Tiny Tapeout GF 0.2 Datasheet

github.com/TinyTapeout/tinytapeout-gf-0p2

November 24, 2025

Table of Contents

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/ttgf0p2-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 `sel` is low).
3. Outputing a counter on the output pins and the bidirectional pins (when `rst_n` is high and `sel` is high).

The following table summarizes the modes:

`rst_n`	`sel`	Mode	uo_out value	uio pins
0	X	Input mirror	ui_in	High-Z
1	0	Bidirectional mirror	uio_in	High-Z
1	1	Counter	counter	counter

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

How to test

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

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]

Zedulo TestChip1

by **Zedulo**

0032 HDL Project

`github.com/ZeduloTech/ttgf-zed_tc1`

"*Integration of UART and SPI IP's*"

How it works

See README.md

How to test

Use e.g TTL-232 to UART interface such that bytes sent via UART to `UART_RX` and observe SPI transfers on `SPI_SCL`, `SPI_CS`, and `SPI_MOSI`. Data returned from SPI is echo via `UART_TX`.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	UART_RX	UART_TX	—
1	SPI_MISO	SPI_SCL	—
2	—	SPI_CS	—
3	—	SPI_MOSI	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Wafer.space Logo VGA Screensaver

by **Uri Shaked**

0033 | 25.175 MHz | HDL Project

`github.com/TinyTapeout/tt-waferspace-vga-screensaver`

"Wafer.space Logo bouncing around the screen (640x480, TinyVGA Pmod)"

How it works

Displays a bouncing Wafer.space logo on the screen, with an animated color gradient.

Figure 33.1: Wafer.space VGA 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	—

Wildcat RISC-V

by **Martin Schoeberl**

0038 25 MHz HDL Project

`github.com/schoeberl/ttgf-wildcat`

"*Wildcat: a 3-stage RISC-V implementation*"

How it works

It is an educational RISC-V core. This tapeout contains the bare minimum: an assembler coded *blinking* LED (In fact it counts up and displays the value on the 7-segment display).

How to test

Just let it run and see the 7-segment display counting up.

External hardware

None

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

TinyQV Risc-V SoC

by **Michael Bell**

0039 24 MHz HDL Project

`github.com/MichaelBell/ttgf0p2-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.

Many of the peripherals making up the SoC are contributed by the Tiny Tapeout community!

Address map

Address range	Device
0x0000000 - 0x0FFFFFF	Flash
0x1000000 - 0x17FFFFF	RAM A
0x1800000 - 0x1FFFFFF	RAM B
0x8000000 - 0x8000033	DEBUG
0x8000040 - 0x800007F	GPIO
0x8000080 - 0x80000BF	UART
0x80000C0 - 0x80001FF	User peripherals 3-7
0x8000400 - 0x800043F	Simple user peripherals 0-3
0xFFFFF00 - 0xFFFFF07	TIME

DEBUG

Register	Address	Description

ID	0x8000008 (R)	Instance of TinyQV: 0x41 (ASCII A)
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_DIVIDER	0x800002C	MTIME counts at clock / (MTIME_DIVIDER + 1). Bits 0 and 1 are fixed at 1, so multiples of 4MHz are supported.
MTIME	0xFFFFF00 (RW)	Get/set the 1MHz time count
MTIMECMP	0xFFFFF04 (RW)	Get/set the time to trigger the timer interrupt

This is a simple timer which follows the spirit of the Risc-V timer but using a 32-bit counter instead of 64 to save area. In this version the MTIME register is updated at 1/64th of the clock frequency (nominally 1MHz), and MTIMECMP is used to trigger an interrupt. If MTIME is after MTIMECMP (by less than 2^30 microseconds to deal with wrap), the timer interrupt is asserted.

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
AUDIO_FUNC_SEL	0x8000050 (RW)	Audio function select for uo7

FUNC_SEL	0x8000060 - 0x800007F (RW)	Function select for out0-7

Function Select	Peripheral
0	Disabled
1	GPIO
2	UART
3 - 15	User peripheral 3-15
16 - 31	User byte peripheral 0-15
32 - 39	User peripheral 16-23

Audio function select	Peripheral
0-3	PSRAM B enabled
4	5 PWL Synth out 7
5	4 Pulse Transmitter out 7
6	?
7	18 Matt PWM out 7

UART

Register	Address	Description
TX_DATA	0x8000080 (W)	Transmits the byte on the UART
RX_DATA	0x8000080 (R)	Reads any received byte
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. Bit 1 indicates whether a received byte is available to be read.
DIVIDER	0x8000088 (R/W)	13 bit clock divider to set the UART baud rate
RX_SELECT	0x800008C (R/W)	1 bit select UART RX pin: `ui_in[7]` when low (default), `ui_in[3]` when high

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. SD2 selects whether half a cycle is subtracted from the read latency by driving the SPI clock on the negative edge.
- 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

At the target 24MHz clock a read latency of 1.5 is probably best (SD2:SD0 = 0b110). The maximum supported latency is 3.

The above should all be handled by some MicroPython scripts for the RP2 on the TT demo PCB.

Build programs using the customised 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.

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

SCμM-BLE-RX

by **Dingyu Zhou**

0101 16 MHz HDL Project

github.com/ZDYnb/ttgf-BLE_Receiver

"Digital Baseband for SCμM3 BLE"

How it works

This project implements a Bluetooth Low Energy (BLE) digital baseband receiver designed for the SCuM (Single-Chip μ-Mote) platform (https://crystalfree.atlassian.net/wiki/spaces/SCUM/overview). The receiver takes I/Q samples from the SCuM RF front-end (or any compatible I/Q-sample interface), processes incoming BLE signals in real time, and performs packet decoding to extract BLE packets.

Core processing stages:

- Matched filtering for GFSK demodulation and bit extraction
- Clock and data recovery for symbol-timing synchronization
- Preamble-detection module for identifying the start of a received BLE packet
- Packet-sniffer module that performs bit de-whitening, CRC checking, and detection of complete BLE packets

How to test

Connect the SCuM chip's I/Q sampling outputs to the BLE digital baseband receiver. Configure the SCuM RF front-end to receive BLE packets, and then observe the decoded packet data on a computer through the Tiny Tapeout chip's output interface.

External hardware

- SCuM Chip – Serves as the RF front-end and BLE transmitter/receiver interface.
- Digital Discovery – Used for signal probing, debugging, and verification.
- Computer – Handles serial communication, visualization, and data logging.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	I_BPF[0]	demod_symbol	channel_sel[0] (input)
1	I_BPF[1]	demod_symbol_clk	channel_sel[1] (input)
2	I_BPF[2]	packet_detected	I_BPF_echo[0] (output)
3	I_BPF[3]	preamble_detected	I_BPF_echo[1] (output)
4	Q_BPF[0]	ena_sync	I_BPF_echo[2] (output)
5	Q_BPF[1]	rst	I_BPF_echo[3] (output)
6	Q_BPF[2]	Q_BPF_echo[0]	Q_BPF_echo[2] (output)
7	Q_BPF[3]	Q_BPF_echo[1]	Q_BPF_echo[3] (output)

Silly-Faust

by **Louis Ledoux** & **Cochard Pierre**

`0102` `44.1 kHz` `HDL Project`

`github.com/Bynaryman/ttgf0p2-faust-mlir-silicon`

"Trial to use the audio circuit compiler for open silicon"

How it works

TTGF0P2 compiles the one-line Faust soft clipper `softclip(x) = tanh(3x) / tanh(3)` into silicon. The flow snapshots every MLIR stage in `src/20251113-120748-faust-tanh-softclip-switchcase/stages`, which mirrors the pipeline below:

1. **Stage 00 – Faust frontend.** Capture the canonical Faust AST with `faust.graph` ops.
2. **Stage 10 – Real arithmetic.** `--faust-to-core-real-arith --configure-faust-real-arith="config=hls-driver/pipelines/tanh-softclip-8bit-config.json"` tags the input domain `[-1, 1]` and keeps the DSP in symbolic real form.
3. **Stage 20 – Uniform piecewise fixed-point.** `--realarith-to-fixed_pt_arith` subdivides the domain into eight regions, emits Horner coefficients per region, and keeps all truncations explicit.
4. **Stage 30 – FixedPointArith to Arith.** `--fixed_pt_arith-to-arith` rewrites everything into `arith` + `scf` so the datapath is purely integer.
5. **Stage 40/50/55 – Switch normalization and CF prep.** `--switch-to-if`, `--convert-scf-to-cf`, and `--strip-real-arith-arg-attrs` sanitize control flow before Dynamatic lowers the result to handshake/RTL.

The resulting RTL lives in `src/faust_core.v` and is wrapped by `tt_um_gf0p2_faust_top`, which multiplexes between an internal ramp and the external sample bus. `ui_in[0]` selects the source (0 = internal ramp, 1 = external data on `ui_in[7:1]`), and the 8-bit result appears on `uo_out`, ready to drive an R-2R ladder.

How to test

1. Install the Python requirements (`pip install -r test/requirements.txt`).
2. Run `make -C test` for the full Cocotb suite or `make -C test sim` for the Icarus-only path.
3. Inspect `test/results.xml` or `test/tb.vcd` for pass/fail information and waveforms.

The regression toggles between the internal ramp and external stimulus to ensure the Horner pipeline matches the saved MLIR snapshots.

External hardware

- Attach the 8-bit Tiny Tapeout DAC (or Digilent Pmod R-2R) to `uo[7:0]` for line-level audio.
- Provide an optional external 7-bit sample stream on `ui[7:1]` (MSB on `ui[7]`) and raise `ui[0]` to route it through the Faust core.
- Keep every `uio` pin unconnected; they remain inputs inside the design.

The Pmod R-2R documentation: https://digilent.com/reference/pmod/pmodr2r/start

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	src_sel (0=ramp,1=external)	dac_out[0]	nc (input only)
1	ext_sample[0]	dac_out[1]	nc (input only)
2	ext_sample[1]	dac_out[2]	nc (input only)
3	ext_sample[2]	dac_out[3]	nc (input only)
4	ext_sample[3]	dac_out[4]	nc (input only)
5	ext_sample[4]	dac_out[5]	nc (input only)
6	ext_sample[5]	dac_out[6]	nc (input only)
7	ext_sample[6] (MSB)	dac_out[7] (MSB)	nc (input only)

Simon's Caterpillar

by **htfab**

0103 50 kHz HDL Project

`github.com/htfab/ttgf0p2-caterpillar`

"Port of Caterpillar Logic to Simon Says PMOD"

How it works

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

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

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

How to test

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

Exploration mode

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

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

(The empty sequence is neither valid nor invalid.)

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

- long-press red: clear buffer
- long-press yellow: erase last color from buffer ("backspace")
- long-press blue: show buffer contents (as a series of led flashes)

- long-press green: activate challenge mode
- short-press green & yellow: show a random valid sequence (and load into buffer)
- short-press red & blue: show a random invalid sequence (and load into buffer)
- short-press blue & yellow: switch to next level
- short-press red & green: switch to previous level
- short-press green & blue: toggle sound

Challenge mode

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

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

Other keys and combinations:

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

External hardware

Simon Says PMOD

Project Pinout

Digital Pins

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

VGA clock

by **Matt Venn**

0128 31.5 MHz HDL Project

github.com/mattvenn/GF180-VGA-clock-waferspace

"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	—

2x2 MAC Systolic array with DFT

by **Julia Desmazes**

`0129` `50 MHz` `HDL Project`

`github.com/Essenceia/Systolic_MAC_with_DFT`

"2x2 multiply and accumulate systolic array supporting signed 8 bit values with design for test infrastructure."

Multiply and accumulate matrix multiplier ASIC with design for test infracture

ASIC design for a 2x2 systolic matrix multiplier supporting multiply and accumulate operations on int8 data alongside a design for test infrastructure to help debug both usage and diagnose design issues in silicon.

MAC

For faster multiplication we are using the booth radix4 algorythme with wallace trees.

If the result of the MAC operation `w*i + a` exeeds the ranges of the int8, they will be clamped to `int8_min` and `int8_max`.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	tck	result_o	data_i[7]
1	data_i[0]	result_o	data_valid_i
2	data_i[1]	result_o	data_mode_i
3	data_i[2]	result_o	data_rst_addr_i
4	data_i[3]	result_o	tdi
5	data_i[4]	result_o	tms
6	data_i[5]	result_o	tdo
7	data_i[6]	result_o	result_v_o

USB CDC (Serial) Device

by **Uri Shaked**

`0130` `48 MHz` `HDL Project`

`github.com/urish/tt-usbcdc-device`

"USB to UART bridge, 115200 baud rate"

How it works

A USB CDC to UART bridge, based on tinyfpga_bx_usbserial.

How to test

1. Connect `usb_p` and `usb_n` pins to D+ / D- USB pins either through 68 ohm resistors or directly (the resistors are recommended, but not mandatory).
2. Connect a 1.5k ohm resistor between `dp_pu_o` and `usb_p` (D+).
3. Connect the RX and TX pins to a UART device or to a logic analyzer.
4. Set the clock frequency to 48 MHz.

The device should appear as a serial port on your computer, with vendor_id=1209 and product_id=5454 (https://pid.codes/1209/5454/). The baud rate for the UART interface is hardcoded at 115200.

External Hardware

USB breakout board, 1.5k ohm resistor

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	—	usb_p
1	—	—	usb_n
2	—	—	dp_pu_o
3	RX	—	—
4	—	TX	—
5	—	—	—
6	—	—	—
7	—	configured	—

VGA Drop (audio/visual demo)

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

0134 25.175 MHz HDL Project

`github.com/rejunity/tt08-vga-drop`

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

How it works

Compressed VGA Logo

How to test

Connect to VGA monitor and run it!

External hardware

TinyVGA PMOD, VGA monitor

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	TinyVGA PMOD - R1	Audio (PWM)
1	—	TinyVGA PMOD - G1	Audio (PWM)
2	—	TinyVGA PMOD - B1	Audio (PWM)
3	—	TinyVGA PMOD - VSync	Audio (PWM)
4	—	TinyVGA PMOD - R0	Audio (PWM)
5	—	TinyVGA PMOD - G0	Audio (PWM)
6	—	TinyVGA PMOD - B0	Audio (PWM)
7	—	TinyVGA PMOD - HSync	Audio (PWM)

raybox-zero TTGF0p2 edition

by **algofoogle (Anton Maurovic)**

0135 | 25.175 MHz | HDL Project

`github.com/algofoogle/ttgf0p2-raybox-zero`

"TTGF0p2 v1.8-dev submission of 'simple VGA ray caster game demo' from TT07/TTCAD25a"

Figure 135.1: TTGF0p2 raybox-zero showing 3D views including textures and doors

How it works

This is an experimental GF180 (gf180mcuD Open PDK) updated submission of ttcad25a-raybox-zero (an updated version of tt07-raybox-zero, from the raybox-zero project).

This project 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.

This version features textured walls (internally-generated or from off-chip QSPI memory), optional doors (sliding panels), flat-coloured floor and ceiling, and a variety of other rendering "hacks" for other simple visual effects. No sprites yet, 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, and note that the design may not be stable above that.

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.

`tex_pmod_type` should be set to 0 when using Leo Moser’s Tiny QSPI PMOD, or 1 for a Digilent QSPI PMOD.

Ideally the `reg` input should be high to make the VGA outputs registered. Otherwise, they are just as they come out of internal combo logic, which may not always meet timing (and hence might be unstable). I’ve done it this way so I can test the difference (if any).

`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.

`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

2048 sliding tile puzzle game (VGA)

by **Uri Shaked**

0194 | 25.175 MHz | HDL Project

`github.com/urish/tt-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

Quickscope

by **Frans Skarman**

`0196` `25 MHz` `HDL Project`

`github.com/TheZoq2/tt-um-thezoq2-quickscope`

"A quick logic analyzer"

How it works

For now, see https://blog.spade-lang.org/quickscope/

How to test

Connect the thing with UART to `uo[0] @ 115200 baud`, then run the quickscope client. When `ui[0]` is high, the logic analyzer will trigger and send the `ui[0]..[7] and uio[0]..[7]` values to the host which will generate a `.vcd` file that can be viewed in Surfer.

