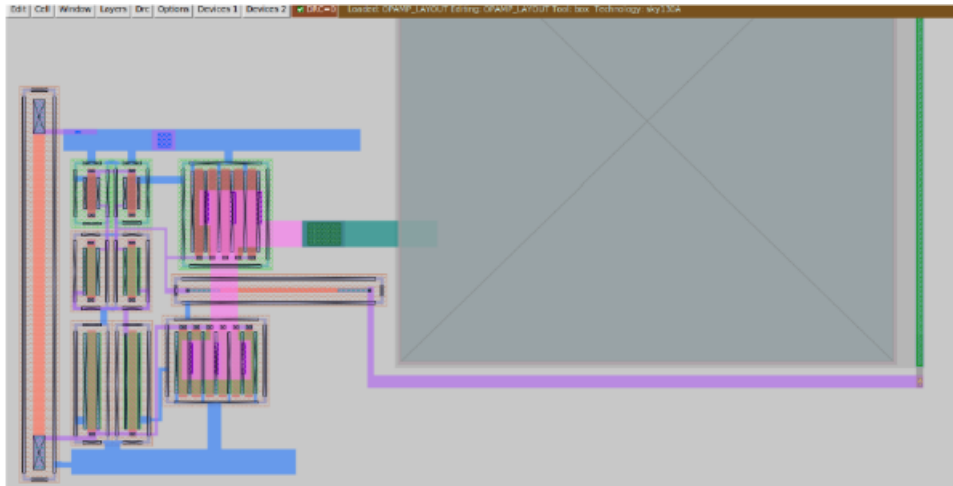


Figure 267.7: Opamp Layout

Spics	Result	Comment
VDD	1.8 v	Achieved
VSS	0 v	Achieved
Load Capacitor	5 ZpF	Achieved
Gian	71.9 dB	Achieved
UGB	16 MHz	Achieved
Slew Rate	10.6761	MATLAB Achieved
Output Range	0.25-1.6	Achieved
Input Range	0.2-1.6	Achieved
Power Dissipation	0.21632 mW	Achieved

Figure 267.8: Opamp Simulation Specifications

POR (Power On Reset) Circuit

Power-on reset (POR) circuit ensures that electronic systems start up in a known, stable state by generating a reset signal when power is initially applied. This circuit detects when the power supply reaches a sufficient voltage level and holds the reset line active until the voltage stabilizes, preventing erratic behavior and data corruption. POR circuits are crucial in microcontrollers, consumer electronics, and industrial systems, as they guarantee reliable initialization and consistent performance, thereby enhancing system stability and functionality during power-up. The circuit designed without capacitor which yields 30% reduction of chip's area compared to the conventional designs

(3) Design by Khalid Alorayir

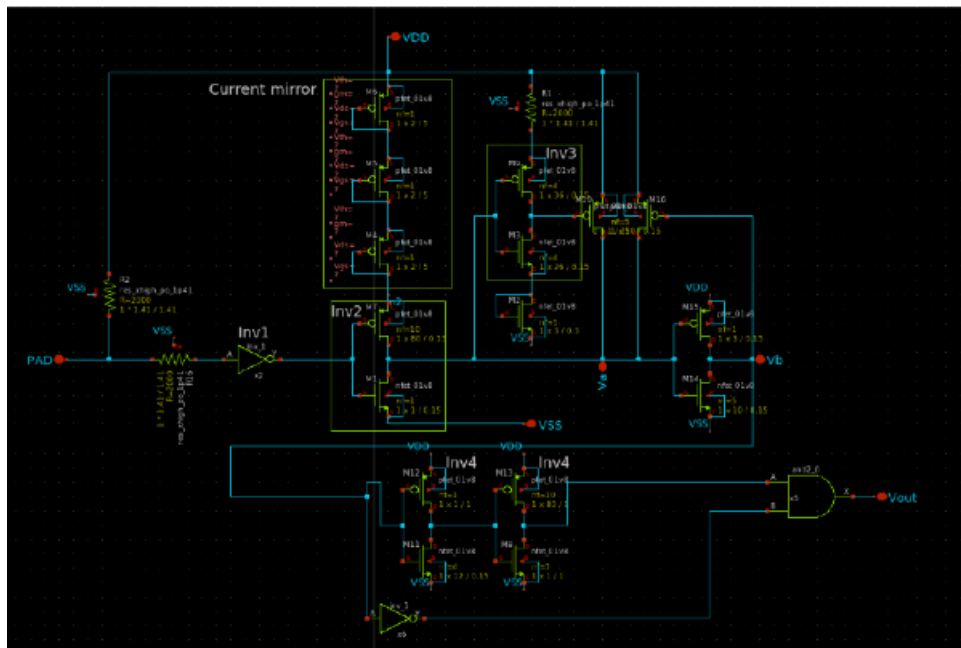


Figure 267.9: POR Schematic

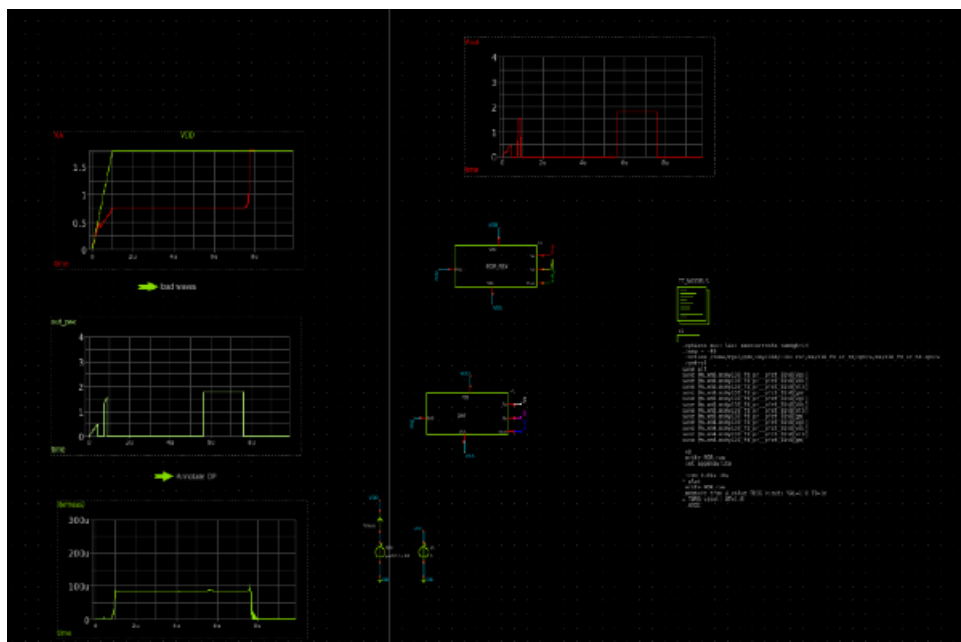


Figure 267.10: POR Testbench

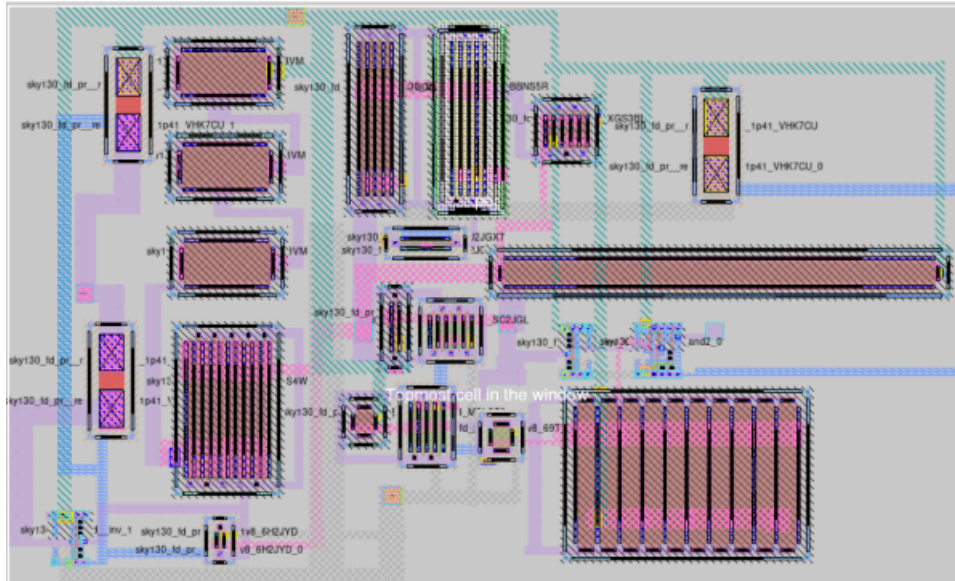


Figure 267.11: POR Layout

How to test

(I) Testing Opamp Circuit

Non-Inverting Configuration

1. **Set Up the Circuit:** Connect the op-amp with the input signal to the non-inverting input (+) and a feedback resistor network to the inverting input (-).
2. **Power the Op-Amp:** Apply the required positive and negative supply voltages.
3. **Apply Input Signal:** Feed a known input signal to the non-inverting input.
4. **Measure Output:** Use an oscilloscope or multimeter to measure the output voltage and verify if $V_{out} = V_{in} \cdot (1 + R_f/R_{in})$ matches the measured output.
5. **Check Frequency Response:** Ensure consistent gain across frequencies.
6. **Evaluate Stability and Linearity:** Look for any oscillations or instability and confirm the output accurately represents the input signal.

Buffer (Voltage Follower) Configuration

1. **Set Up the Circuit:** Connect the op-amp with output to the inverting input (-) and the input signal to the non-inverting input (+).
2. **Power the Op-Amp:** Apply the necessary supply voltages.
3. **Apply Input Signal:** Feed a known input signal to the non-inverting input.

4. **Measure Output:** Use an oscilloscope or multimeter to measure the output voltage and ensure it closely follows the input with minimal offset.
5. **Check Frequency Response and Stability:** Confirm fidelity of output across different frequencies and ensure stable output without oscillations.
6. **Assess Load Driving Capability:** Test with various loads to verify effective driving.

(2) Testing POR Circuit

1. Power up the circuit such that VDD voltage reaches final 1.8V value from 0V in 1 μ s to 10 μ s seep time.
2. Check the reset signal with an oscilloscope to confirm proper activation and deactivation.
3. Verify that the voltage reaches the threshold and that the reset signal duration is sufficient, especially for rise times of 1 μ s to 10 μ s.
4. Test the POR circuit under varying power conditions and ensure correct system initialization post-reset.
5. Power down and repeat the test to ensure consistent performance.

External hardware

Digilent Analog Discovery can be used for various measurements of opamp circuits.

1. **Signal Generation:** Use the Analog Discovery's waveform generator to create test signals for the op-amp and power-on-reset circuits.
2. **Measurement:** Connect the oscilloscope probes to monitor the input and output signals of the op-amp and observe the behavior of the power-on-reset circuit.
3. **Frequency Response:** Analyze the frequency response of the op-amp by sweeping through various frequencies and recording the output using the Analog Discovery's built-in tools.
4. **Transient Analysis:** Measure how the op-amp and power-on-reset circuits respond to transient signals or sudden changes, such as power-up events.
5. **Voltage Levels:** Check the stability and correct operation of the power-on-reset circuit by measuring the voltage levels and timing of the reset pulse.

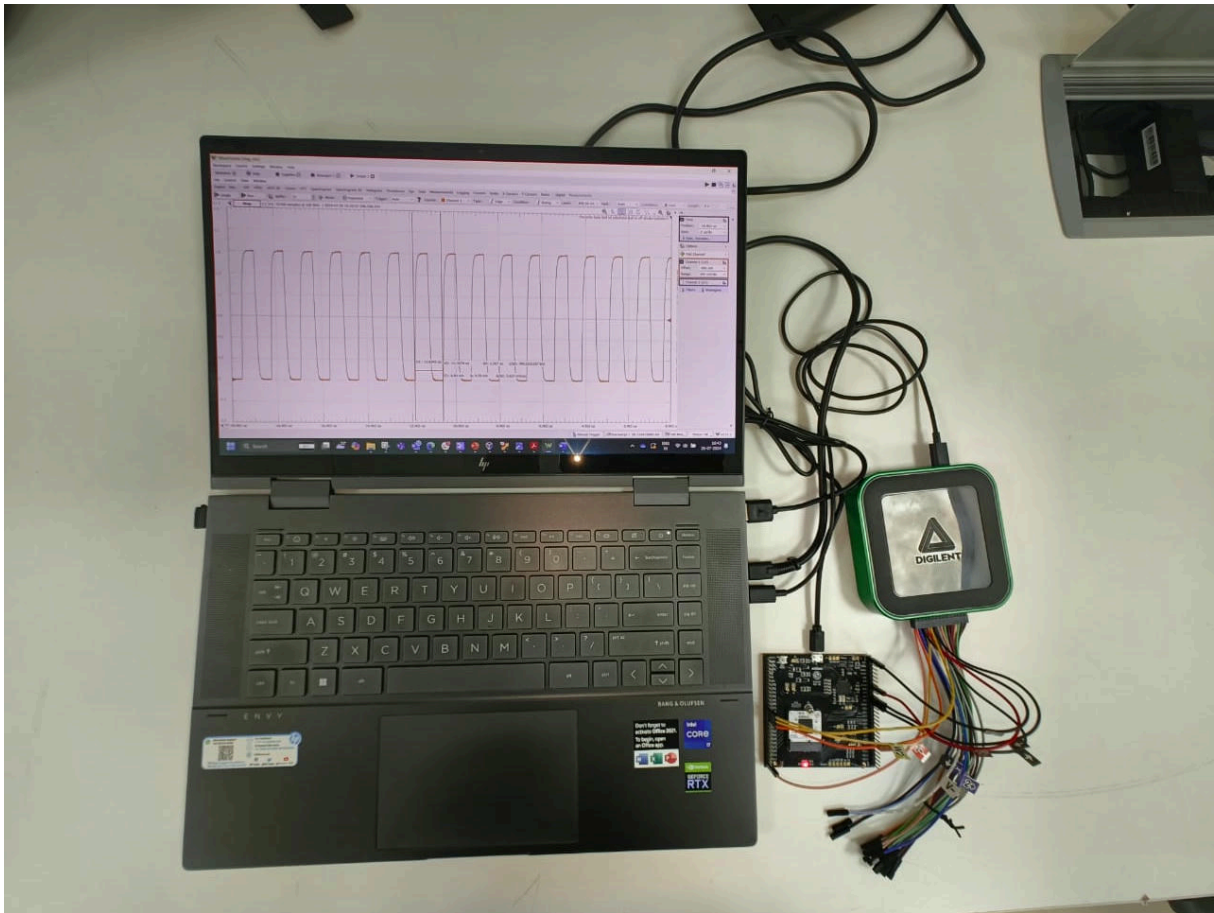


Figure 267.12: Sample Measurement Setup using Analog Discovery 3

Project Pinout

Digital Pins

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

Analog Pins

ua#	analog#	Description
0	11	Vout_POR
1	6	Va_POR
2	10	OUT_Maj

ua#	analog#	Description
3	7	OUT_Amr
4	9	INN
5	8	INP

Dickson Charge Pump

by Uri Shaked

0269

2 MHz

Analog Project

github.com/urish/tt08-charge-pump

"Pumps the input voltage up to ~3.65V"

How it works

A 3-stage dickson charge pump. The output voltage is $V_{out} = 4 \cdot (V_{PWR} - V_d) = \sim 3.6 \text{ V}$ where V_{PWR} is the digital input voltage (1.8 V), and V_d is the diode drop (0.9 V). The output voltage is divided by two and available at the `ua[0]` pin.

How to test

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

Simulation results

Post layout simulation showing the output voltage `x1.vout` and the divided output voltage on the `ua[0]` pin. The output voltage stabilizes at 3.65 V, and the divided output voltage at 1.82 V. The current draw is about 623.5 nA (measured by adding a 1k resistor between `ua[0]` and VGND in simulation).

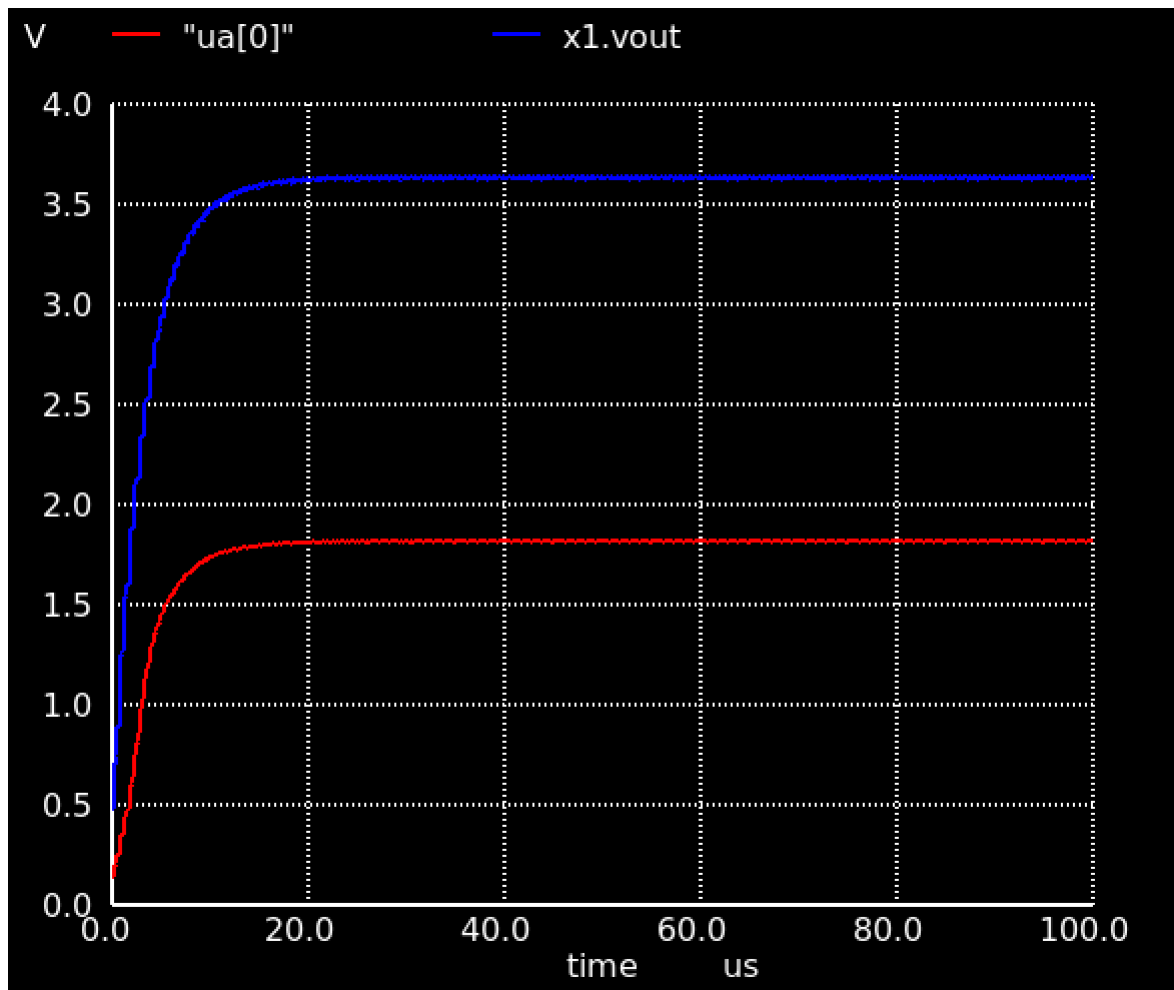


Figure 269.1: output voltage and divided voltage

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

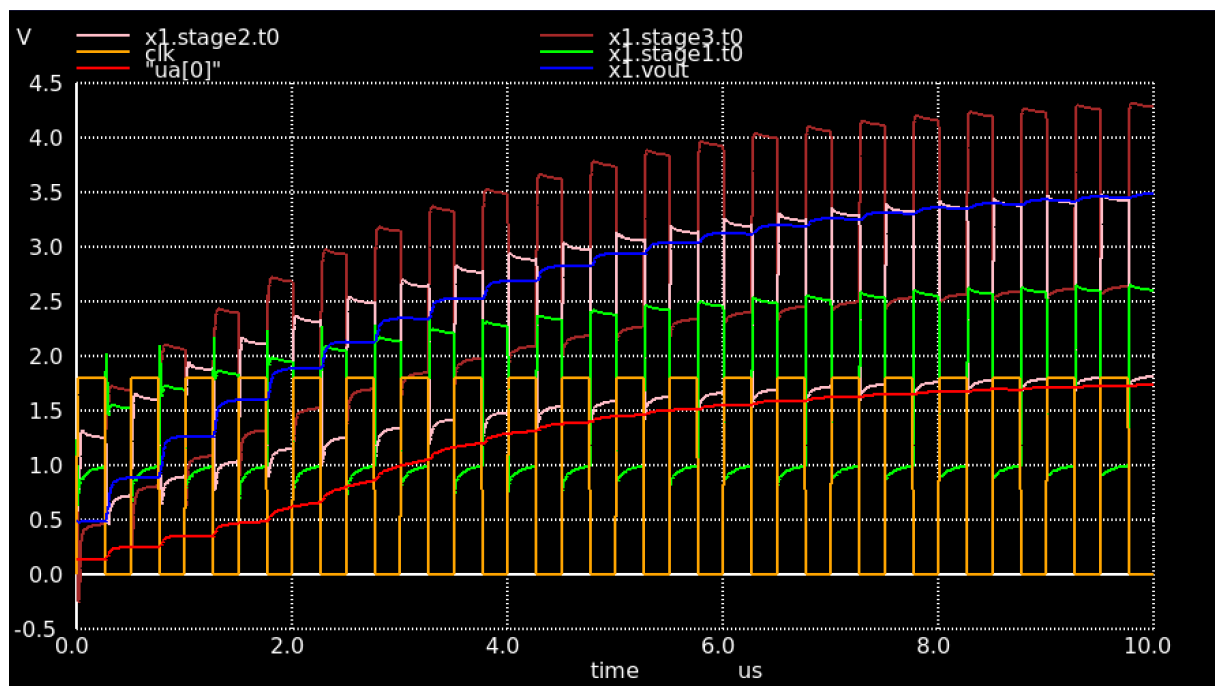


Figure 269.2: output voltage and intermediate voltages

Project layout

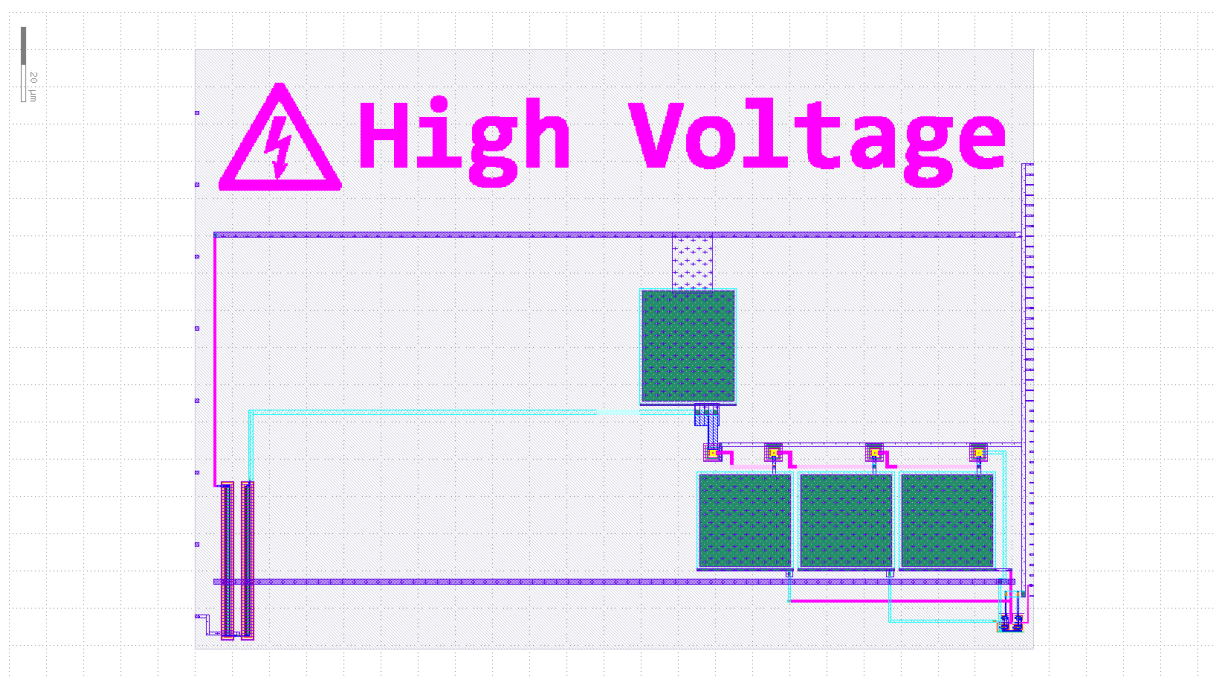


Figure 269.3: Project layout

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	—	—

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

Analog Pins

ua#	analog#	Description
0	9	vout_div

Wildcat RISC-V

by **Martin Schoeberl**

0270

50 MHz

HDL Project

github.com/schoeberl/tt10-wildcat

“Wildcat: a 3-stage RISC-V implementation”

How it works

It is a RISC-V CPU

How to test

TBD

External hardware

Flash and RAM PMOD (not really)

Project Pinout

Digital Pins

#	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

4-Bit Successive Approximation ADC

by **Sudarshana K S, Viraj Bhat B, Shreyas Gowda B N, Sudharsan S, Shylashree N & RV College of Engineering**

0271

5 MHz

Analog Project

github.com/RVCE-DrSSN/tt_um_4Bit_SAR_ADC

"This project implements a 4-Bit Successive Approximation Analog-to-Digital Converter"

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 their knowledge and training.

We extend our special gratitude to Dr. M Uttara Kumari (Dean of Research and Development), Dr. Ravish Aradhya H V (HoD, ECE), Dr. K S Geetha (Vice Principal), Dr. K N Subramanya (Principal) and Rashtreeya Sikshana Samithi Trust for their constant support and encouragement, which enabled us to design a tapeout for Tiny Tapout 10.

Description

The 4-Bit SAR ADC operates by efficiently converting Analog continuous time data to Digital bit stream, enabling seamless interfacing between Analog and Digital systems at high speeds. Here's how it works:

1. Successive Approximation Register (SAR):
 - The SAR is the core block of our ADC design, performing a binary search in each conversion cycle to accurately match the analog input V_{in} to its corresponding digital level.
 - It performs successive approximation, adjusting the DAC output iteratively based on the bit-wise output of the comparator.
2. R-2R Ladder Digital-to-Analog Converter:
 - This block converts the digital output from the SAR into its corresponding analog value, which is then fed to the comparator as a reference.
 - The DAC output must be highly accurate to minimize Differential Non-Linearity (DNL) and Integral Non-Linearity (INL) errors in the ADC.
 - The DAC needs to be driven by a very low resistance Driver circuit achieved by using Wide NMOS and PMOS pair at the Input of the DAC.

3. Complementary Comparator Pair:

- A generic comparator often has a limited common-mode range, making it unreliable when comparing signals outside this range.
- An NMOS differential pair comparator fails at lower voltages, while a PMOS differential pair comparator fails at higher voltages.
- To overcome this limitation, we have designed a novel comparator structure that dynamically switches between the two comparators, allowing accurate voltage comparisons across the full range at high speeds.

Overall, the 4-Bit SAR ADC offers an efficient and reliable method for data conversion even at high speeds, enabling seamless interfacing between the Analog and Digital Domain.

How to test

Analog Input (V_{in}):

- Connect the analog input to the `ua_0` Analog pin.
- The input should be held constant throughout the conversion cycle using an external sample-and-hold circuit.

Clock (`clk`):

- Provide a 5 MHz clock to the `clk` pin.

Reset (`rst_n`):

- The active-low reset `rst_n` must be pulsed each time the `uo_0(EOC)` pin goes high.

Conversion Timing:

- Each conversion takes 5 clock cycles, resulting in a conversion frequency of 1 MHz.

Digital Output:

- The ADC output is available at the pins at the End of conversion cycle:
 - `uo_1` → D3
 - `uo_2` → D2
 - `uo_3` → D1
 - `uo_4` → D0

External hardware

We will be using a Sample and Hold circuit to Hold the V_{in} constant throughout the conversion cycles.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	EOC	—
1	—	D[3]	—
2	—	D[2]	—
3	—	D[1]	—
4	—	D[0]	—
5	—	—	—
6	—	—	—
7	—	—	—

Analog Pins

ua#	analog#	Description
0	10	Vin

8-Bit Calculator

by **Randy Zhu**

0288

HDL Project

github.com/ezchips/tt08-my-calc

"ChipCraft Page 157 Lab ID: C-EQUALS"

How it works

8-Bit Calculator from ChipCraft Lab ID: C-EQUALS

How to test

Tested with Makerchip simulation.

External hardware

None.

Project Pinout

Digital Pins

#	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

i2c peripherals demonstrating address decoding and i2c reads

by **Steve Jenson** <stevej@gmail.com>

0289

HDL Project

github.com/stevej/tt08-stevej-i2c-peripheral

"An i2c peripheral demonstrating address decoding and i2c reads."

How it works

Env-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

Env-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.

Project Pinout

Digital Pins

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

Tiny PLL

by **LegumeEmittingDiode**

0290

10 MHz

HDL Project

github.com/LegumeEmittingDiode/tt08-tiny-pll

“4-channel fractional-N frequency synthesizer”

Overview

This project showcases `tiny_pll`, a completely self-contained fractional-N frequency synthesizer using less than 6% of the area of a 1x1 TinyTapeout tile. The design goals of this project were as follows:

1. The design should be as simple as possible to reduce the chance of failure.
2. The design should be as small as possible so it can be incorporated into future Tiny Tapeout designs with minimal area overhead.

There are 4 `tiny_pll` instances in this project. Each instance multiplies the frequency of a reference clock by a rational number A/B , where A and B can be between 1 and 15. Such a block has two main use cases:

1. Generating several internal clocks from a single off-chip oscillator (e.g., for a large digital design with multiple clock domains)
2. Generating one or more internal clocks at a higher frequency than what can be provided to the tile through the mux and GPIO pins

`tiny_pll` is designed for a 10 MHz reference input, which implies an output frequency between 67 kHz and 150 MHz. The 4 output clocks are connected to the GPIO pins `uo[3:0]`. In reality, the maximum output frequency is limited by 4 factors:

1. The speed of the Caravel I/O cells, which itself is a factor of the off-chip load capacitance
2. The routing between the TT mux and the I/O cells
3. The speed of the TT mux
4. The routing between the project tile and the TT mux The minimum output frequency is limited to roughly 1 MHz due to the minimum speed of the VCO.

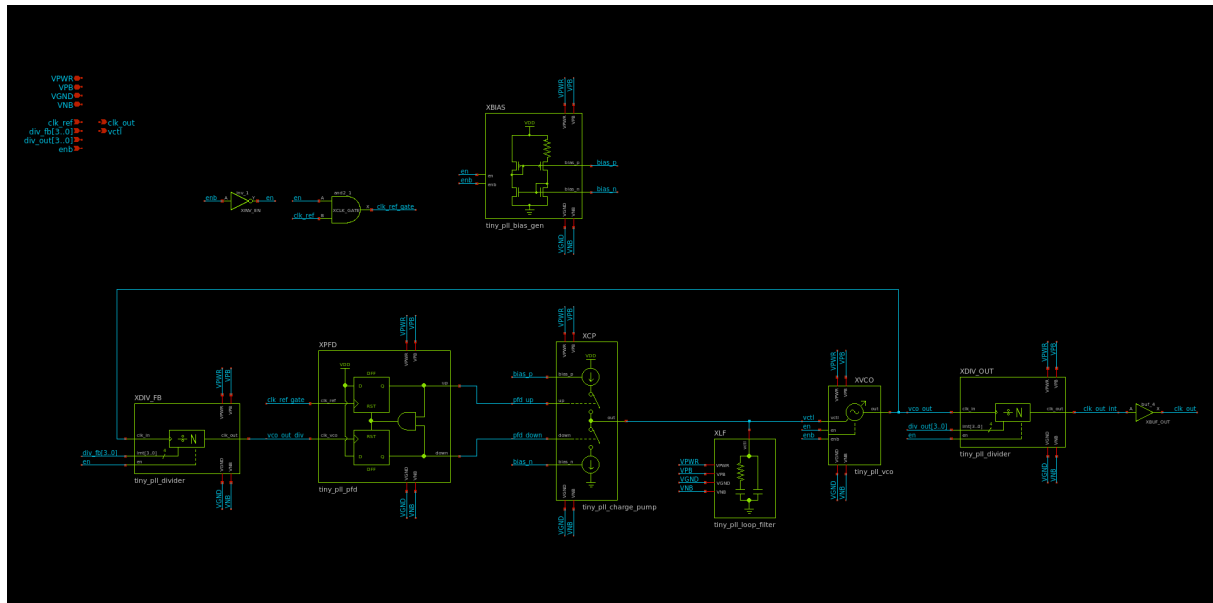
A 1-bit delta-sigma ADC is included to allow measurement of the analog control voltage on `uo[4]`.

This design is inherently mixed-signal due to the analog nature of the PLL. Consequently, the top-level layout is implemented as a custom analog/digital section for the PLL and ADC, surrounded by RTL which implements

the control/status registers (CSRs) and various clock buffering and multiplexing functions. Schematics were created using xschem and simulated with ngspice; custom layout was done using klayout with the Efabless sky130 PDK; digital synthesis and PhR was done using a custom OpenROAD flow; and magic and netgen were used for LVS, DRC and parasitic extraction.

PLL

The top-level schematic of `tiny_pll` is shown below:



The PLL uses a standard fractional-N architecture, where an input and output frequency divider are used to set the frequency multiplication with respect to the reference clock input. The output frequency is $A/B * f_{ref}$, where A is the division ratio of `XDIV_FB`, B is the division ratio of `XDIV_OUT` and f_{ref} is the input clock frequency. Documentation for the PLL subcells is included below.

Throughout the schematics, the pins `VPB` and `VNB` are included to connect the bulk terminals of all PMOS and NMOS devices, respectively. This is done to ensure the corresponding terminals of the standard cell instances at each level of hierarchy are propagated to the top level and connected to `VPWR` and `VGND`.

Divider

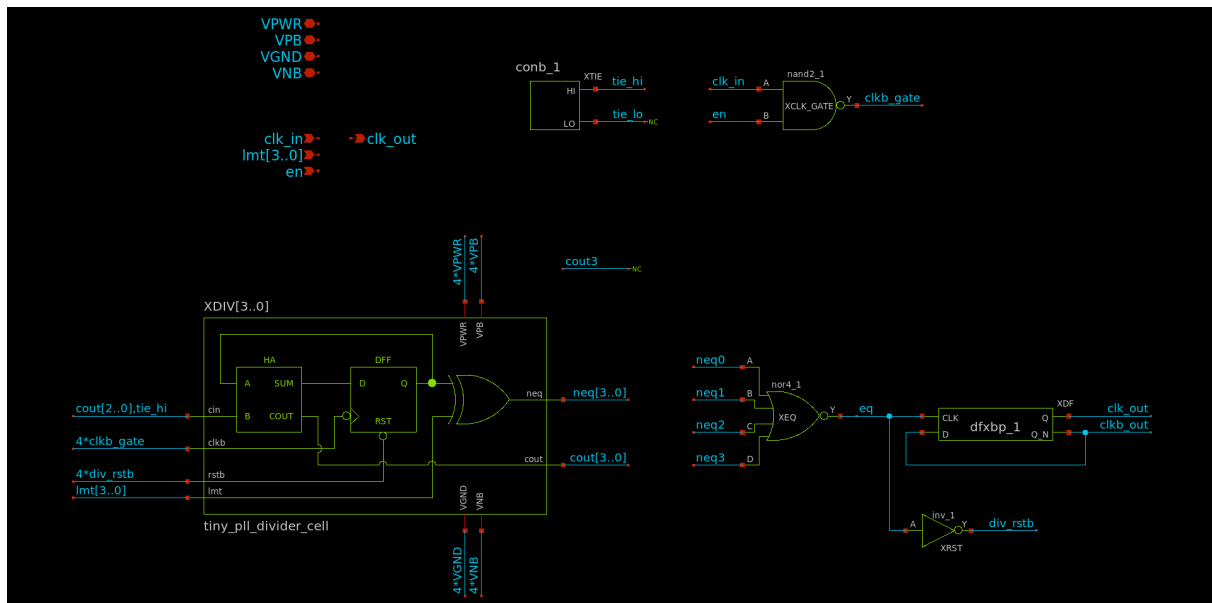


Figure 290.1: Divider schematic

Frequency dividers are implemented using a 4-bit binary counter followed by 4 XOR gates to check for equality with a division ratio input `lmt[3..0]`. When the counter output is equal to `lmt`, `div_rstb` is immediately asserted, which resets the counter to 0 at the rising edge of `clk_in`. As a result, the maximum division ratio from `clk_in` to `eq` is 15, when `lmt == 4'b1111`.

Since the counter is reset as soon as its output is equal to the division ratio, a very short pulse is produced at the `eq` node, with a duration equal to the propagation delay of the counter. This could potentially be a timing concern for XDF, but since the counter delay is at least 3 gate delays, the flip-flop was observed to operate as intended across process, voltage and temperature (PVT) in simulation.

The D flip-flop (DFF) at the output is included to ensure an output duty cycle close to 50%. As a result, the actual output frequency is $f_{ref} / (2 * lmt)$, which implies a division ratio from `clk_in` to `clk_out` between 2 and 30.

The tie cell `sky130_fd_sc_hd__conb_1` is used when gates must be connected to VPWR or VGND to avoid potential ESD issues.

Phase-frequency detector (PFD)

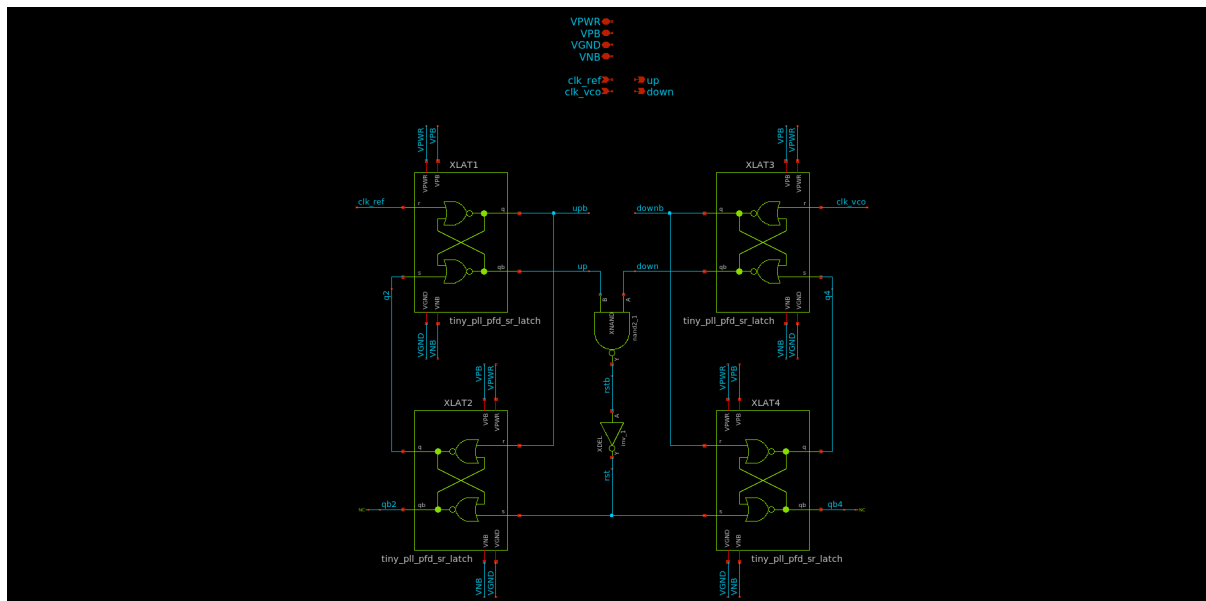


Figure 290.2: PFD schematic

The PFD is composed of two DFFs, clocked by the divided VCO output and the reference input, respectively. Since the input of both DFFs is tied to 1, each DFF can be implemented using two S-R latches, each of which uses two `nor2` gates. The full PFD thus uses 8 `nor2` gates, one `nand2` and one `inv_1`, which is considerably smaller than using discrete DFF standard cells with the D inputs tied to VPWR.