External hardware

You'll need a UART adapter and something to analyze

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	data_in[0] and trigger	Uart out	—
1	data_in[1]	—	—
2	data_in[2]	—	—
3	data_in[3]	—	—
4	data_in[4]	—	—
5	data_in[5]	—	—
6	data_in[6]	—	—
7	data_in[7]	—	—

LISA 8-Bit Microcontroller

by **Ken Pettit**

0197 | 18 MHz | HDL Project

`github.com/kdp1965/ttgf-um-lisa`

"8-Bit Microcontroller SOC with 128 bytes DFFRAM module"

What is LISA?

LISA is a Microcontroller built around a custom 8-Bit Little ISA (LISA) micro-processor core. It includes several standard peripherals that would be found on commercial microcontrollers including timers, GPIO, UARTs and I2C. The following is a block diagram of the LISA Microcontroller:

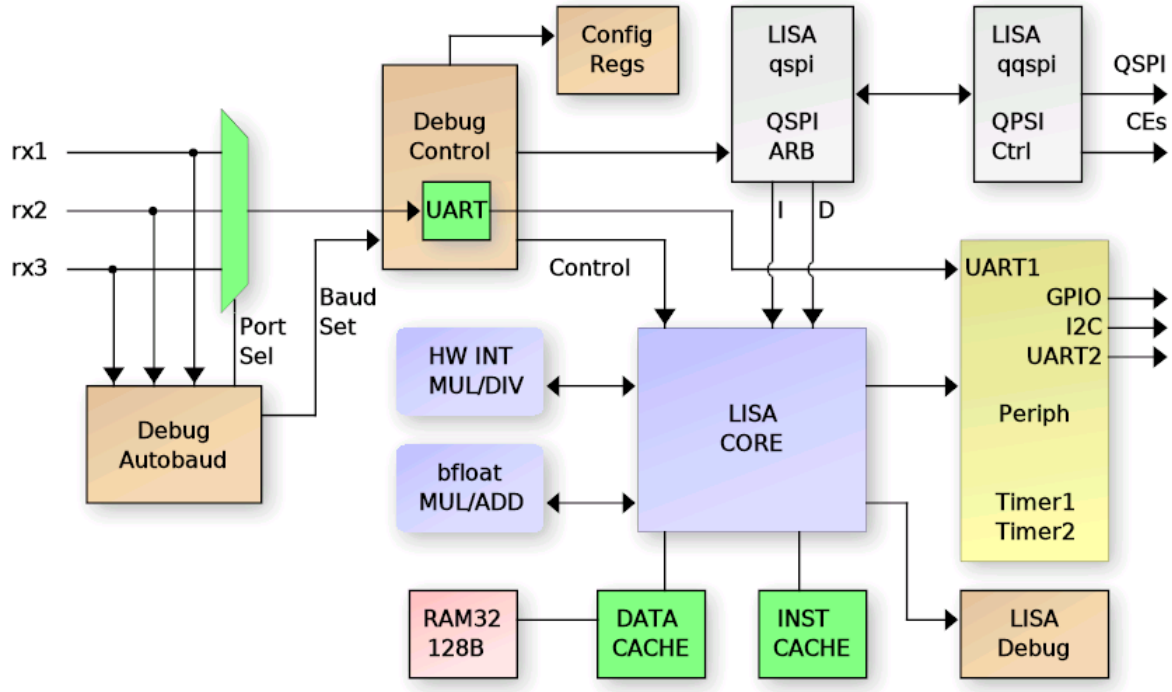


Figure 974.1: LISA Microcontroller Block Diagram

- The LISA Core has a minimal set of register that allow it to run C programs:
  - Program Counter + Return Address Resister
  - Stack Pointer and Index Register (Indexed DATA RAM access)
  - 8-bit Accumulator + 16-bit BF16 Accumulator and 4 BF16 registers

Deailed list of the features

- Harvard architecture LISA Core (16-bit instruction, 15-bit address space)
- Debug interface
  - UART controlled
  - Auto detects port from one of 3 interfaces

    - Auto detects the baud rate
    - Interfaces with SPI / QSPI SRAM or FLASH
    - Can erase / program the (Q)SPI FLASH
    - Read/write LISA core registers and peripherals
    - Set LISA breakpoints, halt, resume, single step, etc.
    - SPI/QSPI programmability (single/quad, port location, CE selects)
- (Q)SPI Arbiter with 3 access channels
    - Debug interface for direct memory access
    - Instruction fetch
    - Data fetch
    - Quad or Single SPI. Hereafter called QSPI, but supports either.
- Onboard 128 Byte RAM for DATA / DATA CACHE
- Data bus CACHE controller with 8 16-byte CACHE lines
- Instruction CACHE with a single 4-instruction CACHE line
- Two 16-bit programmable timers (with pre-divide)
- Debug UART available to LISA core also
- Dedicated UART2 that is not shared with the debug interface
- 8-bit Input port (PORTA)
- 8-bit Output port (PORTB)
- 4-bit BIDIR port (PORTC)
- I2C Master controller
- Hardware 8x8 integer multiplier
- Hardware 16/8 or 16/16 integer divider
- Hardware Brain Float 16 (BF16) Multiply/Add/Negate/Int16-to-BF16
- Programmable I/O mux for maximum flexibility of I/O usage.

It uses a 32x32 1RW DFFRAM macro to implement a 128 bytes (1 kilobit) RAM module. The 128 Byte ram can be used either as a DATA cache for the processor data bus, giving a 32K Byte address range, or the CACHE controller can be disabled, connecting the Lisa processor core to the RAM directly, limiting the data space to 128 bytes. Inclusion of the DFFRAM is thanks to Uri Shaked (Discord urish) and his DFFRAM example.

Reseting the project **does not** reset the RAM contents.

Connectivity

All communication with the microcontroller is done through a UART connected to the Debug Controller. The UART I/O pins are auto-detected by the debug_autobaud module from the following choices (RX/TX):

```
ui_in[3]  / ui_out[4]     RP2040 UART interface
uio_in[4] / uio_out[5]    LISA PMOD board (I am developing)
uio_in[6] / uio_out[5]    Standard UART PMOD
```

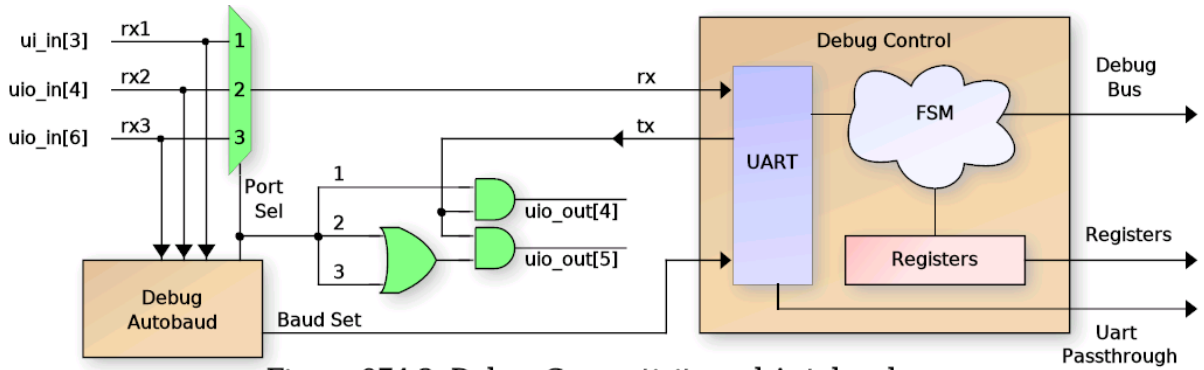


Figure 974.2: Debug Connectivity and Autobaud

The RX/TX pair port is auto-detected after reset by the autobaud circuit, and the UART baud rate can either be configured manually or auto detected by the autobaud module. After reset, the ui_in[7] pin is sampled to determine the baud rate selection mode. If this input pin is HIGH, then autobaud is disabled and ui_in[6:0] is sampled as the UART baud divider and written to the Baud Rate Generator (BRG). The value of this divider should be: clk_freq / baud_rate / 8 - 1. Due to last minute additions of complex floating point operations, and only 2 hours left on the count-down clock, the timing was relaxed to 20MHz input clock max. So for a 20MHz clock and 115200 baud, the b_div[6:0] value would be 42 (for instance).

If the ui_in[7] pin is sampled LOW, then the autobaud module will monitor all three potential RX input pins for LINEFEED (ASCII 0x0A) code to detect baud rate and set the b_div value automatially. It monitors bit transistions and searches for three successive bits with the same bit period. Since ASCII code 0x0A contains a “0 1 0 1 0” bit sequence, the baud rate can be detected easily.

Regardless if the baud rate is set manually or using autobaud, the input port selection will be detect automatically by the autobaud. In the case of manual buad rate selection, it simply looks for the first transition on any of the three RX pins. For autobaud, it selects the RX line with three successive equivalent bit periods.

Debug Interface Details

The Debug interface uses a fixed, Verilog coded Finite State Machine (FSM) that supports a set of commands over the UART to interface with the microcontroller. These commands are simple ASCII format such that low-level testing can be performed using any standard terminal software (such as minicom, tio. Putty, etc.). The ‘r’ and ‘w’ commands must be terminated using a NEWLINE (0x0A) with an optional CR (0x0D). Responses from the debug interface are always terminated with a LINFEED plus CR sequence (0x0A, 0x0D). The commands are as follows (responsce LF/CR ommited):

Command	Description

v	Report Debugger version. Should return: lisav1.2
wAAVVVV	Write 16-bit HEX value ‘VVVV’ to register at 8-bit HEX address ‘AA’.
rAA	Read 16-bit register value from 8-bit HEX address ‘AA’.
t	Reset the LISA core.
l	Grant LISA the UART. Further data will be ignored by the debugger.
+++	Revoke LISA UART. NOTE: a 0.5s guard time before/after is required.

NOTE: All HEX values must be a-f and not A-F. Uppercase is not supported.

Debug Configuration and Control Registers

The following table describes the configuration and LISA debug register addresses available via the debug ‘r’ and ‘w’ commands. The individual register details will be described in the sections to follow.

ADDR	Description	ADDR	Description
0x00	LISA Core Run Control	0x12	LISA1 QSPI base address
0x01	LISA Accumulator / FLAGS	0x13	LISA2 QSPI base address
0x02	LISA Program Counter (PC)	0x14	LISA1 QSPI CE select
0x03	LISA Stack Pointer (SP)	0x15	LISA2 QSPI CE select
0x04	LISA Return Address (RA)	0x16	Debug QSPI CE select
0x05	LISA Index Register (IX)	0x17	QSPI Mode (QUAD, flash, 16b)
0x06	LISA Data bus	0x18	QSPI Dummy read cycles
0x07	LISA Data bus address	0x19	QSPI Write CMD value
0x08	LISA Breakpoint 1	0x1a	The ‘+++’ guard time count
0x09	LISA Breakpoint 2	0x1b	Mux bits for uo_out
0x0a	LISA Breakpoint 3	0x1c	Mux bits for uio
0x0b	LISA Breakpoint 4	0x1d	CACHE control
0x0c	LISA Breakpoint 5	0x1e	QSPI edge / SCLK speed
0x0d	LISA Breakpoint 6	0x20	Debug QSPI Read / Write
0x0f	LISA Current Opcode Value	0x21	Debug QSPI custom command
0x10	Debug QSPI Address (LSB16)	0x22	Debug read SPI status reg

0x11	Debug QSPI Address (MSB8)	—	—

LISA Processor Interface Details

The LISA Core requires external memory for all Instructions and Data (well, sort of for data, the data CACHE can be disabled then it just uses internal DF-FRAM). To accomodate external memory, the design uses a QSPI controller that is configurable as either single SPI or QUAD SPI, Flash or SRAM access, 16-Bit or 24-Bit addressing, and selectable Chip Enable for each type of access. To achieve this, a QSPI arbiter is used to allow multiple accessors as shown in the following diagram:

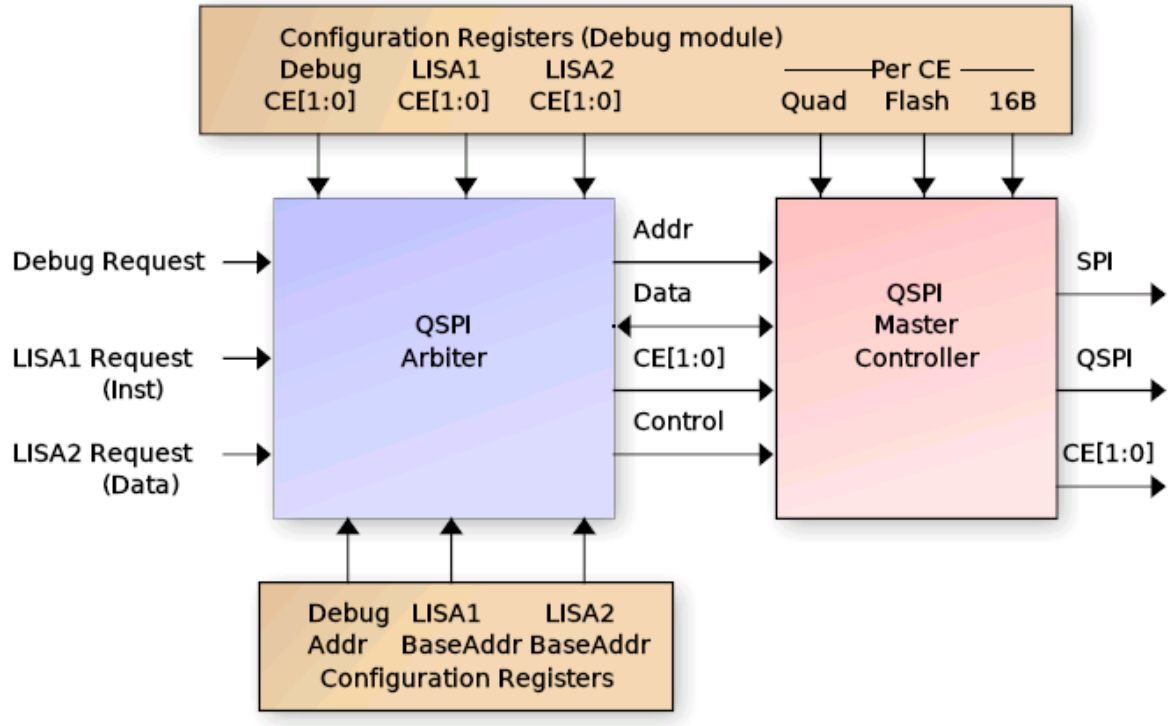


Figure 974.3: (Q)SPI Controller Interface Diagram

The arbiter is controlled via configuration registers (accessible by the Debug controller) that specify the operating mode per CE, and CE selection bits for each of the three interfaces:

- Debug Interface
- LISA1 (Instruction fetch)
- LISA2 (Data read/write)

The arbiter gives priority to the Debug accesses and processes LISA1 and LISA2 requests using a round-robbin approach. Each requestor provides a 24-bit address along with 16-bit data read/write. For the Debug interface, the address comes from the configuration registers directly. For LISA1, the address is the Program Counter (PC) + LISA1 Base and for LISA2, it is the

Data Bus address + LISA2 Base. The LISA1 and LISA2 base addresses are programmed by the Debug controller and set the upper 16-bits in the 24-bit address range. The PC and Data address provide the lower 16 bis (8-bits overlapped that are 'OR'ed together). The BASE addresses allow use of a single external QSPI SRAM for both instruction and data without needing to worry about data collisions.

When the arbiter chooses a requestor, it passes its programmed CE selection to the QSPI controller. The QSPI controller then uses the programmed QUAD, MODE, FLASH and 16B settings for the chosen CE to process the request. This allows LISA1 (Instruction) to either execute from the same SRAM as LISA2 (Data) or to execute from a separate CE (such as FLASH with permanent data storage).

Additionally the Debug interface has special access registers in the 0x20 - 0x22 range that allow special QSPI accesses such as FLASH erase and program, SRAM programming, FLASH status read, etc. In fact the Debug controller can send any arbitrary command to a target device, using access that either provide an associated address (such as erase sector) or no address. The proceedure for this is:

1. Program Debug register 0x19 with the special 8-bit command to be sent
2. Set the 9-th bit (reg19[8]) to 1 if a 16/24 bit address needs to be sent)
3. Perform a read / write operation to debug address 0x21 to perform the action.

Simple QSPI data reads/write are accomplished via the Debug interface by setting the desired address in Debug config register 0x10 and 0x11, then performing read or write to address 0x20 to perform the request. Reading from Debug config register 0x22 will perform a special mode read of QSPI register 0x05 (the FLASH status register).

Data access to the QSPI arbiter come from the Data CACHE interface (described later), enabling a 32K address space for data. However the design has a CACHE disable mode that directs all Data accesses directly to the internal 128 Byte RAM, thus eliminating the need for external SRAM (and limiting the data bus to 128 bytes).

Programming the QSPI Controller

Before the LISA microcontroller can be used in any meaningful manner, a SPI / QSPI SRAM (and optionally a NOR FLASH) must be connected to the Tiny Tapeout PCB. Alternately, the RP2040 controller on the board can be configured to emulate a single SPI (the details for configuring this are outside the scope of this documentation ... search the Tiny Tapeout website for details.). For the CE signals, there are two operating modes, fixed CE output and Mux Mode 3 "latched" CE mode. Both will be described here. The other standard SPI signals are routed to dedicated pins as follows:

Pin	SPI	QSPI	Notes
uio[0]	CE0	CE0	—
uio[1]	MOSI	DQ0	Also MOSI prior to QUAD mode DQ0
uio[2]	MISO	DQ1	Also MISO prior to QUAD mode DQ1
uio[3]	SCLK	SCLK	—
uio[4]	CE1	CE1	Must be enabled via uio MUX bits
uio[6]	-	DQ2	Must be enabled via uio MUX bits
uio[7]	-	DQ3	Must be enabled via uio MUX bits

For Special Mux Mode 3 (Debug register 0x1C uio_mux[7:6] = 2'h3), the pinout is mostly the same except the CE signals are not constant. Instead they are "latched" into an external 7475 type latch. This mode is to support a PMOD board connected to the uio PMOD which supports a QSPI Flash chip, a QSPI SRAM chip, and either Debug UART or I2C. For all of that functionality, nine pins would be required for continuous CE0/CE1, however only eight are available. So the external PMOD uses uio[0] as a CE "latch" signal and the CE0/CE1 signals are provided on uio[1]/uio[2] during the latch event. This requires a series resistor as indicated to allow CE updates if the FLASH/SRAM is driving DQ0/DQ1. The pinout then becomes:

Pin	SPI/QSPI	Notes
uio[0]	ce_latch	ce_latch HIGH at beginning of cycle
uio[1]	ce0_latch/MOSI/DQ0	Connection to FLASH/SRAM via series resistor
uio[2]	ce1_latch/MISO/DQ1	Connection to FLASH/SRAM via series resistor
uio[3]	SCLK	—
uio[6]	-/DQ2	Must be enabled via uio MUX bits
uio[7]	-/DQ3	Must be enabled via uio MUX bits

This leaves uio[4]/uio[5] available for use as either UART or I2C.

Once the SPI/QSPI SRAM and optional FLASH have been chosen and connected, the Debug configuration registers must be programmed to indicate the nature of the external device(s). This is accompilished using Debug registers 0x12 - 0x19 and 0x1C. To programming the proper mode, follow these steps:

1. Program the LISA1, LISA2 and Debug CE Select registers (0x14, 0x15, 0x16) indicating which CE to use.
   - 0x14, 0x15, 0x16: {6'h0, ce1_en, ce0_en} Active HIGH

2. Program the LISA1 and LISA2 base addresses if they use the same SRAM:
    - 0x12: {LISA1_BASE, 8'h0} | {8'h0, PC}
    - 0x13: {LISA2_BASE, 8'h0} | {8'h0, DATA_ADDR}
3. Program the mode for each Chip Enable (bits active HIGH)
    - 0x17: {10'h0, is_16b[1:0], is_flash[1:0], is_quad[1:0]}
4. For Quad SPI, Special Mux Mode 3, or CE1, program the uio_mux mode:
    - 0x1C:
        - [7:6] = 2'h2: Normal QSPI DQ2 select
        - [7:6] = 2'h3: Special Mux Mode 3 (Latched CE)
        - [5:4] = 2'h2: Normal QSPI DQ3 select
        - [5:4] = 2'h3: Special Mux Mode 3
        - [1:0] = 2'h2: CE1 select on uio[4]
5. For RP2040, you might need to slow down the SPI clock / delay between successive CE activations:
    - 0x1E:
        - [3:0] spi_clk_div: Number of clocks SCLK HIGH and LOW
        - [10:4] ce_delay: Number clocks between CE activations
        - [12:11] spi_mode: Per-CE FALLING SCLK edge data update
6. Set the number of DUMMY ready required for each CE:
    - 0x18: {8'h0, dummy1[3:0], dummy0[3:0]
7. For QSPI FLASH, set the QSPI Write opcode (it is different for various Flashes):
    - 0x19: {8'h0, quad_write_cmd}

NOTE: For register 0x1E (SPI Clock Div and CE Delay), there is only a single register, meaning this register value applies to both CE outputs. Delaying the clock of one CE will delay both, and adding delay between CE activations does not keep track of which CE was activated. So if two CE outputs are used and a CE delay is programmed, it will enforce that delay even if a different CE is used. This setting is really in place for use when the RP2040 emulation is being used in a single CE SRAM mode only (i.e. you have no external PMOD with a real SRAM / FLASH chip. In the case of real chips on a PMOD, SCLK and CE delays (most likely) are not needed. The Tech Page on the Tiny Tapeout regarding RP2040 SPI SRAM emulation indicates a delay between CE activations is likely needed, so this setting is provided in case it is needed.

Architecture Details

Below is a simplified block diagram of the LISA processor core. It uses an 8-bit accumulator for most of its operations with the 2nd argument predominately coming from either immediate data in the instruction word or from a memory location addressed by either the Stack Pointer (SP) or Index Register (IX).

There are also instructions that work on the 15-bit registers PC, SP, IX and RA (Return Address). As well as floating point operations. These will be covered in the sections to follow.

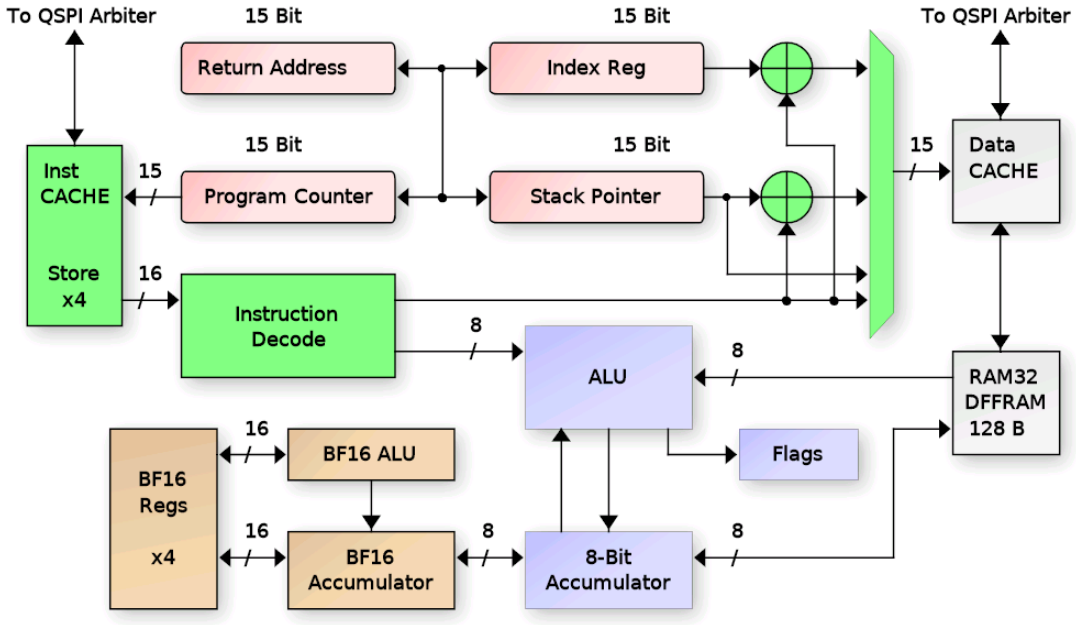


Figure 974.4: Simplified LISA Processor Block Diagram

Addressing Modes

Like most processors, LISA has a few different addressing modes to get data in and out of the core. These include the following:

Mode	Data	Description
Register	Rx[n -: 8]	Transfers between registers (ix, ra, facc, etc.).
Direct	inst[n:0]	N-bit data stored in the instruction word.
NextOp	(inst+1)[14:0]	Data stored in the NEXT instruction word.
Indirect	mem[inst[n:0]]	Address of the data is in the instruction word.
Periph	periph[inst[n:0]]	Accesses to the peripheral bus.
Indexed	mem[sp/ix+inst[n:0]]	The SP or IX register is added to a fixed offset.
Stack	mem[sp]	Stack pointer points to the data (push/ pop).

The Control Registers

To run meaninful programs, the Program Counter (PC) and Stack Pointer (SP) must be set to useful values for accessing program instructions and data. The

PC is automatically reset to zero by rst_n, so that one is pretty much automatic. All programs start at address zero (plus any base address programmed by the Debug Controller). But as far as the LISA core is concerned, it knows nothing of base addresses and believes it is starting at address zero.

Next is to program the SP to point to a useful location in memory. The Stack is a place where C programs store their local variable values and also where we store the Return Address (RA) if we need to call nested routines, etc. The stack grows down, meaning it starts at a high RAM address and decrements as things are added to the stack. Therefore the SP should be programmed with an address in upper RAM. LISA supports different Data bus modes through it's CACHE controller, including CACHE disable where it can only access 128 bytes. But for this example, let's assume we have a full range of 32K SRAM available. The LISA ISA doesn't have an opcode for loading the SP directly. Instead it can load the IX register directly with a 15-bit value using NextOp addressing, and it supports "xchg" opcodes to exchange the IX register with either the SP or RA. So to load the SP, we would write:

```
Example:
  ldx     0x7FFF      // Load IX with value in next opcode
  xchg_sp             // Exchange IX with SP
```

The IX register can be programmed as needed to access other data within the Data Bus address range. This register is useful especially for accessing structures via a C pointer. The IX then becomes the value of the pointer to RAM, and Indexed addressing mode allows fixed offsets from that pointer (i.e. structure elements) to be accessed for read/write.

Loading the PC indirectly can be done using the "jmp ix" opcode which does the operation pc <= ix. Loading ix from the pc directly is not supported, though this can be accomplished using a function call and opcodes to save RA (sra) and pop ix:

```
 Example:
   get_pc:
     sra          // Push RA to the stack (Save RA)
     pop_ix       // Pop IX from the stack
     ret          // Return. Upon return, IX is the same as PC
```

Conditional Flow Processing

Program flow is controlled using flags (zero, carry, sign), arithemetic mode (amode) and condition flags (cond) to determine when program branches should occur. Specific opcode update the flags and condition registers based on results of the operation (AND, OR, IF, etc.). Then conditional branches are made using bz, bnz and if (and variants ifte "if-then-else" and iftt "if-then-then"). Also available are rc "Return if Carry" and rz "Return if Zero", though these are less useful in C programs as typically a routine uses local variables

and the stack must be restored prior to return, mandating a branch to the function epilog to restore the stack and often the return address. Below is a list of the opcodes used for conditional program processing:

Legend for operations below:

- acc_val = inst[7:0]
- pc_jmp = inst[14:0]
- pc_rel = pc + sign_extend(inst[10:0])

Opcode	Operation	Encoding	Description
jal	pc <= pc_jmp	0aaa_aaaa_aaaa_aaaa	Jump And Link (call).
—	ra <= pc	—	—
ret	pc <= ra	1000_1010_0xxx_xxxx	Return
reti	pc <= ra	1000_11xx_iiii_iiii	Return Immediate.
—	acc <= acc_val	—	—
br	pc <= pc_rel	1011_0rrr_rrrr_rrrr	Branch Always
bz	pc <= pc_rel	1011_1rrr_rrrr_rrrr	Branch if Zero.
—	if zero=1	—	—
bnz	pc <= pc_rel	1010_1rrr_rrrr_rrrr	Branch if Not Zero.
—	if zero=0	—	—
rc	pc <= ra	1000_1011_0xxx_xxxx	Return if Carry
—	if carry=1	—	—
rz	pc <= ra	1000_1011_1xxx_xxxx	Return if Zero
—	if zero=1	—	—
call_ix	pc <= ix	1000_1010_100x_xxxx	Call indirect via IX
—	ra <= pc	—	—
jump_ix	pc <= ix	1000_1010_101x_xxxx	Jump indirect via IX
if	cond <= ??	1010_0010_0000_0ccc	If. See below.
iftt	cond <= ??	1010_0010_0000_1ccc	If then-then. See below.
ifte	cond <= ??	1010_0010_0001_0ccc	If then-else. See below.

The IF Opcode

The “if” opcode and it’s variants “if-then-then” and “if-then-else” control program flow in a slightly different manner than the others. Instead of affecting the value of the PC directly, they set the two condition bits “cond[1:0]” to indicate which (if any) of the two following opcodes should be executed. the cond[0] bit represents the next instruction and cond[1] represents the instruction following that. All three “if” forms take an argument that checks

the current value of the FLAGS to set the condition bits. The argument is encoded as the lower three bits of the instruction word ard operate as shown in the following table:

Condition	Test	Encoding	Description
EQ	zflag=1	3'h0	Execute if Equal
NE	zflag=0	3'h1	Execute if Not Equal
NC	cflag=0	3'h2	Execute if Not Carry
C	cflag=1	3'h3	Execute if Carry
GT	cSigned & zflag	3'h4	Execute if Greater Than
LT	cSigned & zflag	3'h5	Execute if Less Than
GTE	cSigned	zflag	3'h6
LTE	cSigned	zflag	3'h7

The "if" opcode will set cond[0] based on the condition above and the cond[1] bit to HIGH. It only affects the single instruction following the "if" opcode. The "iftt" opcode will set both cond[0] and cond[1] to the same value based on the condition above. It means "if true, execute the next two opcodes". And the "ifte" opcode will set cond[0] based on the condition above and cond[1] to the OPPOSITE value, meaning it will execute either the following instruction OR the one after that (then-else).

```
Example:
  ldi     0x41        // Load A immediate with ASCII 'A'
  cpi     0x42        // Compare A immediate with ASCII 'B'
  ifte    eq          // Test if the compare was "Equal"
  jal     L_equal     // Jump if equal
  jal     L_different // Jump if different
```

The above code will load the "jal L_equal" opcode but will not execute it since the compare was Not Equal. Then it will execute the "jal L_different" opcode. Note that if the compare were "ifte ne", it would call the L_equal function and then upon return would not execute the "L_different" opcode. This is because the cond[1] code is saved with the Return Address (RA) during the call and restored upon return. This means the FALSE cond[1] code would prevent the 2nd opcode from executing. As an opcode gets executed, the cond[1] value is shifted into the cond[0] location, and the cond[1] is loaded with 1'b1.