A NAND followed by an inverter is used instead of a single AND to slightly increase the minimum output pulse width and avoid charge pump glitches.

Charge pump

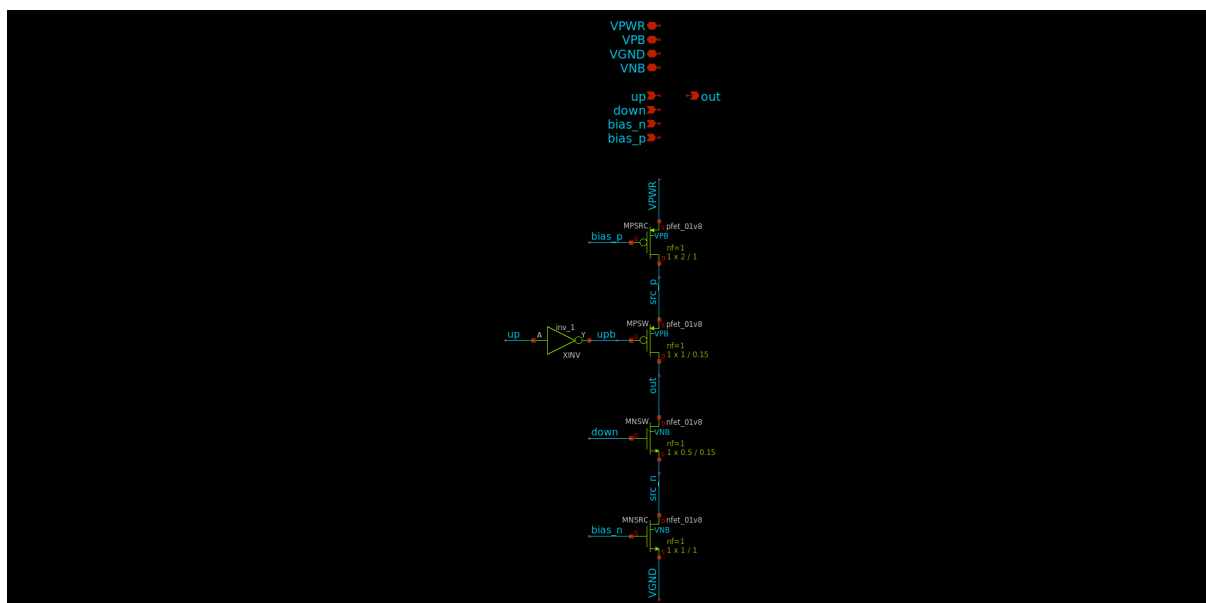


Figure 290.3: Charge pump schematic

The charge pump uses two current sources (MNSRC and MPSRC), which can be interchangeably switched to the output with the up and down inputs. The charge pump current is nominally 1 uA and is set by the bias generator. The switches use nearly minimum width to reduce area, and minimum length to reduce capacitance. The PMOS switch uses 2x the W/L of the NMOS switch to ensure roughly equal drain-source saturation voltages (VDSAT).

Loop filter

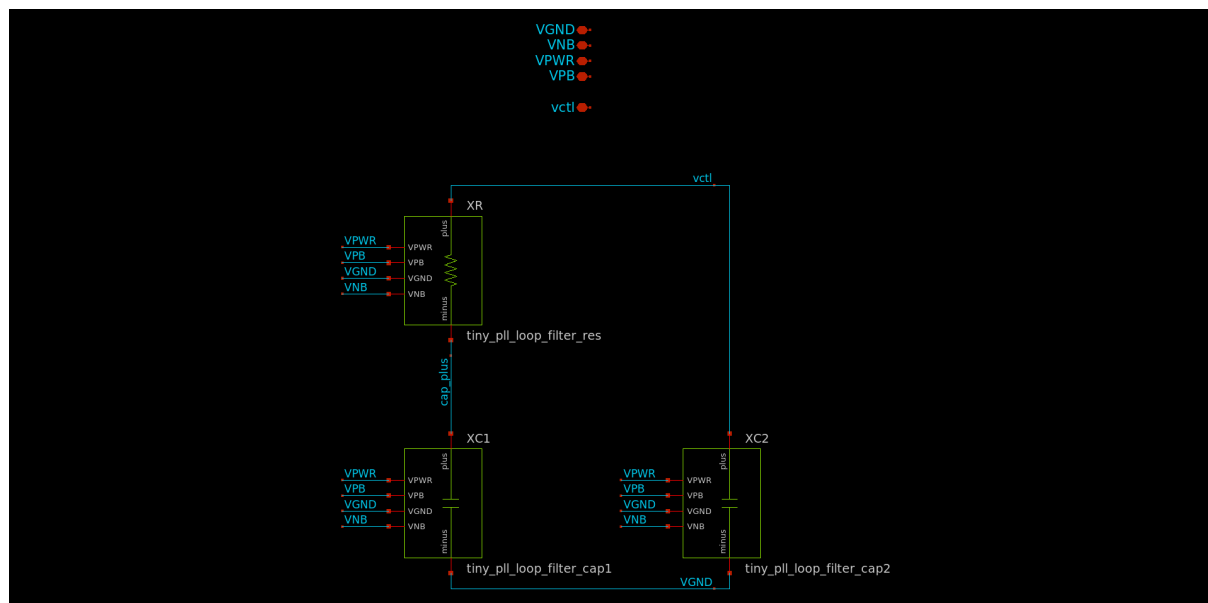


Figure 290.4: Loop filter schematic

The loop filter is implemented using a series R/C combination to compensate the loop transfer function such that a zero is placed below the crossover frequency to ensure stability, and a pole is placed above the crossover frequency to ensure fast settling time. A second capacitor XC2 is included to reduce ripple in the control voltage, which in turn reduces phase noise at the PLL output. Component values were selected using a linearized model developed using schematic-only simulations of the VCO to determine the voltage-to-frequency gain. The loop bandwidth was chosen to be on the order of 100 kHz, with a phase margin of 65 degrees at an output frequency of 10 MHz. The resulting R/C values are $R = 100 \text{ k}\Omega$ and $C1 = 1 \text{ pF}$.

In reality, the loop characteristics vary significantly across output frequency due to the nonlinear gain of the VCO, which was observed to have a nearly exponential voltage-to-frequency characteristic in simulation. This is likely due to the VCO current sources operating in the subthreshold region, where the I_D/V_{GS} characteristic is near-exponential.

The loop filter resistor is implemented using the `urpm` high-resistance poly implant, which is roughly 2 k Ω /square. While e-test values are not provided for this resistor in `sky130`, the value is not critical, and significant variations (+/-50%) were observed to result in a stable loop in simulation.

The loop filter capacitors are implemented using NMOS devices with drain and source shorted to VGND. This is due to the significantly higher capacitance density of MOS devices relative to MIM capacitors (8 vs 2 fF/ μm^2). The MOS capacitance is highly nonlinear and increases at high control voltages due to the inversion charge, but again the capacitor value is not critical and this nonlinearity does not cause instability in the feedback loop.

The loop filter consumes nearly 50% of the area of the PLL. Various methods were explored to reduce loop filter area, including:

1. MIM capacitors could be used and placed on top of the other circuit blocks to reduce area
2. A capacitance multiplier could be used to allow a smaller intrinsic capacitance

The MIM capacitor method is possible, but there is some ambiguity in the sky130 design rules as to whether a MIM capacitor can be placed over met1 and the base layers (see capm.10 in the [sky130 periphery rules](#)). Additionally, this could result in unwanted noise from the digital blocks coupling into the capacitors, which could degrade phase noise performance. Further, the capacitors would have to be divided up to lie between the power rails on met4 which would increase their area.

A capacitance multiplier was implemented using a 100 fF capacitor with a 10:1 multiplication ratio, but the final layout was the same size as the MOS capacitor implementation and was thus excluded from the final design. The capacitance multiplier was additionally seen to have poor high-frequency response compared to a MOS or MIM capacitor, which resulted in unacceptably high control voltage ripple.

Voltage-controlled oscillator (VCO)

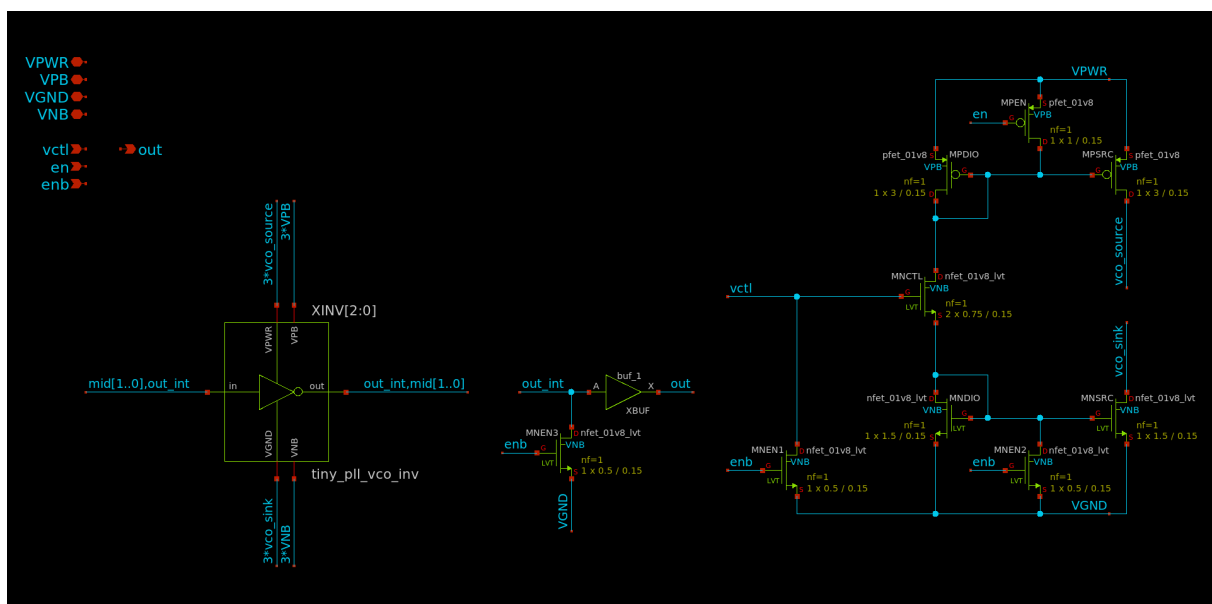


Figure 290.5: VCO schematic

The VCO is a 3-stage current-starved ring oscillator using standard cell inverters. The current sources are minimum-length to maximize W/L, which in turn minimizes VDSAT, and minimize capacitance. The output resistance of these current sources is irrelevant since it only matters that the oscillator current is limited, and not the particular limit value. A triode device MNCTL is used to control the source/sink current of the VCO. LVT NMOS devices are used to ensure the operating control voltage is somewhere near half supply at an output frequency of 10 MHz, which helps ensure the maximum output frequency can be met across process variations. Four “keeper” devices (MNEN1, MNEN2, MNEN3 and MPEN) are included to disable the circuit with zero static power consumption.

Bias generator

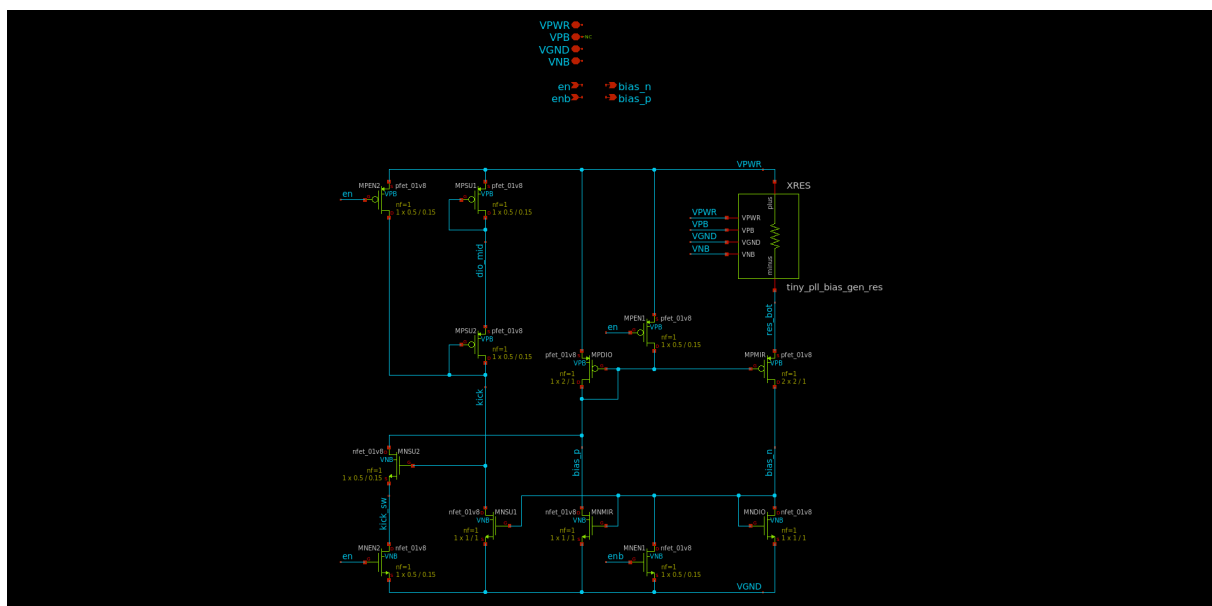


Figure 290.6: Bias generator schematic

The bias generator is a self-biased current mirror, which provides a roughly supply-independent current for the charge pump. The exact current is highly dependent on the poly resistor XRES, but is designed to be nominally 1 uA at 25 degrees C. A startup circuit is included to ensure the bias generator does not fall into an undesirable operating point where $I_{OUT} = 0$. The diode devices MPSU1 and MPSU2 charge the kick node to VPWR when the circuit is enabled, which pulls bias_p low and establishes a current in the mirror devices. Once the mirror is active, MNSU1 pulls kick low and disables the startup circuit. Multiple “keeper” devices are included to disable the circuit with zero static power consumption.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	csr_data_in[0]: Data input for PLL control registers	clk_out[0]: Channel 0 PLL clock output	clk_csr: Clock input for PLL control registers
1	csr_data_in[1]: Data input for PLL control registers	clk_out[1]: Channel 1 PLL clock output	—
2	csr_data_in[2]: Data input for PLL control registers	clk_out[2]: Channel 2 PLL clock output	—
3	csr_data_in[3]: Data input for PLL control registers	clk_out[3]: Channel 3 PLL clock output	—
4	csr_addr_in[0]: Address input for PLL control registers	adc_out: Channel 0 control voltage ADC output	—
5	csr_addr_in[1]: Address input for PLL control registers	—	—
6	csr_addr_in[2]: Address input for PLL control registers	—	—
7	csr_addr_in[3]: Address input for PLL control registers	—	—

TT08 SKY130 ROM 'YOLO' Test

by Sylvain Munaut

0291

HDL Project

github.com/smunaut/tt08-rom-test

“Quick and dirty test of some parts of a SKY130 ROM compiler”

How it works

FIXME

How to test

FIXME

External hardware

FIXME

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	wl[0]	rom_out[0]	bl0_ena
1	wl[1]	rom_out[1]	bl1_ena
2	wl[2]	rom_out[2]	bl_mux_n[0]
3	wl[3]	rom_out[3]	bl_mux_p[0]
4	wl[4]	rom_out[4]	bl_mux_n[1]
5	wl[5]	rom_out[5]	bl_mux_p[1]
6	wl[6]	rom_out[6]	pullup_n
7	wl[7]	rom_out[7]	prechg_n

nVious Graphics

by James Ross

0292

25.175 MHz

HDL Project

github.com/jar/tt08_nvious_graphics

"Submission for VGA Demoscene"

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](#)

Project Pinout

Digital Pins

#	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	—
7	Segment H	HSync	—

Rotary Encoder WS2812B Control

by **Fabio Ramirez Stern**

0293

40 MHz

HDL Project

github.com/faramire/TT08-rotary-encoder-WS2812B

"A rotary encoder controls 12 WS2812B LEDs on a ring PCB."

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	rotary encoder: CLK	DOUT	—
1	rotary encoder: DT	inverted	—
2	rotary encoder: SW	count0	—
3	green	count1	—

#	Input	Output	Bidirectional
4	red	count2	—
5	blue	count3	—
6	intensity1	—	—
7	intensity2	—	—

tt08-octal-alu

by Theo Kachelski

0295

Wokwi Project

github.com/Tkachel101/tt08-wokwi-3bit

wokwi.com/projects/407760296956596225

“Executes binary operations on two octal numbers encoded in an 8-bit instruction and outputs the result to the 7-segment display.”

How it works

This design executes binary operations on two octal numbers encoded in an 8-bit instruction and outputs the result to the 7-segment display. The instruction is made up of 8 total bits.

Bits 0 and 1 make up the operation code. Operations are according to the following table.

Bit 0	Bit 1	Operation
0	0	ADD
0	1	OR
1	0	AND
1	1	XOR

Figure 295.1: OP Bit Table

Bits 2, 3, and 4 make up operand A. Bit 2 is MSB and bit 4 is LSB.

Bits 5, 6, and 7 make up operand B. Bit 5 is MSB and bit 7 is LSB.

A full operation would be decoded like this

Bit	0	1	2	3	4	5	6	7
Decoded	OP1	OP2	A(MSB)	A	A(LSB)	B(MSB)	B	B(LSB)

Figure 295.2: OP Decoding

How to test

1. Craft a instruction following the above decoding table.

2. Enter the instruction on the TT08 PCB's input pin dip switches.
3. View the result on the 7 segment display

Note: An overflow condition during an add operation will illuminate the 7 segment display's DP.

Examples:

1. 0 0 1 0 0 0 0 1 -> ADD 4, 1 -> Displays 5 on 7-Segment Display
2. 1 1 1 0 0 1 0 1 -> XOR 4, 5 -> Displays 1 on 7-Segment Display
3. 1 0 1 0 0 1 0 1 -> AND 4, 5 -> Displays 4 on 7-Segment Display
4. 0 1 1 1 0 1 0 1 -> OR 6, 5 -> Displays 7 on 7-Segment Display
5. 0 0 1 0 1 1 1 0 -> ADD 5, 6 -> 7-Segment Display DP illuminated indicating overflow condition

External hardware

The only external hardware required is the 7 segment display provided by the tt08 PCB.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Operation bit 1	Segment A	—
1	Operation bit 2	Segment B	—
2	Operand A bit 1 (MSB)	Segment C	—
3	Operand A bit 2	Segment D	—
4	Operand A bit 3 (LSB)	Segment E	—
5	Operand B bit 1 (MSB)	Segment F	—
6	Operand B bit 2	Segment G	—
7	Operand B bit 3 (LSB)	Segment DP	—

4-bit ALU

by **Richard Xu, Louis Barbosa, Hallie Ho, Emmy Xu, Gia Bhatia & Emily Chen**

0296

HDL Project

github.com/Richard28277/4bit_alu

"The 4-bit ALU is designed to perform basic arithmetic and logical operations on 4-bit binary numbers"

Overview

The 4-Bit ALU is 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. This capstone project is from the MIT BWSI Basics of ASICs class.

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 *uio_out* wire. The *uio_oe* wire controls the enable or disable functionality for the *uio_in* and *uio_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.
- **ENC:** Encrypts the inputs *a* and *b* using an 8-bit key derived from concatenating *a* and *b*. The encryption function applies an XOR operation between this 8-bit concatenated value and a fixed 8-bit key. The result is an 8-bit encrypted value.

How to Test

To test this 4-bit ALU chip, set the values for a and b based on the 4-bit binary values for each, as well as the 4-bit operation code. The output can be up to 8-bits, using the uo pins along with the first 4 bidirectional pins. The uio pin 6 is used for the carry out, and the uio 7 is used for the overflow.

External hardware

N/A

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	a[0]	result[0]	opcode[0]
1	a[1]	result[1]	opcode[1]
2	a[2]	result[2]	opcode[2]
3	a[3]	result[3]	opcode[3]
4	b[0]	result[4]	—
5	b[1]	result[5]	—
6	b[2]	result[6]	carry_out
7	b[3]	result[7]	overflow

TSAL_TT

by **Ephren Manning**

0299

8 MHz

HDL Project

github.com/ephrenm/tt08-tsal

“FSAE EV Tractive System Active Light”

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.

Project Pinout

Digital Pins

#	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

by Ignatius Bezzam, Dhandeep Challagundla, Jarnail Sanghera & Russell Kim

0301

10 MHz

HDL Project

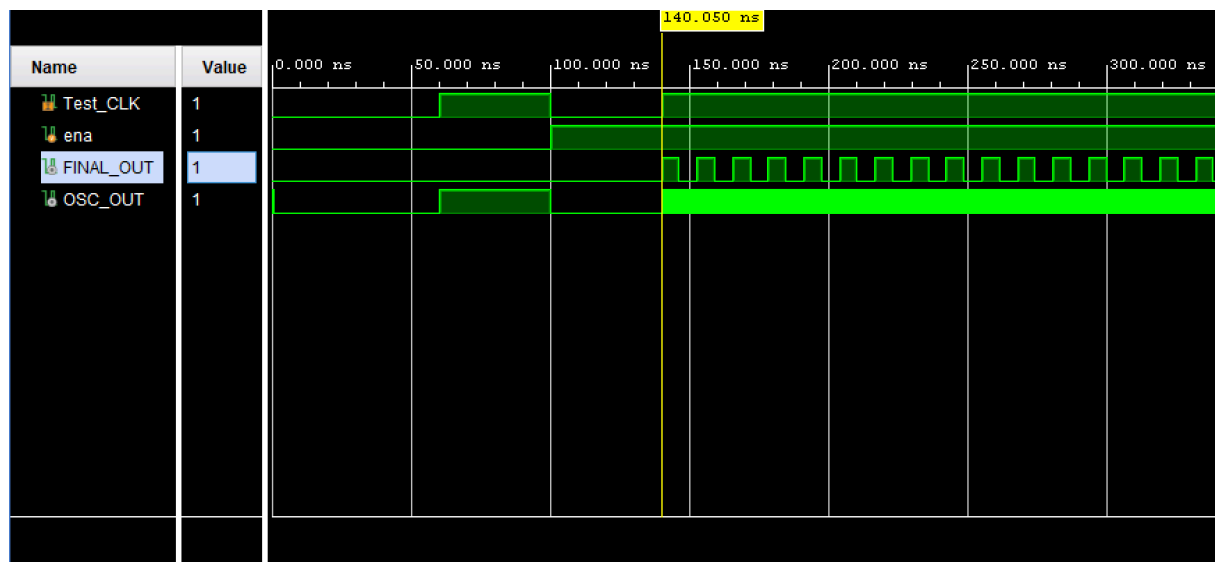
github.com/SmellyJelly01/tt08-ring-divider

"Ring Oscillator"

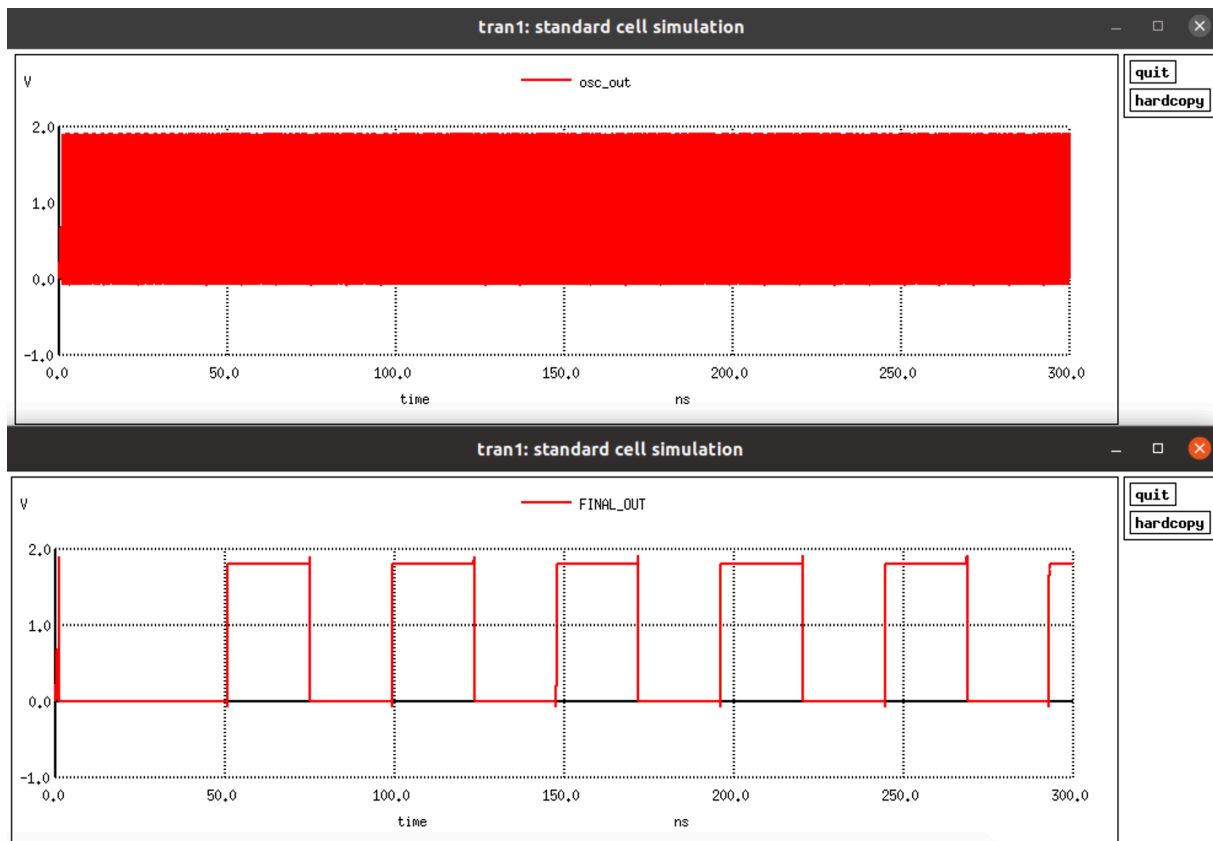
How it works

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).

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	tst_clk	final_out	n1
1	—	osc_out	n3
2	—	ena	—
3	—	clk	—
4	—	rst_n	—
5	—	n2_buf	—
6	—	n4_buf	—

#	Input	Output	Bidirectional
7	—	—	—

HACK CPU

by **Dantong LUO**, **Nour MHANNA** & **Charbel SAAD**

0303

12.5 MHz

HDL Project

github.com/bugavix/hack_cpu

"A 16-bit CPU based on the HACK architecture"

How it works

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

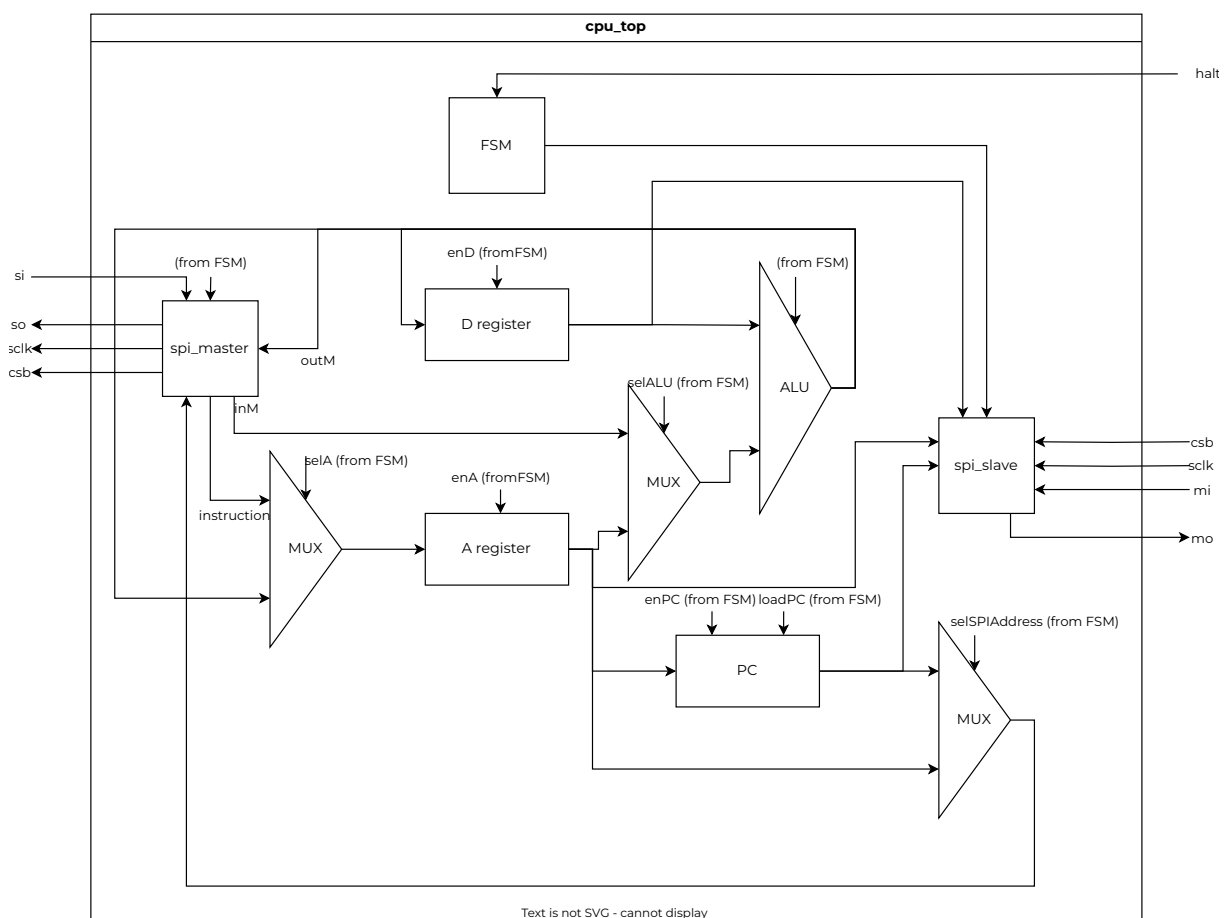


Figure 303.1: The `cpu_top` module content

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.

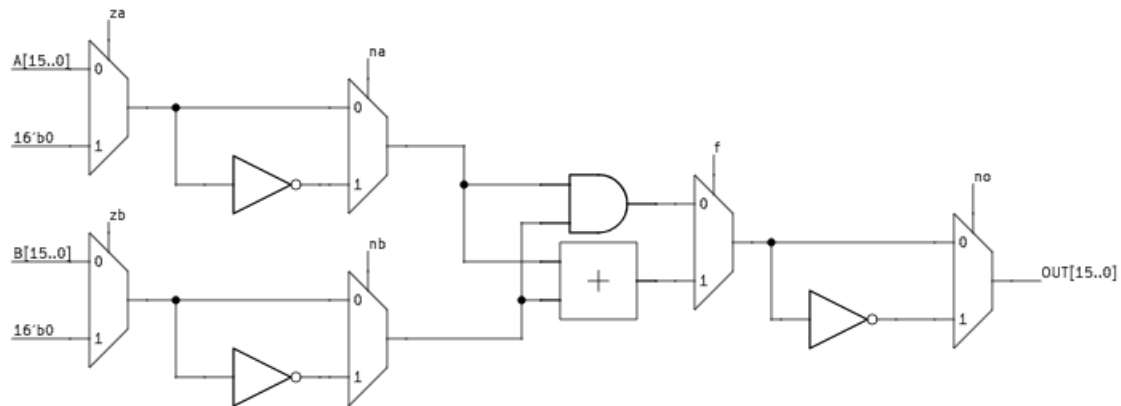


Figure 303.2: The ALU module

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.

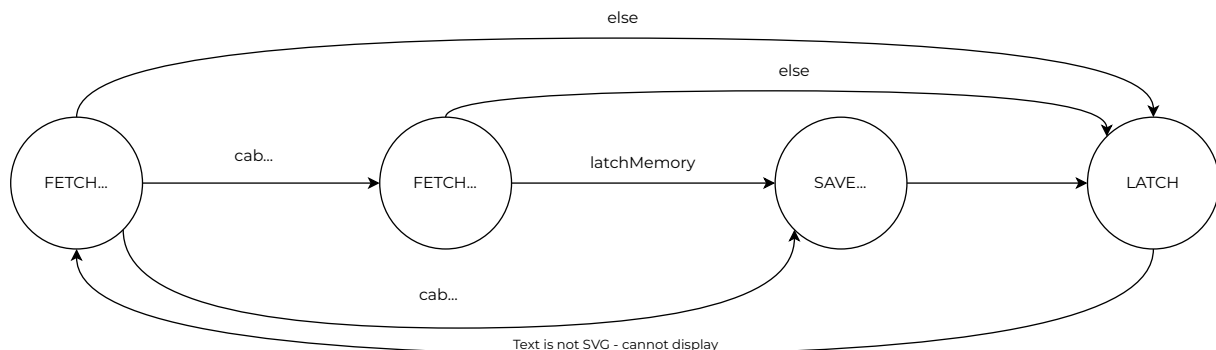


Figure 303.3: The finite state machine

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).

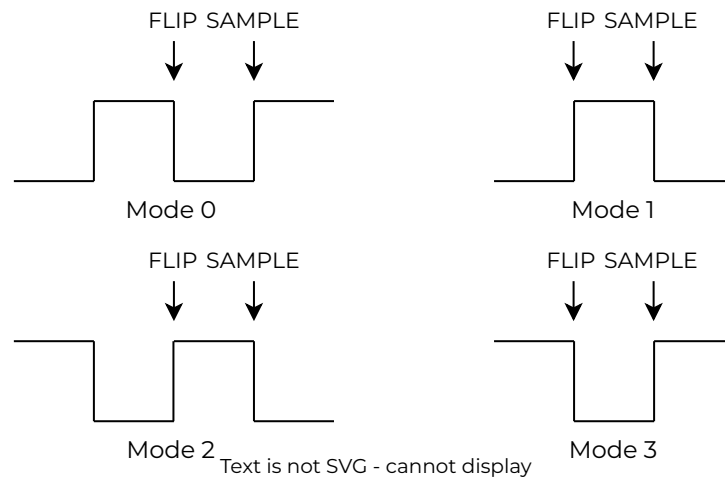
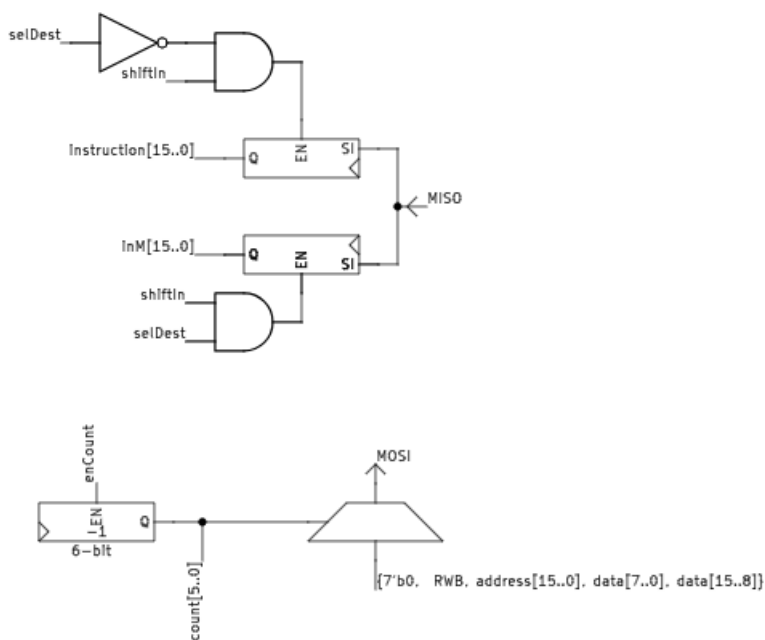


Figure 303.4: The different SPI modes

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.

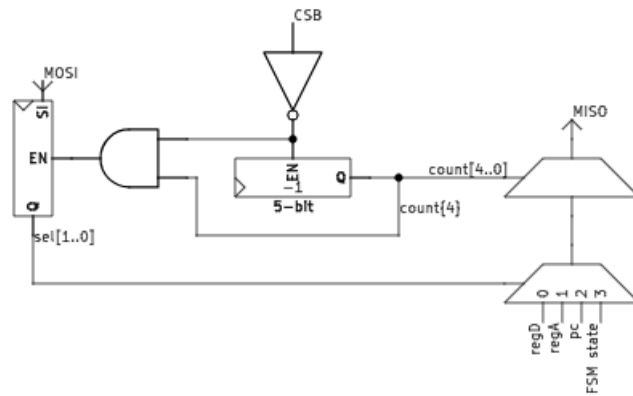


Figure 303.5: The debugging module

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.

Project Pinout

Digital Pins

#	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

#	Input	Output	Bidirectional
4	—	—	DEBUG CS
5	—	—	DEBUG MOSI
6	—	—	DEBUG MISO
7	—	—	DEBUG SCK

DDC

by Armaan Gomes

0322

HDL Project

github.com/arghunter/DDC-Digital-to-DIgital-Converter

“Converts I2S input to PDM output”

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

Project Pinout

Digital Pins

#	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	—

Sequential Shadows Deluxe [TT08 demo competition]

by Toivo Henningsson

0326

50.4 MHz

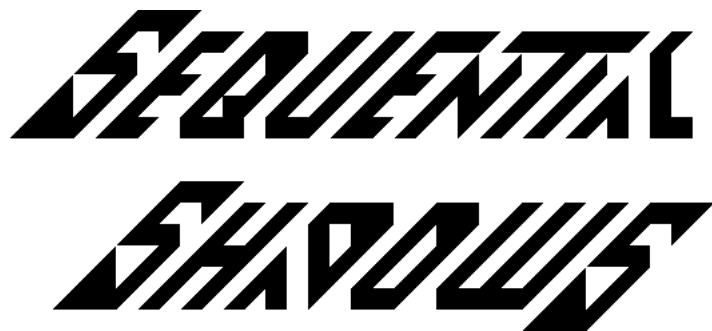
HDL Project

github.com/toivoh/tt08-demo-deluxe

"My contribution to the TT08 demo competition, extended version"

Intro

Curly / Medieval presents



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

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

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

How it works

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

How to test

Plug in a [TinyVGA](#) compatible Pmod on the TT08 demo board's out Pmod. Plug in a Pmod compatible with [Mike's audio Pmod](#) on the TT08 demo board's bidir Pmod. Set all inputs to zero to get the default behavior. **Warning: The default behavior includes some flashing 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.
- `rgb444_mode`: Setting this high sets the output to RGB444 mode instead of the default RGB222
- `pmod_vga_pinout`: Setting this high enables the alternative Pmod VGA pinout.
 - The `t_` outputs are used when `pmod_vga_pinout` is low. This fits the TinyVGA Pmod pinout. (`p_` only outputs are not driven.)
 - The `p_` outputs are used when `pmod_vga_pinout` is high. This fits the Pmod VGA pinout.
- `logo_shadow_off`: When high, removes the logo's shadow (like in the non-deluxe version).

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

For the demo competition, only use a Pmod VGA if you have one and can get sound output while using it. If using Pmod VGA, set `pmod_vga_pinout`, and you can set `rgb444_mode` as well. Don't set any other inputs. If using TinyVGA for output, set all inputs to zero.

External hardware

This project needs

- either
 - a [TinyVGA](#) VGA Pmod.
 - [Mike's audio Pmod](#).
- or a [Pmod VGA](#)

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

The choice of pinout is controlled by the pmod_vga_pinout input.

Project Pinout

Digital Pins

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

AlphaOneSoC

by **Abhiram Gopal Dasika**

0330

64 MHz

HDL Project

github.com/alfadelta10010/tt10-AlphaOneSoC

“A 32-bit RISC-V SoC, based on TinyQV by Michael Bell”

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](#)

Project Pinout

Digital Pins

#	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

2048 sliding tile puzzle game (VGA)

by Uri Shaked

0334

25.175 MHz

HDL Project

github.com/urish/tt10-2048-game

"Slide numbered tiles on a grid to combine them to create a tile with the number 2048."

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:

ui_in 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 `uio` pins. Check out the test bench for more information.

External hardware

- [TinyVGA Pmod](#)
- Optional: [Gamepad Pmod](#)

Project Pinout

Digital Pins

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

Traffic-light-sequence

by **Shaurya Sharma**

0416

Wokwi Project

github.com/ClutchBore/tt08-traffic-light-sequence

wokwi.com/projects/408118380088342529

"A simple traffic light sequence"

How it works

Sequence of Traffic light from Red to Red-Yellow to Green to Yellow to Red
How to test

Input 00= Red, Input 01= Red-Yellow, Input 10= Green, Input 11= Yellow

External hardware

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

Project Pinout

Digital Pins

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

Abacus Lock

by Raunak Singh

0417

Wokwi Project

github.com/Raunak-Singh-Inventor/abacus_lock

wokwi.com/projects/408231820749720577

“digital lock that turns the servo on a specific sequence of switches and displays current combination entered on a LED bar”

How it works

Abacus Lock is a digital combination lock that can be used in various projects. It moves the servo only when a specific sequence of switches is set. An LED Bar displays the status of the current lock combination in binary.

How to test

Follow Wokwi design and run the .ino file using Arduino: <https://wokwi.com/projects/408154128142306305>

External hardware

LED, LED Bar, Servo, Arduino Nano

This project was sponsored by The MITRE Corporation and MIT/LL Beaverworks Summer Institute

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	IN1	OUT1	—
1	IN2	OUT2	—
2	IN3	OUT3	—
3	IN4	OUT4	—
4	IN5	OUT5	—
5	IN6	OUT6	—
6	IN7	OUT7	—
7	IN8	OUT8	—

7 Segment Decode

by Jack Clayton

0418

5 kHz

HDL Project

github.com/JAC-EE/tt08-SegDecode-ASIC

"ASIC implementation of a university CPLD project which drives 4 multiplexed 7 segment displays, and scans a multiplexed keypad."

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.

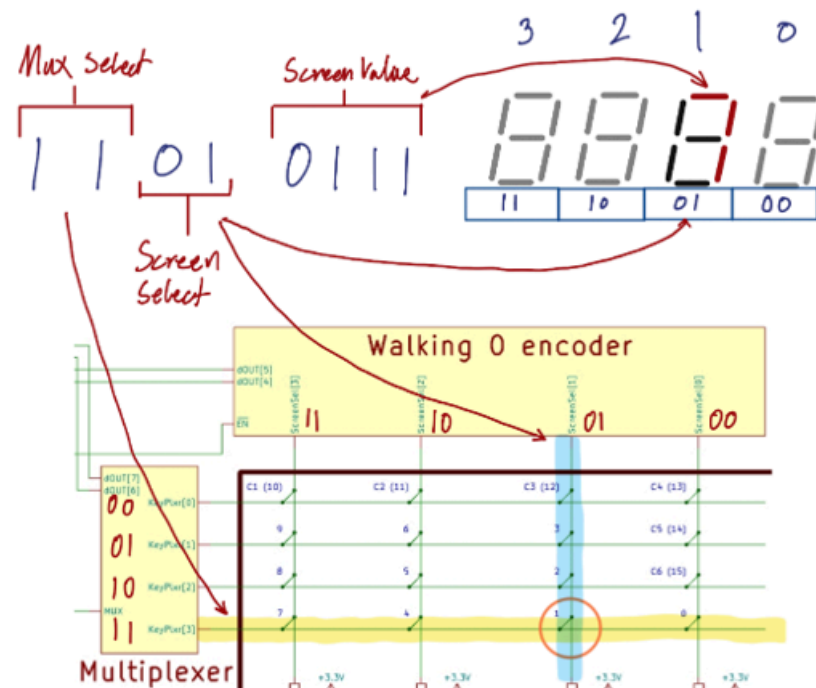


Figure 2.2.0.1 – SPI Protocol visual bit representations

Figure 418.1: 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

Figure 418.2: SPI Protocol bit definitions

2.2.2 Signal structure

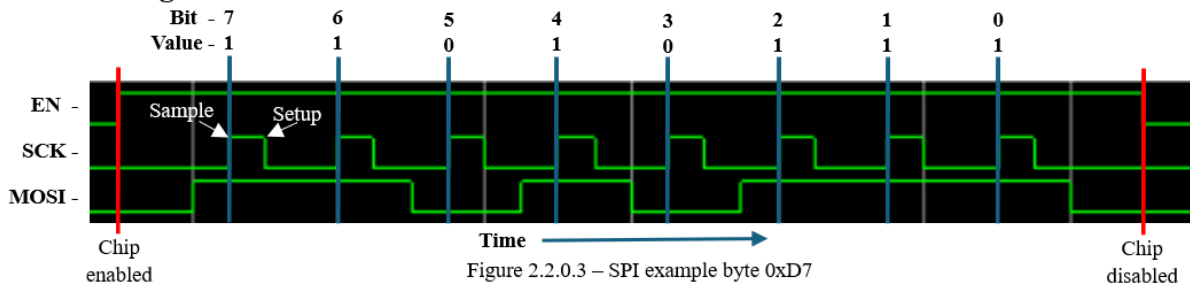
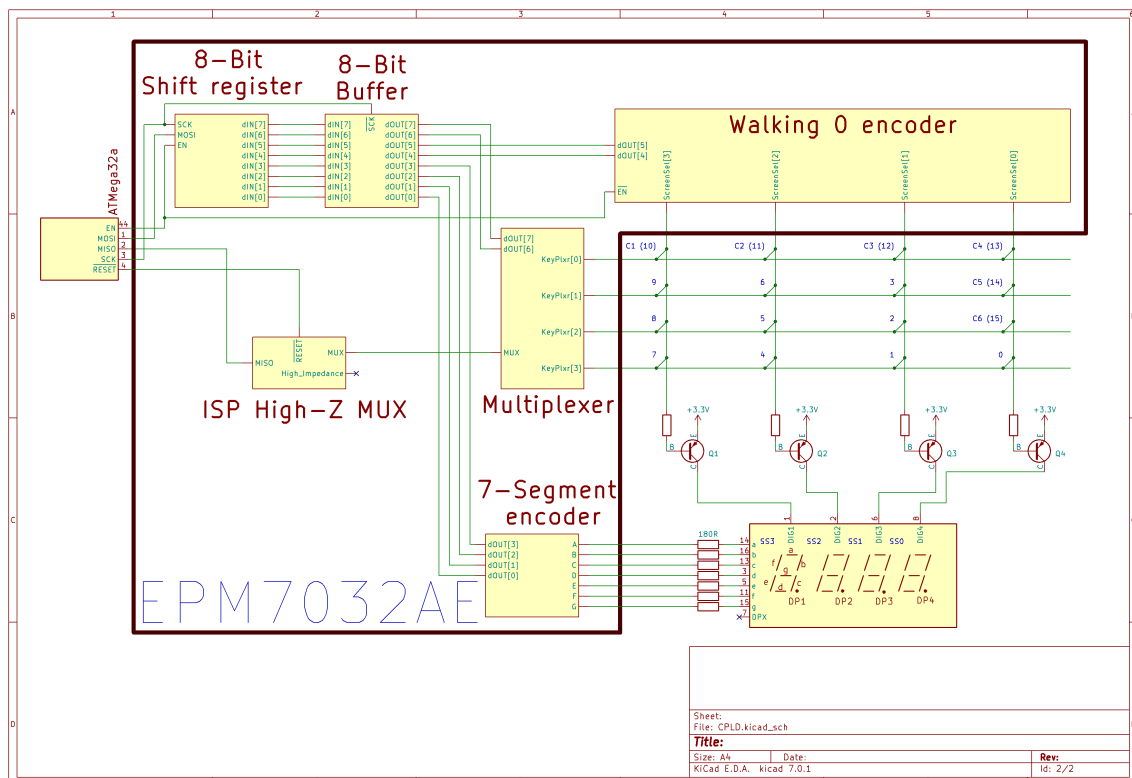


Figure 418.3: SPI example byte 0xD7



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:

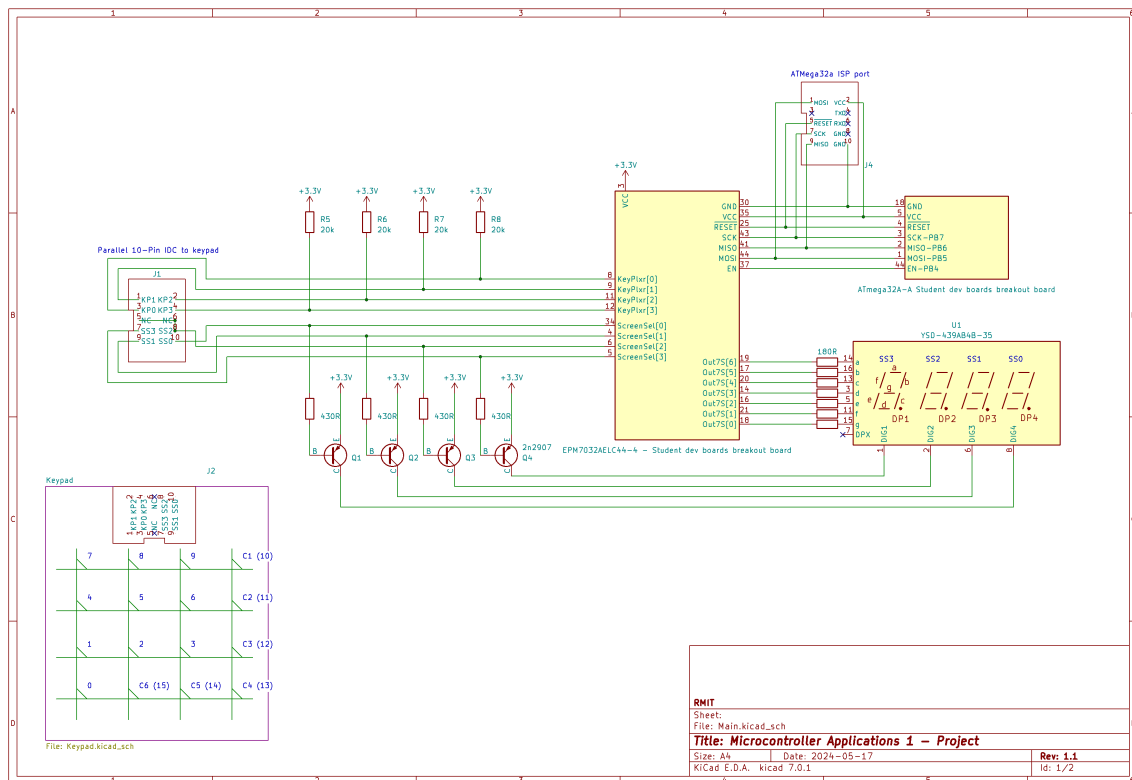


Figure 418.5: Main schematic

Simple keypad:

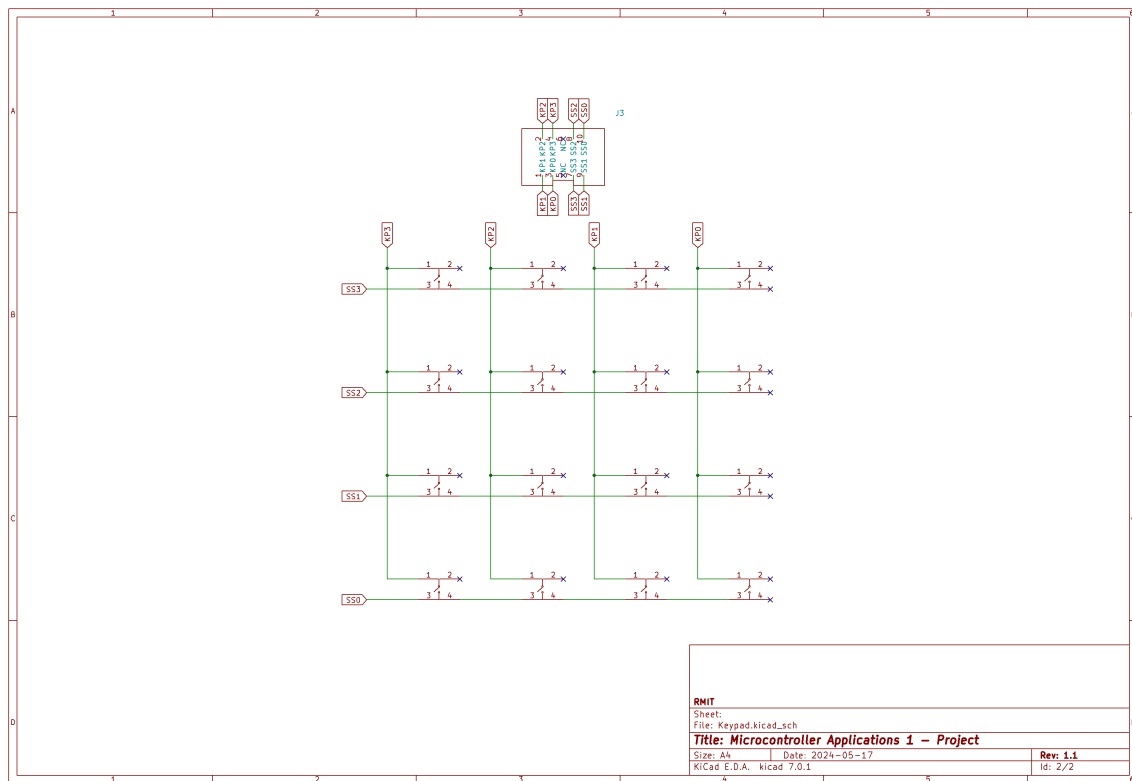


Figure 418.6: Keypad schematic

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	Out7S[0]	ScreenSel[0]
1	MOSI	Out7S[1]	ScreenSel[1]
2	EN	Out7S[2]	ScreenSel[2]
3	RESET	Out7S[3]	ScreenSel[3]
4	KeyPlxr[0]	Out7S[4]	High-Z
5	KeyPlxr[1]	Out7S[5]	—
6	KeyPlxr[2]	Out7S[6]	—
7	KeyPlxr[3]	MISO	—

TT08 - experiments with latch-based shift registers

by **Ciro Cattuto**

0419

HDL Project

github.com/ccattuto/tt08-sr-latch

"A 512-bit latch-based shift register in 1 tile"

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.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	shift register input	shift register output	—
1	shift control (edge-triggered)	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

DPMU

by **Sanjay Kumar M, Shylashree N, Ravish Aradhya H V, RV College Of Engineering, Neha R V & PES Unoversity**

0420

100 Hz

HDL Project

github.com/RVCE-ECE/TT8_DPMU

"Design and Implementation of Dynamic Power management unit"

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

Project Pinout

Digital Pins

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

Micro tile container

by Uri Shaked

0421

HDL Project

github.com/TinyTapeout/tt-micro-tiles-container

“Experimental microtile container”

How it works

Combined 4 micro tile sized projects into a single Tiny Tapeout tile.

Selecting the active project

Use `uio[1:0]` to select the active micro-tile project.

Project 0 - Test

- Repo: <https://github.com/TinyTapeout/tt-micro-tiles-experiment>
- Author: Uri Shaked
- Description: Micro tiles test module

How it works

The micro tiles test module is a simple module that demonstrates the use of the micro tile interface.

It has two modes of operation:

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

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 observe that the counter is output on the output pins (`uo_out`).

Project 1 - Micro Shift Reg ALU

- Repo: <https://github.com/MichaelBell/tt-micro-tiles-serial-compute>
- Author: Michael Bell
- Description: Micro tiles shift register and 8-bit compute

How it works

A 16-bit shift register that is clocked in from ui_in[0]. The low and high byte can be output on uo_out.

Additionally the result of certain computations of the low and high byte of the shift register can be latched and displayed:

- When ui_in[4] is high the result of ADDing the low and high bytes of the shift register is latched
- When ui_in[5] is high the result of ANDing the low and high bytes of the shift register is latched

How to test

Clock data in on ui_in[0].

ui_in[2:1] select the output, as follows

ui_in[2:1]	Output
0	Low byte of shift register
1	High byte of shift register
2	Latched ADD result
3	Latched AND result

Finally, if rst_n is high the outputs mirror the inputs. Reset is otherwise unused.

Project 2 - PDM CIC Filter

- Repo: <https://github.com/gfg-development/tt-micro-tiles-cic>
- Author: Gerrit Grutzeck
- Description: Micro tiles CIC filter for PDM signals with a 2 stage filter and a downsampling of 4

How it works

On ui_in[0] the input PDM datastream is received. Then it is processed in a CIC filter with 2 stages and a downsampling factor of 4. The resulting filtered samples are outputted on uo_out[7:2] and the downsampled clock on uo_out[0]. Furthermore on uo_out[1] the normal clock is available.

How to test

Connect a PDM microphone as follows:

- Clock: uo_out[1]
- Data: ui_in[0]

Then configure the clock generator of the RP2040 to generate a clock, as needed by the microphone (typically around 2 MHz) and reset the design

via `rst_n`. After the reset is removed again, the design is up and running, filtering the incoming datastream. With a connected logic analyzer or the RP2040 the filtered data can now be received on `uo_out[7:2]`, as well as the downsampled clock on `uo_out[0]`. The downsampled clock can be used to latch the filtered data.

Project 3 - Synchronous FIFO

How it works

A generic synchronous First-In-First-Out (FIFO) buffer. It operates on a single clock domain and allows for buffering of data from an input interface (`ui_in`) to an output interface (`uo_out`), while ensuring that data is neither overwritten when full nor read when empty.

Parameters

- **DATA_WIDTH:** The width of the data in bits (default: 6).
- **DEPTH:** The depth of the FIFO in terms of the number of entries (default: 3).

Details:

FIFO Depth:

- The default depth is set to 3 for practical reasons for a micro-tile, but this can be modified via the `DEPTH` parameter.

FIFO Width:

- The FIFO supports a 6-bit data width, which can also be adjusted via the `DATA_WIDTH` parameter.

Reset:

- The `rst_n` signal resets the FIFO, clearing its contents by resetting the write and read pointers (`wr_ptr` and `rd_ptr`).

Data Write:

- Data from the `ui_in` bus is written to the FIFO when the write enable signal (`wr_en`) is asserted (bit 6 of `ui_in`).
- The FIFO prevents writing if the buffer is full, indicated by the `o_full` signal.

Data Read:

- Data is read from the FIFO when the read enable signal (`rd_en`) is asserted (bit 7 of `ui_in`).
- The FIFO prevents reading if the buffer is empty, indicated by the `o_empty` signal.

Control Signals:

- `o_full`: Indicates when the FIFO has reached its maximum capacity.

- `o_empty`: Indicates when the FIFO has no data to read.

How to test

1. Write Operation:

- Set the `wr_en` signal (`ui_in[6]`) high and ensure `o_full` is low (`uo_out[6]`).
- Apply a 6-bit data value on the lower 6 bits of `ui_in[5:0]` until `o_full` is high.

2. Read Operation:

- Set the `rd_en` signal (`ui_in[7]`) high and ensure `o_empty` is low (`uo_out[7]`).
- Observe that data is read from the FIFO and appears on the lower 6 bits of `uo_out[5:0]`.
- Once all the data has been read the `o_empty` signal will go high (`uo_out[7]`).

Inputs and Outputs Table

Signal	Description
<code>ui[0]</code>	FIFO Read Enable
<code>ui[1]</code>	FIFO Write Enable
<code>ui[2]</code>	FIFO Data Input 1
<code>ui[3]</code>	FIFO Data Input 2
<code>ui[4]</code>	FIFO Data Input 3
<code>ui[5]</code>	FIFO Data Input 4
<code>ui[6]</code>	FIFO Data Input 5
<code>ui[7]</code>	FIFO Data Input 6
<code>uo[0]</code>	FIFO Empty Signal
<code>uo[1]</code>	FIFO Full Signal
<code>uo[2]</code>	FIFO Data Output 1
<code>uo[3]</code>	FIFO Data Output 2
<code>uo[4]</code>	FIFO Data Output 3
<code>uo[5]</code>	FIFO Data Output 4
<code>uo[6]</code>	FIFO Data Output 5
<code>uo[7]</code>	FIFO Data Output 6

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	in[0]	out[0]	sel[0]
1	in[1]	out[1]	sel[1]
2	in[2]	out[2]	—
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]	—

MAC

by **Mahaa Santeep G, Shylashree N, Ravish Aradhya H V, RV College Of Engineering, Sneha R V & PES University**

0422

100 Hz

HDL Project

github.com/RVCE-ECE/TT8-MAC

"Design and Implementation of MAC Unit Using Dadda Multiplier and Kogge-Stone Adder"

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.
- `uio_in` (8-bit): Input/Output interface for the second operand.
- `clk` (1-bit): Clock signal to synchronize all operations.
- `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.
- `uio_oe` (8-bit): Output enable signal, set to 0 indicating the `uio` is used as input.
- `uio_out` (8-bit): Unused output path in the current context.