Direct Operations

To do any useful work, the LISA core must be able to load and operate on data. This is done through the accumulator using the various addressing modes. The diagram below details the Direct addressing mode where data is stored directly in the opcode / instruction word:

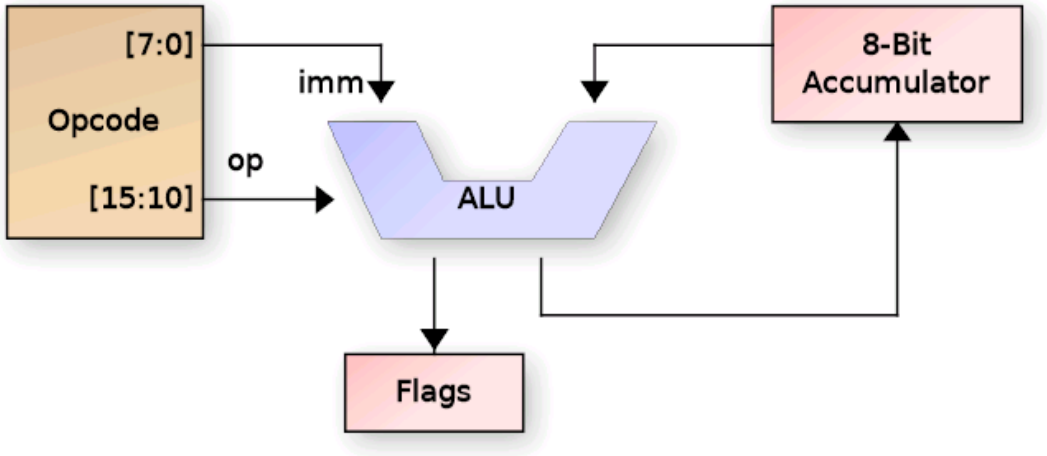


Figure 974.5: Accumulator Direct Operations Diagram

The instructions that use direct addressing are:

Opcode	Operation	Encoding	Description
adc	A <= A + imm + C	1001_00xx_iiii_iiii	ADD immediate with Carry
ads	SP <= SP + imm	1001_01ii_iiii_iiii	ADD SP + signed immediate
adx	IX <= IX + imm	1001_10ii_iiii_iiii	ADD IX + signed immediate
andi	A <= A & imm	1000_01xx_iiii_iiii	AND immediate with A
cpi	Z,C <= A >= imm	1010_01xx_iiii_iiii	Compare A >= immediate
cpi	Z,C <= A >= imm	1010_01xx_iiii_iiii	Compare A >= immediate

Accumulator Indirect Operations

The Accumulator Indirect operations use immediate data in the instruction word to index indirectly into Data memory. That memory address is then used to load, store or both load and store (swap) data with the accumulator.

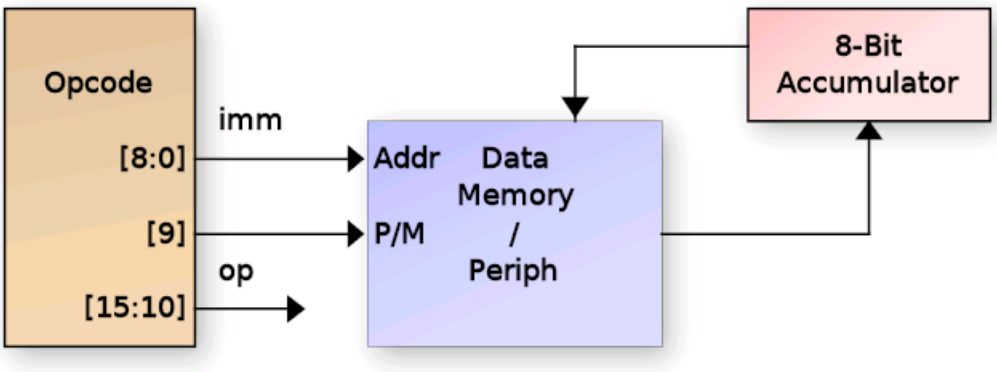


Figure 974.6: Accumulator Indirect Operations Diagram

Opcode	Operation	Encoding	Description
lda	A <= M[imm]	1111_01pi_iiii_iiii	Load A from Memory/Peripheral
sta	M[imm] <= A	1111_11pi_iiii_iiii	Store A to Memory/Peripheral
swapi	A <= M[imm]	1101_11pi_iiii_iiii	Swap Memory/Peripheral with A
—	M[imm] <= A	—	—

- p = Select Peripheral (1'b1) or RAM (1'b0)
- iiii = Immediate data

Indexed Operations

Indexed operations use either the IX or SP register plus a fixed offset from the immediate field of the opcode. The selection to use IX vs SP is also from the opcode[9] bit. The immediate field is not sign extended, so only positive direction indexing is supported. This was selected because this mode is typically used to access either local variables (when using SP) or C struct members (when using IX), and in both cases, negative index offsets aren't very useful. The following is a diagram of indexed addressing:

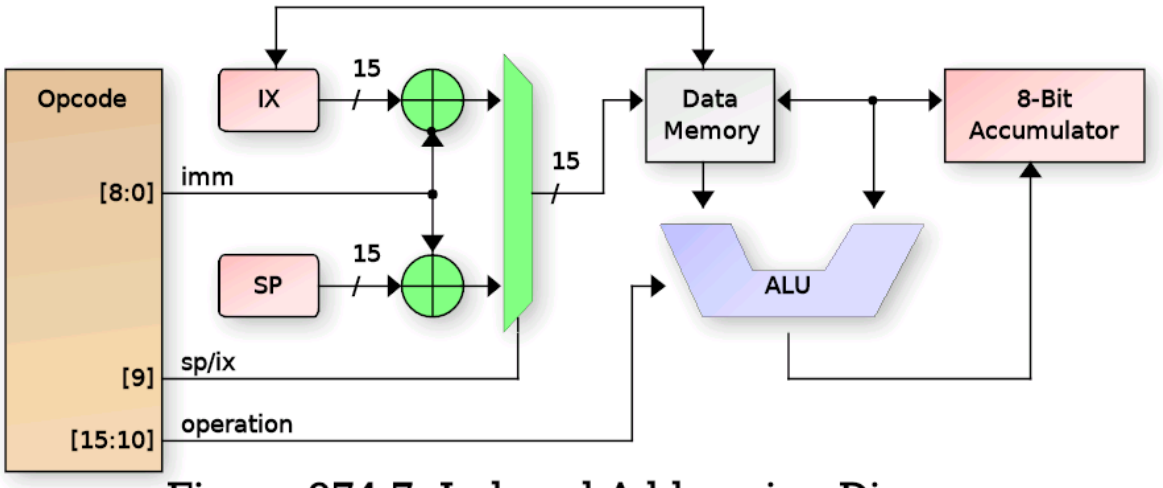


Figure 974.7: Indexed Addressing Diagram

Opcode	Operation	Encoding	Description
add	A <= A+ M[ind]	1100_00si_iiii_iiii	ADD index memory to A
and	A <= A & M[ind]	1101_00si_iiii_iiii	AND A with index memory
cmp	A >= M[ind]?	1110_10si_iiii_iiii	Compare A with index memory
dcx	M[ind] -= 1	1001_11si_iiii_iiii	Decrement the value at index memory
inx	M[ind] += 1	1110_01si_iiii_iiii	Increment the value at index memory

ldax	A <= M[ind]	1111_00si_iiii_iiii	Load A from index memory
ldxx	IX <= M[SP+imm]	1100_110i_iiii_iiii	Load IX from memory at SP+imm
mul	A <= A*M[ind]L	1100_10si_iiii_iiii	Multiply index memory * A, keep LSB
mulu	A <= A*M[ind]H	1000_01si_iiii_iiii	Multiply index memory * A, keep MSB
or	A <= A	M[ind]	1101_10si_iiii_iiii
stax	M[ind] <= A	1111_10si_iiii_iiii	Store A to index memory
stxx	M[SP+imm] <= IX	1100_111i_iiii_iiii	Save IX to memory at SP+imm
sub	A <= A-M[ind]	1100_10si_iiii_iiii	SUBtract index memory from A
swap	A <= M[ind]	1110_11si_iiii_iiii	Swap A with index memory
—	M[ind] <= A	—	—
xor	A <= A ^ M[ind]	1110_00si_iiii_iiii	XOR A with index memory

Legend for table above:

- ind = IX or SP + immediate
- s = Select IX (zero) or SP (one)
- iiii = Immediate data

The Zero and Carry flags are updated for most of the above operations. The Carry flag is only updated for math operations where a Carry / Borrow could occur.

Carry	Zero
adc	add and
add	or xor
sub	cmp sub
cmp	dcx inx
dcx	swap ldax
inx	mul mulu

Stack Operations

Stack operations use the current value of the SP register to PUSH and POP items to the stack in opcode. As items are PUSHed to the stack, the SP is

decremented after each byte, and as they are POPed, the SP is incremented prior to reading from RAM.

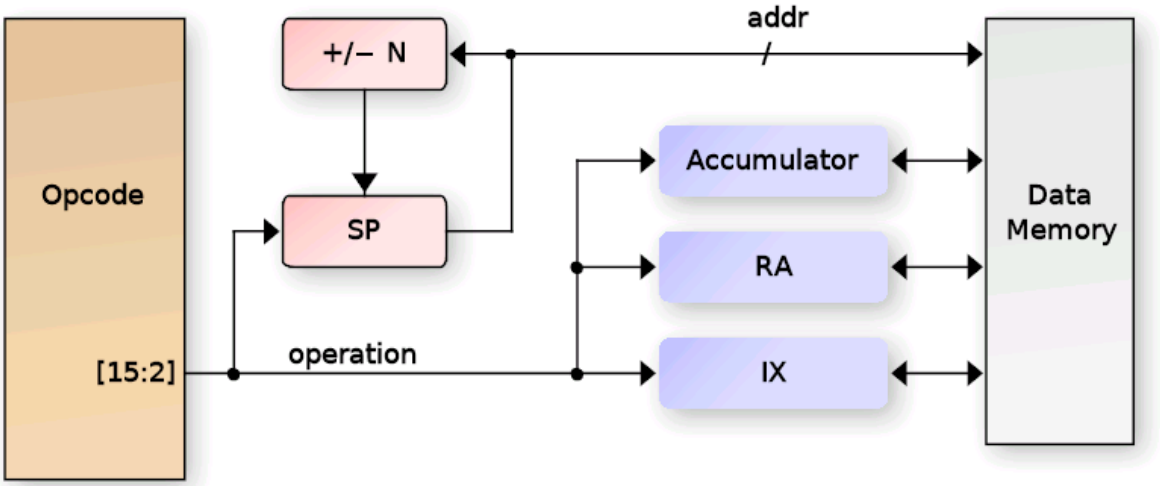


Figure 974.8: Stack Addressing Diagram

Opcode	Operation	Encoding	Description
lra	RA <= M[SP+1]	1010_0001_0110_01xx	Load {cond,RA} from stack
—	SP += 2	—	—
sra	M[SP] <= RA	1010_0001_0110_00xx	Save {cond,RA} to stack
—	SP -= 2	—	—
push_ix	M[SP] <= IX	1010_0001_0110_10xx	Save IX to stack
—	SP -= 2	—	—
pop_ix	IX <= M[SP+1]	1010_0001_0110_11xx	Load IX from stack
—	SP += 2	—	—
push_a	M[SP] <= A	1010_0000_100x_xxxx	Save A to stack
—	SP -= 1	—	—
pop_a	A <= M[SP+1]	1010_0000_110x_xxxx	Load A from stack
—	SP += 1	—	—

How to test

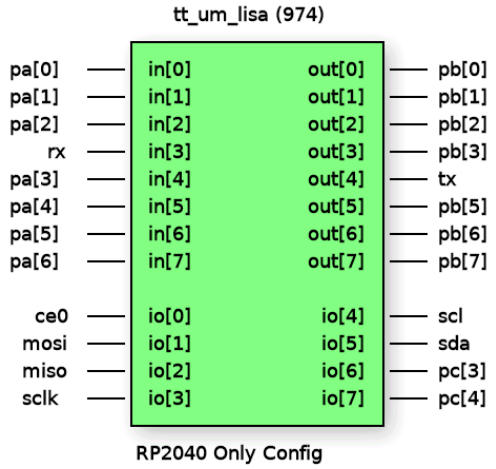
You will need to download and compile the C-based assembler, linker and C compiler I wrote (will make available) Also need to download the Python based debugger.

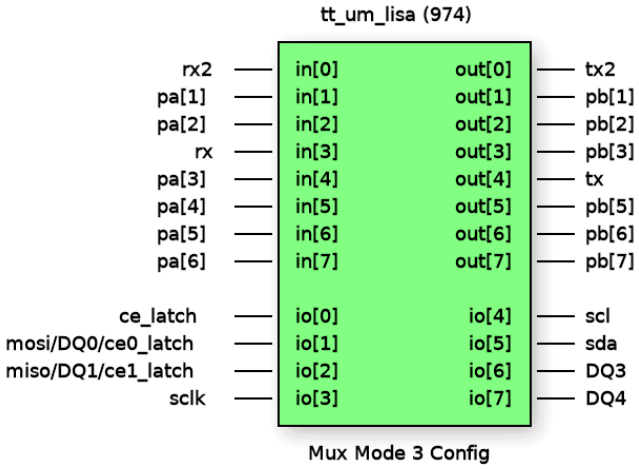
- Assembler is fully functional
  - Includes limited libraries for crt0, signed int compare, math, etc.

    - Libraries are still a work in progress
- Linker is fully functional
- C compiler is somewhat functional (no float support at the moment) but has *many* bugs in the generated code and is still a work in progress.
- Python debugger can erase/program the FLASH, program SPI SRAM, start/stop the LISA core, read SRAM and registers.

Legend for Pinout

- pa: LISA GPIO PortA Input
- pb: LISA GPIO PortB Output
- b_div: Debug UART baud divisor sampled at reset
- b_set: Debug UART baud divisor enable (HIGH) sampled at reset
- baud_clk: 16x Baud Rate clock used for Debug UART baud rate generator
- ce_latch: Latch enable for Special Mux Mode 3 as describe above
- ce0_latch: CE0 output during Special Mux Mode 3
- ce1_latch: CE1 output during Special Mux Mode 3
- DQ1/2/3/4: QUAD SPI bidirection data I/O
- pc_io: LISA GPIO Port C I/O (direction controllable by LISA)





Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	pa[0]/b_div[0]/rx2	pb[0]/tx2	ce0/ce_latch
1	pa[1]/b_div[1]/rx2	pb[1]/tx2	mosi/dq1/ce0_latch
2	pa[2]/b_div[2]/rx2	pb[2]/tx2	miso/dq2/ce1_latch
3	pa[3]/b_div[3]/rx	pb[3]	sclk
4	pa[4]/b_div[4]	pb[4]/tx	rx /pc_io[0]/scl/ce1
5	pa[5]/b_div[5]	pb[5]	tx /pc_io[1]/sda
6	pa[6]/b_div[6]	pb[6]	scl /pc_io[2]/dq2/rx