Internal Architecture

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 han-

dling 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_out (Output)
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, uio_in, uo_out);
end
```

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

Project Pinout

Digital Pins

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

Logic Test

by Eric Ulteig

0423

Wokwi Project

github.com/Beird0/TT08_Memory

wokwi.com/projects/408216451206371329

“Verify”

How it works

Logic

How to test

Test

External hardware

None required

Project Pinout

Digital Pins

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

pulse_add

by Jonny Edwards

0424

HDL Project

github.com/Fountaincoder/trueadd

“a temporal add in digital”

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.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	in[0]	out[0]	—
1	in[1]	out[1]	—
2	in[2]	out[2]	—
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]	—

AES Inverse S-box

by **Dag Arne Osvik**

0425

125 MHz

HDL Project

github.com/daosvik/tt08-aes-invsbox

“Advanced Encryption Standard (AES) Inverse S-box”

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 `uio_out` (registered).

External hardware

None.

Project Pinout

Digital Pins

#	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]	—

nyan

by **Peter Nørlund**

0426

25 MHz

HDL Project

github.com/peter-noerlund/tt08-nyan

"Nyan Cat"

How it works

The Nyan Cat animation and music on infinite repeat

How to test

Connect the TinyVGA PMOD to the 0ut PMOD and Mike' Audio PMOD to Bidir PMOD.

External hardware

- TinyVGA PMOD
- Mike's audio PMOD

Project Pinout

Digital Pins

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

Super Mario Tune on A Piezo Speaker

by **Milosch Meriac**

0427

100 kHz

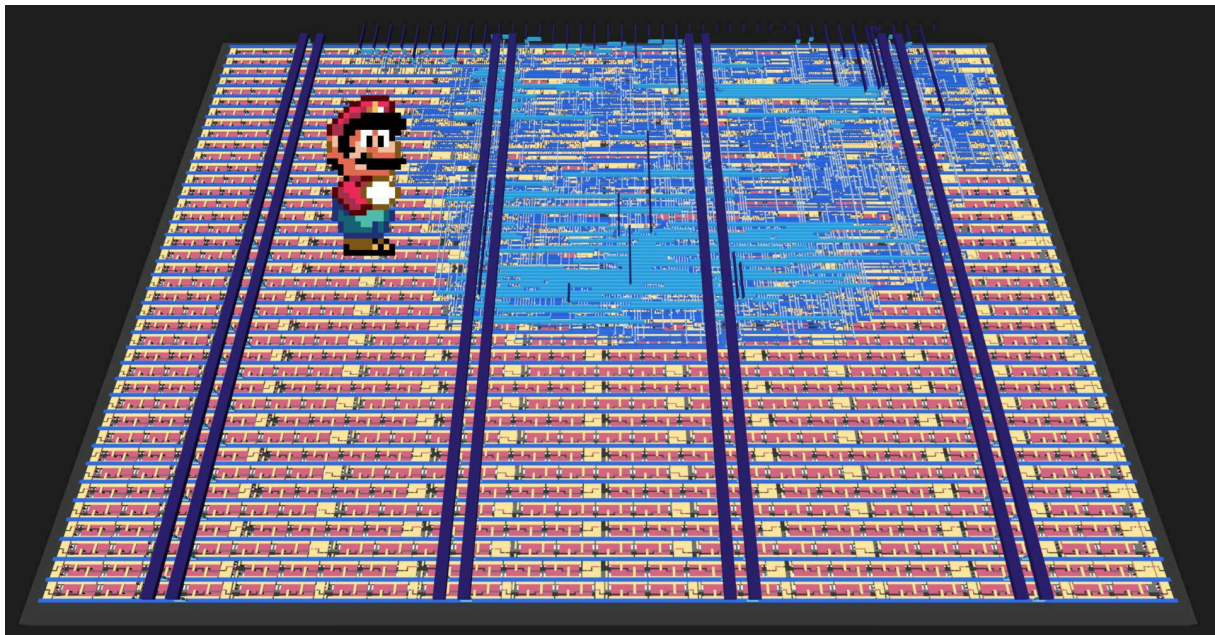
HDL Project

github.com/meriac/tt08-play-tune

*"Plays Super Mario Tune over a Piezo Speaker connected across
uio_out[7:0]"*

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](#))

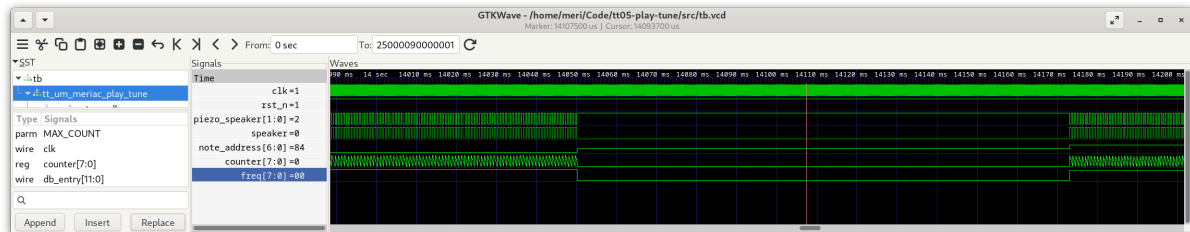
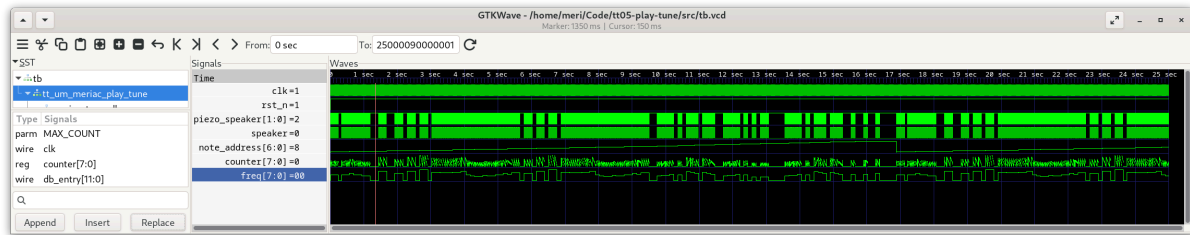
Additionally - for testing purposes, the inputs `ui[7:0]` are copied to the hex segment display `1:1` (`uo[7:0]`).

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.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input pin 0	ui[0]	piezo_speaker_p (uio_out[0])
1	input pin 1	ui[1]	piezo_speaker_n (uio_out[1])
2	input pin 2	ui[2]	GND
3	input pin 3	ui[3]	GND
4	input pin 4	ui[4]	GND
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])

Brailliance

by Akshat B, Evana T, John L, Rittrija M & Riley Gu

0428

5 MHz

HDL Project

github.com/rileyguu/tt08-brailliance

"ASCII-to-Braille Converter"

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

Project Pinout

Digital Pins

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

PS2 Decoder

by Ben Payne

0429

25 MHz

HDL Project

github.com/benpayne/tt08-ps2-68k

"A PS2 keyboard decoder"

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 by 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 tests 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 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 keyboard's 5V to 3.3V for this chip. I have a design for this if anyone wants it.

Project Pinout

Digital Pins

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

Adder with Flow Control

by Yuri Panchul

0430

3 Hz

HDL Project

github.com/yuri-panchul/tt08-adder-with-flow-control

“An adder with a separate flow control for each argument and the result”

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	a_vld	a_rdy	a_data[0]
1	b_vld	b_rdy	a_data[1]
2	sum_rdy	sum_vld	a_data[2]
3	—	sum_data[0]	a_data[3]
4	—	sum_data[1]	b_data[0]
5	—	sum_data[2]	b_data[1]

#	Input	Output	Bidirectional
6	—	sum_data[3]	b_data[2]
7	—	sum_data[4]	b_data[3]

TT08 SKY130 Shift Register 'YOLO' Test

by Sylvain Munaut

0431

HDL Project

github.com/smunaut/tt08-shiftreg-test

“Quick and dirty test of a custom high density shift register”

How it works

FIXME

How to test

FIXME

External hardware

FIXME

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	sr_in[0]	sr_out[0]	clk_ph0
1	sr_in[1]	sr_out[1]	clk_ph1
2	sr_in[2]	sr_out[2]	—
3	sr_in[3]	sr_out[3]	—
4	sr_in[4]	sr_out[4]	—
5	sr_in[5]	sr_out[5]	—
6	sr_in[6]	sr_out[6]	—
7	sr_in[7]	sr_out[7]	—

TinyQV SoC with additional peripherals

by **Michael Bell**

0454

64 MHz

HDL Project

github.com/MichaelBell/ttcad25a-tinyQV

“A Risc-V SoC with Tiny Tapeout contributed peripherals”

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.

This version of the SoC is designed to integrate with many user peripherals.

Address map

Address range	Device
0x00000000 - 0x0FFFFFFF	Flash
0x10000000 - 0x17FFFFFF	RAM A
0x18000000 - 0x1FFFFFFF	RAM B
0x80000000 - 0x80000033	DEBUG
0x80000034 - 0x8000003B	TIME
0x80000040 - 0x8000007F	GPIO
0x80000080 - 0x800000FF	UART (user peripherals 0-1)
0x80000100 - 0x800003FF	User peripherals 2-13
0x80000400 - 0x800004FF	Simple user peripherals 0-15

DEBUG

Register	Address	Description
----------	---------	-------------

SEL	0x800000C (R/W)	Bits 6-7 enable peripheral output on the corresponding bit on out6-7, otherwise out6-7 is used for debug.
DEBUG_UART_DATA	0x8000018 (W)	Transmits the byte on the debug UART
STATUS	0x800001C (R)	Bit 0 indicates whether the debug UART TX is busy, bytes should not be written to the data register while this bit is set.

See also [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

GPIO

Register	Address	Description
OUT	0x8000040 (RW)	Control for out0-7 if the GPIO peripheral is selected
IN	0x8000044 (R)	Reads the current state of in0-7
FUNC_SEL	0x8000060 - 0x800007F	Function select for out0-7

Function Select	Peripheral
0	Disabled
1	GPIO
2	UART TX
3	UART RX
4 - 15	User peripheral 2-13
16	Edge counter
17 - 31	Simple user peripheral 1-15

UART

Register	Address	Description
----------	---------	-------------

TX_DATA	0x8000080 (W)	Transmits the byte on the UART
TX_BUSY	0x8000084 (R)	Bit 0 indicates whether the UART TX is busy, bytes should not be written to the data register while this bit is set.
RX_DATA	0x80000C0 (R)	Reads any received byte
RX_STATUS	0x80000C4 (R)	Bit 0 indicates whether a received byte is available to be read.

Edge Counter

The [edge counter](#) is simple peripheral 0, with base address 0x8000400.

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 tt06 testing, at the target 64MHz clock a read latency of 2 is required. The maximum supported latency is currently 3.

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.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Interrupt 0	UART TX	Flash CS
1	Interrupt 1	UART RTS	SD0
2	—	—	SD1
3	1 MHz clock for time	—	SCK
4	—	—	SD2
5	—	—	SD3
6	—	Debug UART TX	RAM A CS
7	UART RX	Debug signal / PWM	RAM B CS / PWM

TinyQV Risc-V SoC

by **Michael Bell**

0460

64 MHz

HDL Project

github.com/MichaelBell/tt10-tinyQV

“A Risc-V SoC for Tiny Tapeout”

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
0x00000000 - 0x0FFFFFFF	Flash
0x10000000 - 0x17FFFFFF	RAM A
0x18000000 - 0x1FFFFFFF	RAM B
0x7FFFFF00 - 0x7FFFFFFF	Internal RAM (32 bytes, wrapped)
0x80000000 - 0x80000007	GPIO
0x80000010 - 0x8000001F	UART
0x80000020 - 0x80000027	SPI
0x80000028 - 0x8000002B	PWM
0x80000030 - 0x80000033	DEBUG
0x80000034 - 0x8000003B	TIME
0x80000040 - 0x80000047	GAME

GPIO

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

UART

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

Debug UART (Transmit only)

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

SPI

Register	Address	Description
DATA	0x8000020 (W)	Transmits the byte in bits 7-0, bit 8 is set if this is the last byte of the transaction, bit 9 controls Data/Command on out3
DATA	0x8000020 (R)	Reads the last received byte
CONFIG	0x8000024 (W)	The low 4 bits set the clock divisor for the SPI clock to $2 \times (\text{value} + 1)$, bit 8 adds half a cycle to the read latency when set

STATUS	0x8000024 (R)	Bit 0 indicates whether the SPI is busy, bytes should not be written or read from the data register while this bit is set.
--------	---------------	--

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.

Project Pinout

Digital Pins

#	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

VGA Pong with NES Controllers

by **Brandon S. Ramos**

0462

25.175 MHz

HDL Project

github.com/J0NTrollston/tt08-VGA-Pong-with-NES-Controllers

"Pong using 2 NES Controllers with a VGA display"

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_Left[1] clock
3. NES_Controller_Left[2] latch
4. NES_Controller_Right[0] data
5. NES_Controller_Right[1] clock
6. NES_Controller_Right[2] latch
7. NC
8. NC

You will also need to hook up the output to a VGA breakout board. I created my own using a perfboard and some resistors but you can use the TinyTape-out 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

Project Pinout

Digital Pins

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

Dempsey-Gorman DAC

by **htfab**

0480

Analog Project

github.com/htfab/dg-dac

“4+4 bit segmented DAC made from two resistor ladders”

How it works

Implements the Dempsey-Gorman DAC architecture, combining two resistor ladder DACs into a segmented DAC without voltage buffering.

See more details in Analog’s [MT-016 tutorial](#).

How to test

Set the desired voltage using `ui_in[7:0]`. The analog output should map this 8-bit value to the 0V to 3.3V voltage range.

External hardware

Multimeter or any other device that can measure voltages (oscilloscope, microcontroller with ADC, etc.)

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	IN[0]	—	—
1	IN[1]	—	—
2	IN[2]	—	—
3	IN[3]	—	—
4	IN[4]	—	—
5	IN[5]	—	—
6	IN[6]	—	—
7	IN[7]	—	—

Analog Pins

ua#	analog#	Description
0	3	VOUT

Obstacle Detection

by Emmy Xu

0481

HDL Project

github.com/emmyz00/Obstacle-Detector

“Does the logic of when to send certain signals when objects are close.”

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	sensor_left	left_buzz	—
1	sensor_right	right_buzz	—
2	reset	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Analog 8 bit 3.3v R2R DAC

by **Matt Venn**

0482

Analog Project

github.com/mattvenn/tt08-analog-r2r-dac-3v3

"A simple 8 bit DAC with a sine waveform driver and 3.3v output"

How it works

A simple 8 bit R2R DAC. Driven externally or by an digitally generated sine waveform generator.

3.3v output is achieved with level shifting drivers.

How to test

Drive externally

Set the `external` data input high to provide the DAC with external data.

Then drive the 8 inputs and observe the analog output.

Drive with internal sawtooth wave generator

Set the `external` data input low to enable the sine generator. A sine wave should be seen on the analog output. Everytime the sine counter is at 0, digital output 0 should go high for one clock.

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

External hardware

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

Project Pinout

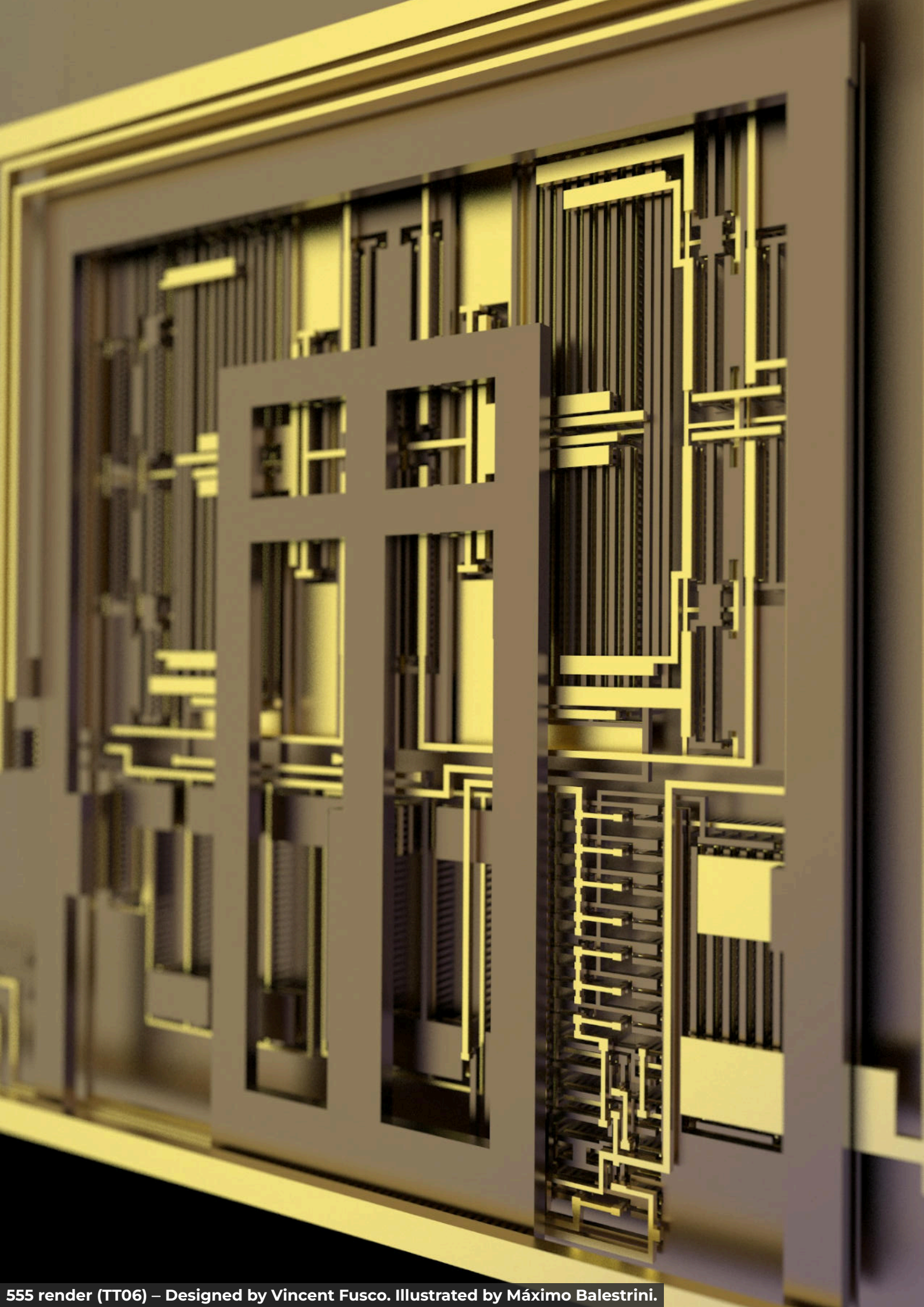
Digital Pins

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

#	Input	Output	Bidirectional
7	bit 7	—	—

Analog Pins

ua#	analog#	Description
0	5	DAC output



Styler

by **Rebecca G. Bettencourt**

0483

HDL Project

github.com/RebeccaRGB/styler

"16x16 bitmap manipulation based on text mode attributes."

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, a 6-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 dim text (even or odd pixels), text blink rate, and cursor blink rate.

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	6-7	Input: ignored; output: 0.
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	Text blink phase.
5	Cursor blink phase.
6	/OE (output enable).
7	/WE (write enable).

How to test

Test cases are to be determined.

External hardware

The styler chip is intended to be used as part of a larger text mode video display hardware project.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	A0 (address)	D0	D0
1	A1 (address)	D1	D1
2	A2 (address)	D2	D2
3	dim text phase	D3	D3
4	text blink phase	D4	D4
5	cursor blink phase	D5	D5

#	Input	Output	Bidirectional
6	/OE (output enable)	D6	D6
7	/WE (write enable)	D7	D7

R2R DAC

by Wallie Everest

0484

Analog Project

github.com/WallieEverest/tt08

“Duplicate of 8-bit DAC from Matt Venn”

How it works

Please refer to Matt Venn’s R2R DAC description, as this is a clone of that project.

How to test

Set the `external` data input high to provide the DAC with external data.

Then drive the 8 inputs and observe the analog output.

External hardware

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	bit 0	count zero	external data
1	bit 1	—	load divider
2	bit 2	—	—
3	bit 3	—	—
4	bit 4	—	—
5	bit 5	—	—
6	bit 6	—	—
7	bit 7	—	—

Analog Pins

ua#	analog#	Description
0	4	DAC output

resfuzzy

by roshan

0485

HDL Project

github.com/ROSHAN0120/tt08-verilog-resfuzzy

“calculation”

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]

Project Pinout

Digital Pins

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

Pi Snake

by **htfab**

0486

Analog Project

github.com/htfab/pi-snake

“Voltage divider that generates π volts”

How it works

The blue circle has unit radius and therefore an area of π units. We can enclose it by a polygon with exactly 3.3 units of area, adding the parts marked red. Using precision resistors that meander through the blue and red areas respectively we can construct a voltage divider that subdivides the the 3.3 V power and ground rails to achieve a π V output.

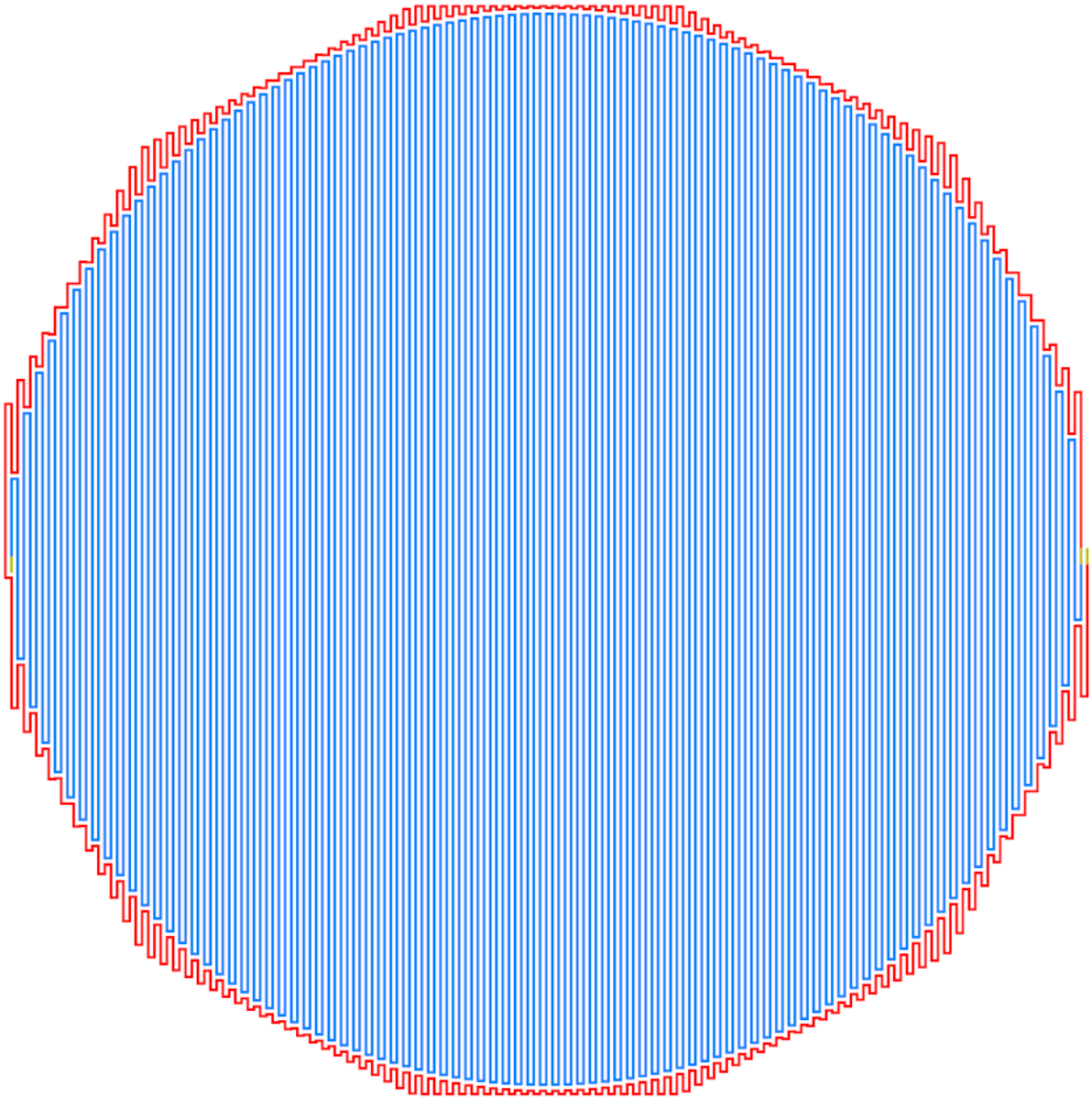


Figure 486.1: blue line meanders through circle, red line meanders through polygon minus circle

How to test

Measure the voltage between the single analog output pin and ground. It should read approximately π volts.

External hardware

Multimeter or other test equipment

Project Pinout

Digital Pins

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

Analog Pins

ua#	analog#	Description
0	5	voltage output

CEJMU Beers and Adders

by **Prof. Dr.-Ing. Matthias Jung, Philipp Wetzstein, Derek Christ & Jonathan Hager**

0487

12 MHz

HDL Project

github.com/CEJMU/tt08_cejmu

"Several projects to show in lectures. Includes a simple state-machine, a decoder and two 24 bit adders. Refer to documentation for details"

How it works

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

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

How to test

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

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

1. Select the adder you want to use: `uio_in[1:0] == 10` (RCA) or `11` (CLA)
2. Reset the chip for at least one cycle
3. `ui_in[7:0]` should now be assigned `a[23:16]`
4. Wait for one cycle, repeat with `a[15:8]`, `a[7:0]`
5. Repeat with `b[23:16]`, `b[15:8]`, `b[7:0]`

6. The inputs are now read into the design and will be send to the adders by asserting uio_in[2] 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

Project Pinout

Digital Pins

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

I2S to PWM

by Armaan Gomes

0489

HDL Project

github.com/arghunter/I2S-to-PWM-Verilog

"An 8-bit I2S to PWM convertor"

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

Project Pinout

Digital Pins

#	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	—

Bandgap Reference

by Asal Golmanesh

0490

Analog Project

github.com/AsalGolmanesh/tt08-bgr

“Bandgap 1.2V at 1.8 supply with skywater 130nm from 0 to 80deg”

How it works

This project involves designing a bandgap reference circuit using Skywater 130nm CMOS technology to provide a stable 1.17V output across a temperature range of 0 to 80°C. A startup circuit is included to initialize the system upon power-up. The circuit operates at 1.8V and consumes 0.35 mW of total power.

How to test

To test the circuit, apply the specified power supply voltage of 1.8V. Upon power application, the startup circuit should activate, enabling the entire circuit. The following major tests are required to evaluate the bandgap reference (BGR) circuit:

1. Temperature Testing: Use a temperature chamber or hotplate to vary the temperature and measure the reference voltage at different points:
 - Start at room temperature (25°C).
 - Test at low temperatures (e.g., 0°C).
 - Test at high temperatures (e.g., 80°C).
2. Line Regulation: Vary the supply voltage (e.g., from 1.6V to 2.0V) and measure the output reference voltage to assess the circuit's ability to maintain a stable output.
3. Load Regulation: Apply different loads to the bandgap reference output and measure the change in output voltage to ensure stability under varying load conditions.
4. Power Consumption: Use a multimeter or source-measure unit (SMU) to measure the current consumption of the bandgap circuit.
5. Long-Term Drift: If feasible, operate the circuit for an extended period (e.g., hours or days) under typical conditions and periodically measure the output voltage to verify that the reference voltage does not drift significantly over time.

External hardware

Ensure a proper testing environment with equipment such as oscilloscopes, multimeters, SMUs, temperature chambers, and power supplies. An accurate power supply is essential to provide the required 1.8V (or the specified voltage in your design) to the circuit.

Project Pinout

Digital Pins

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

Analog Pins

ua#	analog#	Description
0	4	—
1	1	—
2	3	—
3	2	Vbgr

DMTD

by **Armaan Gomes**

0491

HDL Project

github.com/arghunter/DMTD-Verilog

"A Dual Mixer Timer Differential"

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

Project Pinout

Digital Pins

#	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	—

Raw_Transistors

by Isshu Wakita

0492

Analog Project

github.com/wakki-0123/Raw_Transistors

"Performance evaluation of Raw_Transistors"

How it works

The circuit consists of raw NMOS and PMOS transistors.

How to test

ua[0] and ua[1] are connected to the source and drain of the MOSFETs (NMOS1, NMOS2, and PMOS1). The length/width are 0.25um/1.0um for all MOSFETs. ua[2] provides the gate bias for NMOS1, ua[3] provides the gate bias for NMOS2, and ua[4] provides the gate bias for PMOS1. Set the appropriate bias voltages, then measure the current.

External hardware

A source-measure unit is used to measure currents.

Project Pinout

Digital Pins

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

Analog Pins

ua#	analog#	Description
0	0	source
1	4	gate1
2	1	gate2

ua#	analog#	Description
3	3	gate3
4	2	drain

PWM generator

by **Matea Samuel**

0493

50 MHz

HDL Project

github.com/MateaSamuel/tt08-pwm-generator

"Generate pwm signal with configurable period - 12-bit - and duty cycle - 1%-99%."

How it works

This design is intended to function as a PWM generator. It contains two 12-bit registers: one for the duty cycle (duty_reg) and one for the period (period_reg). When the sel signal is set to "0," the duty_reg is selected, and when sel is "1," the period_reg is selected. If values for the duty cycle or period are provided at the input, they are written to the registers only when wr_en is set to "1." For the duty cycle, only 7 bits (from 0 to 6) are used, with the remaining bits hardcoded to 0. The value for period_reg can range from 2 to 4095 (using 12 bits). The pwm_out signal will be available only when out_en is set to "1."

How to test

Connect the output to an oscilloscope and verify whether the frequency and duty cycle correspond to your expectations.

External hardware

The only external hardware required is the wire for the pwm_out signal and 15 inputs (such as a microcontroller, digital pattern generator, etc.) to set the period, duty cycle and controls signals: sel, wr_en, and out_en.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	in[0]	pwm_out	in[8]
1	in[1]	—	in[9]
2	in[2]	—	in[10]
3	in[3]	—	in[11]
4	in[4]	—	—
5	in[5]	—	out_en
6	in[6]	—	sel

#	Input	Output	Bidirectional
7	in[7]	—	wr_en

TT08 Analog Factory Test

by Sylvain Munaut

0494

Analog Project

github.com/smunaut/tt08-analog-factory-test

“Test structures for TT08 analog support”

How it works

FIXME

How to test

FIXME

External hardware

FIXME

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	ena_1v8_n	digital_out	digital_out
1	ena_3v3_n	digital_out	digital_out
2	—	digital_out	digital_out
3	—	digital_out	digital_out
4	—	digital_out	digital_out
5	—	digital_out	digital_out
6	uio_oe	digital_out	digital_out
7	digital_in	digital_out	digital_out

Analog Pins

ua#	analog#	Description
0	5	ibias
1	0	vapwr_sense
2	4	vgnd_sense
3	1	vdpwr_sense
4	3	loopback_a
5	2	loopback_b

Simple Stopwatch

by **Fabio Ramirez Stern**

0495

1 MHz

HDL Project

github.com/faramire/TT08-simple-stopwatch

“A simple stopwatch counting in 100th seconds and outputting it via SPI to a MAX7219 chip controlling an 8 digit 7-segment display.”

How it works

A clock divider turns 1 MHz into 100 Hz, which drives a stopwatch going from 00:00:00 to 59:59:99. To achieve this, a chain of two types of counting circuit, one per digit gives it's output to an SPI master that encodes the result to be displayed on a 7-segment display with at least 6 digits.

How to test

The start/stop button toggles the clock, the lap time button pauses the display, while the clock keeps running in the background. Pressing it again re-enables the display. The time can be reset with the reset button on input 2, or with the chip/PCB wide reset. The PCB wide reset affects everything, the input pin driven reset does only resets the counters.

External hardware

2-3 buttons, one for start/stop and one for lap times. For the reset, either a third button or the dev board's reset for the whole chip can be used. 1 MAX7219/MAX7221 driven 7-segment display, or something that can interpret the SPI signal according to the MAX's specifications.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	start/stop	SPI MOSI	—
1	lap time	SPI CS (active low)	—
2	reset (active high)	SPI CLK	—
3	skip display setup (only output time, active high during reset)	stopwatch enabled (counting up)	—
4	—	display enabled (goes low when showing lap time)	—

#	Input	Output	Bidirectional
5	—	—	—
6	—	—	—
7	—	—	—

Current-Mode Bandgap Reference

by **Christoph Weiser**

0514

Analog Project

github.com/christoph-weiser/tt10-vref

“PVT stable reference voltage and current.”

How it works

This circuit is a Current-Mode Bandgap Reference which is fully self-biased. It is powered from a nominal 3.3V supply.

It produces a 1V reference voltage and a 5.6uA output current.

In order to measure the very sensitive output voltage of the bandgap reference, a testbuffer is also included in the design. The testbuffer is biased from an external current source (5uA) through pin ua[2].

To characterize the testbuffer a loop-through mechanism from ua[1] to ua[0] can be enabled. The test voltage can be applied to ua[1] and then measured back at ua[0].

The bandgap core can be enabled/disabled using the signal on uio_in[2].

To trim out process variations the core can be trimmed with a 8 bit correction resistor controlled from the tiny-tapeout management area using ui_in[7:0].

How to test

Testing the testbuffer.

1. Power VAPWR to 3.3V
2. Apply 5uA bias current to the testbuffer on ua[2]. SMU or 620kOhm from 3.3V.
3. Apply external test voltage to input on ua[1].
4. Route external voltage to testbuffer input. uio_in[0]=1
5. Route buffered voltage to output. uio_in[1]=1
6. Measure voltage back on ua[0].

Testing the bandgap voltage.

1. Power VAPWR to 3.3V
2. Apply 5uA bias current to the testbuffer on ua[2]. SMU or 620kOhm from 3.3V.
3. Enable bandgap core. uio_in[0]=1
4. Route bandgap voltage to testbuffer input. uio_in[0]=0

5. Route buffered bandgap voltage to output pin. uio_in[1]=1
6. Measure voltage on ua[0].

Testing the reference current.

1. Power VAPWR to 3.3V
2. Enable bandgap core. uio_in[0]=1
3. Route reference current to output pin. uio_in[1]=1
4. measure current from ua[0] to GND.

External hardware

Multimeter to measure voltage and current.

Optional: Oscilloscope to measure startup waveform. Precise power supply for VAPWR SMU for precision bias current. Oven to measure temperature stability.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	trim 7	—	Select Vext or Vbg
1	trim 6	—	Select Vbuf or Iout
2	trim 5	—	Enable bandgap core
3	trim 4	—	—
4	trim 3	—	—
5	trim 2	—	—
6	trim 1	—	—
7	trim 0	—	—

Analog Pins

ua#	analog#	Description
0	7	Buffered output voltage or reference current
1	9	External test input voltage
2	8	Testbuffer bias current (~5uA)

Mixed-Signal

by **Alexander Co Abad & Isabelle Rose Sta Rita**

0516

Analog Project

github.com/alexandercoabad/mixedsignal

"Inverter Biased for Linear Mode Operation"

How it works

Inverter Biased for Linear Mode Operation

How to test

First, add a 20nF capacitor then input a small signal sinewave. Then, see the output using an oscilloscope.

External hardware

Oscilloscope 20nF capacitor

Project Pinout

Digital Pins

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

Analog Pins

ua#	analog#	Description
0	6	output
1	10	input

Level Shifter

by Matt Venn

0518

Analog Project

github.com/mattvenn/tt10-level-shifter

"Analog Level Shifter Experiment"

Danger!

- Enabling this design without first removing the 3.3v VAPWR will blow the design.
- The 3.3v VAPWR rail is used as a variable supply to a single inverter.
- You must remove the VDDA 0 ohm resistor on the breakout board, and provide 1.8v.

How it works

The [traditional level shifter](#) only works down to about 1v.

This [level shifter](#) was designed by Sylvain Munaut and operates down to around 0.2v.

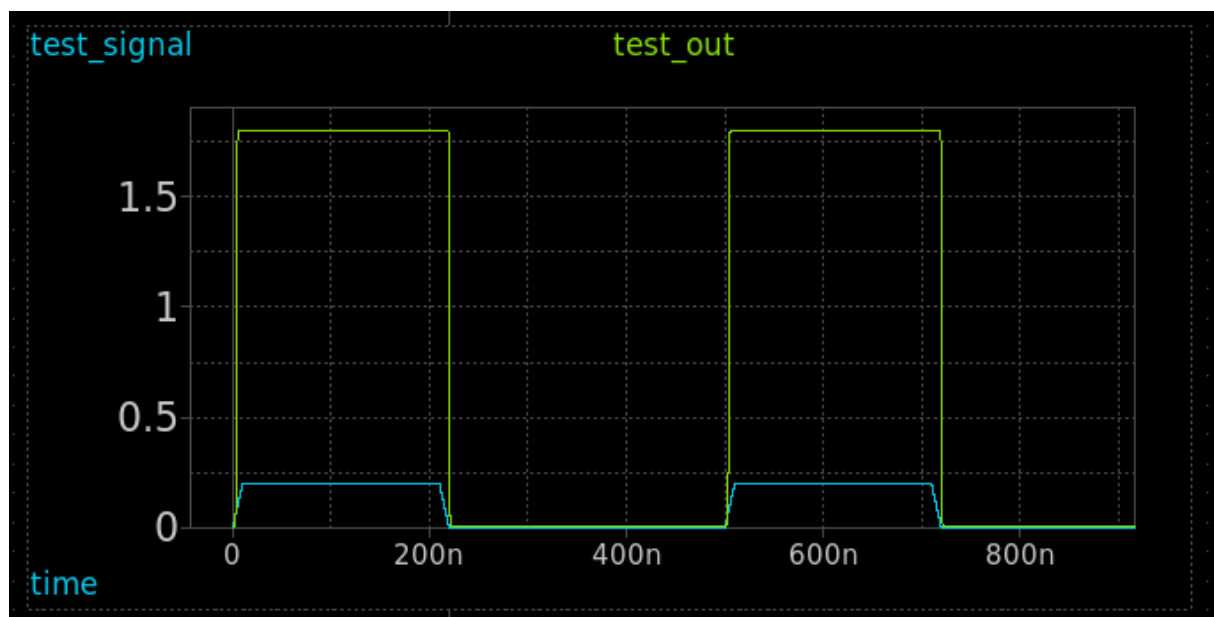


Figure 518.1: graph

Graph shows an input signal of 0.2v resulting in a full swing 1.8v on the output.

Theory of operation

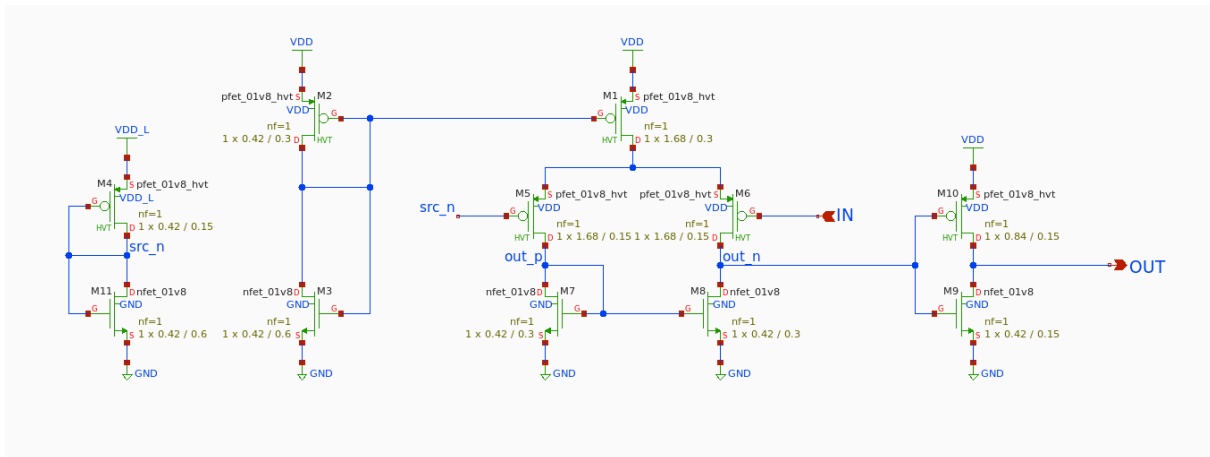


Figure 518.2: level

- M4 and M11 create a reference voltage `src_n` about half of `VDD_L`.
- M2, M3 and M1 create a current source for the differential pair.
- M5 and M6 are the differential pair, with the `src_n` on one side and the input on the other.
- M7 and M8 are the current mirror, along with M1, M5 and M6 they create the classical opamp comparator.
- Finally the output is buffered through M10 and M9 as an inverter.

This design features 2 of these level shifters, one connected to the digital pins and one to the analog.

Digital version

`ui[0]` is connected to the input of an inverter standard cell. The inverter is powered by the low power rail (VAPWR).

The level shifter receives input from the inverter and outputs to `uo[0]`.

Analog version

Analog pins `In`, `Out` and `VDD_L` are connected to the level shifter. `VDD` comes from the 1.8v supply.

How to test

Digital version

- Set up the variable PSU (see above), and set it to 1.8v.
- Input a test signal to A: `ui[0]`.
- Observe output signal on Y: `uo[0]`.
- Lower the variable PSU and the output signal should stay full swing until about 0.2v.

Analog version

- Supply 1.8v or less to `vdd_1` pin.

- Supply test input signal to the in pin.
- Observe the out pin.
- Lower the variable PSU and the output signal should stay full swing until about 0.2v.

External hardware

- Adjustable power supply
- Soldering iron to remove the resistor
- Scope to observe output signal

Project Pinout

Digital Pins

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

Analog Pins

ua#	analog#	Description
0	11	out
1	10	in
2	6	vdd_l

TT08 VGA FUN!

by algofoogle (Anton Maurovic)

0520

25 MHz

Analog Project

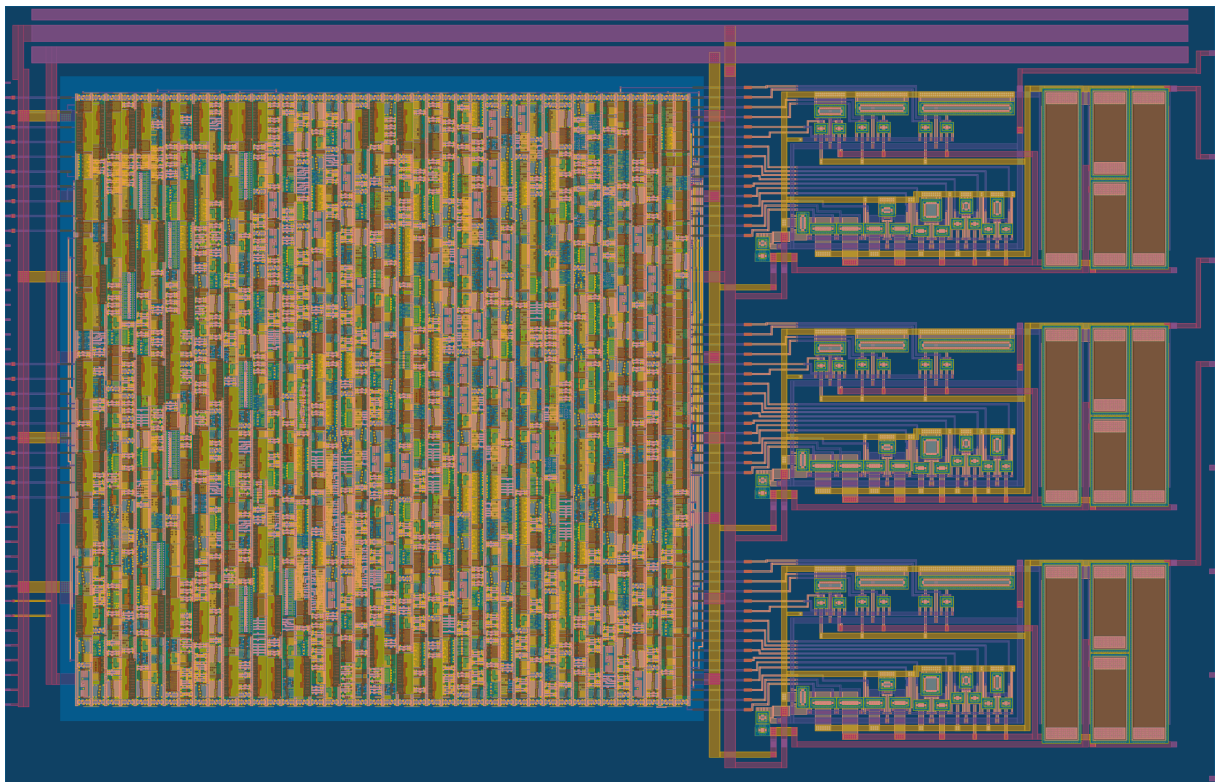
github.com/algofoogle/tt08-vga-fun

“Rough 24-bit VGA DAC tests with digital control block”

Overview

This project (tt08-vga-fun) uses (roughly-designed) current steering DACs to hopefully produce analog outputs that can produce an adequate RGB888 (24-bit) VGA image, based on patterns that can be generated from a simple digital controller. This improves on my previous [tt06-grab-bag](#) – my 1st analog ASIC project, [included](#) on [TT06](#), using 3 RDAC instances instead.

With these current steering DACs, I’m hoping for an improved slew rate (estimated to be about 60-80nS; still below the target of 40nS, but better than the TT06 version which was estimated to be about 240nS).

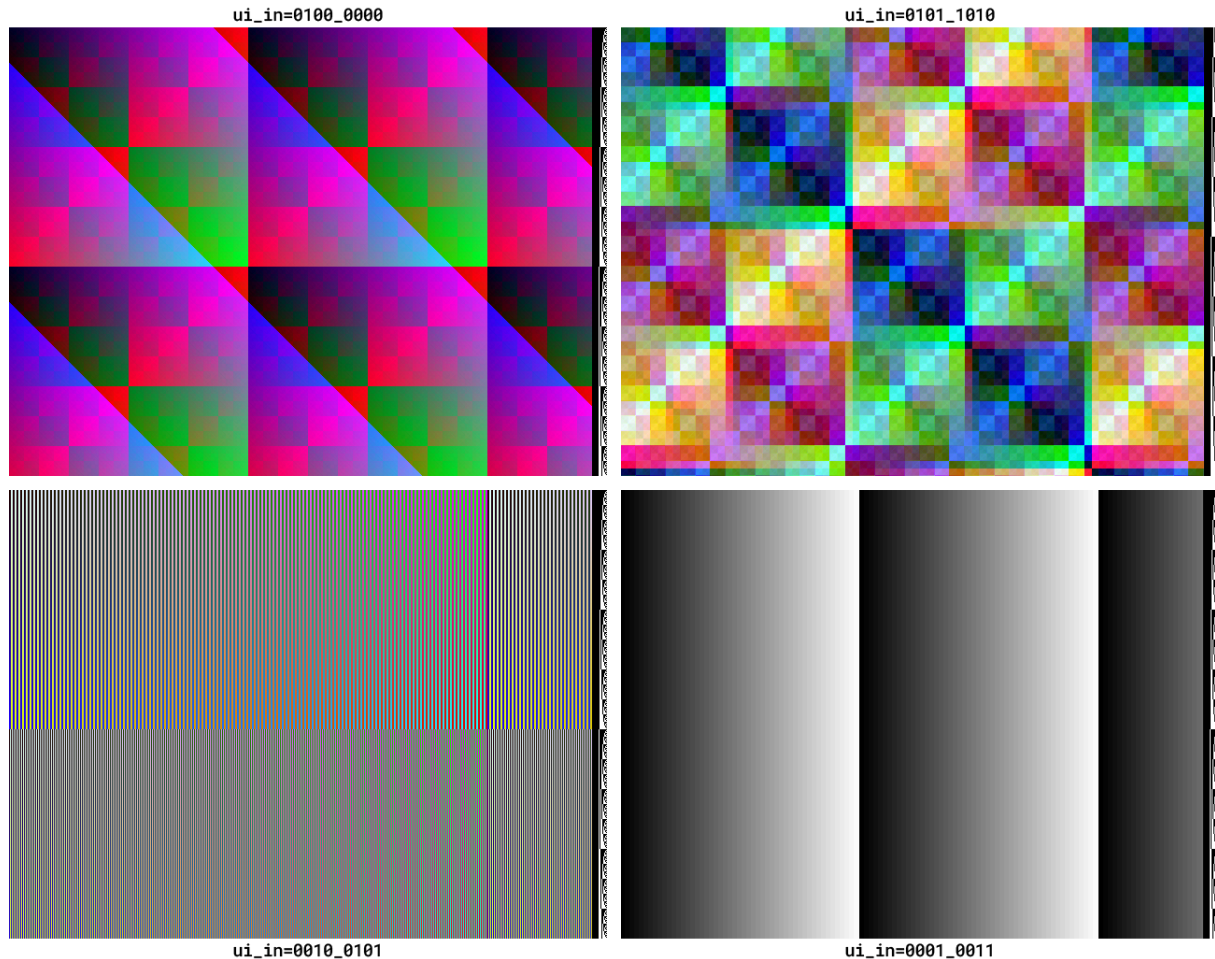


Note that the analog R/G/B outputs (ua[1:3]) are expected to be in the range 0.9-1.8V, and high impedance, while VGA requires a 0.0-0.7V range and 75Ω impedance. Thus, external opamps will be required.

How it works

There is a digital control block which can be [controlled by the state of the ui_in pins](#) at reset. It has various test modes, and a pass-through mode.

Here are some of the test patterns it can produce, but note that the image probably won't be this clear because of: (a) poor matching; and (b) slew simulated to be worse than 40nS will lead to a little bit of horizontal smearing:



The digital control block internally drives 3 (RGB) colour channels, each of which has 8 positive and 8 negative polarity bits. This complementary polarity is required for switching the binary-weighted current steering transistors either one way or the other, maintaining an equal (estimated) current of 500 μ A per channel. Each channel's internal current sum is then converted to a voltage with a pull-up resistor that is about 2.3k Ω .

Additionally the first analog output pin (ua[0]) is the internal VbiasR of the red channel DAC (gate voltage for current mirroring); this is for testing, but could possibly also be pulled up or down a little to see what effect it has on the red channel's output.

How to test

TBC.

External hardware

Probably an op-amp on each analog output, plus a VGA connector.

TBC.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	mode[0] / dac_in[0]	r7	vblank_out
1	mode[1] / dac_in[1]	g7	hblank_out
2	mode[2] / dac_in[2]	b7	—
3	mode[3] / dac_in[3]	vsync	—
4	mode[4] / dac_in[4]	r6	—
5	mode[5] / dac_in[5]	g6	bias1_in
6	mode[6] / dac_in[6]	b6	bias2_in
7	mode[7] / dac_in[7]	hsync	bias3_in

Analog Pins

ua#	analog#	Description
0	10	VbiasR
1	7	r
2	9	g
3	8	b

Sine Wave Synthesizer

by Maximilian Scherzer

0522

Analog Project

github.com/m-scherzer/tt_um_08_sws

"Generate Sine Wave"

How it works

Sine wave synthesizer (SWS), that generates a stepwise approximation of a sinusoid.

How to test

The SWS produces a sinosoid with an output frequency depding on the clk.

Drive externally

NN

Drive with internal sawtooth wave generator

NN

External hardware

Pins (ion,iop) need a 1kohm resistor to gnd.

Project Pinout

Digital Pins

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

Analog Pins

ua#	analog#	Description
0	6	bias

ua#	analog#	Description
1	10	ib_hi
2	7	ib_lo
3	9	iop
4	8	ion

AICD Playground

by **Leo Moser**

0526

1 MHz

Analog Project

github.com/mole99/tt08-aicd-playground

“A mixed-signal test project for the analog IC design course at Graz University of Technology.”

How it works

AICD Playground is a mixed-signal test project for the analog IC design course at Graz University of Technology.

It integrates a digital 8-bit controller with several analog IPs such as up/down-levelshifters, an R2R-DAC and a comparator. The IPs are connected in a way that a simple SAR-ADC is formed.

How to test

To initialize the memory, you need to set the `mode` pin to high. Now data can be send via the SPI interface, which is then written into the memory. After a rising edge of `mode`, the address points to zero and is incremented with each SPI transaction. Send 64 bytes to initialize the whole memory.

Next, set `mode` to low. The CPU starts executing the program. Depending on the program that is loaded, an analog voltage is applied to `ua[1]` via the R2R-DAC. This voltage is compared with the voltage at `ua[0]` and the result can be read by the CPU. In this way, a simple SAR-ADC can be programmed.

External hardware

No external hardware necessary.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	port_i[0]	port_o[0]	CS
1	port_i[1]	port_o[1]	MOSI
2	port_i[2]	port_o[2]	MISO
3	port_i[3]	port_o[3]	SCK
4	port_i[4]	port_o[4]	mode
5	port_i[5]	port_o[5]	debug_i

#	Input	Output	Bidirectional
6	port_i[6]	port_o[6]	debug_o[0]
7	port_i[7]	port_o[7]	debug_o[1]

Analog Pins

ua#	analog#	Description
0	11	adc_in
1	6	dac_out

Clock Divider

by Armaan Gomes

0544

HDL Project

github.com/arghunter/Clock-Divider-Verilog

"A clock divider with lag correction"

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

Project Pinout

Digital Pins

#	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

by **Sanjay Kumar M, Shylashree N, Ravish Aradhya H V, RV College Of Engineering, Neha R V & PES University**

0546

100 Hz

HDL Project

github.com/RVCE-ECE-Shylashree/TT8-DPMU

"Design and Implementation of Dynamic Power management unit"

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 (uio_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.

Time (ns)	ui_in (Input)	State	uio_out (Expected Output)	uo_out (Expected Output)
0	11110010	NORMAL	01010110	010_010010
10	00010010	PERFORMANCE	11111111	111_111111
30	11110010	NORMAL	01010110	010_010010
50	11110010	THERMAL_MANAGEMENT	01010111	011_011011
70	11110010	NORMAL	01010110	010_010010
90	11101010	THERMAL_MANAGEMENT	01010111	011_011011
110	11111010	BATTERY_SAVING	00000000	000_000000
130	11111110	BATTERY_SAVING	00000000	000_000000

150	11111010	BATTERY_SAVING	00000000	000_000000
-----	----------	----------------	----------	------------

Testbench 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.

Project Pinout

Digital Pins

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

6Digit7SegClock

by Patrick Lampl

0547

32.768 kHz

HDL Project

github.com/meisterlampe99/tt10-6Digit7SegClock

“A digital clock multiplexed to a 6 digit 7 segment display”

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.

Project Pinout

Digital Pins

#	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

#	Input	Output	Bidirectional
6	—	segment a	—
7	—	segment dot	—

Generate VGA output for Color Blindness Test

by [Krushnasis Pradhan](https://www.linkedin.com/in/krushnasis-pradhan-35881518b/) & [Aniruddha Ranade](https://www.linkedin.com/in/aniruddha-ranade/)

0548

HDL Project

github.com/AniruddhaRanade/TT08_VGA_CBTTest

“Generate VGA output which shall display the pattern similar to Ishihara Plates”

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

Project Pinout

Digital Pins

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

MAC Operation

by **Sachin & Sandeep**

0549

100 kHz

HDL Project

github.com/OnSachinSharma/tt10-verilog-MAC_Accelerator

"Multiplication & accumulation"

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

Project Pinout

Digital Pins

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

4-bit CLA

by **Wei Zhang**

0550

HDL Project

github.com/Electom/tt08_CSA_4bits

"A 4 bit carry look-ahead adder"

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.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	a[0]	s[0]	ci
1	a[1]	s[1]	—
2	a[2]	s[2]	—
3	a[3]	s[3]	—
4	b[0]	co	—
5	b[1]	—	—
6	b[2]	—	—
7	b[3]	—	—

Xor-Logic

by Haohua Li

0551

HDL Project

github.com/eggfly/tt10-xor-logic

"A xor logic from input to output pins"

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.

Project Pinout

Digital Pins

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

SkyKing Demo

by Nicklaus Thompson

0552

25.2 MHz

HDL Project

github.com/FangameEmpire/tt08_skyking

"Types some text over an image of a plane flying into the sunset"

How it works

The demo consists of a static image of a passenger jet flying off into the sunset with a text overlay at the bottom that fills in character-by-character. The text begins typing immediately after reset, so it is likely that the entire text animation will complete before the VGA monitor recognizes the signal. It is best to view this demo in the VGA playground due to the timing issue. The project also includes demos to test some oscilloscope XY display PMODs I'm working on. The demos for these PMODs are both circles and they can be accessed by setting `ui_in[0]` high and using `ui_in[1]` to select the demo.

How to test

The demo runs automatically if all inputs are low. If `ui_in[1:0] = 2'b01`, an unrelated demo for a 1-PMOD XY display driver will play. If `ui_in[1:0] = 2'b11`, a demo for a 2-PMOD XY display driver will play.

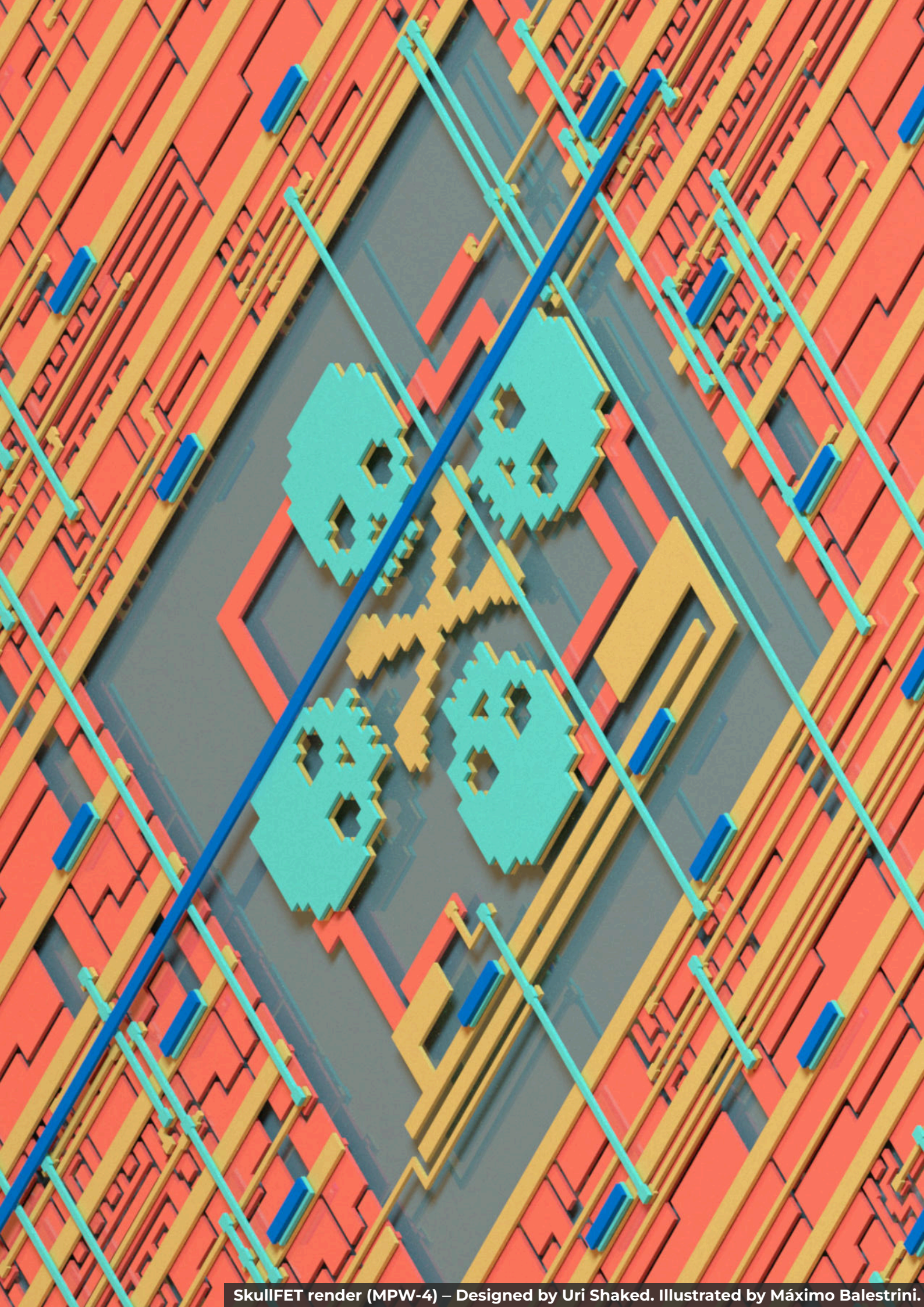
External hardware

The demo requires the Tiny VGA PMOD on UO. The XY demos require either a 1-PMOD driver on UO, or a 2-PMOD driver on UO and UIO. The demo does not include audio.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	0: VGA, 1: XY	<code>ui[1:0] = 0 -> HS, 1 -> Trig, 3 -> Y0</code>	<code>ui[1] = 0 -> 1'b0, 1 -> X0</code>
1	0: XY 1, 1: XY 2	<code>ui[1:0] = 0 -> R0, 1 -> Y5, 3 -> Y2</code>	<code>ui[1] = 0 -> 1'b0, 1 -> X2</code>
2	—	<code>ui[1:0] = 0 -> G0, 1 -> X7, 3 -> Y4</code>	<code>ui[1] = 0 -> 1'b0, 1 -> X4</code>
3	—	<code>ui[1:0] = 0 -> B0, 1 -> X5, 3 -> Y6</code>	<code>ui[1] = 0 -> 1'b0, 1 -> X6</code>
4	—	<code>ui[1:0] = 0 -> VS, 1 -> Y6, 3 -> Y1</code>	<code>ui[1] = 0 -> 1'b0, 1 -> X1</code>
5	—	<code>ui[1:0] = 0 -> R1, 1 -> Y4, 3 -> Y3</code>	<code>ui[1] = 0 -> 1'b0, 1 -> X3</code>
6	—	<code>ui[1:0] = 0 -> G1, 1 -> X6, 3 -> Y5</code>	<code>ui[1] = 0 -> 1'b0, 1 -> X5</code>
7	—	<code>ui[1:0] = 0 -> B1, 1 -> X4, 3 -> Trig</code>	<code>ui[1] = 0 -> 1'b0, 1 -> X7</code>



SkullFET render (MPW-4) – Designed by Uri Shaked. Illustrated by Máximo Balestrini.

Simon Says memory game

by Uri Shaked

0553

50 kHz

HDL Project

github.com/urish/tt10-simon-game

"Repeat the sequence of colors and sounds to win the game"

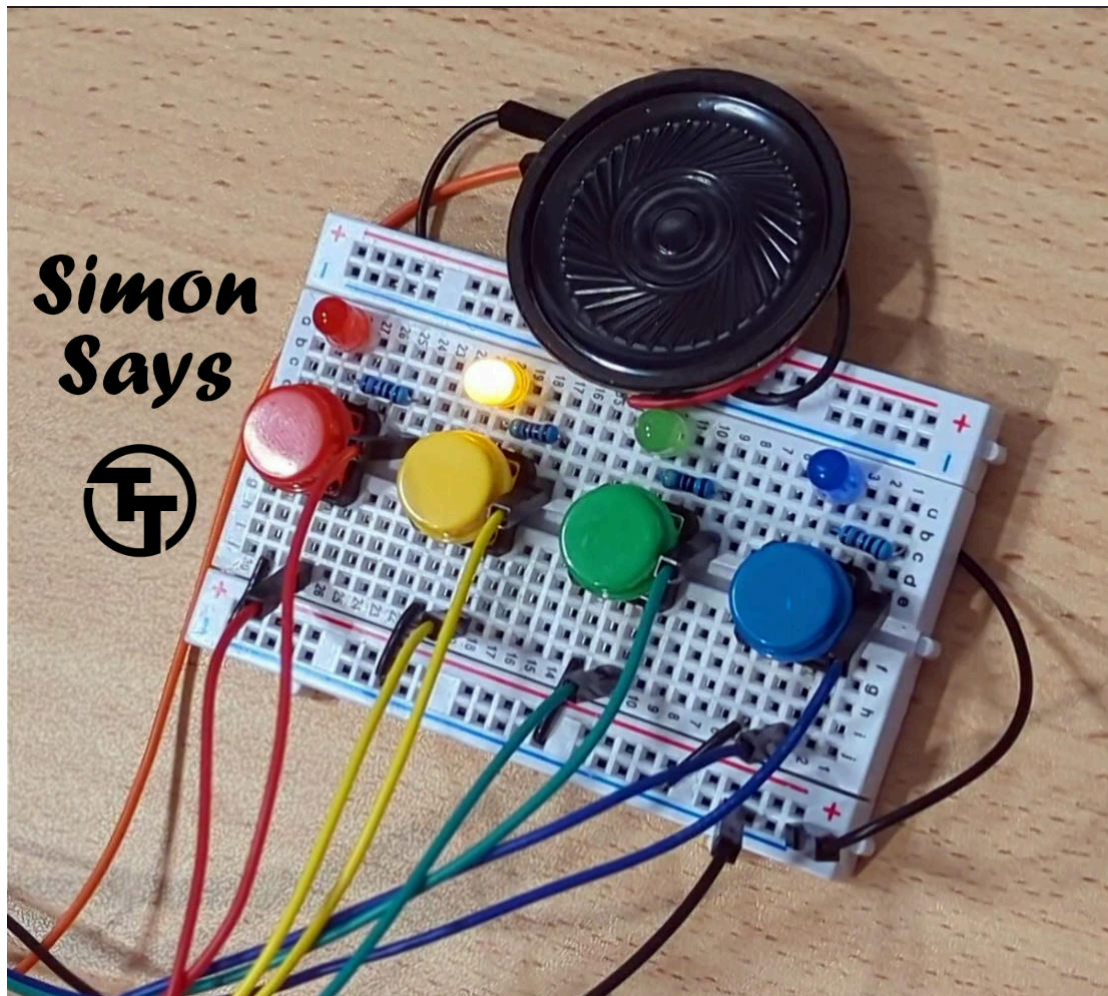


Figure 553.1: Simon Says Game

How it works

Simon says is a simple electronic memory game: the user has to repeat a growing sequence of colors. The sequence is displayed by lighting up the LEDs. Each color also has a corresponding tone.

In each turn, the game will play the sequence, and then wait for the user to repeat the sequence by pressing the buttons according to the color sequence. If the user repeated the sequence correctly, the game will play a

“leveling-up” sound, add a new color at the end of the sequence, and move to the next turn.

The game continues until the user has made a mistake. Then a game over sound is played, and the game restarts.

Check out the online simulation at <https://wokwi.com/projects/408757730664700929> (including wiring diagram).

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 62.5 KHz.

The internal clock is generated by a 13-stage ring oscillator, divided by 8192 to get the desired frequency. The divider value was determined by measuring the output frequency of `tt_um_urish_ringosc_cnt` from Tiny Tapeout 5.

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 `sev_g` to individual segments, `dig1` to the common pin of the tens digit, `dig2` to the common pin of the ones digit. Set `seginv` according to the type of 7 segment display you have: high for common anode, low for common cathode.
5. Reset the game, and then press any button to start it. Enjoy!

External Hardware

[Simon Says Pmod](#) or four push buttons (with pull-down resistors), four LEDs, and optionally a speaker/buzzer and two digit 7-segment display.

Project Pinout

Digital Pins

#	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	—

Flame demo

by **Konrad Beckmann** & **Linus Mårtensson**

0554

25 MHz

HDL Project

github.com/kbeckmann/tt08-flame

"Flame demo"

Flame - Konrad & Linus tinytapeout08 demo compo entry

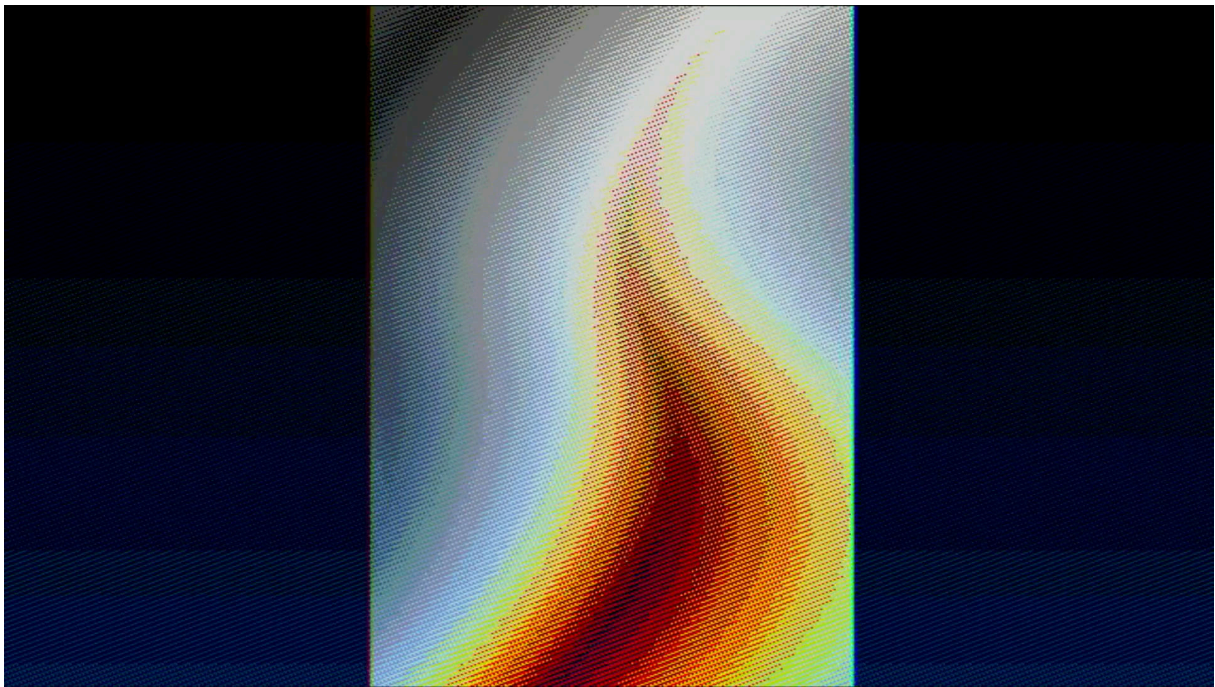


Figure 554.1: preview

How it works

It shows a flame and plays audio. The VGA output is standard [640x480@60Hz](#), audio is simple 1 bit PWM.

How to test

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

External hardware

Follows the [democompo hardware rules](#):

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

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

Project Pinout

Digital Pins

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

SPI 7-segment display

by **Garima Bajpayi**

0555

HDL Project

github.com/garima19bajpayi/tt10-spi-7seg

"A small SPI slave module driving the 7-segment display on the TT-Carrier board"

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	SS	Segment A	—
1	MOSI	Segment B	—
2	—	Segment C	—
3	—	Segment D	—
4	—	Segment E	—
5	—	Segment F	—
6	—	Segment G	—
7	SCLK	Segment DP	—

Metaballs

by Johannes Hoff

0556

50 MHz

HDL Project

github.com/johshoff/tt08-metaballs

"You can't prove it's not metaballs"

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

Project Pinout

Digital Pins

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

AtomNPU

by Aakash Apoorv

0557

100 MHz

HDL Project

github.com/ToyMath/AtomNPU-tt10

"A 4-bit Neural Processing Unit performing multiply-accumulate operations"

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input_data[0]	output_data[0]	weight[0]
1	input_data[1]	output_data[1]	weight[1]
2	input_data[2]	output_data[2]	weight[2]
3	input_data[3]	output_data[3]	weight[3]
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Micro tile container (group 2)

by Uri Shaked

0558

HDL Project

github.com/TinyTapeout/tt08-micro-tiles-container-group2

“Experimental microtile container”

Selecting the active project

Use `uio[1:0]` to select the active micro-tile project.

Project 0 - Test

- Repo: <https://github.com/TinyTapeout/tt-micro-tiles-experiment>
- Author: Uri Shaked
- Description: Micro tiles test module

How it works

The micro tiles test module is a simple module that demonstrates the use of the micro tile interface.

It has two modes of operation:

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

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 observe that the counter is output on the output pins (`uo_out`).

Project 1 - Wokwi Doodle

- Repo: <https://github.com/dlmiles/tt08-micro-wokwi-doodle> (micro)
- Repo: <https://github.com/dlmiles/tt08-wokwi-doodle> (1x1)
- Wokwi: <https://wokwi.com/projects/408272151035187201>
- Module: `tt_um_wokwi_408272151035187201`
- Author: Darryl Miles
- Description: TT08 Wokwi Doodle

How to build instructions in README.md of `tt08-micro-wokwi-doodle`

Picture of circuit also in README.md

How it works

I don't know how it works, it is a doodle.

The aim is to see if a random doodle can be made to count on the 7SEG.

Maybe it can, maybe it can't, let the truth tables work it out.

How to test

Send all possible input combinations to the project and see what happens.

It has never been tested to find out if it is possible to observe a full set of 7SEG font states at the output.

External hardware

None, just the standard Tiny Tapeout PCB.

Project 2 - NCO

- Repo: <https://github.com/gfg-development/tt-micro-tiles-nco>
- Author: Gerrit Grutzeck
- Description: Micro tiles nummerical controlled oscillator, which generates a PDM stream of a sawtooth

How it works

On `ui_in` the desired frequency is set. The highest output frequency is the clock divided by $2^{21}/(2^8 - 1)$. The lowest possible frequency is the clock divided by 2^{21} . First a phase accumulator is used to generate the sawtooth. The resulting waveform is then converted into a PDM stream in a second stage. This generated PDM datastream is output via `uo_out[7]`.

How to test

Connect the audio Pmod to the output ports or any other low pass filter with a speaker to `uo_out[7]`. Then configure the clock to a meaningful frequency (e.g. 50 MHz for frequencies between 6 kHz and 24 Hz). Finally set the `ui_in` pins to the desired frequency ($= \text{clk} / 2^{21} * \text{ui_in}$). After applying the reset, the sawtooth will be generated.

Project 3 - Micro Maze

- Repo: <https://github.com/htfab/micro-maze>
- Author: htfab
- Description: A simple fixed maze game with 7-segment output

How it works

The player can walk around the maze, showing the adjacent walls on the 7-segment display.

How to test

Use the first four inputs to move up, down, left or right.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	in[0]	out[0]	sel[0]
1	in[1]	out[1]	sel[1]
2	in[2]	out[2]	—
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]	—

Tiny Neural Network Accelerator

by **Greg Chadwick**

0582

50 MHz

HDL Project

github.com/GregAC/tt10-tiny-nn

"A toy neural network accelerator targetting CNNs"

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'hX

- 16'hX
- 16'hX
- 16'hX
- 16'hX
- 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.

Project Pinout

Digital Pins

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

SIC-1 8-bit SUBLEQ Single Instruction Computer

by Uri Shaked

0590

HDL Project

github.com/urish/tt09-sic1

"Hardware implementation of the 8-bit Single Instruction Computer"

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 uio pins are used to load a program into the computer, and to control the computer:

uio 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 (uio pins)
5	dbg[0]	output	Debug select bit 0
6	dbg[1]	output	Debug select bit 1
7	dbg[2]	output	Debug select bit 2

Debug interface

The dbg pins are used to expose internal signals for debugging on the uio pins. When the dbg pins are set to 0, the uio pins will output the @OUT value. For other values of dbg, the uio pins will output the following signals:

dbg[2:0]	Signal
0	None
1	PC

2	A
3	B
4	C
5	valA
6	result (valA - valB)
7	state (3 bits)

The `state` signal is a 3-bit value that represents the current state of the computer, corresponding to the execution cycle stages described above (1-6), and a 3-bit value of 0 when the computer is halted.

Programming the SIC-1

You can use the [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 `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.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	in[0]	out[0]	run
1	in[1]	out[1]	halted
2	in[2]	out[2]	set_pc
3	in[3]	out[3]	load_data
4	in[4]	out[4]	out_strobe

#	Input	Output	Bidirectional
5	in[5]	out[5]	dbg[0]
6	in[6]	out[6]	dbg[1]
7	in[7]	out[7]	dbg[2]

TinyFPGA resubmit for TT08

by **Emilian Miron**

0608

HDL Project

github.com/diferential/tt08-muxpga

"TinyFPGA"

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.

Project Pinout

Digital Pins

#	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

VSLC - Very Small Logic Controller with a timer and servo control

by James Keener

0609

12 MHz

HDL Project

github.com/jimktrains/vslc-tto

"A small PLC-like controller"

How it works

The VSLC (Very Small Logic Controller) is a stack machine with 8 inputs and 8 outputs and two timers. It takes inspiration from relay logic, early PLCs and their processors, such as the Motorola [MC14500B](#) (known to me through the [UE-1/UEVTC/UE1450](#)). While not the first time I heard of relay logic, one of the more in depth views of it I've had was from [Technology Connections](#) video series on the [logic inside an William's 1976 Aztec pinball machine](#)

Another not-my-first-introduction, but Ben Eater's series on building a [6502-based computer](#) and an [8-bit TTL computer](#) reinvigorated my interest in electronics and the low-level aspects of computer hardware.

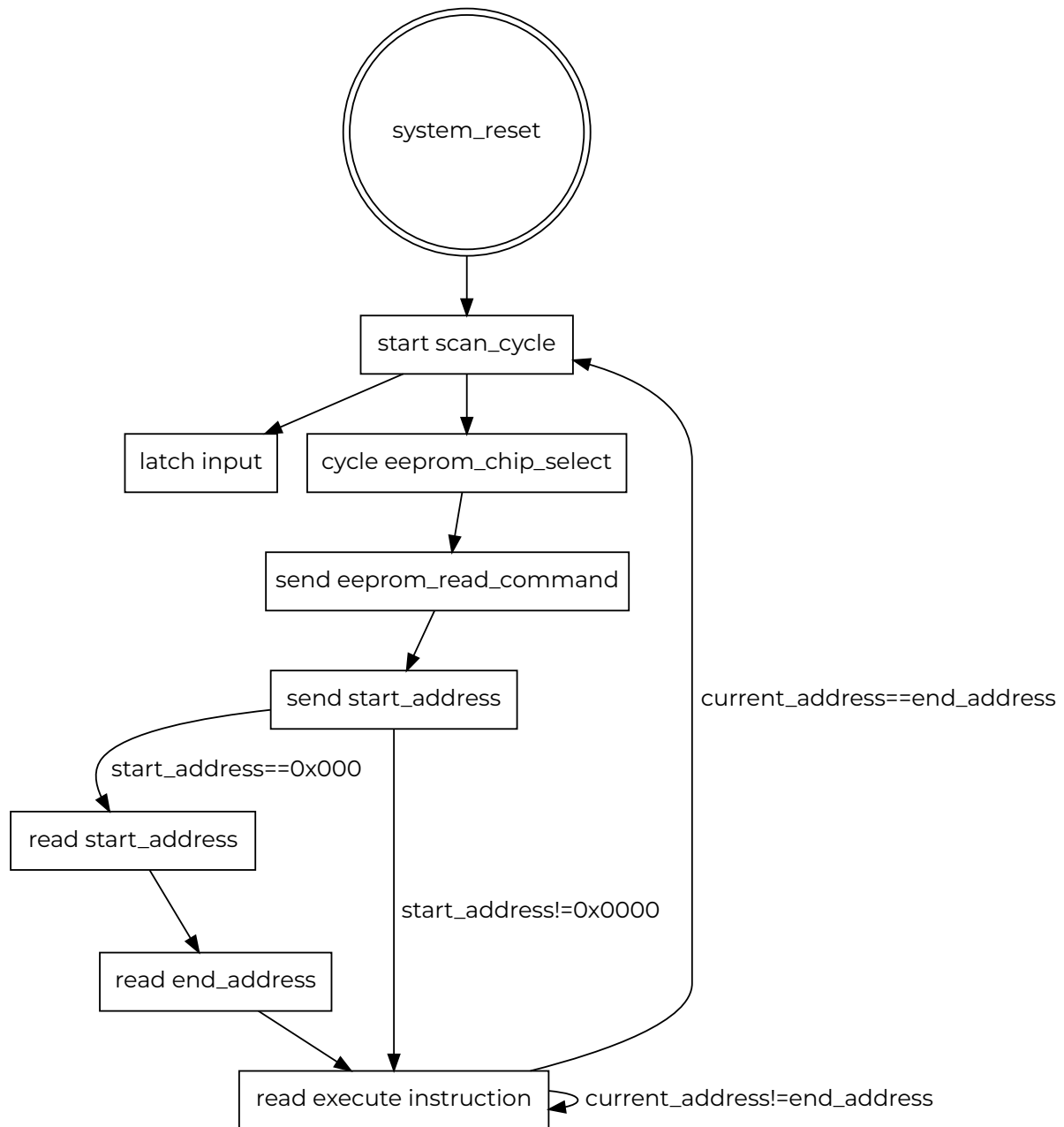
My design goal was to have a small processor that could be programmed to handle small control tasks. Specifically, I wanted something that could run crossing gates on my model train layout. This requires sensing rising and falling edges of sensors, flashing lights, and actuating a servo.

Given the relay and PLC "heritage", I reference "scan cycles", "setting", and "resetting" registers. The scan cycle is one trip through the program and sees the input as it was at the start of the cycle. "Setting" a register turns it "on"/sets it to 1. "Resetting" is turning it off/setting it to 0. I try to refer to cycling the reset line of the module, i.e. `rst_n` as a "system reset" to avoid confusion. I am not a professional in industrial automation, just a hobbyist, if I'm using any terms wrong or confusingly please let me know.

Once powered on, the controller needs to be held in a system reset for at least a cycle.

On system reset, the start address is set to 0x0000. The controller will send a READ command (0x03), a 16-bit 0 address, and expects to receive a 2 byte scan cycle vector (i.e. the first instruction for subsequent runs), a 2 byte scan cycle end address (i.e. the last valid instruction), followed by instructions. Once the end address is reached a new scan cycle is initiated. A scan cycle begins by latching input while cycling the EEPROM chip select for 1 cycle, followed by a READ command (0x03) and the 16-bit cycle start vector.

During a cycle, the value of an input will be unchanging even if the actual input to the pin changes during the execution of the cycle. **RISING** and **FALLING** are defined as changes between cycle starts. If a full cycle happens during cycle execution, it will not be seen by the program.



How to test

Place a program on EEPROM (or an emulator), and use the IO pins.

Example 0 is a short and a good place to start.

The file [examples/prog0.vslc](#) is:

```

PUSH i0
DUP
POP o0
NOT
PUSH o1

```

This program begins by taking the value of input 0 at the last positive edge of the scan cycle clock and placing it on the stack.

```
top of stack: [i0](i0)
```

It then duplicates that value.

```
top of stack: [i0](i0) [i0](i0)
```

followed by popping one instance from the top of the stack and placing it in the output 0 register.

```
top of stack: [i0](i0)
```

The top of the stack is then inverted.

```
top of stack: /[i0](i0)
```

Finally, that value is popped to output register 1.

If `o0` and `o1` were attached to active-high LEDs and `i0` attached to an active high momentary button, one LED would be on when the button isn't pressed. Once the button is pressed, that LED turns off and the other turns on. When the button is let go, the lights toggle.

To assemble the source code, we can run `./vslcc examples/prog0.vslc` and get the output:

```

line_num=1 line='PUSH i0'
  cur_addr=00 => PUSH I0 => 00(00000000)
line_num=2 line='DUP'
  cur_addr=01 => DUP => b5(10110101)
line_num=3 line='POP o0'
  cur_addr=02 => POP O0 => 18(00011000)
line_num=4 line='NOT'
  cur_addr=03 => NOT => 8a(10001010)
line_num=5 line='PUSH o1'
  cur_addr=04 => PUSH O1 => 09(00001001)
cur_addr=9
start_addr=b'\x00\x04' end_addr=b'\x00\x08'

```

The compiler/assembler generates `examples/prog0.eeprom.bin`. Checking that it exists with `ls -l examples/prog0.eeprom.bin`

```
-rw-rw-r-- 1 jim jim 1024 Feb 13 20:28 examples/prog0.eeprom.bin
```

shows us that the file is 1kB (8kb) in size! The compiler generates a file adequate for a 8kb EEPROM, and pads the rest of the file with `0xff`.

We can then look at the generated output with `xxd examples/prog0.eeprom.bin | head -n 2`

```
00000000: 0004 0008 00b5 188a 09ff ffff ffff ffff .....  
00000010: ffff ffff ffff ffff ffff ffff ffff ffff .....
```

we see the padded `0xffs`. The actual program also seems to be 4 bytes longer than we would expect, as our program is 5 instructions long. The first 16-bits are the start address for a new cycle. The second 16-bits are the last valid program address, after which a new cycle should be started.

There is also a disassembler. We can try `./vs1cd examples/prog0.eeprom.bin` and get

```
; starting address is the default  
; .start 4  
PUSH i0  
DUP  
POP o1  
NOT  
PUSH o2  
; ending address is the default  
; .end 8
```

Nothing too exciting.

Once we start the controller, it will cycle the EEPROM chip select, start a scan cycle which will latch the input, send a read command (0x03) for address 0x0000, read the starting address of 0x0004, then the end address of 0x0008 then the end address of 0x0008. It will then begin executing the next byte it receives. It will continue to send the SPI clock and expect bytes in return. Once it has executed the instruction at the ending address, the EEPROM chip select is cycled and a new scan cycle is started which latches the input. The controller will then send the read command (0x03) for address 0x0004 and begin execution at the first byte it receives. This will continue until the controller is powered down.

External hardware

The controller expects something that presents as an EEPROM that accepts 16-bit addresses and will provide continuous data while the clock and chip select are active.

General Architecture

- Stack machine
- Code executes until the end of code memory is reached, in which case a new cycle is started

- 8 input bits/registers mapped to pins
- 8 output bits/registers mapped to pins
- 16 bit data stack
- 1 timer/counter with clock divisor
- 1 timer/counter for driving a servo

Opcodes

General Decoding rules

- Bit 7
- 0: Register Operations
- Bit 6
 - 00: IO Registers
 - 01: Special Function Registers (SFR)
- Bit 5,4 * 00 Push * 01 Pop * 10 Set * 11 Reset
- 1: Operations
- Bit 6
- 0: Logical Operations These all write to the top of stack. The shifts below are the final shifts, not the number of pops before the operation pushes. e.g. NOT conceptually does a pop and then a push, which is a shift of 0. *
 - Bit 5,4
 - 00: No Shift
 - 01: Shift 1 Right
 - 10: Reserved
 - 11: Shift 1 Left * Bits 3,2,1,0: Truth table. The result is bit 3 – {nos, tos}
- 1: Other * Bit 5
 - 0: Temporal (Rising/Falling)
 - Bit 4 - Expected previous state
 - 1: Other
 - Bit 4:
 - 0: Set Parameters
 - 1: Other

Register Operations

PUSH reg	0000	IRRR
POP reg	0001	1RRR
SET reg	0010	1RRR
RESET reg	0011	1RRR
PUSH sfr	0100	RRRR
POP sfr	0101	RRRR
SET sfr	0110	RRRR
RESET sfr	0111	RRRR

Logical Operations

AND	1010	0001
NAND	1010	1110

OR	1010 0111
NOR	1010 1000
XOR	1010 0110
BICOND	1010 1001
IMPL	1010 1101
NIMPL	1010 0010
CONV	1010 1011
NCONV	1010 0100

DUP	1000 1100
OVER	1000 1010
DROP	1010 1010
ZERO	1011 0000
ONE	1011 1111
NOT	1001 0011
OVERNOT	1010 0101

Temporal Operations

RISING reg 1100 IRRR
 FALLING reg 1101 IRRR

Parameters

These are followed by a byte with the value to set

SPARAM0 parm 1110 0PPP XXXX XXXX
 SPARAM1 parm 1110 1PPP XXXX XXXX

Other

CLR	1111 0000
SETALL	1111 0001
SWAP	1111 0010
ROT	1111 0011
NOP	1111 1111

Special Function Registers SFR

- 0x0 Timer Enable
- 0x1 Timer Output
- 0x2 Servo Enable
- 0x3 Servo Value
- 0x4 Servo Output

Parameter

- 0x0 SPI Clock Divider (Not Implemented)
- 0x1 Timer0 Clock Divider (4 bits)
- 0x2 Timer0 Counter A (Not Implemented)
- 0x3 Timer0 Counter B (Not Implemented)
- 0x4 Servo0 Clock Divider (Not Implemented)
- 0x5 Servo0 Frequency Value (Not Implemented)
- 0x6 Servo0 Reset Value (Not Implemented)

- 0x7 Servo0 Set Value (4 bits)
-

Timer

Due to space constraints, the timer will only output a 50% duty cycle signal and has a period of 366 timer-ticks long. The clock divider for the timer can be adjusted from 0 (input clock) to 16 (input clock / 216). At 12MHz, the default clock divider of 14 cycles the timer at 2Hz.

The timer's output is always available on `uio_out[4]`. The complement is available on `uio_out[5]`. There is a cycle of dead-time inserted between transitions.

SFR

- 0x00 Timer Enable - When set the timer will count
- 0x01 Timer Output - The current output of the timer (even when disabled). This is available to the program.

Parameters

- 0x1 Timer0 Clock Divider (4 bits)
 - 0x2 Timer0 Counter A (Not Implemented)
 - 0x3 Timer0 Counter B (Not Implemented)
-

Servo

The servo timer is designed to produce a 50Hz with a duty-cycle between 5% and 13% or pulse-length of between 1ms and 2.6ms. The servo can be set to one of two positions based on the value of the servo's value SFR. Due to space constraints, only the set value (i.e. when the servo's value is 1) can be changed. The reset value is fixed at 11 (1ms pulse). The set value can be set to any value up to 31 (2.6ms pulse). The default set value is 23 (1.9ms pulse). Servos typically accept between 1ms and 2ms to go from 0° and 180°. Practically, this leaves 12 steps (of 15° each) between the servo's home position and max rotation.

The servo's output is always available on `uio_out[6]`;

SFR

- 0x2 Servo Enable - Turns the servo counter on when set.
- 0x3 Servo Value - Place the servo at either its set or reset position.
- 0x4 Servo Output - The output of the timer. (Even when the servo is disabled.) This is available to the program.

Parameter

- 0x4 Servo0 Clock Divider (Not Implemented)
- 0x5 Servo0 Frequency Value (Not Implemented)

- 0x6 Servo0 Reset Value (Not Implemented)
 - 0x7 Servo0 Set Value (4 bits)
-

Assembler

The compiler will generate a 1kB `.eeprom.bin` containing the code, and a `.final.vslc` with the post-processed code.

Comments

Anything after a semicolon, `;` on a line will be ignored.

.define

```
.define alias original
```

Allows aliasing values and register names.

Example

```
.define leftsensor i0
push leftsensor
```

is equivalent to

```
push i0
```

.logic

Allows writing propositional logic statements that will compile to assembly. Parentheses, `(` and `)`, logical-and `&`, logical-or `|`, xor `^`, and logical-not `~` are supported. Other values are assumed to be register names and may be aliased via a `.define`.

Example

See [examples/prog2.vslc](#) and [examples/prog2.final.vslc](#) for an example of implementing a counter and 7-segment decoder.