#	Input	Output	Bidirectional
7	pa[7]/b_set(autobaud_disable)	pb[7]/baud_clk	sda/pc_io[3]/dq3

easy PAL

by **Ken Pettit**

0198 HDL Project

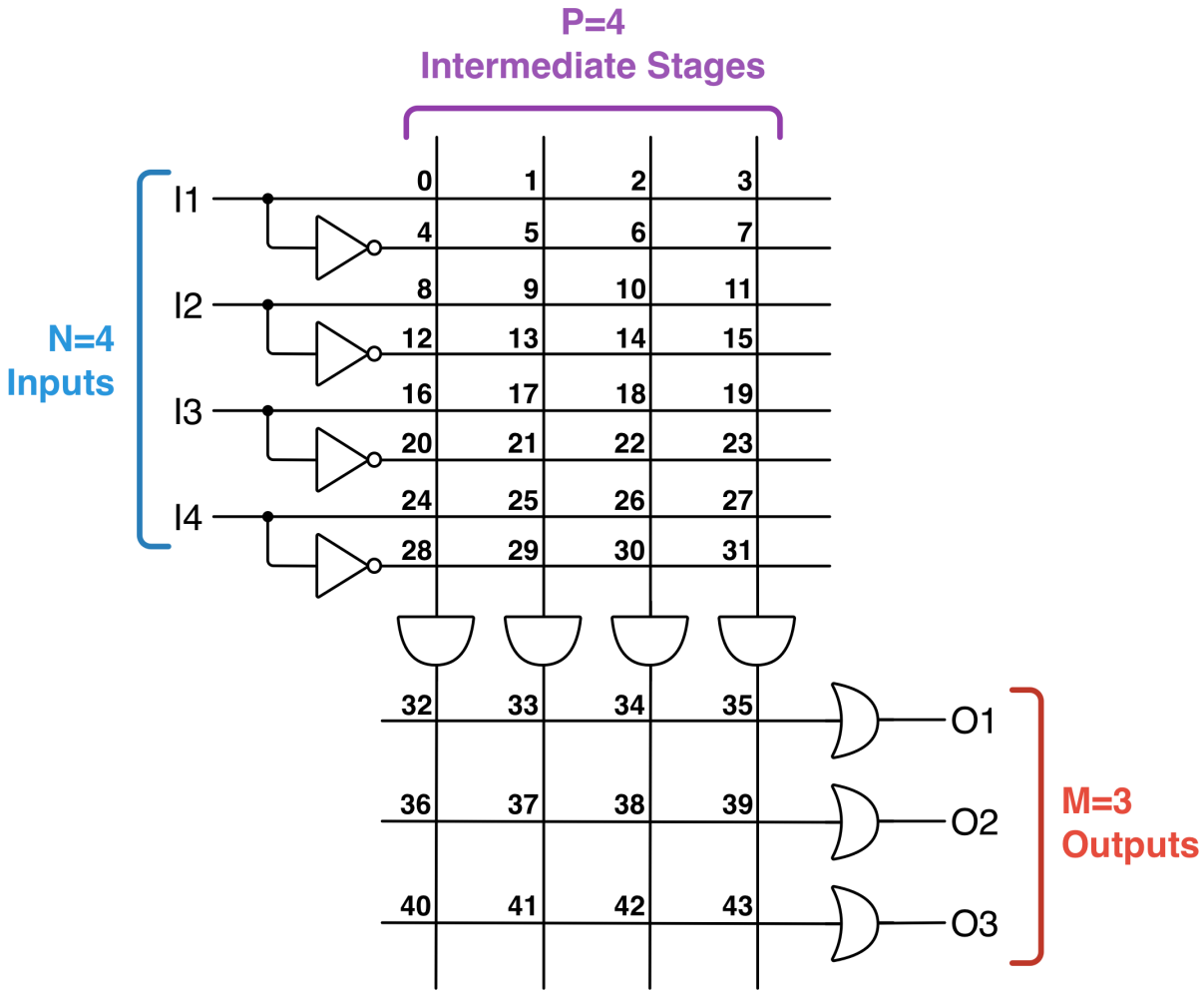
`github.com/kdp1965/ttgf-um-easy-PAL`

"This is a simple PAL (by Matthias Musch) using shift-register based configuration"

How it works

This project is a PAL (programmable array logic device). It is programmed with a shift register. It was written by Matthias Musch and included in TTGF180 by me (Ken).

easy_pal is a simple and naive PAL implementation that can be (re)programmed via a shift-register chain. The PAL is fully parametric and thus number of inputs (N), number of intermediate stages (P) and the number of outputs (M) can be configured in a flexible way in the verilog sources.



Example configuration

To generate a bitstream the python script has to be run. In the top of the file the logic function and the size of the PAL-device has to be provided. After displaying the truth table the script generates the following output:

```
Bitstream is:
000000110000000100000010110
Visualization of bitstream:
I0 -> 000
~I0-> 110
I1 -> 000
~I1-> 000
I2 -> 100
~I2-> 000
I3 -> 010
~I3-> 110
      &&&
      110-> 00
---
Bitstream for verilog is:
27'b000000110000000100000010110
```

The logic function was given in the following way in the Python code: “O0 = I0 | I1 & (I2 & I3)”

Looking at the following waveform we can see that it does indeed work! :)

Taped-out configuration and pin assignment

Because I do not want to update the text below too often I write the configuration of the physical PAL device in terms of:

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

- 7 outputs

Pin assignment

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

Programming

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

Generate bitstreams

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

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

Using the PAL

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

How to test

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

External hardware

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

Project Pinout

Digital Pins

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

VGA Nyan Cat

by **Andy Sloane**

0199 | 25.175 MHz | HDL Project

`github.com/a1k0n/ttgf0p2-a1k0n-nyan`

"Displays the classic nyan.cat animation"

VGA nyan cat

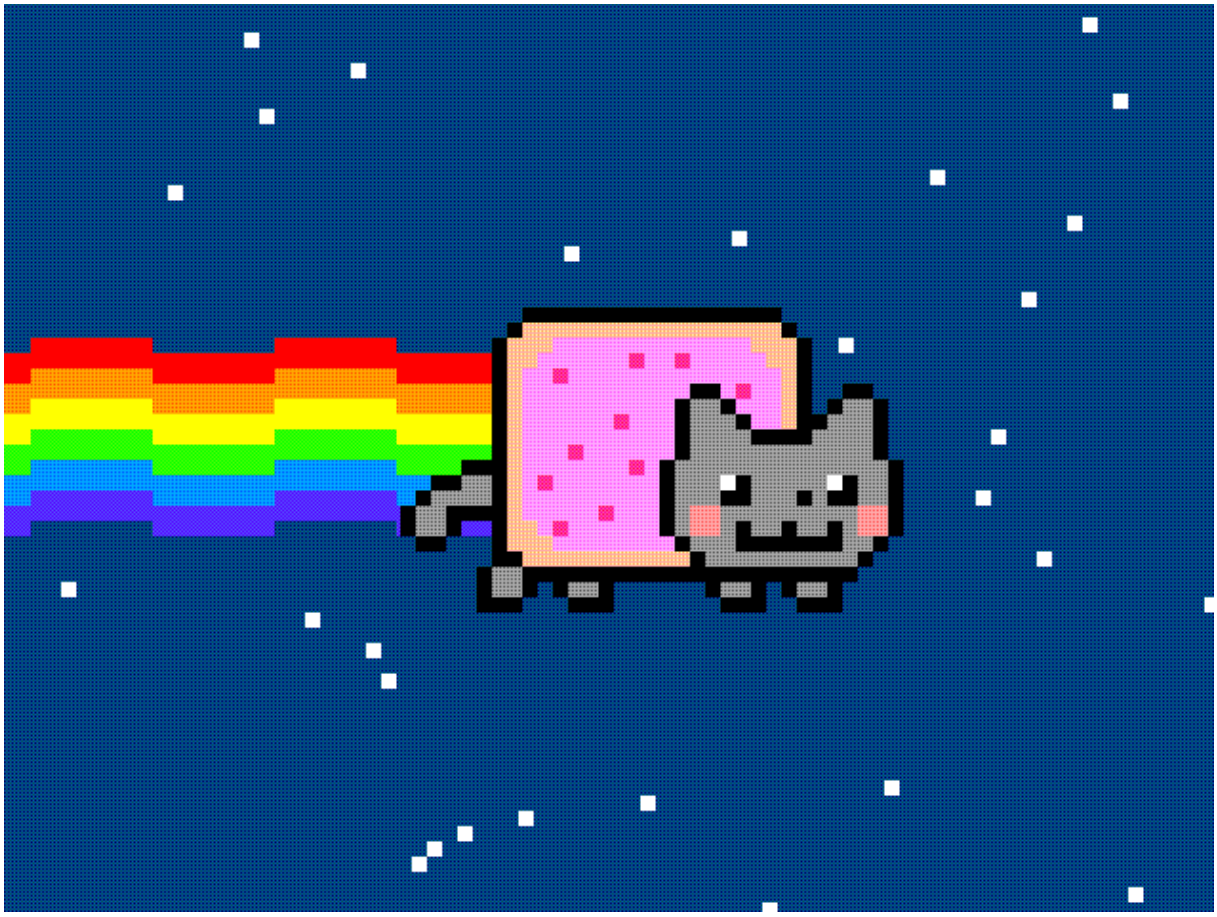


Figure 199.1: nyancat preview

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

Notre Dame - Lockpick Game TT Example

by **Matthew Morrison**

288 | 7 MHz | HDL Project

github.com/mmorri22/lockpick

"An example SystemVerilog Implementation of a pipelined input/output for future TinyTapeout use in the classroom"

How it works

There are two 256-bit inputs that are pipelined through the 8-bit inputs using the enable signal

How to test

The testbench lockpick_game.sv is included to test the design.

External hardware

We will use the conventional 8-bit LED on the tiny tapeout, as well as the output_valid signal to verify that the output is correct.

Project Pinout

Digital Pins

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

KianV uLinux SoC

by **Hirosh Dabui**

0295 HDL Project

`github.com/TinyTapeout/KianV-RV32IMA-RISC-V-uLinux-SoC`

"A RISC-V ASIC that can boot µLinux."

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

Notre Dame - CSE 30342 - DIC - Advanced FSM Final Project Example

by **Matthew Morrison**

358 | 10 kHz | HDL Project

github.com/mmorri22/tt_um_mmorri22_cse_30342

"*Example of SystemVerilog code with FSM and IO pipelining.*"

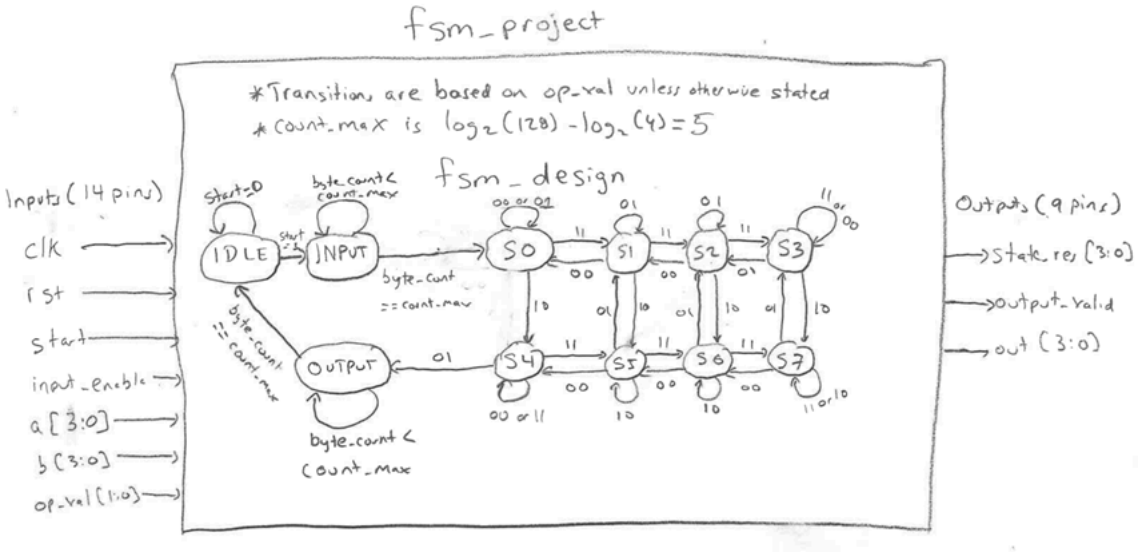
How it works

This is an example final project SystemVerilog example for the CSE 30342 Digital Integrated Circuits course at the University of Notre Dame The objective of this project is to develop a robust finite-state machine that instructs students on how to implement a design in their final course project. Since the project requirement is that the Finite State Machine has at least five states (not including the IDLE, INPUT, and OUTPUT states) where at least two of those five have a self-feedback loop, and two of those five have an edge that goes backwards, this design goes to an extreme where every state has a self-feedback loop and every state has at least one edge that goes backwards. I will explain how this FSM meets the requirement in this report.

The require project diagram is listed below. In the student project, there will be a purpose behind the FSM, such as a vending machine or stop light or something the students have developed themselves. In this section, describe the purpose and operation of your design.

This purpose of this project is a "memory calculator", where the output of the states are driven by the initial 128-bit inputs a and b, as well as the previous result. Each state has a unique calculation that are designed to help students understand certain SystemVerilog coding techniques, such as concatenation, as well as an example of how to implement modulo using loops, which is one of the forbidden operators in Genus Synthesis Solution. List each input and output, and state the count for ease of grading. The project has 23 total pins, meeting the project requirement. To meet the input and output requirements of the project, the pins are implemented as follows: • One clk pin • One rst pin • One start pin • One input_enable pin • A four-bit input for a • A four-bit input for b • A two-bit op_val, since each state can have up to 4 possible transitions • A four-bit out signal representing the output • A one-bit output_valid signal • A four-bit signal representing the current state, since there are 11 possible states. In the INPUT state, the inputs for a and b are read in parallel, and are controlled using an input_enable signal. Parameter N is set to 128 by default, and N_width is set to 4. The INPUT state requires

the input_enable signal to be positive to overwrite the previous result. This choice meets the requirement that there be at least one input that is 32-bits in length. The OUTPUT state pipelines the output on the out pin, and is controlled by sending an output_valid signal.



In this design, we define the “internal states” as every state with the exception of IDLE, INPUT, and OUTPUT. Every internal state has a feedback loop and a return edge, with the exception of S0, which does not have a return edge, meeting the project requirements that at least two internal states have a feedback loop, and that two internal states have a return edge. For example, S1’s feedback loop occurs when op_val is 01, and its feedback loop occurs when op_val is 00, and its return edges are 00 (back to S0) and 01 (back to S5). In your report, list them explicitly to make the grading easier. For example: To meet the project design requirements, the FSM is designed as follows • There are eight internal states: S0, S1, S2, S3, S4, S5, S6, and S7 o S0 produces simple bitwise logic o S1 produces a+b, and then xor with the previous result o S2 produces the absolute difference of a and b, then XOR with prev_result o S3 produces the min(a,b) which blended with prev_result using concatenation o S4 produces max(a,b) plus a small shift of prev_result o S5 produces saturation of add of a and b, then AND with prev_result o S6 produces the average of a and b, then OR with prev_result o S7 produces simple rotate-left of a by 1, then XOR with b and the previous result • Feedback loops: S0 when op_val is 00 or 01, S1 when op_val is 01, S2 when op_val is 01, and so on... • Return edges: S1 when op_val is 00 or 01, S2 when op_val is 00 or 01, S3 when op_val is 00, and so on... The unique case is state S4, which has a transition to the OUTPUT state when op_val is 01. The OUTPUT state, upon completion, goes back to the IDLE state to wait for the next start signal.

How to test

The fsm_example_tb.sv file is included

External hardware

None required other than the TinyTapeout LED.

Project Pinout

Digital Pins

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

Simple RISC-V

by **kinako71-2**

`0359` `HDL Project`

`github.com/kinako71-2/TTGF0p2`

"Simple RISC-V based on the university lecture"

How it works

A simple one-stage RISC-V CPU. This CPU was created with reference to the lecture: https://eeic-vlsi.github.io/2025/. Instructions are 32-bit, while the input bus is 8-bit, so each instruction is loaded over four clock cycles.

How to test

Very simple instructions are written in the testbench to verify that the calculations are correct.

External hardware

The chip is not being shipped at this time.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	1st bit of xth instruction byte	1st bit of xth alu_out byte	reset to CPU
1	2nd bit of xth instruction byte	2nd bit of xth alu_out byte	write enable to instruction memory
2	3rd bit of xth instruction byte	3rd bit of xth alu_out byte	not used
3	4th bit of xth instruction byte	4th bit of xth alu_out byte	not used
4	5th bit of xth instruction byte	5th bit of xth alu_out byte	not used
5	6th bit of xth instruction byte	6th bit of xth alu_out byte	not used
6	7th bit of xth instruction byte	7th bit of xth alu_out byte	not used
7	8th bit of xth instruction byte	8th bit of xth alu_out byte	not used

LEDs Racer

by **KerCrafter**

`0384` `50 MHz` `HDL Project`

`github.com/KerCrafter/FPGA-LEDs-Racer`

"Funny electronic game with WS2812B LEDs"

How it works

A fun little game, 4 players (Red, Blue, Green, Yellow) around a table, each with an arcade button, must challenge each other by pressing their button as fast as possible.

How to test

Pressing the button causes their led to progress to the center circle. The first to reach the center wins.

External hardware

I'm putting here the material I used for the project, but of course if it's compatible this list can be adapted.

- 4 x **Arcade buttons** (https://www.amazon.fr/EG-STARTS-Nouveau-Boutons-poussoirs-lumineux/dp/B01MSNXLN0/)
- 1 x **LEDs Circles (with WS2812B Leds)** (https://www.amazon.fr/Treedix-anneaux-WS2812B-adressable-Raspberry/dp/B0CD3DYJRK/)

Should be connected to the FPGA inputs with a pull down resistor

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	BLUE_BTN	LEDS_LINE	—
1	RED_BTN	TP_SCREEN_0	—
2	GREEN_BTN	TP_SCREEN_1	—
3	YELLOW_BTN	TP_BLUE_READY_TO_PLAY	—
4	—	TP_RED_READY_TO_PLAY	—
5	—	TP_GREEN_READY_TO_PLAY	—
6	—	TP_YELLOW_READY_TO_PLAY	—
7	—	TP_UPDATE_FRAME	—

Frequency Counter SSD1306 OLED

by **Pawel Sitarz (embelon)**

0385 | 1 MHz | HDL Project

`github.com/embelon/ttgf-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

Freqquency 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	—	—	—

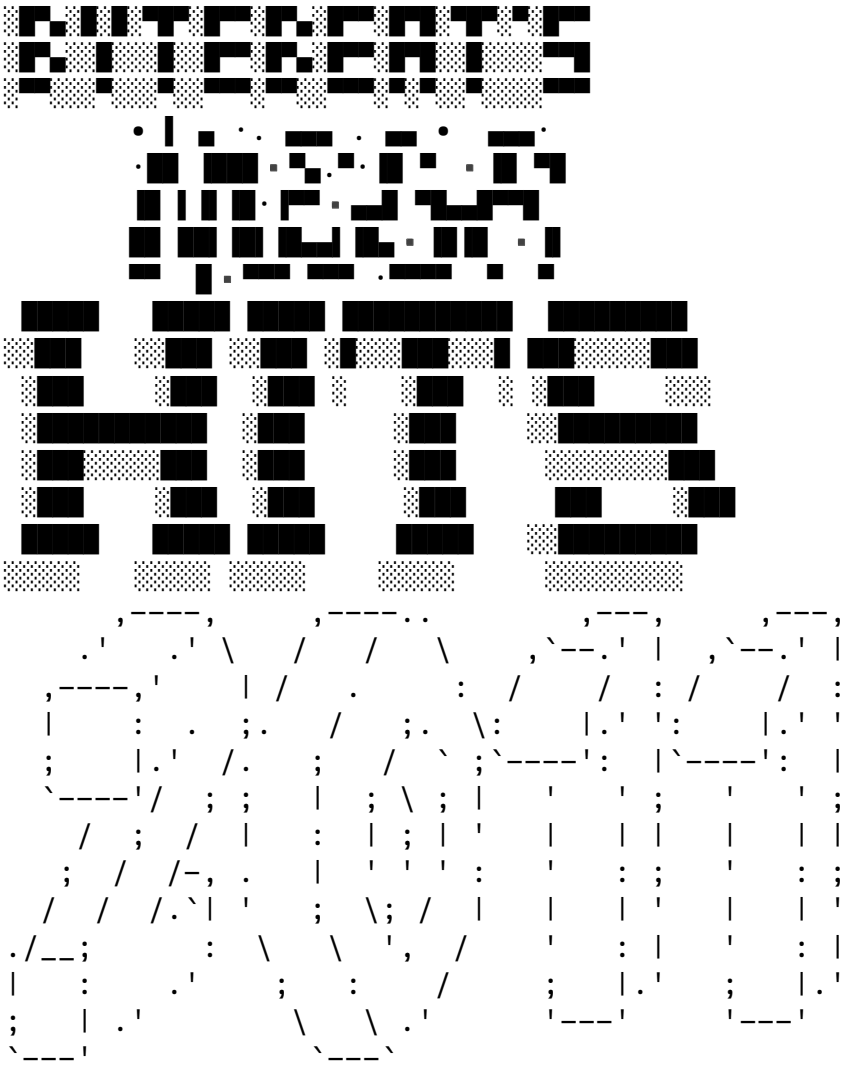
megabytebeat

by **proppy**

0391 | 2.048 MHz | HDL Project

`github.com/proppy/ttgf-bytebeat`

“Best-of compilation of bytebeat 2011's greatest hits on-a-chip.”



Extended Play of some of the 2011’s BYTEBEAT greatest hits on-a-chip:

Pin	Track	Artist
`out0`	the 42 melody	people on irc
`out1`	fractal trees	danharaj
`out2`	untitled	droid
`out3`	a tune to share	Niklas_Roy
`out7`	sierpinski harmony	miiro

How it works

The main module accept parameters from 4x 4-bit parameters buses and generate PWM audio signal on each output pins according to the following bytebeat formulas:

Pin	Formula	Original Params
out0	t*({b,a}&t>>{d,c})	a=0xa,b=0x2,c=0xa,d=0x0
out1	t|t%{b,a}|t%(2+{d,c})	a=0xf,b=0xf,c=0xf,d=0xf
out2	t>>a&b?t>>c:-t>>d	a=0x6,b=0x1,c=0x5,d=0x4
out3	t*(t>>a|t>>b)&{d,c}	a=0x9,b=0xd,c=0x0,d=0x1
out7	t*a&(t>>b)|t*c&(t>>d)	a=0x5,b=0x7,c=0x4,d=0xa

How to test

- Connect a speaker to the pin you want to “play”.
- Tweak parameters pins using binary rotary switches.

External hardware

- Rotaty switches w/ 16 positions.
- Speakers or TinyTapeout Audio PMOD if you want to hear `out7`.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	param a bit 0/3	track #0 PWM output	param c bit 0/3
1	param a bit 1/3	track #1 PWM output	param c bit 1/3
2	param a bit 2/3	track #2 PWM output	param c bit 2/3
3	param a bit 3/3	track #3 PWM output	param c bit 3/3
4	param b bit 0/3	N/A	param d bit 0/3
5	param b bit 1/3	N/A	param d bit 1/3
6	param b bit 2/3	N/A	param d bit 2/3
7	param b bit 3/3	track #7 PWM output	param d bit 3/3

Asicle v2

by **htfab**

0451 | 25.175 MHz | HDL Project

`github.com/htfab/ttgf0p2-asicle2`

"Wordle clone in raw silicon"

How it works

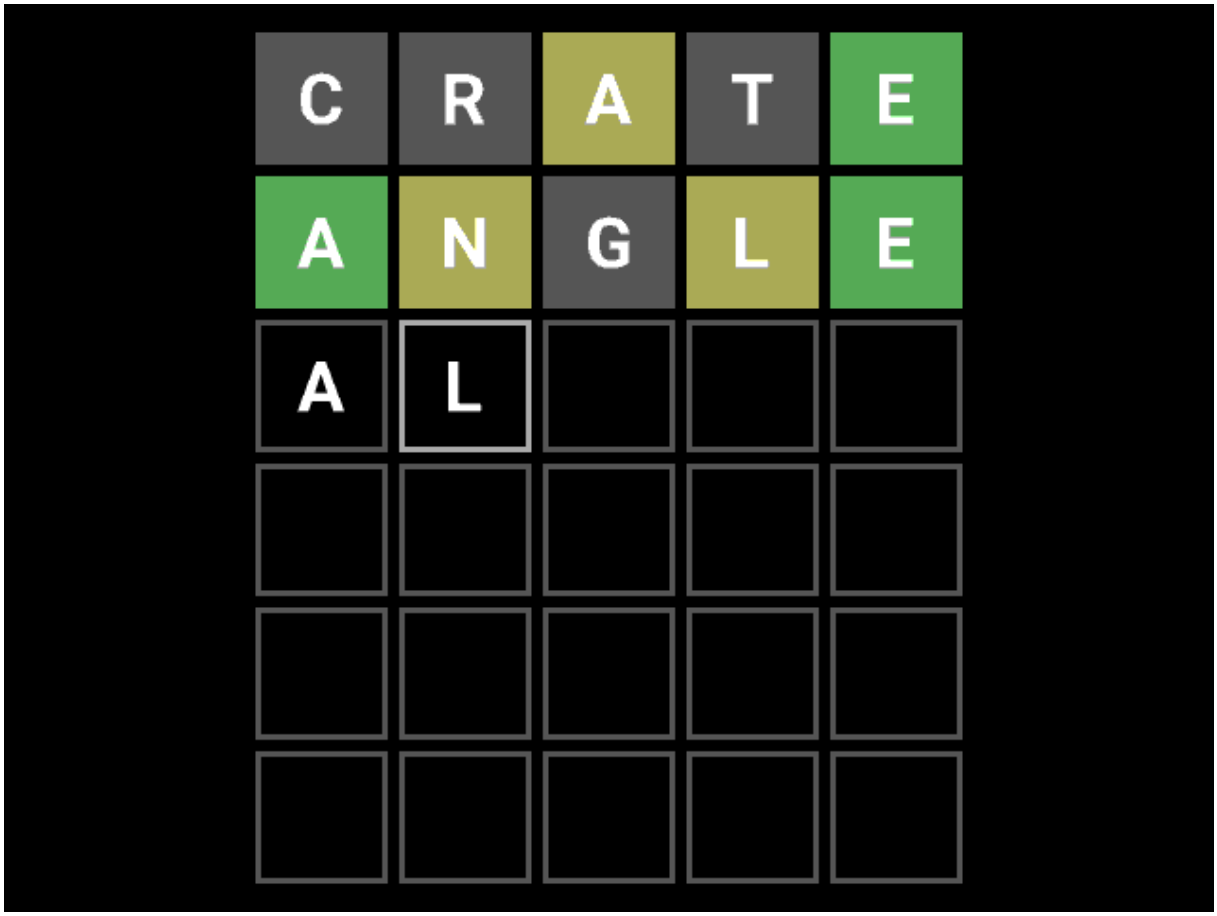


Figure 451.1: VGA screenshot with questions CRATE and ANGLE

Asicle is a Wordle clone implemented directly in hardware. A first version of it was taped out on the Google-Skywater MPW6 shuttle. This second version is a rewrite for Tiny Tapeout.

The 25-fold decrease in area mostly comes from moving the word list and font bitmaps to external flash on the QSPI Pmod, with some architectural changes to compensate for slower memory access. The design was also adapted to the Tiny Tapeout ecosystem by using the Gamepad Pmod for input and the Tiny VGA Pmod for output.

How to test

- Connect the Pmods:
  - Gamepad to input port (optional, you can also drive the input pins using the commander app or momentary push buttons)
  - QSPI to bidirectional port
  - Tiny VGA to output port
- Flash the data file to the QSPI Pmod using the Tiny Tapeout flasher
- Select the design
- Set the clock to 25.175 MHz and reset the design
- Play the game

If you haven't played Wordle before, the aim is to guess a five-letter English word in six attempts. Each time you get feedback: a green square indicates that the letter is correct, a yellow square indicates that it appears in the hidden word but at a different position, and a grey square means that the letter doesn't appear in the solution at all.

Gamepad controls:

- ↑ ↓: change the letter in the selected position
- ← →: move the selection
- A: make a guess
- START: start a new game (if the current one is finished)
- SELECT+START: start a new game (any time)
- SELECT+X: show the solution*
- SELECT+Y: re-roll the solution*

(* only in debug mode)

Direct input using `ui_in`:

- 0 1: change the letter in the selected position
- 2 3: move the selection
- 4: make a guess
- 5: start a new game
- 6: show the solution
- 7: re-roll the solution

External hardware

- Tiny VGA Pmod
- QSPI Pmod
- Gamepad Pmod (optional)

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	up	tinyvga: r1	qspi: cs0 (flash)
1	down	tinyvga: g1	qspi: sd0/mosi
2	left	tinyvga: b1	qspi: sd1/miso
3	right	tinyvga: vsync	qspi: sck
4	guess / gamepad: latch	tinyvga: r0	qspi: sd2
5	new game / gamepad: clock	tinyvga: g0	qspi: sd3
6	peek (debug) / gamepad: data	tinyvga: b0	qspi: cs1 (unused)
7	roll (debug)	tinyvga: hsync	qspi: cs2 (unused)

CAN Controller for Rocket

by **Noritsuna Imamura**

0454 HDL Project

`github.com/noritsuna/ttgf0p2-tt_um_noritsuna_CAN_CTRL`

"*CAN Controller for Rocket*"

How it works

This CAN only has a transmit function.
It is intended only to transmit GPS data.

How to test

- Arty Z7
  - This code is running.
- ZYBO Z7
  - This board is transmitting dummy GPS data to Arty Z7.
- Raspberry Pi
  - This is the destination for CAN data.
- Analog Discovery 3
  - This generates the clock.

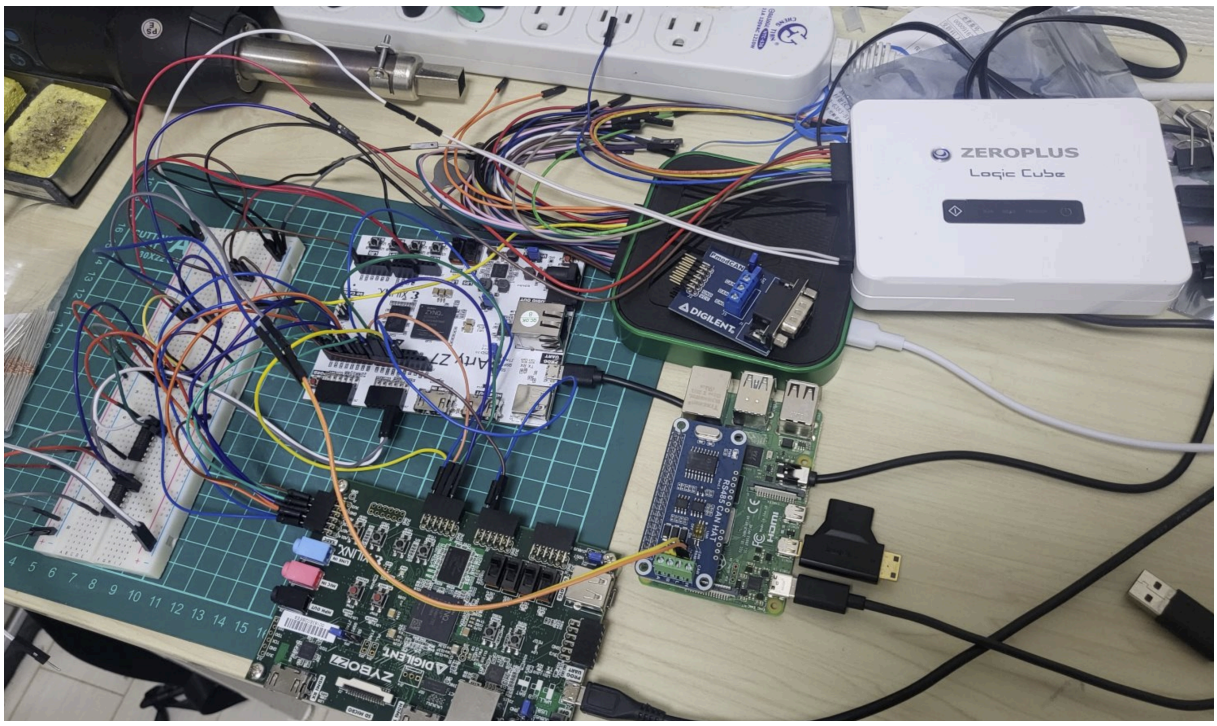


Figure 454.1: Development Environment

External hardware

A GPS data receiver and a mechanism to transmit that data to this chip are required.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	CAN_RX	CAN_TX	transmit_data[0]
1	send_data	TXING	transmit_data[1]
2	CLOCK_SIGNAL_IN	TT_Enable	transmit_data[2]
3	RESET_N	TT_Clock	transmit_data[3]
4	can_addr[0]	TT_Reset	transmit_data_counter[0]
5	can_addr[1]	—	transmit_data_counter[1]
6	can_addr[2]	—	transmit_data_counter[2]
7	can_addr[3]	—	transmit_data_counter[3]

Zilog Z80

by **ReJ aka Renaldas Zioma**

`0455` `4 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 reigsters which may be used as either 8-bit or 16-bit register pairs.
- One maskable and one non-maskable interrupt.
- Instruction set derived from Datapoint 2200, Intel 8008 and Intel 8080A.