```
.define segC o2
```

```
.define t0 s6
.define t1 s7
.define t2 s8
.define t3 s9
```

```
.logic (~t3 & t2) | (t3 & ~t2) | (~t1 & t0) | (~t2 & ~t1) | (~t2 & t0)
```

```
POP segC
```

will compile to

```
PUSH S9
NOT
PUSH S8
```

```

AND
PUSH S9
PUSH S8
NOT
AND
PUSH S7
NOT
PUSH S6
AND
PUSH S8
NOT
PUSH S7
NOT
AND
PUSH S8
NOT
PUSH S6
AND
OR
OR
OR
OR
POP o2

```

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Input	Output	EEPROM /CS
1	Input	Output	SPI COPI
2	Input	Output	SPI CIPO
3	Input	Output	SPI CLK
4	Input	Output	Timer Output
5	Input	Output	Complementary Timer Output
6	Input	Output	Servo Output
7	Input	Output	(unused)

Dummy Counter

by Chinmay

0610

HDL Project

github.com/pyamnihc/tt08-dummy-counter

"A 16-bit counter"

How it works

Like a 16-bit counter

How to test

Like a 16-bit counter

External hardware

NA

Project Pinout

Digital Pins

#	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

Gamepad Pmod Demo

by Uri Shaked

0611

25.175 MHz

HDL Project

github.com/urish/tt10-gamepad-pmod-demo

"Gamepad Pmod + Tiny VGA demo from VGA Playground"

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](#)

Project Pinout

Digital Pins

#	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	—

Why not?

by **sylefeb**

0612

25 MHz

HDL Project

github.com/sylefeb/tt08-fun

"One tile something"

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

Project Pinout

Digital Pins

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

VGA Screensaver with Tiny Tapeout Logo

by Uri Shaked

6613

25.175 MHz

HDL Project

github.com/TinyTapeout/tt10-logo-screensaver

"Tiny Tapeout Logo bouncing around the screen (640x480, TinyVGA Pmod)"

How it works

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



Figure 613.1: Tiny Tapeout screensaver

How to test

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

- `tile` (`ui_in[0]`) to repeat the logo and tile it across the screen,
- `solid_color` (`ui_in[1]`) 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](#)

Project Pinout

Digital Pins

#	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	—

Counter

by **Jasmin Mittelman**

0614

50 kHz

Wokwi Project

github.com/jasmin1659/tt08-wokwi-jasmin

wokwi.com/projects/408237988946759681

“Counts till 99 when button is pressed and shows light depending if counter is even or odd.”

How it works

There are three modules: MyCounter (tests if the button is pressed or released), score (receives the increment signal from MyCounter and sends the 7 segment information back to Wokwi module), and Wokwi (Takes the inputs from the buttons and clock. Then it sends the information to MyCounter. Next, it sends the information received by MyCounter back out to the 7 segment display and LEDs).

How to test

1. Click the red button to increment the counter
2. Blue LED indicates the counter is an even number
3. Red LED indicates the counter is an odd number
4. Press the black reset button to reset the counter back to 0

External hardware

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	CLK	LED0	SEG_G
1	RST	LED1	DIG1
2	BTN0	SEG_A	DIG2
3	—	SEG_B	—
4	—	SEG_C	—
5	—	SEG_D	—
6	—	SEG_E	—
7	—	SEG_F	—

SPI test

by **Matt Venn**

0615

50 MHz

HDL Project

github.com/mattvenn/tt10-spi-test

"SPI test design"

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	cpol	—	—
1	cpha	—	—
2	—	—	—
3	—	—	spi_miso
4	—	—	spi_cs_n
5	—	—	spi_clk
6	—	—	spi_mosi
7	—	—	—

RGB Mixer

by Tianmin (Kevin) Kong

0616

10 MHz

HDL Project

github.com/ktm0228/tt08-rgb-mixer

"First ASIC Project!"

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.

Project Pinout

Digital Pins

#	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

Huffmann_Coder

by Marvin Barth

0617

HDL Project

github.com/MarvinBrth/tt10-huffman-coder

“compresses ASCII input into Huffman codes”

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.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	ASCII_in[0]	Huffman_out[0]	Huffman_out[8]
1	ASCII_in[1]	Huffman_out[1]	Huffman_out[9]
2	ASCII_in[2]	Huffman_out[2]	Valid_out
3	ASCII_in[3]	Huffman_out[3]	Bit_length[0]
4	ASCII_in[4]	Huffman_out[4]	Bit_length[1]
5	ASCII_in[5]	Huffman_out[5]	Bit_length[2]
6	ASCII_in[6]	Huffman_out[6]	Bit_length[3]
7	Load	Huffman_out[7]	—

RGB Mixer demo

by **Matt Venn**

0618

10 MHz

HDL Project

github.com/mattvenn/tt08-rgb-mixer

“Zero to ASIC demo project”

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.

Project Pinout

Digital Pins

#	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

Vedic multiplier

by Vivek Chiranjit

0619

HDL Project

github.com/ChiranjitPatel/tt10-multiplier_UART_SPI

"8-bit binary unsigned multiplier"

How it works

This is a 8-bit binary multiplier

Inputs: mul_ip_A mul_ip_B

Ouputs: prod_low
prod_high

How to test

Give the two 8-bit inputs thorough ui_in and uio_in. The multiplied data will be received at uo_out and uio_out

External hardware

None

Project Pinout

Digital Pins

#	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

VGA clock

by **Matt Venn**

0620

31.5 MHz

HDL Project

github.com/mattvenn/tt08-vga-clock

"Shows the time on a VGA screen"

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

Project Pinout

Digital Pins

#	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	—

NAND

by **kofi**

0621

Wokwi Project

github.com/filokot/TinyTapeoutDesign

wokwi.com/projects/422964384478997505

“NAND gate”

How it works

The setup has two inputs and a single output. The setup consists of a NOT gate and a NAND gate, where the NOT gate is connected to the first input and feed of one pin of the NAND gate. The second input of the setup is connected to the second input of the NAND gate.

How to test

Send the following signals to IN2 and IN3 to check if the proper NAND output results: INs Out 00 1 10 1 01 0 11 1

External hardware

You can test the chip with LEDs or other indicators.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	IN2	OUT3	—
1	IN3	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

synth_simple

by MM

0622

50 MHz

HDL Project

github.com/maurizio-martina/synth_simple

"Simple monophonic synth with PWM output"

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	note_enn_i[0]	pwm_left	sw_i[0]
1	note_enn_i[1]	pwm_right	sw_i[1]
2	note_enn_i[2]	—	sw_i[2]
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]

DaliaProjekt

by Dalia

0623

Wokwi Project

github.com/DaliaPireh/-U00RH8C_Dalia_Wokwi

wokwi.com/projects/422964754310747137

“LearningToDesign”

How it works

It is just a test to try fabricating a chip, special function other than som simple logics applied

How to test

It is just a test to try fabricating a chip, special function other than som simple logics applied, see if the logics actually give the desired output

External hardware

It is just a test to try fabricating a chip, special function other than som simple logics applied, no extra HW

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	IN1	—	—
1	—	—	—
2	IN3	—	—
3	—	OUT3	—
4	IN4	—	—
5	—	OUT5	—
6	IN6	—	—
7	IN7	OUT7	—

VGA donut

by **Andy Sloane**

0642

48 MHz

HDL Project

github.com/a1k0n/tt08-vga-donut

"Renders a 3D torus on a VGA display"

VGA Donut

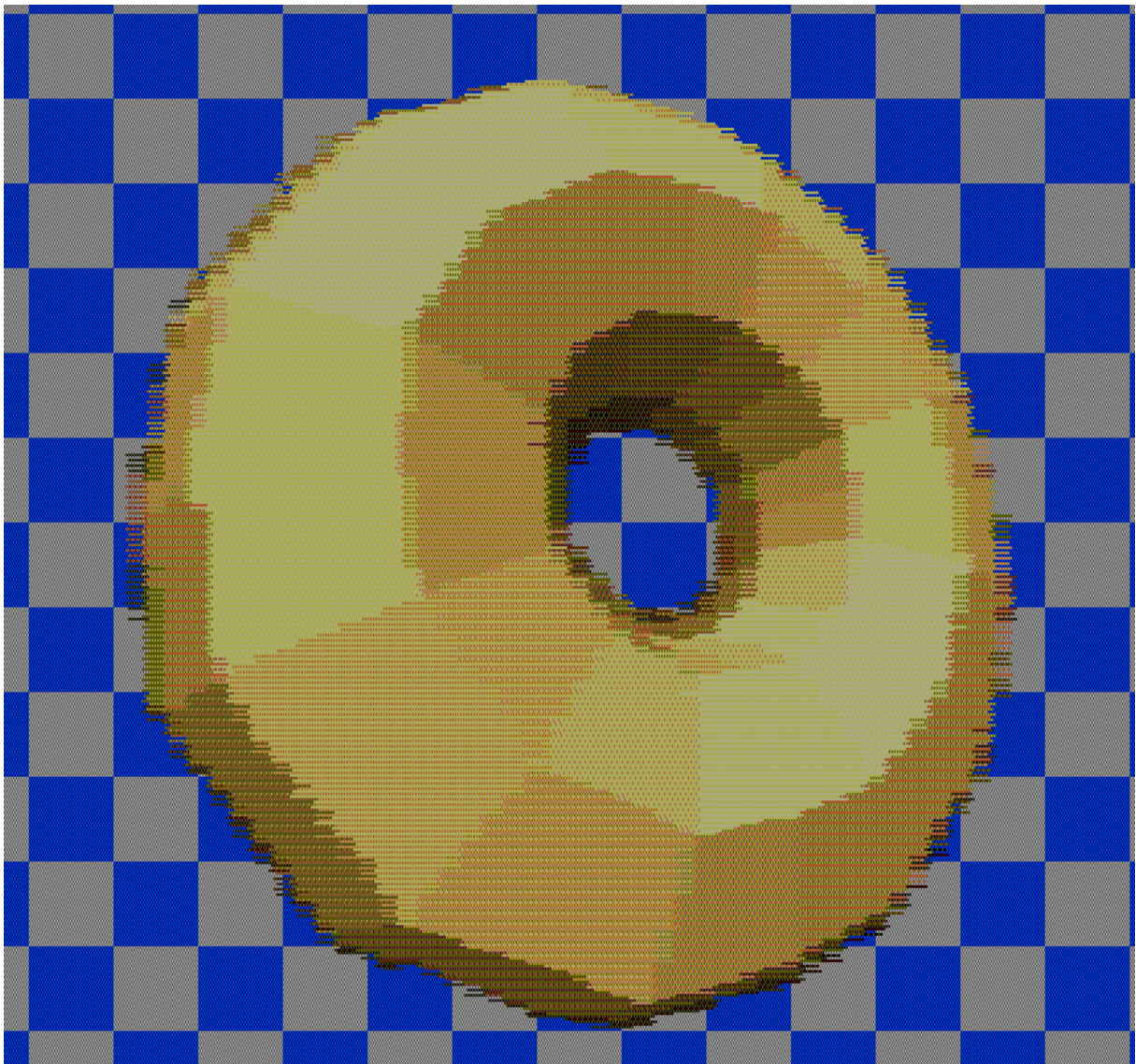


Figure 642.1: preview

How it works

Renders a faceted donut to a VGA monitor.

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

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

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

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

How to test

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

External hardware

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

Project Pinout

Digital Pins

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

raybox-zero TTCAD25a edition

by algofoogle (Anton Maurovic)

654

25.175 MHz

HDL Project

github.com/algofoogle/ttcad25a-raybox-zero

"TTCAD25a v1.7 resub of 'simple VGA ray caster game demo' from TT07"

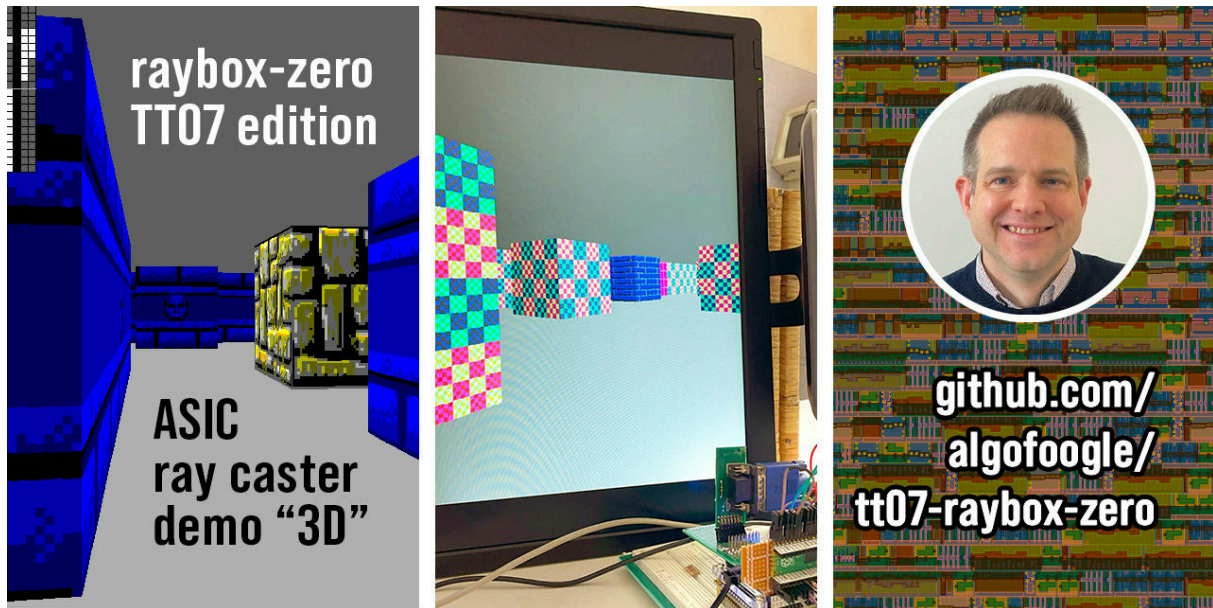


Figure 654.1: TT07 raybox-zero showing 3D views in simulation and on an FPGA

How it works

This is a sky130 resubmission of [ttihp0p2-raybox-zero](https://github.com/algofoogle/ttihp0p2-raybox-zero) (an updated version of [tt07-raybox-zero](https://github.com/algofoogle/tt07-raybox-zero), from the [raybox-zero](https://github.com/algofoogle/raybox-zero) project).

This resubmission of [tt07-raybox-zero](https://github.com/algofoogle/tt07-raybox-zero) on TTIHP0p2 is a framebuffer-less VGA display generator (i.e. it is 'racing the beam') that produces a simple implementation of a "3D"-like ray casting game engine... just the graphics part of it. It is inspired by Wolfenstein 3D, using a map that is a grid of wall blocks, with basic texture mapping.

There is nothing yet but textured walls, and flat-coloured floor and ceiling. No doors or sprites, sorry. Maybe that will come in a future version.

The 'player' POV ("point of view") registers allow the player position, facing X/Y vector, and viewplane X/Y vector in one go, and (along with other visual effects registers) are controlled by a single SPI interface.

NOTE: To optimise the design and make it work without a framebuffer, this renders what is effectively a portrait view, rotated. A portrait monitor (i.e. one

rotated 90 degrees anti-clockwise) will display this like the conventional first-person shooter view, but it could still be used in a conventional landscape orientation if you imagine it is for a game where you have a first-person perspective of a flat 2D platformer, endless runner, “Descent-style” game, whatever.

TBC. Please contact me if you want to know something in particular and I’ll put it into the documentation!

How to test

TBC. Please contact me if you want to know something in particular and I’ll put it into the documentation!

Supply a clock in the range of 21-31.5MHz; 25.175MHz is ideal because this is meant to be “standard” VGA [640x480@59.94Hz](#).

Start with `gen_texb` set low, to use internally-generated textures. You can optionally attach an external QSPI memory (`tex_ . . .`) for texture data instead, and then set `gen_texb` high to use it.

`debug` can be asserted to show current state of POV (point-of-view) registers, which might come in handy when trying to debug SPI writes.

If `reg` input is high, VGA outputs are registered. Otherwise, they are just as they come out of internal combo logic. I’ve done it this way so I can test the difference (if any).

`inc_px` and `inc_py` can be set high to continuously increment their respective player X/Y position register. Normally the registers should be updated via SPI, but this allows someone to at least see a demo in action without having to implement the SPI host controller. NOTE: Using either of these will suspend POV updates via SPI.

Unlike the TT07 version, this one combines the two separate SPI peripheral interfaces into one, allowing both the POV and other registers to be updated from the same interface.

External hardware

Tiny VGA PMOD on dedicated outputs (`uo`).

Optional SPI controllers to drive `ui_in[2:0]`.

Optional external SPI ROM for textures.

TBC. Please contact me if you want to know something in particular and I’ll put it into the documentation!

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	spi_sck	red[1]	Out: digilent_tex_csb / Out: moser_tex_csb
1	spi_sdi	green[1]	I/O: digilent_tex_io0 / I/O: moser_tex_io0
2	spi_csb	blue[1]	In: digilent_tex_io1 / In: moser_tex_io1
3	debug	vsync_n	Out: digilent_tex_sclk / Out: moser_tex_sclk
4	inc_px	red[0]	In: SPARE / In: moser_tex_io2
5	inc_py	green[0]	In: gen_texb / In: moser_tex_io3
6	reg	blue[0]	In: digilent_tex_io2 / N/A: moser_CS1
7	tex_pmod_type	hsync_n	In: digilent_tex_io3 / N/A: moser_CS2

Tiny piano

by **Kenneth Petersen**

0672

10 MHz

HDL Project

github.com/kentrane/tt-tiny_spectrum

"A tiny musical note generator with 16 notes across 4 octaves and tremolo effect"

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)

Project Pinout

Digital Pins

#	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	—



my_own_chip

by Abdala

0673

Wokwi Project

github.com/Abdala-Alhourri-Nordicclays/My_own_chip

wokwi.com/projects/422959974126748673

“Build in tiny tapeout workshop”

How it works

you input the secret code and the 7-segment display will display an “A”

How to test

try to guess the secret code

External hardware

Empty

Project Pinout

Digital Pins

#	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	—

4 Bit Adder with Overflow Counter

by Yousif

0674

Wokwi Project

github.com/Yousif-dk/Tinytapeout

wokwi.com/projects/422962760920307713

"Standard 4 bit adder"

How it works

This is a simple 4 bit adder circuit. I built it out of the following digital logic gates: AND XOR OR It has a 5 bit output to handle overflow.

How to test

The inputs are elementwise pairs. So the first two input pins will be the least significant bit of both numbers. The next two input pins will be the second least significant bit of both numbers.

If you'd like to test it, two a 4 bit number addition on paper, and try to do it using the circuit. the results should be the same

External hardware

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

Project Pinout

Digital Pins

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

Hero on Tape

by **Marcus Sand**

0675

HDL Project

github.com/HeroGamers/Tiny-Tapeout-Hero

“Echoes out a predefined text onto a 16x2 character LCD.”

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)

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	CLK	RS	—
1	—	E	—
2	—	D4	—
3	—	D5	—
4	—	D6	—
5	—	D7	—
6	—	—	—
7	—	—	—

First design

by **Mathias Vestergaard**

0676

Wokwi Project

github.com/Mathiasvest2109/Tiny-Tapeout-Template

wokwi.com/projects/422960174616660993

“Very simple project to display mirrored 3 (M) on a 7-seg display”

How it works

By making 1,3,5,7 switches on the dip-switch-8 component high, we display a mirrored 3 (‘M’ sideways) on the 7-segment display. An LED will light as well when the correct sequence is inputted.

How to test

Flip the 1,3,5,7 switches to display the mirrored 3 and light the LED

External hardware

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

Project Pinout

Digital Pins

#	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	—

digital LIF Neuron

by Kosmas Wernhard

0677

HDL Project

github.com/Strider93/tt_verilog_test

"digital LIF Neuron"

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Weight0	Spike_OUT	—
1	Weight1	—	—
2	Weight2	—	—
3	Weight3	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	Spike_IN	—	—

example1

by Hassan Sirelkhatim

0678

Wokwi Project

github.com/sirelkhatim/tapeout-sirelkhatim

wokwi.com/projects/422959954857061377

"Wokwi diagram"

How it works

This is a series of NOT gates and AND gate.

How to test

Just flip some switches until you are able to see an 8

External hardware

I am not sure

Project Pinout

Digital Pins

#	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	—

ANDNOT

by HKG

0679

10 kHz

Wokwi Project

github.com/GattyInstruments/TinyTapeout20250215

wokwi.com/projects/422960491546730497

“NOTAND”

How it works

AND and NOT and FlipFlop logic gates

How to test

LEDs

External hardware

LED display

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	INO	OUT0	—
1	IN1	OUT1	—
2	IN2	OUT2	—
3	IN3	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

test/15/02/25

by Viktor Hougaard Jørgensen

0680

Wokwi Project

github.com/ViktorHJ/TinyTapeoutGDS_pipeline

wokwi.com/projects/422964381148718081

"Nothing just yet"

How it works

It doesn't work.

How to test

You Can't.

External hardware

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

Project Pinout

Digital Pins

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

Holm's TinyTapeOut 4-bit adder

by Jakob Holm

0681

Wokwi Project

github.com/jaholmDTU/TinyTapeOutChip

wokwi.com/projects/422962959838345217

"4-bit adder, outputting 5-bit number"

How it works

4-bit adder outputting a 5-bit number. Two input numbers: IN0-IN3 and IN4-IN6 Output bits are OUT0-OUT4

How to test

Enable the inputs to create two numbers. See if output matches.

External hardware

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

Project Pinout

Digital Pins

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

Extremely cool stuff (secret)

by **Alexander Aakersø**

0682

Wokwi Project

github.com/WAAALex1/TT-Workshop

wokwi.com/projects/422957954050029569

"4bit ripple carry adder without carry-in"

How it works :)

Simple 4 bit carry adder with no initial carry-in. Inputs A/B0 reference bit 0, A/B1 reference bit 1 and so forth. Output O0 reference output bit 0, O1 reference output bit 1 and so forth.

How to test

Simply connect VCC to some of the input pins and scope the output pins.

External hardware

Attach one LED to each output - dependent on the choice of LED a resistor in series can also be included. To make testing easier, switches can be connected to the input pins.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	A0	O0	—
1	B0	O1	—
2	A1	O2	—
3	B1	O3	—
4	A2	O4	—
5	B2	—	—
6	A3	—	—
7	B3	—	—

mp_LIF_neuron

by **Andreas Schorer**

0683

HDL Project

github.com/schorand/LIF_CORE

"Mixed precision Leaky integrate and fire (LIF) neuron"

How it works

This project implements a mixed precision leaky integrate 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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	w0	spk_upper	ready
1	w1	spk_lower	—
2	w2	—	—
3	w3	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Tinytapeout_design_ANP

by Ashrfun Naher Pinky

0684

10 kHz

Wokwi Project

github.com/Ashrafun-naher-pinky/Tinytapout_design_anp

wokwi.com/projects/422961309631153153

"Show a letter"

How it works

Its a logic circuit which Output the letter a all the time in 7 segment display. When any input is on then the output changes.

How to test

When the clock running it should show A. If it any input turned on. It will change the display

External hardware

LED display

Project Pinout

Digital Pins

#	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	—

Full Adder

by May Wang

0685

Wokwi Project

github.com/wwwe18/tt10-wokwi

wokwi.com/projects/422957657550394369

"Full Adder"

How it works

Full adder with input A, B, Cin and output S, Cout

How to test

A B Cin S Cout 0 0 0 0 0 0 1 1 0 0 1 0 1 0 0 1 1 0 1 1 0 0 1 0 1 0 1 0 1 1 1 1 1 1

External hardware

LED

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	A	S	—
1	B	—	—
2	Cin	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	Cout	—

TinyTapeout 4 bit ripple carry adder

by Georg Brink Dyvad

0686

Wokwi Project

github.com/GorgTheFrog/Tiny-Tapeout-Adder

wokwi.com/projects/422957918936350721

“An adder made of 4 full-adders with the first full-adders c_in connected to ground.”

How it works

The project adds two 4-bit numbers and outputs a 5-bit number as the result.

How to test

Inputs A0, A1, A2, and A3 are the bits for the first number to be added. A0 is the LSB. Same logic for the B input. The S0, S1, S2 and S3 output are the first 4 bits of the result with the 5th bit named C_OUT since its the carry-out.

External hardware

I used LED's for the output and switches for the inputs.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	A0	S0	—
1	A1	S1	—
2	A2	S2	—
3	A3	S3	—
4	B0	C_OUT	—
5	B1	—	—
6	B2	—	—
7	B3	—	—

Special code for letter n

by **Nuno Jorge**

0687

Wokwi Project

github.com/NunoJorge-NC/TinyTapeout_Workshop

wokwi.com/projects/422960080008854529

“Specific input binary word to display the letter n”

How it works

The project simply displays different symbols in a 7-segment LED display according to the input binary word.

How to test

To test the project, one should try different input binary words. The output should only display the letter n (without the dot; outputs 0, 1, 2, 4, and 5 HIGH and the others LOW) when the input is 0b00010101 (21 in binary).

External hardware

8 input switches; 7-segment LED display.

Project Pinout

Digital Pins

#	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	—

Neural Net ASIC

by **Neural Navigators: Linyang Lee, Harsha Ganta, Stephanie Shen, William Li & Kiana Dai**

0710

10 MHz

HDL Project

github.com/piboi/tt08-nn

"MNIST Handwriting prediction on a neural network"

How it works

Neural network

How to test

Test

Project Pinout

Digital Pins

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

Zoom Zoom

by **Justin T, Andrew H, Simon Y, Kellen Y, Vallabh A & Nicole C**

0718

60 MHz

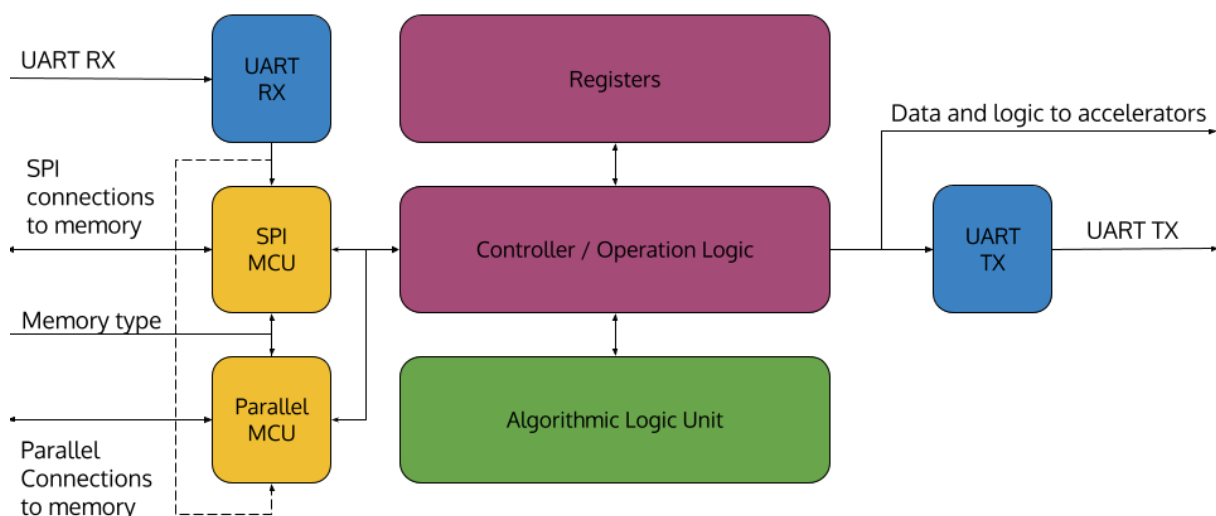
HDL Project

github.com/ringedSquid/BWSI-ASICS-24-Zoom-Zoom

"Custom Cpu with custome external memory bus and sha-3 and CORDIC accelerators"

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](#))

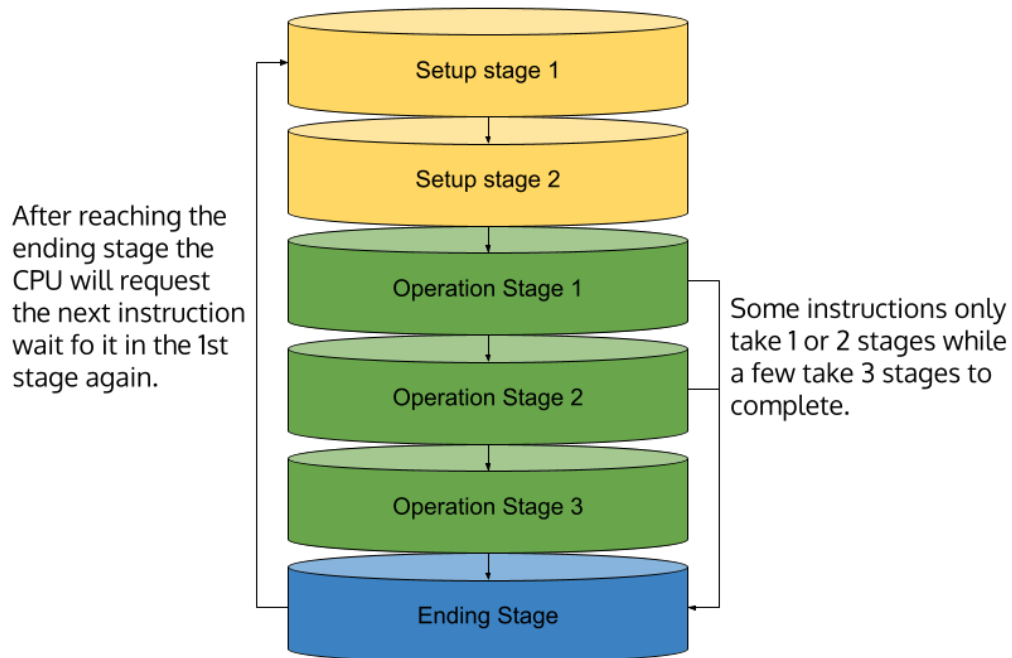


Detailed List of Features

- 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 Memeroy 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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Type		
reg out			reg2			reg1			settings			OPCODE			A			
Literal 9 bit									reg out			OPCODE			L			
Literal 8 bit						reg out			Setting			OPCODE			I			
Unrecommended General Purpose			Settings						k reg 1 extend			k reg 1			reg 1			OPCODE

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] * reg2[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) { inst addr = reg1 }
jmpg	Jump if Greater	A	0100	010	reg_out = inst addr; if (GF) { inst addr = reg1 }
jmpe	Jump if Equal	A	0100	111	reg_out = inst addr; if (EF) { inst addr = reg1 }
jmp1	Jump if Less	A	0100	011	reg_out = inst addr; if (!GF) { inst addr = reg1 }
jmpm	Jump if Memory Flagged	A	0100	100	reg_out = inst addr; if (MF) { inst addr = reg1 }
jmpu	Jump if UART Flagged	A	0100	101	reg_out = inst addr; if (UF) { inst addr = reg1 }
jmp1	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 permutation 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 of parallel storage with custom protocol

Project Pinout

Digital Pins

#	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

Emil Njor's Design

by Emil Njor

0736

Wokwi Project

github.com/Ekhao/tiny_tapeout

wokwi.com/projects/422962914561876993

"Nothing much really, just a couple of logic gates connecting the input to the output."

How it works

It is literally just a couple of logic gates between input and output pins.

How to test

There is nothing to test.

External hardware

Nothing.

Project Pinout

Digital Pins

#	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	—

Encoder

by **Damianos**

0737

Wokwi Project

github.com/NorthCdt/typeout

wokwi.com/projects/422968416190311425

"8x3 encoder"

How it works

following to a special code created from an 8 port switch it displays the letter U.

How to test

like this

External hardware

8 port switchc, 7 segment display

Project Pinout

Digital Pins

#	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	—

Asynchronous I2C Registerfile Interface

by DTU

0738

HDL Project

github.com/SimoneGuglielmino/I2C_register

"async. i2c if to regfile"

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

Project Pinout

Digital Pins

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

simplePass

by **Shawko**

0739

Wokwi Project

github.com/BitisG/tinytapeout10

wokwi.com/projects/422962904571040769

"If inputs 1, 2, 3 and 4 are high, all outputs are high."

How it works

Power goes in, power maybe comes out?

How to test

No testing needed, it's perfect.

External hardware

No external hardware is used

Project Pinout

Digital Pins

#	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	—

3-bit register print

by **Emad Maroun**

0740

Wokwi Project

github.com/Emoun/tiny-tapeout-workshop

wokwi.com/projects/422960054456096769

"Prints 3-bit value on 7-segment display"

How it works

Shows the value of the input (first 3 bit) on the 7-segment

How to test

Should show the numbers

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any 7-segment

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Bit 0	Segment	N/A
1	Bit 1	Segment	N/A
2	Bit 2	Segment	N/A
3	N/A	Segment	N/A
4	N/A	Segment	N/A
5	N/A	Segment	N/A
6	N/A	Segment	N/A
7	N/A	Segment	N/A

SSMCI

by **Oliver Keszocze**

0741

HDL Project

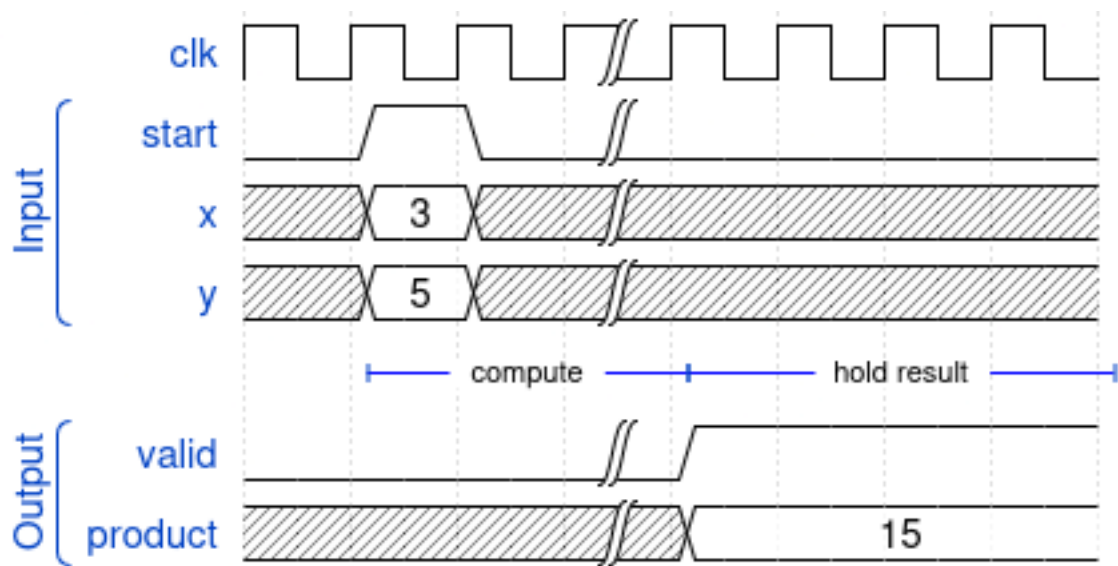
github.com/keszocze/tt10_SSMCI

"Slow 3-bit unsigned multiplier"

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 wavetrave of the 3-bit multiplier

Figure 741.1: simplified wave trace for 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!).

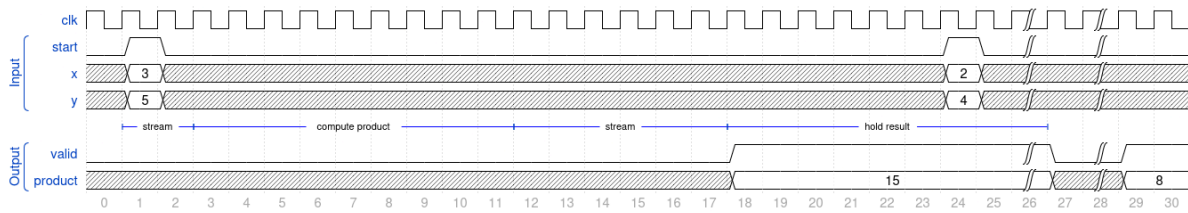


Figure 741.2: long wave trace for the 3-bit multiplier

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.

Project Pinout

Digital Pins

#	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	—

Four basic building blocks

by Thomas novotny

0742

Wokwi Project

github.com/ThomasNovotny/template2

wokwi.com/projects/422960130190575617

"The 1st input and the 1st output is connected to an inverter. The 2nd and 3rd input and the 2nd and 3rd output is connected to an AND-gate. The 3rd and 4th inputs and the 3rd and 4th output is connected to an OR-gate. The last 2 inputs and the 2nd-last output is connected to an Earle-latch, where the last input bit is the Enable-signal."

How it works

The 1st input and the 1st output is connected to an inverter. The 2nd and 3rd input and the 2nd and 3rd output is connected to an AND-gate. The 3rd and 4th inputs and the 3rd and 4th output is connected to an OR-gate. The last 2 inputs and the 2nd-last output is connected to an Earle-latch, where the last input bit is the Enable-signal.

How to test

For the NOT-gate, AND-gate and OR-gate, test every input combination. For the Earle latch, test if the latch can hold onto a 0, and test if the latch can hold onto a 1.

External hardware

No addons.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	IN0	OUT0	—
1	IN1	OUT1	—
2	IN2	OUT2	—
3	IN3	—	—
4	IN4	—	—
5	—	—	—
6	IN6	OUT6	—
7	IN7	—	—

TT10_Luke_Clock

by Luca Pezzarossa

0743

25.175 MHz

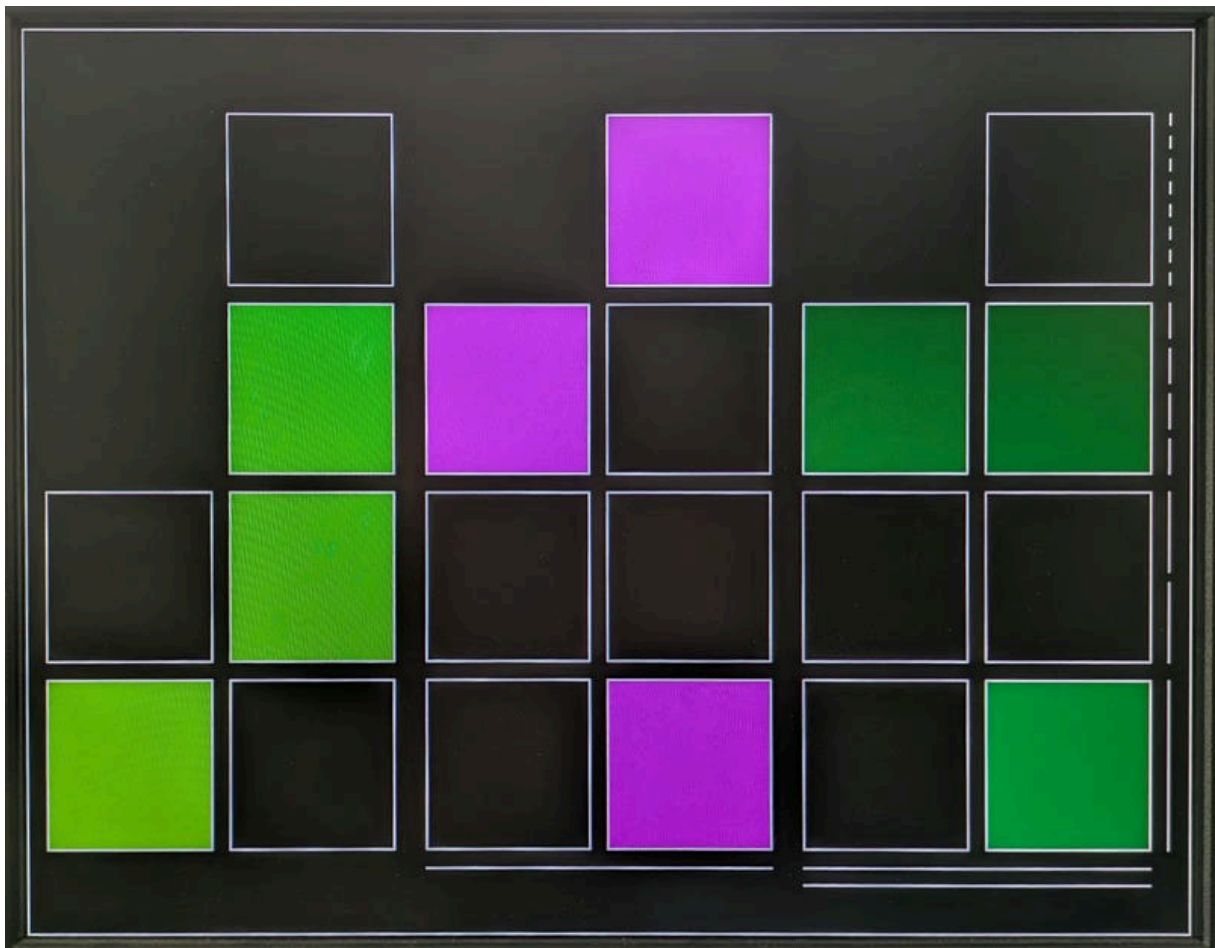
HDL Project

github.com/lucapezza/tt10_luke_clock

"A VGA based binary clock"

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 columns 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]**
- ui_in(1): **Select time clock source [bit 1]**
 - 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 (uio)

The following bidirectionals (all used as output) provide debugging information for internal signals.

- `uio_out(0)`: **(Debug output) tClk**
- `uio_out(1)`: **(Debug output) cntReg [bit 0]**
- `uio_out(2)`: **(Debug output) cntReg [bit 1]**
- `uio_out(3)`: **(Debug output) cntReg [bit 2]**
- `uio_out(4)`: **(Debug output) cntReg [bit 3]**
- `uio_out(5)`: **(Debug output) inDisplayArea**
- `uio_out(6)`: **(Debug output) modeReg [bit 0]**
- `uio_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 **uio_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.

Project Pinout

Digital Pins

#	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]

#	Input	Output	Bidirectional
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]

3 Bit Adder

by Victor Ding

0744

Wokwi Project

github.com/vicclu/Tinytapeout

wokwi.com/projects/422965035809389569

"3 bit adder"

How it works

dup

How to test

dup

External hardware

Noo

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	IN0	OUT0	—
1	IN1	OUT1	—
2	IN2	OUT2	—
3	IN3	OUT3	—
4	IN4	—	—
5	IN5	—	—
6	—	—	—
7	—	—	—

Verilog based clock to 7-segment counter

by EnJens

0745

1 Hz

HDL Project

github.com/EnJens/tt10-workshop-verilog

"Verilog based clock to 7-segment counter"

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

Project Pinout

Digital Pins

#	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	—

TinyChipDesign

by Oliver

0746

Wokwi Project

github.com/S234412/TT10Design

wokwi.com/projects/422958894385882113

"Initial design "

How it works

Template Explain how your project works

How to test

Template Explain how to use your project

External hardware

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	IN0	OUT0	—
1	IN1	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—



Simple NAND 2

by Mad

0747

50 MHz

Wokwi Project

github.com/git-mad-max/tt10-chapter-03

wokwi.com/projects/422968696249282561

"Following soon ;)"

How it works

Well this part will follow... for sure.

How to test

This will be an outstanding explanation how this entire project will work.

External hardware

LED Display, Flux compensator, quantum computer

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	IN0	OUT0	—
1	IN1	OUT1	—
2	IN2	OUT2	—
3	IN3	—	—
4	IN4	—	—
5	IN5	—	—
6	IN6	—	—
7	IN7	—	—

Tappu

by **Tobias Jensen**

0748

20 Hz

HDL Project

github.com/tobias1012/tt10-tappu

"A simple 8 bit CPU inspired by the esoteric language BF"

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

Project Pinout

Digital Pins

#	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	—

Adder

by Ehsan

0749

Wokwi Project

github.com/EhsanKhodadad/tinytapeout

wokwi.com/projects/422960085743520769

"An adder :D"

How it works

By magic

How to test

Use it

External hardware

Switches and LED displays

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	IN0	OUT0	—
1	IN1	OUT1	—
2	IN2	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Enter-Code

by **Imad**

0750

Wokwi Project

github.com/imaad17/tinytapeout

wokwi.com/projects/422960332734617601

“Enter the code (267) to display number (4)”

How it works

You enter the code 267 on the input switch and see if it shows number 4.

How to test

Turn different switches off or on.

External hardware

See number on display.

Project Pinout

Digital Pins

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

XOR Cipher

by **Damian**

0751

50 MHz

HDL Project

github.com/gantover/tt10-verilog-gantover

"Simple XOR Cipher with UART"

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"

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	updateKey	tx	—
1	—	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—

#	Input	Output	Bidirectional
7	rx	—	—

LGN hand-written digit classifier (MNIST, 16x16 pixels)

by Renaldas Zioma & Jogundas Armaitis

0782

1 MHz

HDL Project

github.com/rejunity/tt10-lgn-mnist

"Experiment with Deep Differential Logic Networks"

How it works

Experiment with Deep Differential Logic Networks. Classifier network.

How to test

TO DO: Provide inputs. Read out the classifier results.

External hardware

MCU to provide input data and read out the response

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Shift-in 8-bit input packet (LSB)	Digit (LSB)	(out) Confidence score (LSB)
1	Shift-in 8-bit input packet	Digit	(out) Confidence score
2	Shift-in 8-bit input packet	Digit	(out) Confidence score
3	Shift-in 8-bit input packet	Digit (MSB)	(out) Confidence score
4	Shift-in 8-bit input packet	—	(out) Confidence score
5	Shift-in 8-bit input packet	—	(out) Confidence score
6	Shift-in 8-bit input packet	—	(out) Confidence score (MSB)
7	Shift-in 8-bit input packet (MSB)	—	(in) /WE - Pause input

zc-sushi-demo

by Zachary Chen

800

25.2 MHz

HDL Project

github.com/zacharyzgc/zc-sushi-demo

"sushi running"

How it works

Sushi running across the screen

How to test

connect using TinyVGA PMOD and reset

External hardware

TinyVGA PMOD

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	R1	—
1	—	G1	—
2	—	B1	—
3	—	vsync	—
4	—	R0	—
5	—	G0	—
6	—	B0	—
7	—	hsync	—

Sine Synth

by **R. Timothy Edwards**

801

50 MHz

HDL Project

github.com/RTimothyEdwards/tt10-sine-synth

“Keyboard synthesizer with one octave of notes and a sine wave generator”

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.

Project Pinout

Digital Pins

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

#	Input	Output	Bidirectional
6	Key B	8-bit Output, bit 6	—
7	Key C	8-bit Output, bit 7	PWM Output

Crossyroad

by **Matt, Jovan & Ryan**

0802

25 MHz

HDL Project

github.com/mchen26/tt10-vga-crossyroad

"Crossyroad game recreated on an ASIC"

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

- Debounced button to ui_in[0]
- 25Mhz clock to clk pin

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	move_btn	red	—
1	—	green	—
2	—	blue	—
3	—	vsync	—
4	—	red	—
5	—	green	—
6	—	blue	—
7	—	hsync	—

DRUM

by Gökçe Aydos

0803

HDL Project

github.com/goekce/tinytapeout10-design

“an approximate multiplier”

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.

Project Pinout

Digital Pins

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

JTAG TAP

by Sean Patrick O'Brien

0804

HDL Project

github.com/obrienp/tt10-spo-jtag

"JTAG TAP with Boundary Scan"

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 [BSD 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]`

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	output_mode	seven_segment	—
1	output_value[0]	seven_segment	—
2	output_value[1]	seven_segment	—
3	output_value[2]	seven_segment	—
4	output_value[3]	seven_segment	TCK

#	Input	Output	Bidirectional
5	—	seven_segment	TMS
6	—	seven_segment	TDI
7	—	seven_segment	TDO

Tiny Hash Table

by **Sasha Krassovsky**

805

HDL Project

github.com/save-buffer/tt10-hash-table

"Hash table with 8 slots, 4-bit keys, 4-bit values"

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 G0 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 | | STATUS_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 G0 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 G0 to 0 for at least one cycle before triggering another command.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	VAL0	OVAL0	CMD0
1	VAL1	OVAL1	CMD1

#	Input	Output	Bidirectional
2	VAL2	OVAL2	GO
3	VAL3	OVAL3	—
4	KEY0	—	—
5	KEY1	—	—
6	KEY2	—	STATUS0
7	KEY3	—	STATUS1

7-segment with LFSR

by Jun-ichi OKAMURA

806

50 MHz

HDL Project

github.com/junlokamura/tt10_um_junlokamura

"TEST 7-segment/LFSR"

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.
- Inputs[1]: 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.

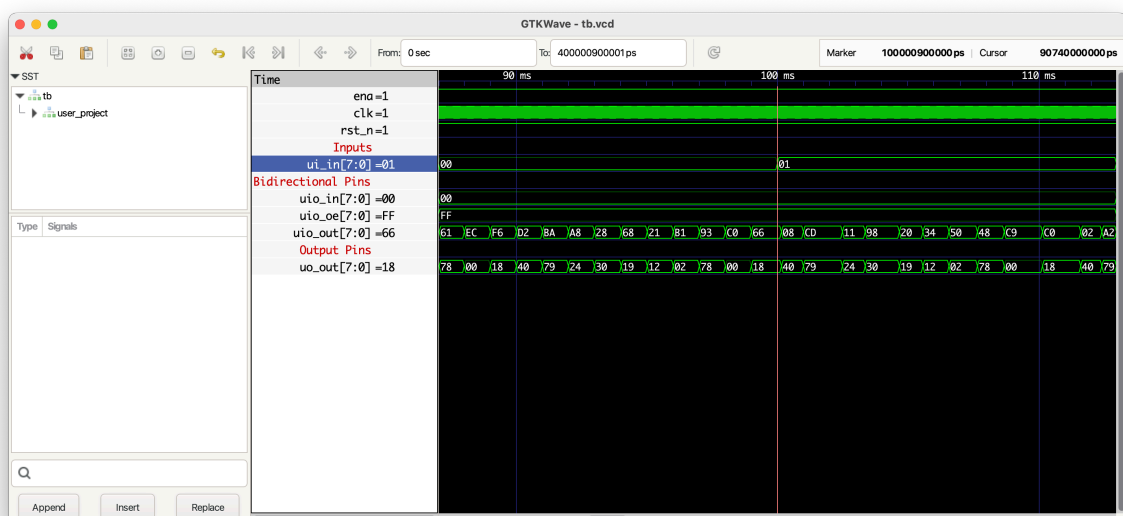


Figure 806.1: WAVEFORM

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 “junlokamura”, supported by Open-SUSI (non-profit) and AIST Solutions inc. in Japan.

Project Pinout

Digital Pins

#	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

Asynchronous FIFO

by **RMKGSN**

0807

60 MHz

HDL Project

github.com/reemashivva/CF-2024-TT10-06

“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”

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_in[1]) – Enables reading data from the FIFO.
- reset (ui_in[2]) – 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_out[1]) is LOW (FIFO contains data).
 - Set read_enable (ui_in[1]) 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.

Project Pinout

Digital Pins

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

Synchronous FIFO

by **Monish V.R**

0809

HDL Project

github.com/Monish-VR/CF-2024-TT10-02

"It's a synchronous fifo which has 4 bit of width"

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

Project Pinout

Digital Pins

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

KCH CD101 Saw Synth

by Johannes Pfau

0810

25 MHz

HDL Project

github.com/jpf91/cd101-tt-saw

“Fork of the KCH CD101 Synth Generating Saw Waves”

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 de-asserted 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.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Trigger	Audio Data	—
1	SPI CLK	—	—
2	SPI MOSI	—	—
3	SPI nSS	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Pulse Width Modulation

by **Nithish Reddy KVS**

0811

50 MHz

HDL Project

github.com/nithishreddykvs/CF-2024-TT10-05

"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."

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_in[1]`) 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_in[1]` 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_in[1]. 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.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	clk	PWM_OUT	—
1	ui_in[0]	—	—
2	ui_in[1]	—	—

#	Input	Output	Bidirectional
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

tt10_zhouzhouthezhou_adder

by Kyle Zhou

0812

16 kHz

HDL Project

github.com/zhouzhouthezhou/tt10-zhouzhouthezhou-adder

"Adds 2 4bit numbers and displays the output on the 8 segment"

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

Project Pinout

Digital Pins

#	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

by Tórrur Biskopstø Strøm

0814

HDL Project

github.com/torurstrom/tt10-async-lock

"An asynchronous lock with 8 request signals and 8 grant signals"

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.

Project Pinout

Digital Pins

#	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	—

Simple shift Reg

by **test**

0815

1 Hz

HDL Project

github.com/fromaf1/TT10_Verilog

"Simple shift Reg"

How it works

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

How to test

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

External hardware

None wher used

Project Pinout

Digital Pins

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

TRNG

by **Muhammad Bilal**

842

50 MHz

HDL Project

github.com/engrbilal1992/tt10-TRNG

"Generate Raw and Hashed Random Numbers"

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](https://github.com/sp800-90b))

External hardware

- ZCU102 FPGA Board
- UART-to-USB Adapter (for serial communication)
- Oscilloscope (for debugging noise source if needed)

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	TRNG_Enable	failure	—
1	ctrl_mode	hash_rdy	—
2	—	UART_Tx	—
3	—	—	—
4	—	—	—

#	Input	Output	Bidirectional
5	—	—	—
6	—	—	—
7	—	—	—

Find The Damn Issue

by **Leonel Gouveia Ergin (Synogate)** & **Michael Offel (Synogate)**

846

12 MHz

HDL Project

github.com/synogate/find-the-damn-issue

"USB to UART/SPI/I2C/JTAG/GPIO adapter"

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.

Project Pinout

Digital Pins

#	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

kch cd101

by Johannes Pfau

0864

25 MHz

HDL Project

github.com/kit-kch/cd101-tt

“KCH Things”

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 de-asserted 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.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Trigger	Audio Data	—
1	SPI CLK	—	—
2	SPI MOSI	—	—
3	SPI nSS	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Muller C-element

by **simon2024**

0865

HDL Project

github.com/simon2024/tt10v

“An attempt to do a standard cell implementation of Muller C-element”

How it works

TBD

How to test

TBD

Project Pinout

Digital Pins

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

SimpleSPIDEv

by Zedulo

0866

HDL Project

github.com/ZeduloAdmin/TT10_SPICtrl

"SPI device with GPIO/mem interface"

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.

Project Pinout

Digital Pins

#	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	—

Tiny Shader

by **Leo Moser**

0867

25.175 MHz

HDL Project

github.com/mole99/tt10-tiny-shader

"With Tiny Shader you can write a small program to create different images and even animations."

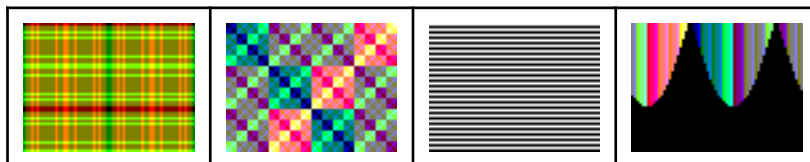
How it works

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 accessing the LUT, the entries are automatically mirrored to form one half of a sine wave with a total of 32 6-bit values.

Instructions

The following instructions are supported by Tiny Shader. A program consists of 10 instructions and is executed for each pixel individually. The actual resolution is therefore one tenth of the VGA resolution (64x48 pixel).

Output

Instruction	Operation	Description
SETRGB RA	RGB <= RA	Set the output color to the value of the specified register.
SETR RA	R <= RA[1:0]	Set the red channel of the output color to the lower two bits of the specified register.
SETG RA	G <= RA[1:0]	Set the green channel of the output color to the lower two bits of the specified register.

SETB RA	B <= RA[1:0]	Set the blue channel of the output color to the lower two bits of the specified register.
---------	--------------	---

Input

Instruction	Operation	Description
GETX RA	RA <= X	Set the specified register to the x position of the current pixel.
GETY RA	RA <= Y	Set the specified register to the y position of the current pixel.
GETTIME RA	RA <= TIME	Set the specified register to the current time value, increases with each frame.
GETUSER RA	RA <= USER	Set the specified register to the user value, can be set via the SPI interface.

Branches

Instruction	Operation	Description
IFEQ RA	TAKE <= RA == R0	Execute the next instruction if RA equals R0.
IFNE RA	TAKE <= RA != R0	Execute the next instruction if RA does not equal R0.
IFGE RA	TAKE <= RA >= R0	Execute the next instruction if RA is greater then or equal R0.
IFLT RA	TAKE <= RA < R0	Execute the next instruction if RA is less than R0.

Arithmetic

Instruction	Operation	Description
DOUBLE RA	RA <= RA * 2	Double the value of RA.
HALF RA	RA <= RA / 2	Half the value of RA.
ADD RA RB	RA <= RA + RB	Add RA and RB, result written into RA.

Load

Instruction	Operation	Description
CLEAR RA	RA <= 0	Clear RA by writing 0.
LDI IMMEDIATE	RA <= IMMEDIATE	Load an immediate value into RA.

Special

Instruction	Operation	Description
SINE RA	$RA \leftarrow \text{SINE}[R0[4:0]]$	Get the sine value for R0 and write into RA. The sine value LUT has 32 entries.

Boolean

Instruction	Operation	Description
AND RA RB	$RA \leftarrow RA \& RB$	Boolean AND of RA and RB, result written into RA.
OR RA RB	$RA \leftarrow RA \mid RB$	Boolean OR of RA and RB, result written into RA.
NOT RA RB	$RA \leftarrow \neg RB$	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	$RA \leftarrow RA \ll RB$	Shift RA with RB to the left, result written into RA.
SHIFTR RA RB	$RA \leftarrow RA \gg RB$	Shift RA with RB to the right, result written into RA.

Pseudo

Instruction	Operation	Description
NOP	$R0 \leftarrow R0 \& R0$	No operation.

How to test

First set the clock to 25.175 MHz and reset the design. For a simple test, simply connect a Tiny VGA to the output Pmod. A shader is loaded by default and an image should be displayed via VGA.

For advanced features, connect an SPI controller to the bidir pmod. If ui[0], the mode signal, is set to 0, you can write to the user register via SPI. Note that only the last 6 bit are used.

If the mode signal is 1, all bytes transmitted via SPI are shifted into the shader memory. This way you can load a new shader program. Have fun!

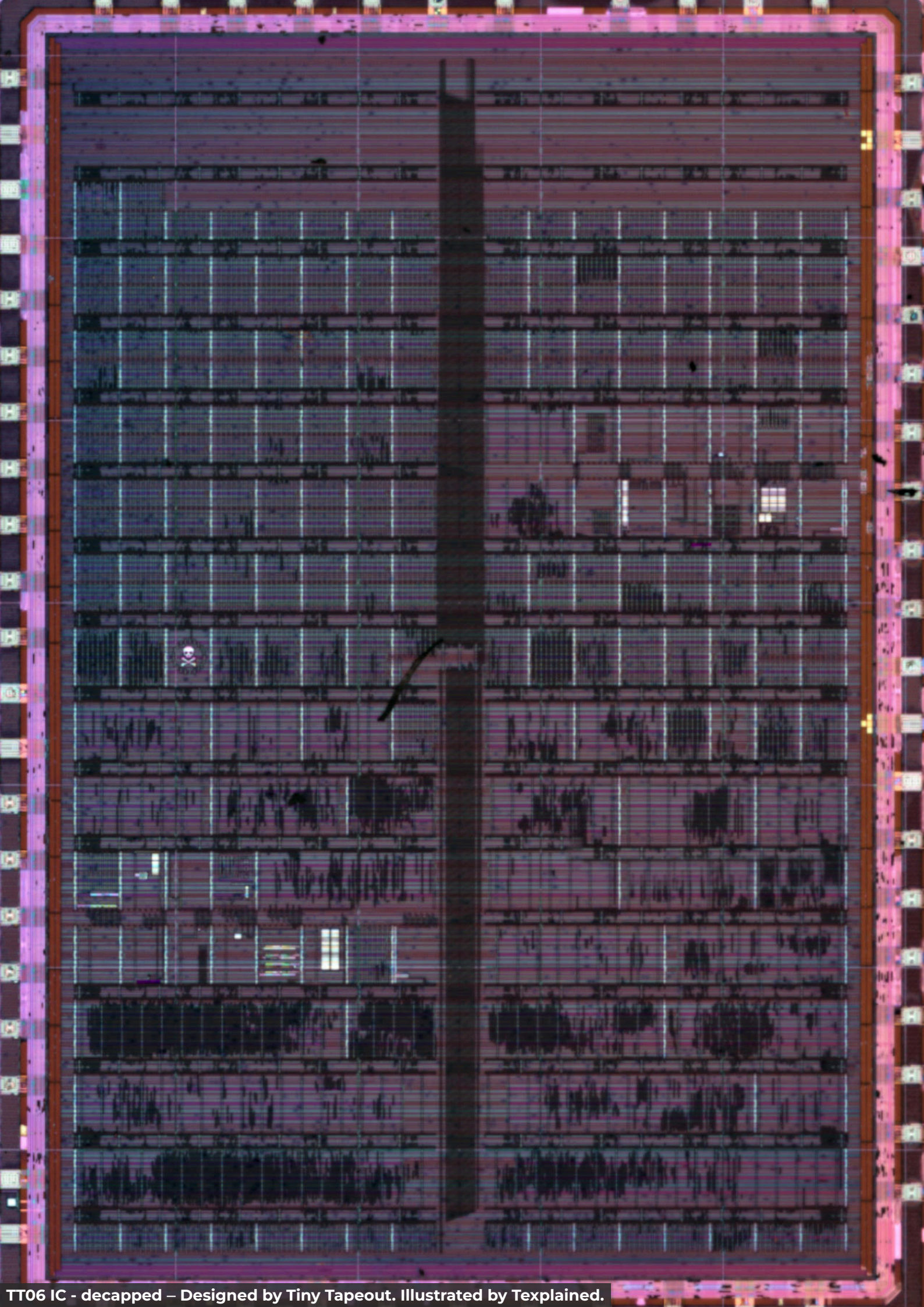
External hardware

- [Tiny VGA](#) or similar VGA Pmod
- Optional: SPI controller to write the user register and new shaders

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	mode	R[1]	CS
1	debug_i[0]	G[1]	MOSI
2	debug_i[1]	B[1]	MISO
3	—	vsync	SCK
4	—	R[0]	next_vertical
5	—	G[0]	next_frame
6	—	B[0]	debug_o[0]
7	—	hsync	debug_o[1]



RNG_test

by Ba-Anh Dao

0868

50 MHz

HDL Project

github.com/daobaanh/tt10-trng-test

"trying to implement lightweight all digital trng"

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	RX_Serial	TX_Serial	—
1	—	o_RO	—
2	—	o_RG	—
3	—	led	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Linear Timecode (LTC) generator

by Thomas Flummer

0869

12 MHz

HDL Project

github.com/flummer/tt-um-flummer-ltc

"Timecode generator for audio video synchronization"

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 ui[2] and ui[3]

ui[3]	ui[2]	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.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	—	—
1	—	—	—
2	FRAMERATE_0	—	—
3	FRAMERATE_1	—	—
4	—	—	—

#	Input	Output	Bidirectional
5	—	—	—
6	—	—	—
7	—	—	LTC_OUT

15bit GCD

by **stephan**

0870

100 kHz

HDL Project

github.com/StephanAAU/tt10-verilog

"Greatest common denominator"

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	—	—	—	—	—	—	—	—	—	—	—	—	—	—	LSB
uio[6]	uio[5]	uio[4]	uio[3]	uio[2]	uio[1]	uio[0]	ui[7]	ui[6]	ui[5]	ui[4]	ui[3]	ui[2]	ui[1]	ui[0]	

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]	uo[2]	uo[1]	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.

Project Pinout

Digital Pins

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

Priority-encoder

by Ole Henrik Moller

0871

HDL Project

github.com/ole-moller/Priority_encoder

"8-bit priority encoder to decimal 7-segment code"

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	data[0]	segments[0]	—
1	data[1]	segments[1]	—
2	data[2]	segments[2]	—
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	—

XY Spacewar

by Nicklaus Thompson

0872

50 MHz

HDL Project

github.com/FangameEmpire/tt10_xy_spacewar

"A scaled-down Spacewar game with a VGA mode. Might be Asteroids instead."

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.

Project Pinout

Digital Pins

#	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	—

UltraTiny-CPU

by Roméo Estezet

0873

HDL Project

github.com/Romultra/UltraTiny-CPU

"Simple 8-Bit CPU"

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 `uio[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.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Memory load address bit 0 (when <code>ui[7]=1</code>)	Accumulator output bit 0	Data bus bit 0 for load mode (input), otherwise tri-stated
1	Memory load address bit 1 (when <code>ui[7]=1</code>)	Accumulator output bit 1	Data bus bit 1 for load mode (input), otherwise tri-stated
2	Memory load address bit 2 (when <code>ui[7]=1</code>)	Accumulator output bit 2	Data bus bit 2 for load mode (input), otherwise tri-stated
3	Memory load address bit 3 (when <code>ui[7]=1</code>)	Accumulator output bit 3	Data bus bit 3 for load mode (input), otherwise tri-stated
4	Unused (available for other custom inputs)	Accumulator output bit 4	Data bus bit 4 for load mode (input), otherwise tri-stated
5	Unused (available for other custom inputs)	Accumulator output bit 5	Data bus bit 5 for load mode (input), otherwise tri-stated
6	Unused (available for other custom inputs)	Accumulator output bit 6	Data bus bit 6 for load mode (input), otherwise tri-stated

#	Input	Output	Bidirectional
7	Load mode enable (1=program load, 0=run)	Accumulator output bit 7	Data bus bit 7 for load mode (input), otherwise tri-stated

16-bit Logarithmic Approximate Floating Point Multiplier

by **Anwesh Rao, B S Gurucharan, Shreyas M Iliger, Tushar M, Shylashree N**
& **RV College of Engineering**

874

50 MHz

HDL Project

github.com/RVCE-ECE-Shylashree/TT10_LogAFPM-16

"A 16-bit floating-point multiplier utilizing logarithmic approximation to achieve fast and power-efficient floating-point multiplication with reduced hardware."

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 **uio_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) uio_in (Input B) Reset State uo_out (Output)		
Clock		
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 | | 0x436c | 0x45aa | 0x4d4c
| | 0x44df | 0x483d | 50x12c |
```