Z80 registers

- `AF`: 8-bit accumulator (A) and flag bits (F)
- `BC`: 16-bit data/address register or two 8-bit registers
- `DE`: 16-bit data/address register or two 8-bit registers
- `HL`: 16-bit accumulator/address register or two 8-bit registers
- `SP`: stack pointer, 16 bits

- `PC`: program counter, 16 bits
- `IX`: 16-bit index or base register for 8-bit immediate offsets
- `IY`: 16-bit index or base register for 8-bit immediate offsets
- `I`: interrupt vector base register, 8 bits
- `R`: DRAM refresh counter, 8 bits (msb does not count)
- `AF'`: alternate (or shadow) accumulator and flags (toggled in and out with `EX AF, AF'`)
- `BC'`, `DE'` and `HL'`: alternate (or shadow) registers (toggled in and out with `EXX`)

Z80 Pinout

```
              ,---------.__.---------.
<--     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     -->
              `----------------------'
```

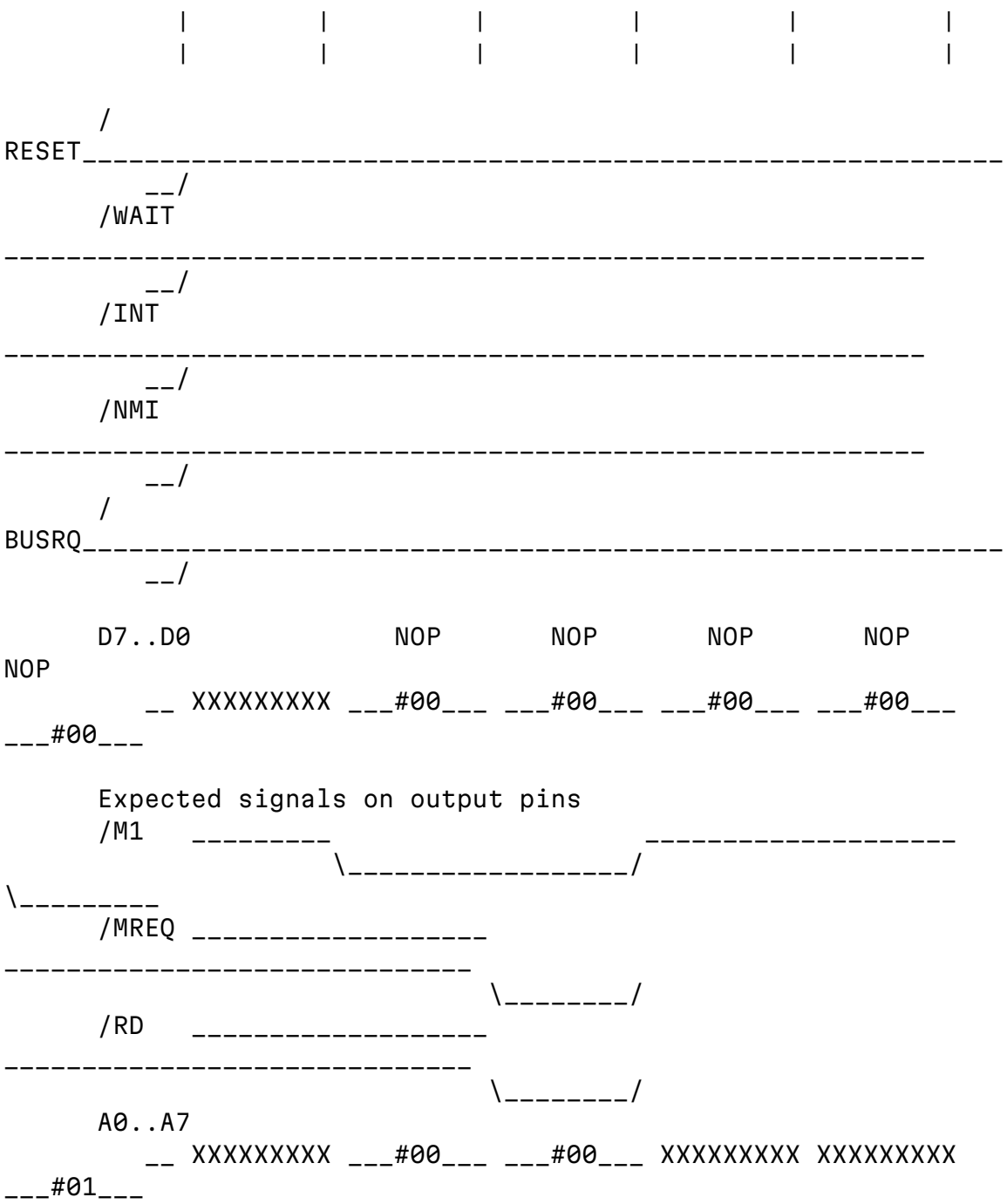
How to test

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

      Z80CLK____      ____      ____      ____      ____
____
         __/    \____/    \____/    \____/    \____/    \____/
`____ ...
```



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 starts early	/WR, A4, A12	D4
5	CONFIG – 10: long MREQ IORQ / 11: long MREQ IORQ RD starts 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

ROTFPGA v2

by **htfab**

481 | 10 MHz | HDL Project

`github.com/htfab/ttgf0p2-rotfpga`

> "*A reconfigurable logic circuit made of identical rotatable tiles*"

How it works

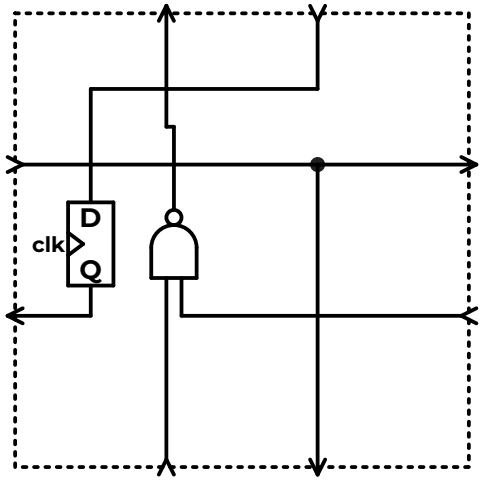


Figure 481.1: Logic tile

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

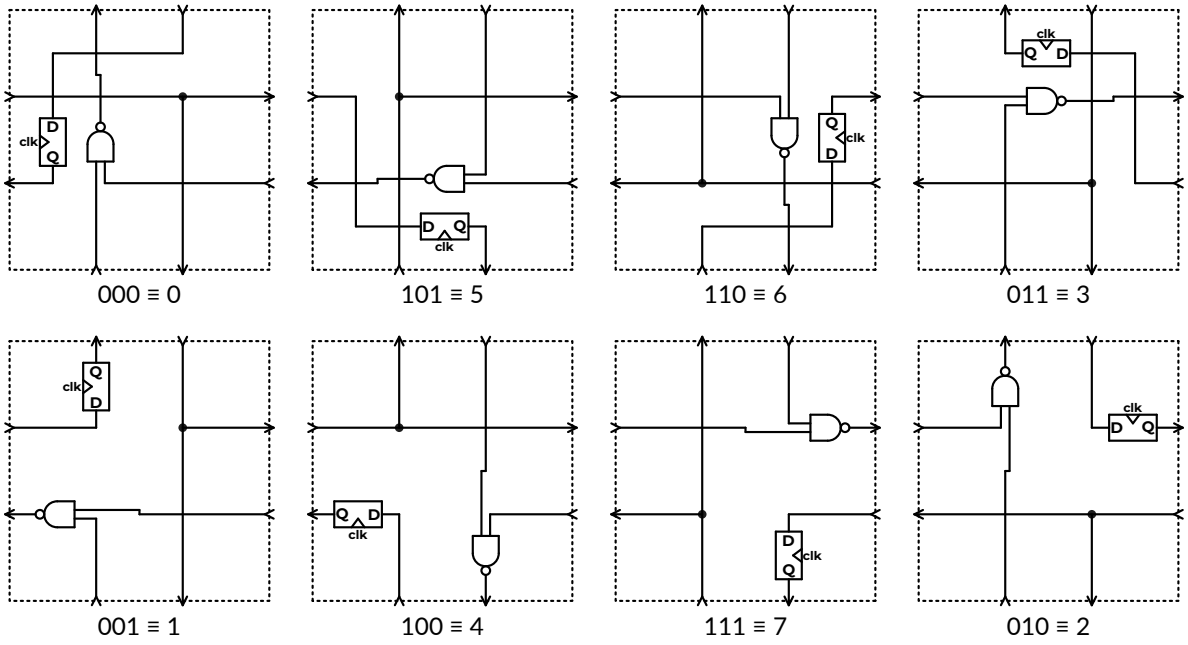
Porting the design required a 50-fold decrease in chip area which was achieved using a combination of cutting corners, heavy optimization and a few design changes. In particular:

- The FPGA was reduced from 24×24 to 8×8 tiles. There are 8 inputs and 8 outputs instead of 12 each.
- To compensate for smaller size, tiles can also be mirrored in addition to rotation.
- Tiles (being the most repeated part of the design) were rewritten as hand-optimized gate-level Verilog.
- Each tile only contains 1 flip-flop (the one exposed to the user). Configuration is now stored in latches.
- Configuration and reset are performed using a routing-efficient scan chain, so the design is no longer routing constrained. This allows standard cells to be placed with >80% density.

- Openlane and its components are 2 years more mature, hardening the same HDL more efficiently.

Configuration

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



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

Tiles are arranged in an 8×8 grid:

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

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

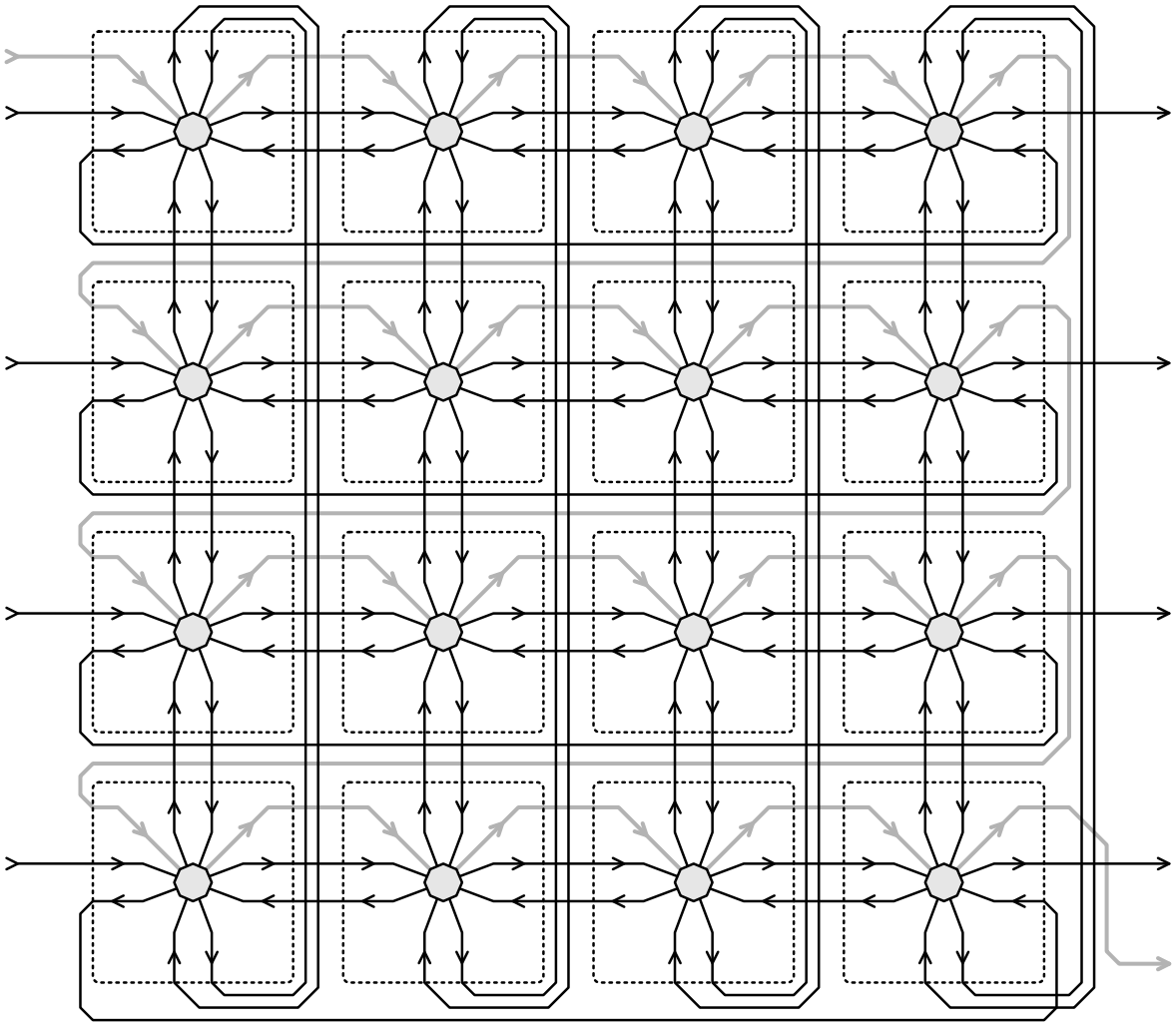


Figure 481.2: Grid model

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

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

One can thus configure the FPGA by sending the sequence of all *diagonal flip* bits through the scan chain, then setting *configuration* to 11 and back to 00, then sending all *horizontal flip* bits, setting *configuration* to 10 and back to 00, and finally sending the *vertical flip* bits and setting *configuration* to 01 and back to 00.

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

Loop breaker

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

Tiles are assigned to loop breaker classes:

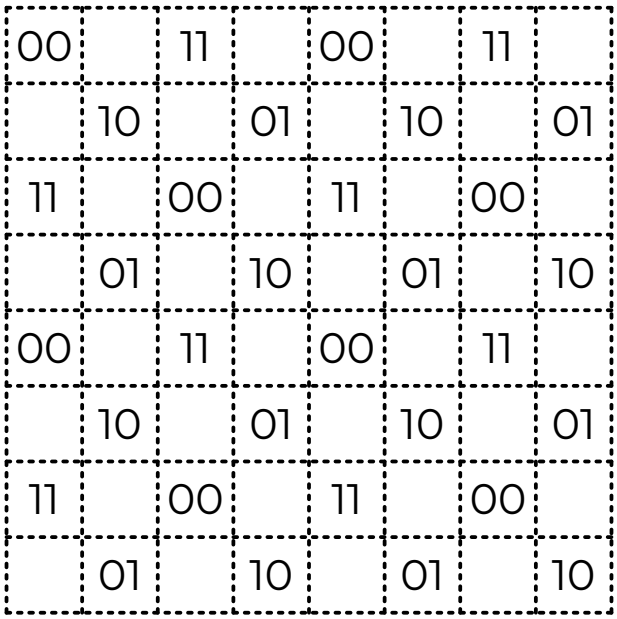


Figure 481.3: Loop breaker tile classes

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

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

The loop breaker has the following properties:

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

Reset

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

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

Pin mapping

Input pins:

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

Output pins:

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

Bidirectional pins:

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

How to test

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

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

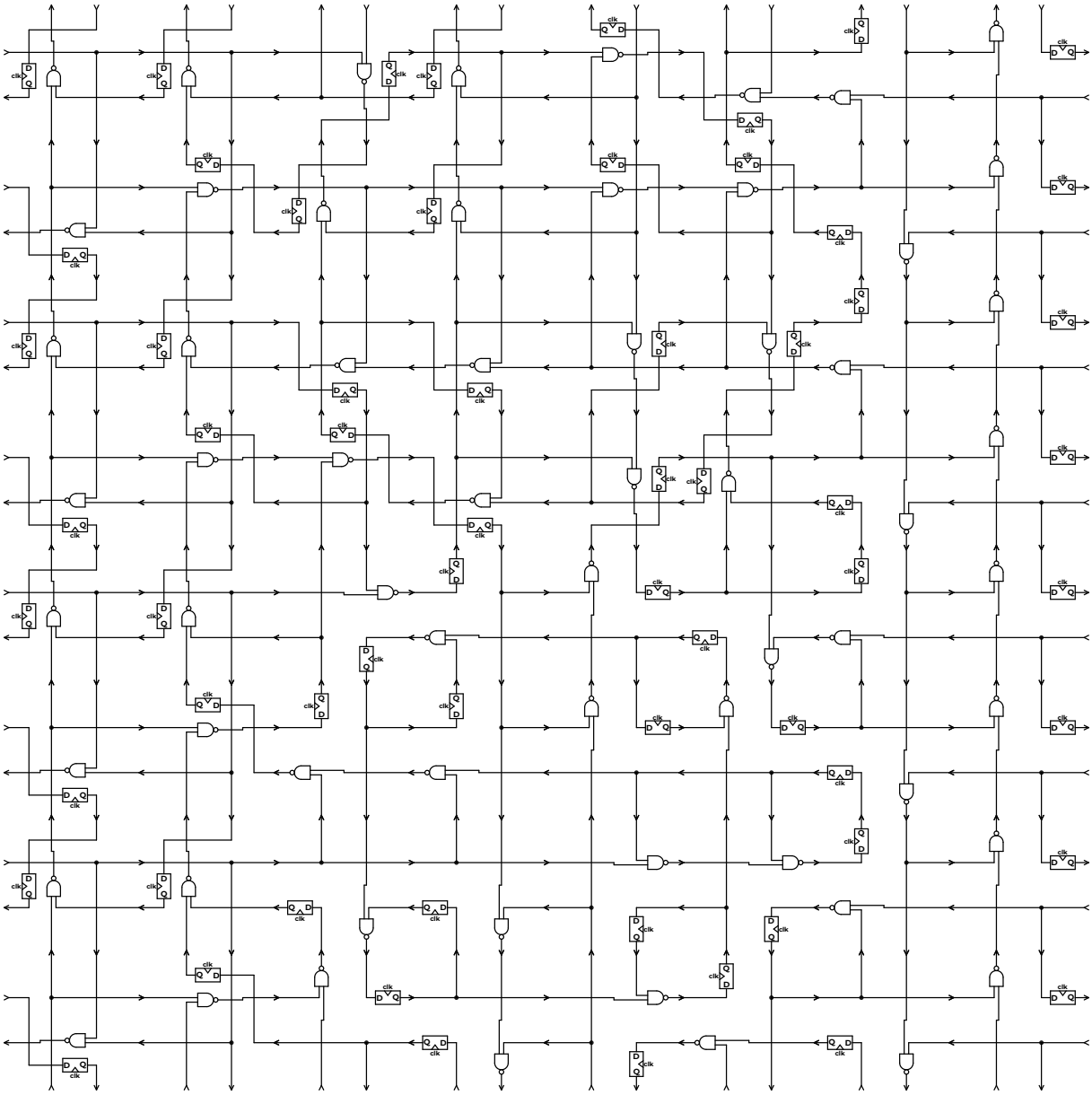


Figure 481.4: Diagram corresponding to `fpga_config` in `test.py`

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

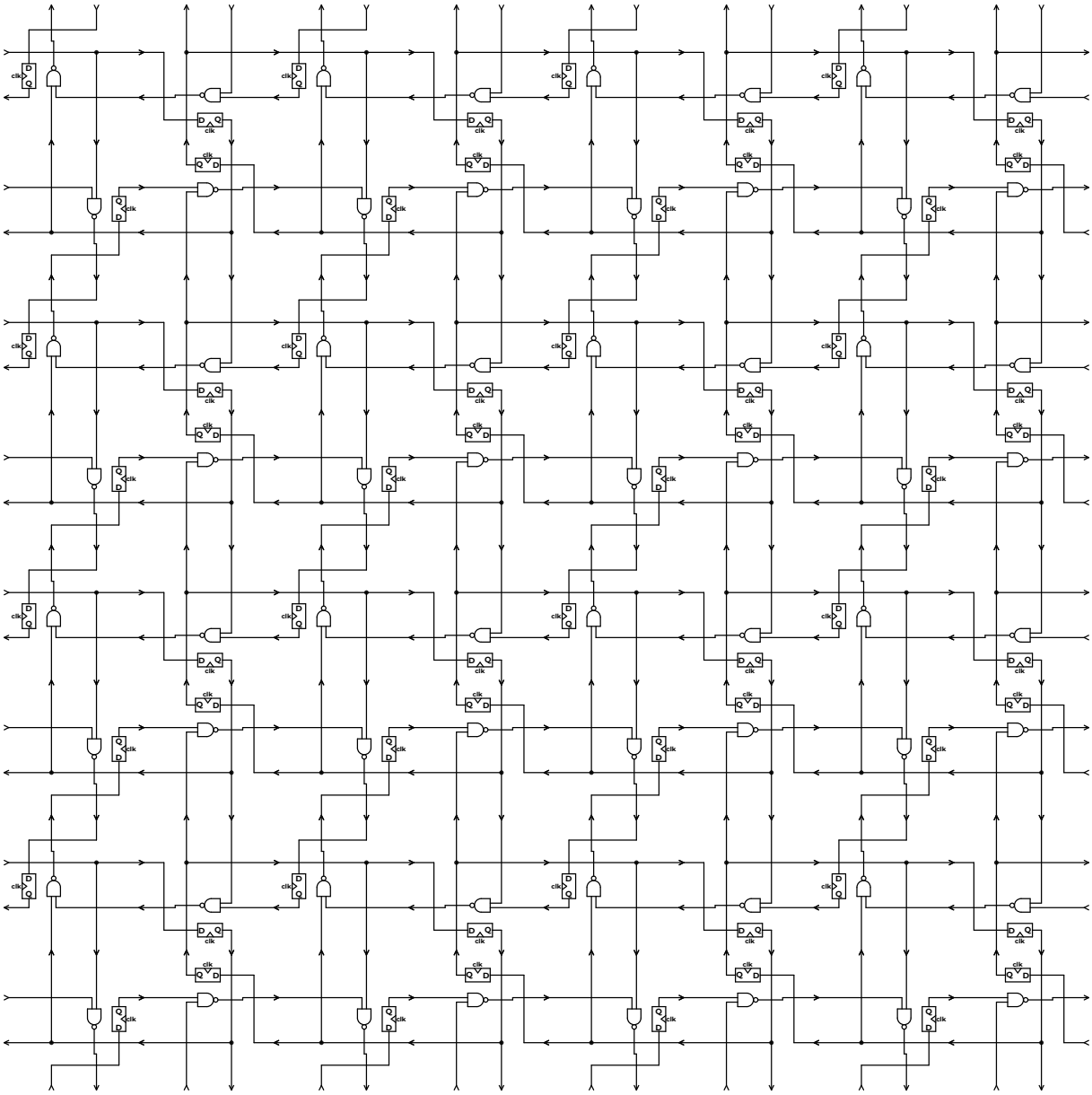


Figure 481.5: Diagram corresponding to `cfg` from the loop breaker test in `test.py`

External hardware

None

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
`0`	tile(0,0) left in	tile(7,0) right out	*scan enable* input
`1`	tile(0,1) left in	tile(7,1) right out	*scan chain* input
`2`	tile(0,2) left in	tile(7,2) right out	*configuration* input bit 0

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

7-Segment Digital Desk Clock

by **Samuel Ellicott**

0483 | 50 MHz | HDL Project

`github.com/sellicott/digital_clock_gf0p2`

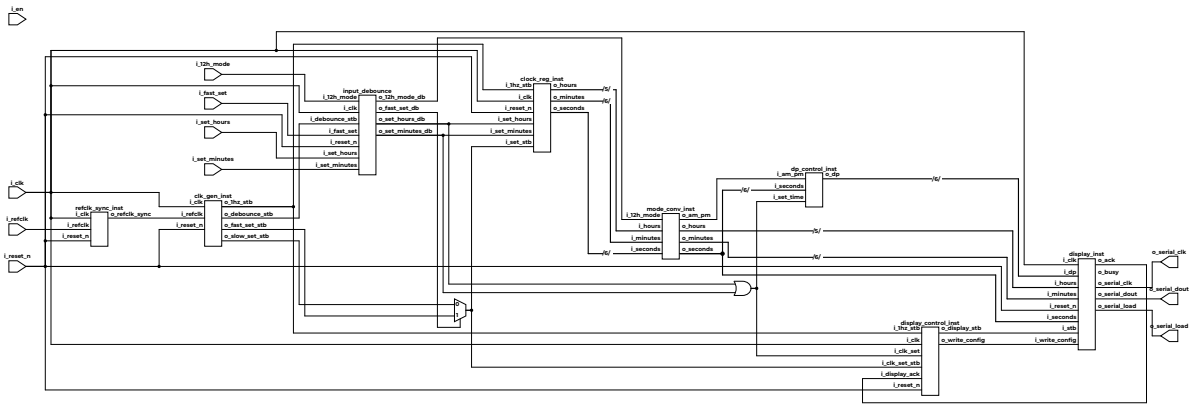
"7-Segment Desk Clock"

How it works

This is a rehardening of a test design from Skywater and IHP processes for GF180

Simple digital clock, displays hours, minutes, and seconds in either a 24h format. Since there are not enough output pins to directly drive a 6x 7-segment displays, the data is shifted out over SPI to a MAX7219 in 7-segment mode. The time can be set using the `hours_set` and `minutes_set` inputs. If `set_fast` is high, then the the hours or minutes will be incremented at a rate of 5Hz, otherwise it will be set at a rate of 2Hz. Note that when setting either the minutes, rolling-over will not affect the hours setting. If both `hours_set` and `minutes_set` are presssed at the same time the seconds will be cleared to zero.