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	A[0]/A[8]	P[0]/P[8]	B[0]/B[8]
1	A[1]/A[9]	P[1]/P[9]	B[1]/B[9]
2	A[2]/A[10]	P[2]/P[10]	B[2]/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]

E-ink display driver

by **Tim Edwards**

0875

50 Hz

HDL Project

github.com/htfab/tt10-eink-driver

“Test driver for Adafruit 2.13 inch e-ink display”

How it works

This is an example hardware driver for an e-ink display. Adafruit makes a nice series of small e-ink displays, but they are designed for an Arduino and driven by software. This project shows how to build a display driver in verilog. To keep memory overhead to a minimum, it operates like a VGA screen saver, displaying simple patterns that can be computed in real time as the pixel positions are counted and transmitted to the driver.

The driver instantiates an SPI master which communicates with the SSD1680 chipset on the e-ink display. Whenever a bit from the input PMOD is set to “1”, an initialization sequence is sent to the display, followed by a transmission of the display image, followed by a deep sleep power-down. Once in deep sleep mode, the displayed image will remain indefinitely, even if the display is disconnected from the development board.

How to test

The input/output PMOD is used to connect to the e-ink display pins. Since the e-ink display is not PMOD-compatible, it is necessary to install a header onto the e-ink display and create a bundle of jumper wires to connect to the PMOD as follows:

pin signal direction PMOD pin

ECS: uio[0] output 1 MOSI: uio[1] output 2 MISO: uio[2] input 3 SCK: uio[3] output 4 SRCS: uio[4] output 7 RST: uio[5] output 8 BUSY: uio[6] input 9 D/C: uio[7] output 10 GND: 11 or 5 VIN: 12 or 6 —————

To test the eight example patterns, raise one of the input pins to value “1”. This can be done with a set of external buttons on the input PMOD, or the input PMOD value can be set from software.

ui[5] is a special case in which the contents of the display board’s SRAM are copied directly to the e-ink display. This uses an unusual method in which the SRAM is set to a sequential read mode and then is left enabled while the e-ink display is initialized. Commands being sent to the display are ignored by the SRAM, which outputs one bit on every clock cycle. The SRAM contents

are then copied into the display starting at offset address 30 (which is the number of SPI bytes clocked while initializing the display). The SRAM is volatile and so unprogrammed at power-up. It can be programmed using the “pass-through” mode, in which the SRAM’s SPI can be bit-banged from the ui[] port using software. Enable “pass-through” mode by setting ui[7:4] to 0xf, then bit-bang using ui[0] for clock and ui[1] for data (if the SRAM is given a READ command, then output from the SRAM can be read from uo[0]). First put the SRAM into sequential mode with command 0x01 0x40. End pass-through mode with ui = 0x00, then re-enter pass-through mode with ui = 0xf0. Continue with the command 0x02 0x00 0x1e and then write 3904 bytes of image data (32 bytes x 122 lines). End pass-through mode again with ui = 0x00, then display the image data with ui = 0x20.

External hardware

Every e-ink display has a very specific driver, and making a general-purpose driver is prohibitive for Tiny Tapeout. The project is designed to drive the Adafruit 2.13” e-ink display, Product ID: 4197, URL <https://www.adafruit.com/product/4197> (as of this writing, cost is \$22.50).

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	All white Bitbang SCK	SRAM MISO (out)	ECS (out)
1	All black Bitbang MOSI	—	MOSI (out)
2	Vertical stripes	—	MISO (in, unused)
3	Horizontal stripes	—	SCK (out)
4	Small checkerboard	—	SRCS (out, =1)
5	User SRAM contents	—	RSTB (out)
6	Large checkerboard	—	BUSY (in)
7	Low-res smiley face	—	D/C (out)

TT_spiralPattern

by rkarl

0876

25 MHz

HDL Project

github.com/rkarl2/TT_spiral

"Creates A Controllable Spiral Pattern"

How it works

Creates a arcmedes 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)

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	ui_in[0]	uo_out[0]	—
1	ui_in[1]	uo_out[1]	—
2	ui_in[2]	uo_out[2]	—
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]	—

I2C EEPROM Project Selection

by Uri Shaked

0877

20 MHz

HDL Project

github.com/TinyTapeout/tt-autosel

“Prototype for reading the selected Tiny Tapeout design address from an I2C EEPROM”

How it works

This is a prototype for automatic project selection in Tiny Tapeout. It reads the design id from an I2C EEPROM and selects the corresponding project. The design id is stored in the first two bytes of the EEPROM memory as a big endian 16-bit number. The design id is then used to select the project by setting the `ctrl_sel_inc` signal to the lowest 10 bits of the design id.

The project also prints the 16-bit number read from the EEPROM to the UART port on `uo_out[4]` (baud rate 115200, 8N1) for debugging purposes.

The design also contains a ring oscillator that can be enabled by setting `rosc_clk_en` (`ui_in[7]`) to 1. The ring oscillator is used to generate a clock signal on the `rosc_clk_out` pin (`uo_out[7]`) that can be fed back to the `clk` input, so no external clock is needed.

The ring oscillator has 11 stages and a 5-bit divider. We estimate the frequency of the ring oscillator to be around 550 MHz, and the frequency of the divided clock to be around 17.2 MHz.

How to test

You need to connect a 1 kbit or 2 kbit I2C EEPROM (e.g. 24C01 or 24C02), program the selected Tiny Tapeout design address into the first two bytes of the EEPROM memory, and connect the EEPROM to the SCL/SDA pins with a 10k pull-up resistor. After connecting the EEPROM, reset the design, and then observe the `ctrl` output pins to see if the design id is correctly read from the EEPROM:

- `ctrl_sel_rst_n` should go low, and then high
- `ctrl_sel_inc` should pulse high according to the lowest 10 bits of the number stored in the EEPROM
- `ctrl_ena` should go high

For more information about those signals, please refer to the [Tiny Tapeout Multiplexer documentation](#).

You can clock the design externally with a 20 MHz clock signal, or enable the ring oscillator by setting `rosc_clk_en` to 1 and connect the `rosc_clk_out` pin to the `clk` input.

External hardware

I2C EEPROM (24C01 or 24C02) and two 10k pull-up resistors on SCL and SDA pins.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	<code>ctrl_sel_rst_n</code>	SCL
1	—	<code>ctrl_sel_inc</code>	SDA
2	—	<code>ctrl_ena</code>	—
3	—	—	—
4	—	<code>uart_tx</code>	—
5	—	—	—
6	—	—	—
7	<code>rosc_clk_en</code>	<code>rosc_clk_out</code>	—

ledtest

by **jellyant**

0878

10 MHz

HDL Project

github.com/jellyant/helloworld

"Turns one output ON."

How it works

It turns ON an output.

How to test

Don't do anything.

External hardware

No need.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	A[0]	—
1	—	A[1]	—
2	—	A[2]	—
3	—	A[3]	—
4	—	A[4]	—
5	—	A[5]	—
6	—	A[6]	—
7	—	A[7]	—

Overengineered Checkers

by **htfab**

0879

25.175 MHz

HDL Project

github.com/htfab/tt10-py-checkers

“Recreation of the Checkers demo with each layer generated by a different Python-based HDL”

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

Project Pinout

Digital Pins

#	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	—

Dice

by ZHU QUANHAO

0897

50 MHz

HDL Project

github.com/DanielZhu123/tt08-verilog-Ztemplate

"after you press the button the system will generate a random number from 0-F"

How it works

It generate number by inverter ring

How to test

press the button to capture number

External hardware

2 LED display

Project Pinout

Digital Pins

#	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

INTERCAL ALU

by **Rebecca G. Bettencourt**

899

HDL Project

github.com/RebeccaRGB/intercal-alu

“An ALU for the five operators of the INTERCAL programming language.”

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 bit-wise logic are surprisingly common, it is much rarer to encounter one that can calculate the five operations of the INTERCAL programming language. Due to either the cost-prohibitive nature of Warmenhovian logic gates or general lack of interest, such a feat has never been performed until now. With chip production finally within reach of the average person, all it takes is one person who has more dollars than sense to design the fabled INTERCAL ALU (Arrhythmic Logic Unit).

The pin assignments for this design are roughly as follows. The /OE (output enable) and /WE (write enable) signals are active low, so should be set HIGH by default.

#	Dedicated Input	Dedicated Output	Bidirectional I/O
0	A0 (address)	D0 (output only)	D0 (input and output only)
1	A1 (address)	D1 (output only)	D1 (input and output only)
2	S0 (selector)	D2 (output only)	D2 (input and output only)
3	S1 (selector)	D3 (output only)	D3 (input and output only)
4	S2 (selector)	D4 (output only)	D4 (input and output only)
5	S3 (selector)	D5 (output only)	D5 (input and output only)
6	/OE (output enable)	D6 (output only)	D6 (input and output only)
7	/WE (write enable)	D7 (output only)	D7 (input and output only)

This ALU has two 32-bit registers, B and A (in no particular order). (These may also be thought of as four 16-bit registers, AL, AH, BL, and BH.) To write a byte to a register, set A0 and A1 to the byte address, set S0 LOW for the A register or HIGH for the B register, set S1 through S3 LOW, set the bidirectional I/O pins to the byte value, set /WE LOW, then set /WE HIGH again. (Do not set S1 through S3 HIGH when writing, or else something unpredictable will happen, most likely nothing.)

To read a register or result, set A0 and A1 to the byte address, set S0 through S3 to the desired operation, set /OE LOW, read the byte value from the bidirectional I/O pins, then set /OE HIGH. Results can also be read from the dedicated outputs; the dedicated outputs are not affected by the /OE signal, as they do not need to care about your feelings.

The operations supported are listed below. An attempt was made to make it understandable.

						Address				
						A	3	2	1	0
						A1	1	1	0	0
						A0	1	0	1	0
Selector										
S	S3	S2	S1	S0	Operation					
0	0	0	0	0	A	AH		AL		
1	0	0	0	1	B	BH		BL		
2	0	0	1	0	AND16	& AH		& AL		
3	0	0	1	1	AND32	& A				
4	0	1	0	0	OR16	V AH		V AL		
5	0	1	0	1	OR32	V A				
6	0	1	1	0	XOR16	? AH		? AL		
7	0	1	1	1	XOR32	? A				
8	1	0	0	0	MINGLE16L	AL \$ BL				
9	1	0	0	1	MINGLE16H	AH \$ BH				
10	1	0	1	0	SELECT16	AH~BH		AL~BL		
11	1	0	1	1	SELECT32	A ~ B				

Operations 0 and 1 simply return the current value of the A or B register, respectively. This corresponds with the values of S0 through S3 used in write mode. This is not unintentional. This might also explain why S1 through S3 must be LOW in write mode.

Operations 2 through 7 correspond to INTERCAL's unary AND, unary OR, and unary XOR operators, represented by ampersand (&), book (V), and what (?), respectively. From the INTERCAL manual:

These operators perform their respective logical operations on all pairs of adjacent bits, the result from the first and last bits going into the first bit of the result. The effect is that of rotating the operand one place to the right and ANDing, ORing, or XORing with its initial value. Thus, `#&77` (binary = 1001101) is binary 0000000000000100 = 4, `#V77` is binary 1000000001101111 = 32879, and `#?77` is binary 1000000001101011 = 32875. Operations 2, 4, and 6 work on the 16-bit halves of the A register independently, while operations 3, 5, and 7 work on the 32-bit whole of the A register.

Operations 8 and 9 correspond to INTERCAL's *interleave* (also called *mingle*) operator, represented by big money (\$). From the INTERCAL manual:

The interleave operator takes two 16-bit values and produces a 32-bit result by alternating the bits of the operands. Thus, `#65535$#0` has the 32-bit binary form 101010....10 or 2863311530 decimal, while `#0$#65535` = 0101....01 binary = 1431655765 decimal, and `#255$#255` is equivalent to `#65535`. Operation 8 returns the interleave of the lower halves of A and B, while operation 9 returns the interleave of the upper halves of A and B. (Should the chip fabrication process allow for it, operation 8½ will, of course, return the interleave of the middle halves of A and B.)

Operations 10 and 11 correspond to INTERCAL's *select* operator, represented by sqiggle (). From the INTERCAL manual:

The select operator takes from the first operand whichever bits correspond to 1's in the second operand, and packs these bits to the right in the result. Both operands are automatically padded on the left with zeros. [...] For example, `#179 #201` (binary value 10110011 11001001) selects from the first argument the 8th, 7th, 4th, and 1st from last bits, namely, 1001, which = 9. But `#201 #179` selects from binary 11001001 the 8th, 6th, 5th, 2nd, and 1st from last bits, giving 10001 = 17. `#179 #179` has the value 31, while `#201 #201` has the value 15. To help understand the select operator, the INTERCAL manual also provides a helpful [circuitous diagram](#).

Use of operations 12 and above is not recommended, unless undefined behavior is required.

How to test

The following example calculations found in the INTERCAL manual should be particularly illuminating.

S	A	B	F
MINGLE16L (8)	0	256	65536
MINGLE16L (8)	65535	0	2863311530
MINGLE16L (8)	0	65535	1431655765

MINGLE16L (8)	255	255	65535
SELECT16 (10)	51	21	5 *
SELECT16 (10)	179	201	9
SELECT16 (10)	201	179	17
SELECT16 (10)	179	179	31
SELECT16 (10)	201	201	15
AND16 (2)	77	—	4
OR16 (4)	77	—	32879
XOR16 (6)	77	—	32875

These test cases are included in the (unfortunately Python and not INTERCAL) `test.py` file. As these are likely more INTERCAL operations than any sensible person will ever perform, they should be sufficient for testing purposes. However, for curiosity's sake, an extensive set of additional test cases have also been included.

* Not found in the INTERCAL manual.

External hardware

The ALU may be used without external hardware, although seeing the output values may present a challenge. Instead, it is recommended to use a microcontroller of some sort to drive the inputs and read the outputs, as microcontrollers are designed to do. The implementation of the rest of the INTERCAL language is left as an exercise for the reader.

Further reading

[The INTERCAL Programming Language Revised Reference Manual](#) by Donald R. Woods and James M. Lyon with revisions by Louis Howell and Eric S. Raymond (can recommend highly enough)

Project Pinout

Digital Pins

#	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

#	Input	Output	Bidirectional
6	/OE (output enable)	D6	D6
7	/WE (write enable)	D7	D7

4-bit minicomputer ALU

by Mike McCann

0901

HDL Project

github.com/kb2ghz/tt_um_kb2ghz_xalu

"this design provides basic arithmetic and logic functions"

How it works

The project is a 4-bit ALU section that is usfull 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).

Project Pinout

Digital Pins

#	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	—

Hardware UTF Encoder/Decoder

by **Rebecca G. Bettencourt**

6903

HDL Project

github.com/RebeccaRGB/hardware-utf8

“Converts Unicode code points between UTF-8, UTF-16, and UTF-32.”

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 ($\geq 0x110000$).
- 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 ($\geq 0x110000$ or, if CHK is LOW, $\geq 0x80000000$).

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 ($\geq 0x110000$).

Outputting UTF-32

1. Set READ or /WRITE (input 4) HIGH.
2. Set /CIO (input 5, character I/O) LOW.
3. Pulse CLK and read the first byte of the UTF-32 word from the bidirectional I/O.
4. Pulse CLK and read the second byte of the UTF-32 word from the bidirectional I/O.
5. Pulse CLK and read the third byte of the UTF-32 word from the bidirectional I/O.
6. Pulse CLK and read the fourth byte of the UTF-32 word from the bidirectional I/O.
7. Set /CIO (input 5, character I/O) HIGH.
8. If the UTF-32 word is within range, the input and output are both valid.
9. If the UTF-32 word is not within range, then the input was either incomplete or invalid.

Outputting UTF-16

1. Set READ or /WRITE (input 4) HIGH.
2. If UEOF (output 6) is HIGH, then the input was either incomplete or invalid.
3. Set /UIO (input 6, UTF-16 I/O) LOW.
4. Pulse CLK and read the next byte of the UTF-16 sequence from the bidirectional I/O.
5. Repeat step 4 until UEOF (output 6) is HIGH.
6. Set /UIO (input 6, UTF-16 I/O) HIGH.

Outputting UTF-8

1. Set READ or /WRITE (input 4) HIGH.
2. If BEOF (output 7) is HIGH, then the input was either incomplete or invalid.
3. Set /BIO (input 7, byte I/O) LOW.

4. Pulse CLK and read the next byte of the UTF-8 sequence from the bidirectional I/O.
5. Repeat step 4 until BEOF (output 7) is HIGH.
6. Set /BIO (input 7, byte I/O) HIGH.

Error status

When ERRS or /PROPS (input 1) is HIGH, the dedicated outputs will be:

#	Name	Meaning
0	READY	The input and output are complete sequences.
1	RETRY	The previous input was invalid or the start of another sequence and was ignored. Process the output, reset, and try the previous input again.
2	INVALID	The input and output are invalid.
3	OVERLONG	The UTF-8 input was an overlong sequence.
4	NONUNI	The code point value is out of range ($\geq 0x110000$). (This is set independently of the CHK input; the CHK input only changes whether this counts as an error.)
5	ERROR	Equivalent to (RETRY or INVALID or OVERLONG or (NONUNI and CHK)).

If all of these outputs are LOW, the accumulated input is incomplete and more input is required (underflow).

Character properties

When ERRS or /PROPS (input 1) is LOW, the dedicated outputs will be:

#	Name	Meaning
0	NORMAL	The code point value is valid and not a C0 or C1 control character, surrogate, private use character, or noncharacter.
1	CONTROL	The code point value is valid and a C0 or C1 control character (0x00-0x1F or 0x7F-0x9F).
2	SURROGATE	The code point value is valid and a UTF-16 surrogate (0xD800-0xDFFF).
3	HIGHCHAR	The code point value is valid and either a high surrogate (0xD800-0xDBFF) or a non-BMP character ($\geq 0x10000$).

4	PRIVATE	The code point value is valid and either a private use character (0xE000-0xF8FF, ≥0xF0000) or the high surrogate of a private use character (0xDB80-0xDBFF).
5	NONCHAR	The code point value is valid and a noncharacter (0xFDD0-0xFDEF or the last two code points of any plane).

If all of these outputs are LOW, there is no valid code point in the output.

How to test

The `test.py` file covers a comprehensive set of test cases which are listed in [a separate file](#) to avoid bloating the TT08 manual.

External hardware

Any device that needs to process Unicode text.

Project Pinout

Digital Pins

#	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

RGB Mixer demo5

by **Matt Venn**

0905

10 MHz

HDL Project

github.com/rubenet6/ruben

“Zero to ASIC demo project”

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.

Project Pinout

Digital Pins

#	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	—	—

Universal Binary to Segment Decoder

by **Rebecca G. Bettencourt**

0907

HDL Project

github.com/RebeccaRGB/ubcd

“Decodes various binary codes to various segmented displays.”

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	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

	1010	1011	1100	1101	1110	1111		1010	1011	1100	1101	1110	1111	
V0=0 V1=0 V2=0								V0=0 V1=0 V2=1	-	=	=	=	-	
V0=1 V1=0 V2=0	c	3	4	5	6			V0=1 V1=0 V2=1	-	L	C	r	E	
V0=0 V1=1 V2=0	o	o	-	-	-			V0=0 V1=1 V2=1	-	E	H	L	P	
V0=1 V1=1 V2=0	0	1	2	3	4	5		V0=1 V1=1 V2=1	A	b	C	d	E	F

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
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		-	'	"	1	5	'	t	-	C	J	o	4	J	-	-	7
D6=0 D5=1 D4=1	0	1	2	3	4	5	6	7	8	9	-	J	4	=	7	7	7
D6=1 D5=0 D4=0	P	A	b	C	d	E	F	G	H	I	J	K	L	M	N	O	
D6=1 D5=0 D4=1	P	9	r	S	7	U	Y	8	=	Y	2	C	4	J	7	-	
D6=1 D5=1 D4=0	4	2	b	c	d	E	F	9	H	7	J	K	I	K	N	O	
D6=1 D5=1 D4=1	P	9	r	S	t	U	C	8	=	Y	2	4	I	7	-		

FS=1:

	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111	
D6=0 D5=1 D4=0		_	'	"	1	'	'	6	-	c	3	0	4	J	-	.	7
D6=0 D5=1 D4=1	0	1	2	3	4	5	6	7	8	9	-	]	[	=	3	2	
D6=1 D5=0 D4=0	P	A	b	C	d	E	F	G	H	I	J	K	L	M	N	O	
D6=1 D5=0 D4=1	P	q	r	S	T	U	V	W	X	Y	Z	[	]	^	_		
D6=1 D5=1 D4=0	4	7	b	c	d	L	P	Q	H	I	J	K	L	M	N	O	
D6=1 D5=1 D4=1	P	q	r	S	T	U	V	W	X	Y	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).

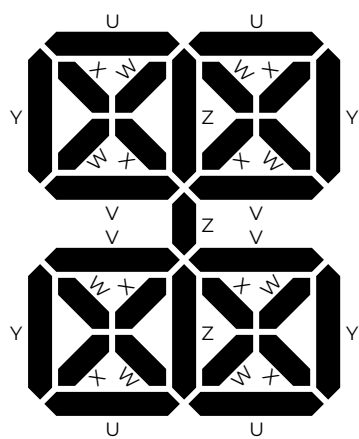
The pin assignments in this mode are:

—	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

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

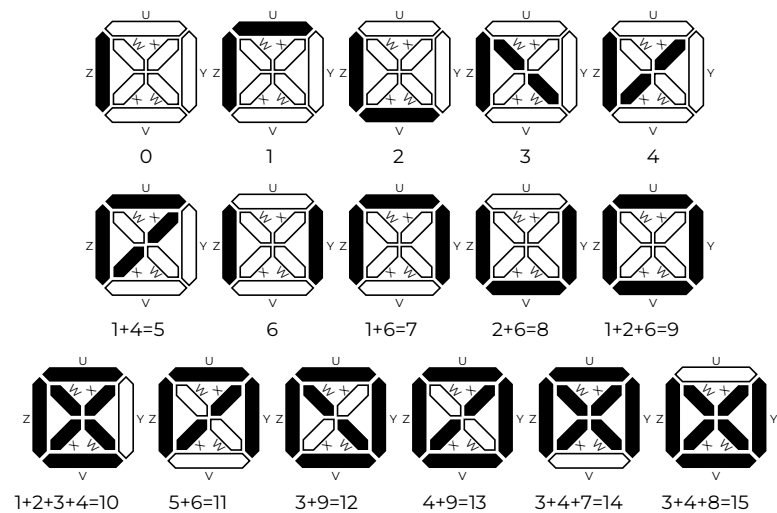
Dual BCD to Cistercian numeral decoder

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:

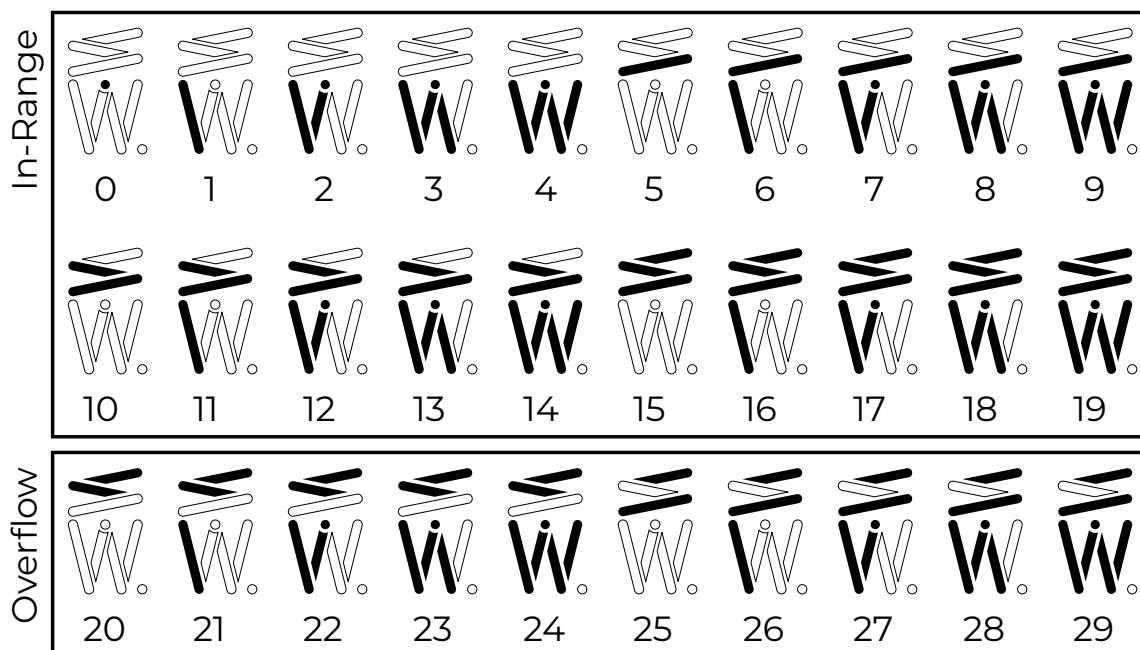
—	Dedicated Input	Dedicated Output	Bidirectional
0	A1	Segment U1	Output - Y1
1	B1	Segment U2	Output - Y2
2	C1	Segment V1	Input - /LT1
3	D1	Segment V2	Input - /LT2
4	A2	Segment W1	Input - /BI
5	B2	Segment W2	Input - /AL
6	C2	Segment X1	Input - LOW
7	D2	Segment X2	Input - HIGH

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	—
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

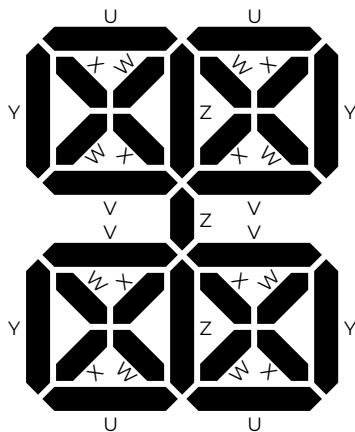
How to test

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](#).



Project Pinout

Digital Pins

#	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)

Asynchronous Multiplier

by Tommy Thorn

0909

50 MHz

HDL Project

github.com/tommythorn/tt08-maxbw

"An asynchronous multiplier"

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, so what is actually emitted is $r \wedge (r \gg 15)$.

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:

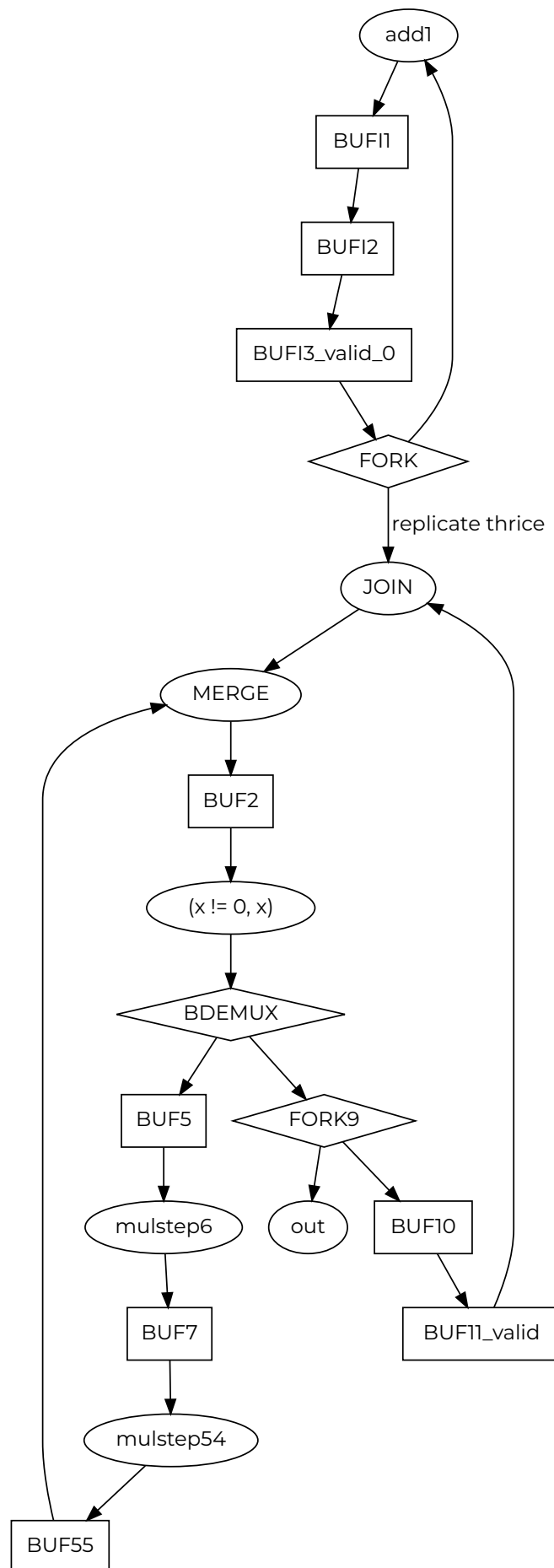


Figure 909.1: token-flow graph

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.

Project Pinout

Digital Pins

#	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

Supermic

by **Armaan Gomes, Asmi Sawant, Ria Saheta, Vikhaash Kanagavel Chithra, Morgan Packard & Sanjay Ravishankar**

0910

HDL Project

github.com/arghunter/Supermic-tt08

"A 8 channel customizable beamforming signal processor"

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

Project Pinout

Digital Pins

#	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	—

VGA Tiny Logo (1 tile)

by Renaldas Zioma

0911

25.175 MHz

HDL Project

github.com/rejunity/tt08-huge-480x480-tiny-tapeout-logo-in-1-tile

"Large 480x480 pixels Tiny Tapeout logo with bling and dithered colors crammed into 1 tile!"

How it works

Compressed VGA Logo

How to test

Connect to VGA monitor

External hardware

TinyVGA PMOD, VGA monitor

Project Pinout

Digital Pins

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

TTCAD25a SKY130 FF YOLO Test

by Sylvain Munaut

0938

HDL Project

github.com/smunaut/ttcad25a-ff-yolo-test

“Test fliflop”

How it works

See design data for information

How to test

No test procedure has been written yet and it's not meant to be tested by the “demo board” but instead on a dedicated test platform.

External hardware

To do any meaningful timing testing you'll need some FPGA hardware to drive the various control signal in sequence with precise timings.

The exact testing platform is still TBD.

Project Pinout

Digital Pins

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

Padlock

by **htfab**

0940

10 kHz

Wokwi Project

github.com/htfab/ttcad25a-padlock

wokwi.com/projects/422192563267712001

“Set a code for your precious safe”

How it works

- Stores a 7-bit secret code using flip-flops. Another flip-flop stores whether the safe is open.
- Unlocks the safe when the input (bits 0-6) matches the stored secret code. Locks the safe when RESET is pressed.
- Updates the secret code when input bit 7 is high.
- The 7-segment display shows “L” if the safe is locked and “U” if unlocked.

How to test

- Set the clock to free-running mode, e.g. at 10 kHz.
- Set the desired secret code using pins 0-6. Toggle pin 7 ON, then OFF.
- Change the inputs 0-6 to prevent the safe from automatically unlocking.
- Press RESET to lock the safe. The display should show “L” (for locked).
- Change the inputs 0-6 to try opening the safe. It should only show “U” (for unlocked) once the secret code is entered. Then it should stay unlocked until the next RESET.

External hardware

None

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	code bit 0	segment a	—
1	code bit 1	segment b	—
2	code bit 2	segment c	—
3	code bit 3	segment d	—
4	code bit 4	segment e	—
5	code bit 5	segment f	—
6	code bit 6	segment g	—

#	Input	Output	Bidirectional
7	set code	—	—

sky130_as_sc_hs dyn test

by Tholin

6942

10 MHz

HDL Project

github.com/89Mods/tt_sky130_as_sc_hs__dyn_test

"Testing sky130_as_sc_hs SCL with dynamic DFFs"

How it works

Contains a LFSR that updates on every clock cycle and outputs on all output pins. It uses dynamic DFFs, which means they will lose their state if the clock speed gets too low.

How to test

Release from reset with all ui_in pins zero.

External hardware

RP2040 or LEDs to see the output state.

Project Pinout

Digital Pins

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

Lab and Lectures SoC

by **Aloke Kumar Das**

0961

50 Hz

HDL Project

github.com/alokerdas/tt_um_LnL_SoC

"A tiny SoC comprising of a cpu, memory and SPI protocol"

How it works

This project implements a tiny system on chip. It has a 16 bit microprocessor, a boot rom, a PWM, a timer and a spi protocol.

The boot rom has 32 words. After reset it runs a program to get input from outside and display to outside. The program has all the instructions that this processor supports. This tapeout is done to test the microprocessor on silicon. The SPI, PWM and timers are memory mapped. The processor writes the data to SPI, PWM and timers so that those IPs can be tested also.

The SPI protocol can be used for serial communication. The data can be loaded to and from cpu. This IP is mapped at 0020. If the cpu attempts to write to the address 0020 the data will be transmitted through the SPI protocol. It can accept data from outside of the SoC as specified in the spi protocol. The signals load and unload can be used to enable this IP.

The PWM resolution is 8. The duty cycle can be varied from 12.5 to 87.5 percent. It is memory mapped at the address 0040. It has a 3-bit register which can be written by the processor to set the duty cycle value. The timer is 8-bit without any pre-scalar. The timer is auto reload and can not be stopped. The output signals can be chosen from divide by 2/4/.../128. It is memory mapped at 0080. It has a 3-bit register which can be written by the processor to set the divisor value.

The microprocessor is a basic one. The data bus is 16-bits, address bus is 12-bits. Address and data busses are connected to internal boot rom, RAM and SPI. They cannot access outside memory. There is a parallel input port of 8-bits which is also input of the SoC. Similarly, there is a parallel output port of 8-bits that is also output of SoC. The Instructions that are supported are as follows: LDA - Load the content of a memory location to accumulator AC ADD - Add the content of a memory location to AC AND - And the content of a memory location with AC STA - Store the content of AC to a memory location BUN - Branch unconditionally BSA - Branch to a memory location storing the return address ISZ - Increment the content of a memory location and check if zero, skip the next instruction Indirect addressing mode of all the above instruction are also supported

CLA - Clear the content of the AC CLE - Clear the overflow flag E CMA - Complement the content of the AC CME - Complement the overflow flag E CIR - Shift right the content of AC and E, circular CIL - Shift left the content of AC and E, circular INC - Increment the content of AC SPA - Skip next instruction if the content of AC is positive SNA - Skip next instruction if the content of AC is negative SZA - Skip next instruction if the content of AC is zero SZE - Skip next instruction if E is zero INP - Accept 8-bit input from input port if inp flag is high OUT - Send 8-bit output to output port and set the outp flag SKI - Skip next instruction if input flag is high SKO - Skip next instruction if output flag is high HLT - Halt the cpu

How to test

After power on the cpu starts running automatically. No extra effort is required. The boot rom has a program inbuilt. It check for input. If input flag is high the 8-bit value is written to accumulator from ui_in pins. Immediately the same value is output to uo_out pins so that it can be displayed on 7-segment. After that all the other instructions are executed. Those tests the direct as well as indirect addressing modes. The program write addresses 0020, 0040 and 0080. This is the space for SPI, timer and PWM. The data comes out serially of uio_out[5] pin (mosi of spi), uio_out[4] and uio_out[3].

External hardware

Keypad, 7-segment or LCD or LED. Some kind of storage or data source. To be decided later.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	keyboard 0	display 0	cpu keyboard in flag
1	keyboard 1	display 1	miso of spi
2	keyboard 2	display 2	ssn in of spi
3	keyboard 3	display 3	clock of spi (future use)
4	keyboard 4	display 4	ssn out of spi
5	keyboard 5	display 5	mosi of spi
6	keyboard 6	display 6	sclk of spi
7	keyboard 7	display 7	cpu display flag

asic design is my passion

by **Nicholas Junker**

0963

25.175 MHz

HDL Project

github.com/Quarren42/tt08_Quarren42_demoscene

"baby's first asic - cheeky little text meme"

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!

Project Pinout

Digital Pins

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

Crispy VGA

by James Meech

0965

HDL Project

github.com/JamesTimothyMeech/James-Meech-TT08-Demoscene

“The scrolling VGA example from the vga playground but as you set more inputs high it gets successively more crispy”

How it works

This project “Crispy VGA” takes as input the output of a standard tiny tapeout VGA project. Crispy VGA then adds a programmable amount of random noise to the VGA signal and passes it through to the output. The `uio_in[0]` input sets the noise on the `hsync` signal. The `uio_in[1]` input sets the noise on the B signal. The `uio_in[2]` input sets the noise on the G signal. The `uio_in[3]` input sets the noise on the R signal. The `uio_in[4]` input sets the noise on the `vsync`. The `uio_in[5]` signal sets the noise level applied to the R, G, and B wires to high or low. The `uio_in[0:5]` inputs set the succesively increasing noise levels on the audio signal.

How to test

Plug an existing tiny tapeout VGA project into the input of this design. Plug the output of this design into a standard VGA input monitor. Power up both tiny tapeout boards and select the appropriate control bits for the level of noise that you want to see on the output VGA signal.

External hardware

You will need a VGA input monitor and a computer that can output a VGA signal or a second tiny tapeout ASIC with a working VGA design that follows the standard pinout. You will also need two tiny tapeout VGA adapters and two VGA cables.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	R[1] vga input	R[1] vga input	Crispy input bit 0 that toggles the noise on the <code>hsync</code> signal on or off. Also adds one bit of noise to audio.

#	Input	Output	Bidirectional
1	G[1] vga input	G[1] vga input	Crispy input bit 1 toggles the noise on the B signal on or off. Also adds one bit of noise to audio.
2	B[1] vga input	B[1] vga input	Crispy input bit 2 toggles the noise on the G signal on or off. Also adds one bit of noise to audio.
3	vsync vga input	vsync vga input	Crispy input bit 3 toggles the noise on the R signal on or off. Also adds one bit of noise to audio.
4	R[0] vga input	R[0] vga input	Crispy input bit 4 that toggles the noise on the vsync signal on or off. Also adds one bit of noise to audio.
5	G[0] vga input	G[0] vga input	Crispy input bit 5 that sets the noise level applied to the R, G, and B wires to high or low. Also adds one bit of noise to audio.
6	B[0] vga input	B[0] vga input	Audio input bit
7	hsync vga input	hsync vga input	Audio output bit

Calculator

by JING Shuangyu

0967

10 MHz

HDL Project

github.com/Cooooobra/tt08_calculator

"A calculator do basic calculation"

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.

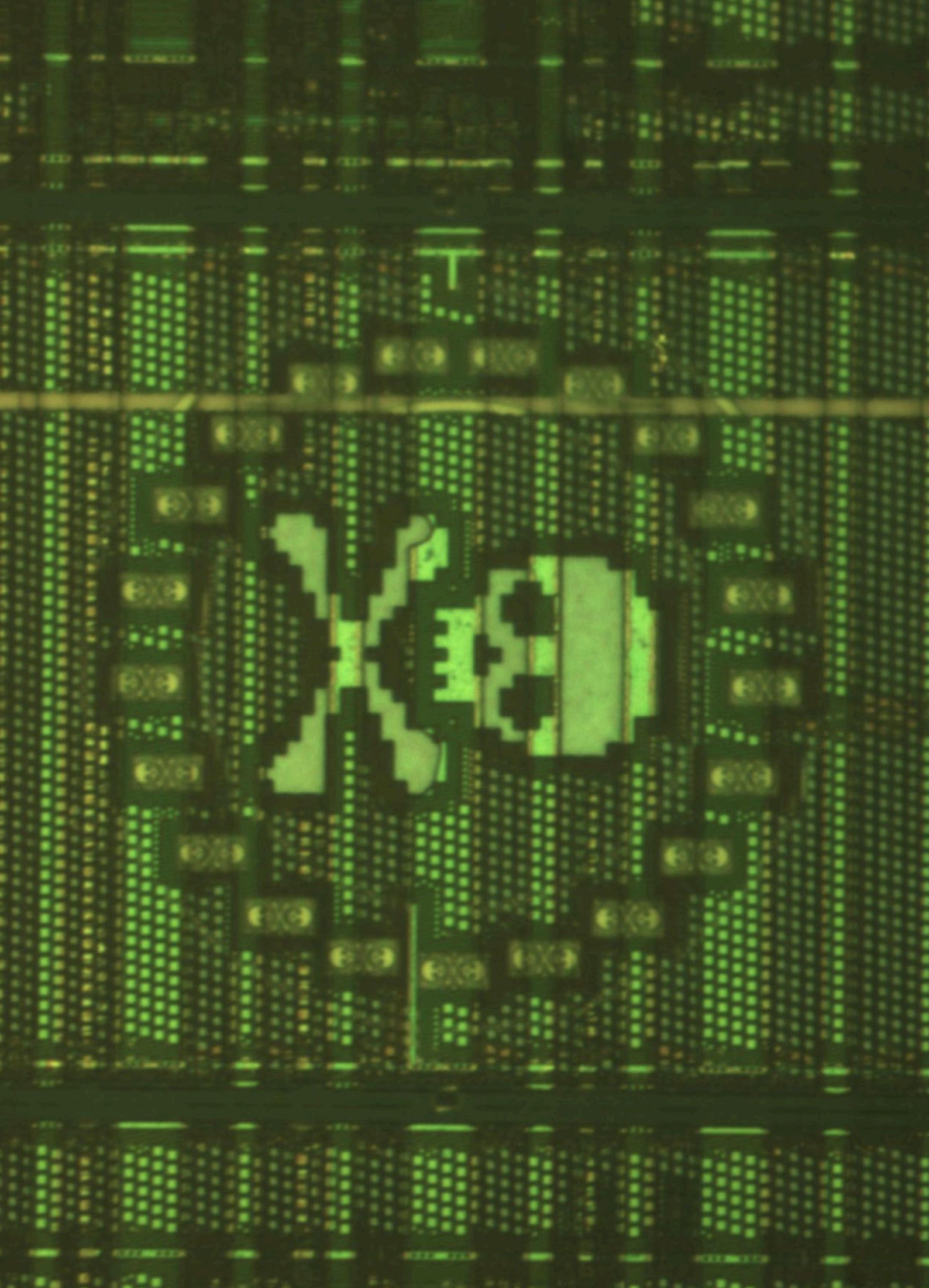
External hardware

The calculator need a 4x4 matrix keypad for input and a 3-digit seven segment display to show the calculated result.

Project Pinout

Digital Pins

#	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



Oscillating Bones – Designed by Uri Shaked. Illustrated by Texplained.

VGA Scroller

by **FavoritoHJS**

0971

25 MHz

HDL Project

github.com/FavoritoHJS/tt08-favoritohjs-scroller

"Scrolls across a very pixelated cityscape"

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.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	Rh	—
1	—	Gh	—
2	—	Bh	—
3	—	vsync	—
4	—	RI	—
5	—	GI	—
6	—	BI	—
7	—	hsync	—

Glyph Mode

by James Ross

0973

25.175 MHz

HDL Project

github.com/jar/tt08_vga_glyph_mode

"Submission for VGA Demoscene"

How it works

This is a standalone VGA demo that runs with or without input. It will accept two pins `ui_io[0]` and `ui_io[1]` for palette color selection:

ui_io[1:0]	Palette
0	Green (default)
1	Red
2	Blue
3	Pride

How to test

Plug into a VGA monitor and select this circuit to test

External hardware

Requires the [TinyVGA PMOD](#)

Project Pinout

Digital Pins

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

KianV uLinux SoC

by Hirosh Dabui

0974

HDL Project

github.com/TinyTapeout/KianV-RV32IMA-RISC-V-uLinux-SoC

“A RISC-V ASIC that can boot μ Linux.”

How it works

32-bit RISC-V IMA processor, capable of booting Linux. Features 16 MiB of external SPI flash memory, 16 MiB of external PSRAM (8 MiB per bank), a UART peripheral, and a SPI peripheral.

System Memory Map

The system memory map is as follows:

Address	Size	Purpose
0x10000000	0x14	UART Peripheral
0x10500000	0x14	SPI Peripheral
0x10600000	0x0c	GPIO Peripheral
0x11100000	0x04	Reset / HALT control
0x20000000	16 MiB	SPI Flash
0x80000000	16 MiB	PSRAM

The system boots from the SPI flash memory. After reset, the CPU starts executing code from 0x20100000 (corresponding to the offset 0x100000 into the SPI flash memory), where the bootloader is expected to be.

UART Peripheral registers

Address	Name	Description
0x10000000	UART_DATA	Write to transmit, read to receive
0x10000005	UART_LSR	UART line status register
0x10000010	UART_DIV	Clock divider for UART baud rate

SPI Peripheral registers

Address	Name	Description
0x10500000	SPI_CTRL0	SPI Peripheral Control
0x10500004	SPI_DATA0	SPI Data

0x10500010	SPI_DIV	Clock divider for SPI peripheral
------------	---------	----------------------------------

GPIO Peripheral registers

Address	Name	Description
0x10600000	GPIO_UO_EN	Enable bits for uo_out pins
0x10600004	GPIO_UO_OUT	Write to uo_out pins
0x10600008	GPIO_UI_IN	Read from ui_in pins(read-only)

CPU control register

Address	Name	Description
0x11100000	CPU_RESET	Write 0x7777 to reset the CPU, 0x5555 to halt the CPU.

How to test

Build the system image as described in the [kianRiscV repo](#) and load it into the SPI flash memory:

Flash offset	File name	Description
0x100000	bootloader.bin	Bootloader
0x180000	kianv.dtb	Device Tree Blob
0x200000	Image	Linux kernel + rootfs

The system runs at 30 MHz, with a maximum tested speed of 34.5 MHz.

External hardware

[QSPI Pmod](#) - can be purchased from the [Tiny Tapeout store](#).

Project Pinout

Digital Pins

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

#	Input	Output	Bidirectional
7	gpio[7]	gpio[7]	cs2 psram

RO

by Arna Roy

0975

20 MHz

HDL Project

github.com/Secure-Embedded-Systems/tt08-trng

"Implementation of simple RO"

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.

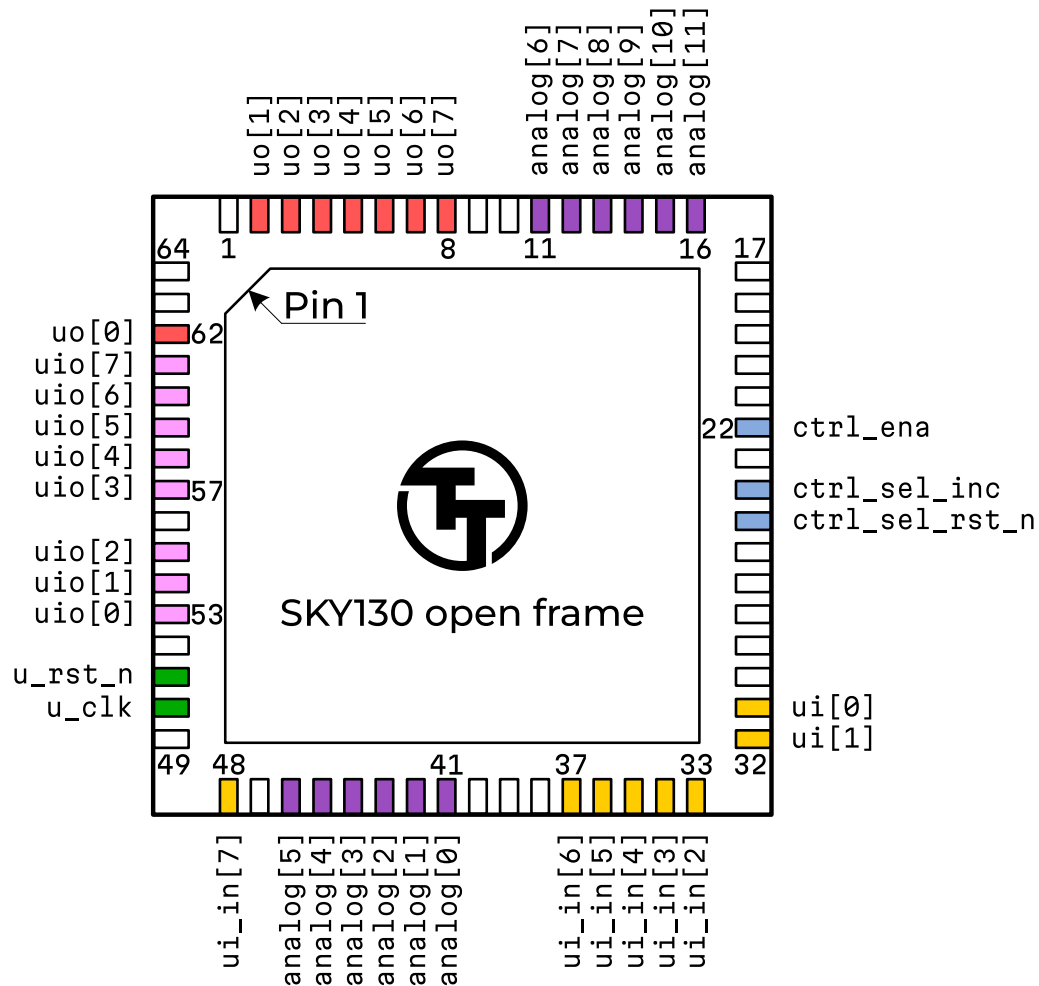
Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	ui_in[0]	uo_out[0]	—
1	ui_in[1]	uo_out[1]	—
2	ui_in[2]	uo_out[2]	—
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]	—

Pinout

The chip is packaged in a 64-pin QFN package. The pinout is shown below.



Bottom View

Note

You will receive the chip mounted on a breakout board (github.com/tinytapeout/breakout-pcb).

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 outputs
- 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 — connects the spine to individual user designs

The Controller

The mux controller has 3 input lines:

Input	Description
ena	Sent as-is (buffered) to the downstream mux units
sel_rst_n	Resets the internal address counter to 0 (active low)
sel_inc	Increments the internal address counter by 1

It outputs the address of the currently selected design on the `si_sel` port of the spine (see below).

For instance, to select the design at address 12, you need to pulse `sel_rst_n` low, and then pulse `sel_inc` 12 times:

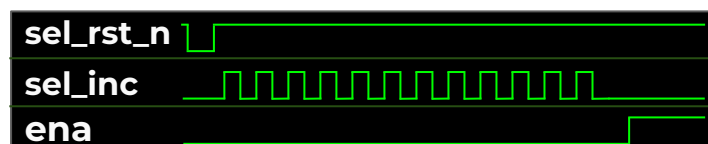


Figure 1: Mux signals for activating the design at address 12

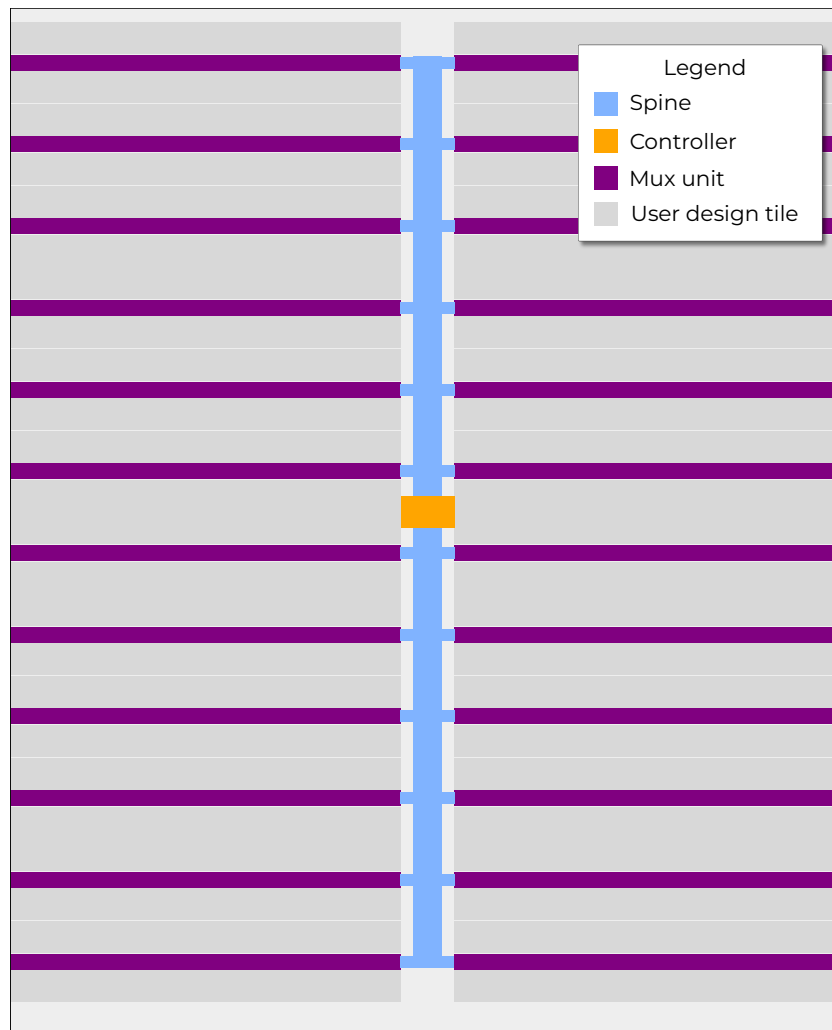


Figure 2: Mux Diagram

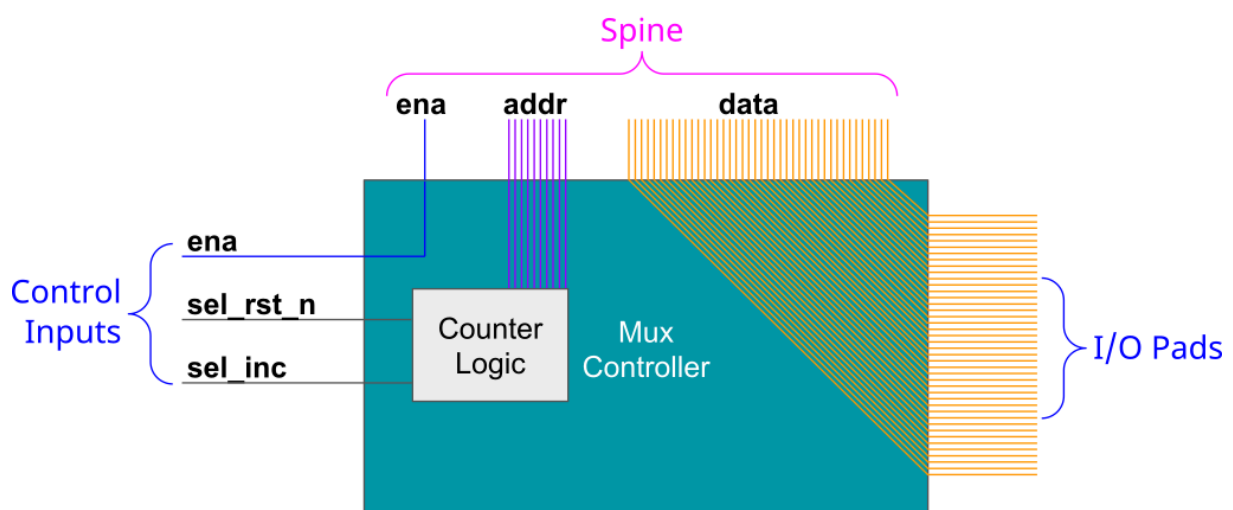


Figure 3: Mux Controller Diagram

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 project demonstrates this setup: wokwi.com/projects/36434780766. It contains an Arduino Nano that decodes the currently selected mux address and displays it on a 7-segment display. Click on the button labeled RST_N to reset the counter, and click on the button labeled INC to increment the counter.

The Spine

The controller and all muxes are connected together through the spine. The spine has the following signals going to 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 chip I/O pads.

The Multiplexer

Each mux branch is connected to up to 16 designs. It also has 5 bits of a 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	mprj_io_pin	Function	Signal
1	vssa2	Ground	GND
2	25	Output	uo[1]
3	26	Output	uo[2]
4	27	Output	uo[3]
5	28	Output	uo[4]
6	29	Output	uo[5]
7	30	Output	uo[6]
8	31	Output	uo[7]
9	vdda2	Power	VDD Analog
10	vssd2	Ground	GND
11	32	Analog	analog[6]
12	33	Analog	analog[7]
13	34	Analog	analog[8]
14	35	Analog	analog[9]
15	36	Analog	analog[10]
16	37	Analog	analog[11]
17	vddio	Power	VDD IO
18	vccd	Power	VDD Core
19	NC	—	—
20	vssa	Ground	GND
21	resetb	—	—
22	38	Mux	ctrl_ena
23	vssd	Ground	GND
24	39	Mux	ctrl_sel_inc
25	40	Mux	ctrl_sel_rst_n
26	41	Reserved	(none)
27	42	Reserved	(none)
28	43	Reserved	(none)
29	vssio	Ground	GND
30	vdda	Power	VDD Analog
31	0	Input	ui[0]
32	1	Input	ui[1]
33	2	Input	ui[2]
34	3	Input	ui[3]
35	4	Input	ui[4]

QFN64 pin	mprj_io_pin	Function	Signal
36	5	Input	ui[5]
37	6	Input	ui[6]
38	vssa1	Ground	GND
39	vssd1	Ground	GND
40	vdda1	Power	VDD Analog
41	7	Analog	analog[0]
42	8	Analog	analog[1]
43	9	Analog	analog[2]
44	10	Analog	analog[3]
45	11	Analog	analog[4]
46	12	Analog	analog[5]
47	vdda1	Power	VDD Analog
48	13	Input	ui[7]
49	vccd1	Power	VDD Core
50	14	Input	u_clk †
51	15	Input	u_rst_n †
52	vssa1	Ground	GND
53	16	Bidirectional	io[0]
54	17	Bidirectional	io[1]
55	18	Bidirectional	io[2]
56	vssio	Ground	GND
57	19	Bidirectional	io[3]
58	20	Bidirectional	io[4]
59	21	Bidirectional	io[5]
60	22	Bidirectional	io[6]
61	23	Bidirectional	io[7]
62	24	Output	uo[0]
63	vccd2	Power	VDD Core
64	vddio	Power	VDD IO
EPAD	—	Ground	—

† Internally, there's no difference between u_clk, u_rst_n, and ui 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.

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 makes**
- **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

Using This Datasheet

Structure

Projects are ordered by their mux address, in ascending order. Documentation is user-provided from their GitHub repositories and are merged into the final shuttle once the deadline is reached.

In general, each project should contain:

- The user-provided title & a list of authors
- A link to the GitHub repository used for submission
- A link to the Wokwi project (if applicable)
- A “How it works” section
- A “How to test” section
- An “External hardware” section (if applicable)
- A pinout table for both digital & analog designs

Badges

This datasheet uses “badges” to quickly convey some information about the content. These badges are explained in the table below.

Badge	Description
	Used to showcase artwork from our community.
 	Mux address of the project, in decimal. For microtile designs, their sub-address is placed after the forward slash. In this example, it would be 2.
	Clock frequency of the project. May be truncated from actual value or omitted completely.
  	Project type, indicating if it was made with a HDL, Wokwi, or if it is analog.
 	Indicates the risk that the project presents to the ASIC. Medium danger projects can damage the ASIC under certain conditions, whilst high danger projects <i>will</i> damage the ASIC.

Callouts

In addition to **Medium Danger** and **High Danger** badges being used, a callout is placed before the project documentation begins to alert the user.

A callout for **Medium Danger** may look something like:

This project will damage the ASIC under certain conditions.
There is an error in the schematic which may lead to ASIC failure under certain clocking conditions.

Similarly, a callout for **High Danger** may look something like:

This project will damage the ASIC.
There is an error in the schematic which may cause permanent damage when powered on in a certain configuration.

Should there be a project that poses a danger, the callout will explain the reasoning behind the danger level.

Callouts may also provide some additional information, and look something like so:

Information
Silicon melts at 1414°C, and boils at 3265°C. Don't let your chip get too hot!

Figures & Footnotes

Numbering for figures and footnotes within the “Project” chapter is formed by combining the address of the project with the current figure number. For example, the second figure for a project with an address of 256 will be captioned with “Figure 256.2”. Likewise, the third footnote for a project of address 128 will be shown as “128.3”.

The numbering outside of the “Project” chapter resumes as normal, being formatted with a simple number, e.g. “Figure 3”.

Updates

This datasheet is intended to be a living and breathing document. Please update your projects’ datasheet with new information if you have it, by creating a pull request against the shuttle repository.

Where is your design?

Go from idea to chip design in minutes, without breaking the bank.

Interested and want to get your own design manufactured? Visit our website and check out our educational material and previous submissions!

How?

New to this? Use our basic Wokwi template to see what's possible. If you're ready for more, use our advanced Wokwi template and unlock some extra pins.

Know Verilog and CocoTB? Get stuck in with our HDL templates.

When?

Multiple shuttles are run per year, meaning you've got an opportunity to manufacture your design at any time.

Stuck? Need help? Want inspiration?

Come chat to us and our community on Discord! Scan the QR code below.

Website



tinytapeout.com

Digital design guide



[tinytapeout.com/
digital_design](https://tinytapeout.com/digital_design)

Discord server



[tinytapeout.com/
discord](https://tinytapeout.com/discord)