A block diagram of the system is shown below.



How to test

Apply a 5MHz clock to the clock pin and 32.786Khz signal to the refclk pin. Use the `hours_set` and `minutes_set` pins to set the time.

External hardware

Connect the BIDIR PMOD to a MAX7219 7-segment display, For reference Tiny Tapeout SPI

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	refclk	—	Display CS
1	—	—	Display MOSI
2	Fast/Slow Set	—	—
3	Set Hours	—	Display SCK
4	Set Minutes	—	—
5	12-Hour Mode	—	—
6	—	—	—
7	—	—	—

MarcoPolo

by **Javier Munoz Saez**

0485 | 50 MHz | HDL Project

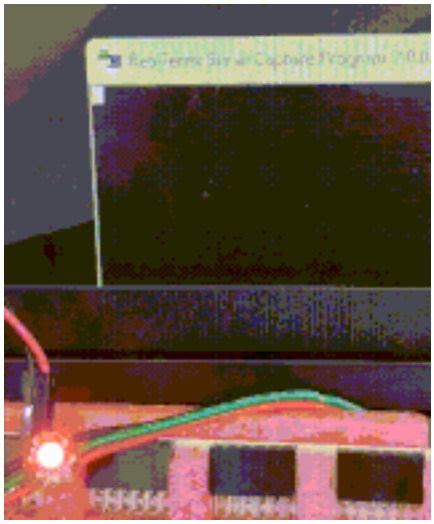
`github.com/javiBajoCero/ttgf-verilog-template`

"when this UART receives 'MARCO' answers 'POLO!' , adapted from https://github.com/javiBajoCero/ttihp-verilog-template"

How it works

This is just an UART + small WS2812b single led driver copied from my old ttsky submission https://github.com/javiBajoCero/ttsky-verilog-template

Listens to ascii 'MARCO' and once detected, with 10us delay (mega fast) replies with '! and blips the LED from red to green.



FPGA tested :)

How to test

The device will need 50Mhz clock. (provided by the RP2)

Connect ui[0]: "uartRX" and uo[0]: "uartTX" with your favourite UART gizmo, i used my PC and FTDI FT2232HQ USB-UART bridge builtin my ArtyS7 evaluation board.

with default putty serial settings 9600 bauds 8 data bits 1 stop bit parity NONE

and type uppercase 'MARCO' trough UART RX pin you should receive a '! on UART TX pin

Some extra debugging signals are also exposed :

- uo[1]: “baud_tick_rx”
- uo[2]: “baud_tick_tx”
- uo[3]: “trigger_send”
- uo[4]: “uartTxbusy”
- uo[5]: “led_data_out”

External hardware

probably a USB to TTL dongle like CP2102, and a WS2812b led

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	uartRX	uartTX	unused
1	—	baud_tick_rx	—
2	—	baud_tick_tx	—
3	—	trigger_send	—
4	—	uartTxbusy	—
5	—	led_data_out	—
6	—	1	—
7	—	1	—

Linear Timecode (LTC) generator

by **Thomas Flummer**

0487 | 12 MHz | HDL Project

`github.com/flummer/tt-um-flummer-ltc`

"Timecode generator for audio video syncronization"

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

Classic 8-bit era Programmable Sound Generator SN76489

by **ReJ aka Renaldas Zioma**

513 | 4 MHz | HDL Project

github.com/rejunity/tt05-psg-sn76489

"The SN76489 Digital Complex Sound Generator (DCSG) is a programmable sound generator chip from Texas Instruments."

How it works

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

The goals of this project

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

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

The future work

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

Chip technical capabilities

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

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

Channel	Volume registers	Tone & noise registers

0	Channel #0 attenuation	Tone #0 frequency
1	Channel #1 attenuation	Tone #1 frequency
2	Channel #2 attenuation	Tone #2 frequency
3	Channel #3 attenuation	Noise type and frequency

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

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

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

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

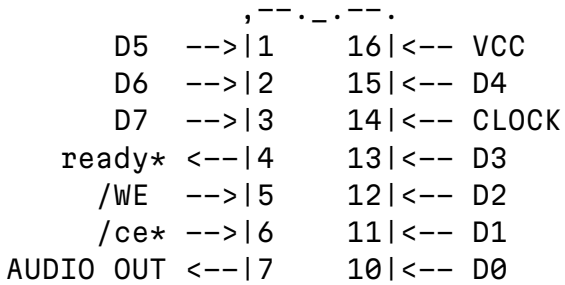
Historical use of the SN76489

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

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

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

The original pinout of the SN76489AN



```
GND ---|8     9|    not connected*
       `-------'
* -- omitted from this Verilog implementation
```

Difference from the original hardware

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

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

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

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

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

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

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

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

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

A configurable clock divider was introduced in this implementation.

1. the original SN76489 with the master clock internally divided by 16. This classical chip was intended for PAL and NTSC frequencies. However in BBC Micro 4 MHz clock was employed.
2. SN94624/SN76494 variants without internal clock divider. These chips were intended for use with 250 to 500 KHz clocks.

3. high frequency clock configuration for TinyTapeout, suitable for a range between 25 MHz and 50 Mhz. In this configuration the master clock is internally divided by 128.

The reverse engineered SN76489

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

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

How to test

Summary of commands to communicate with the chip

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

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

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

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

Note frequency

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

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

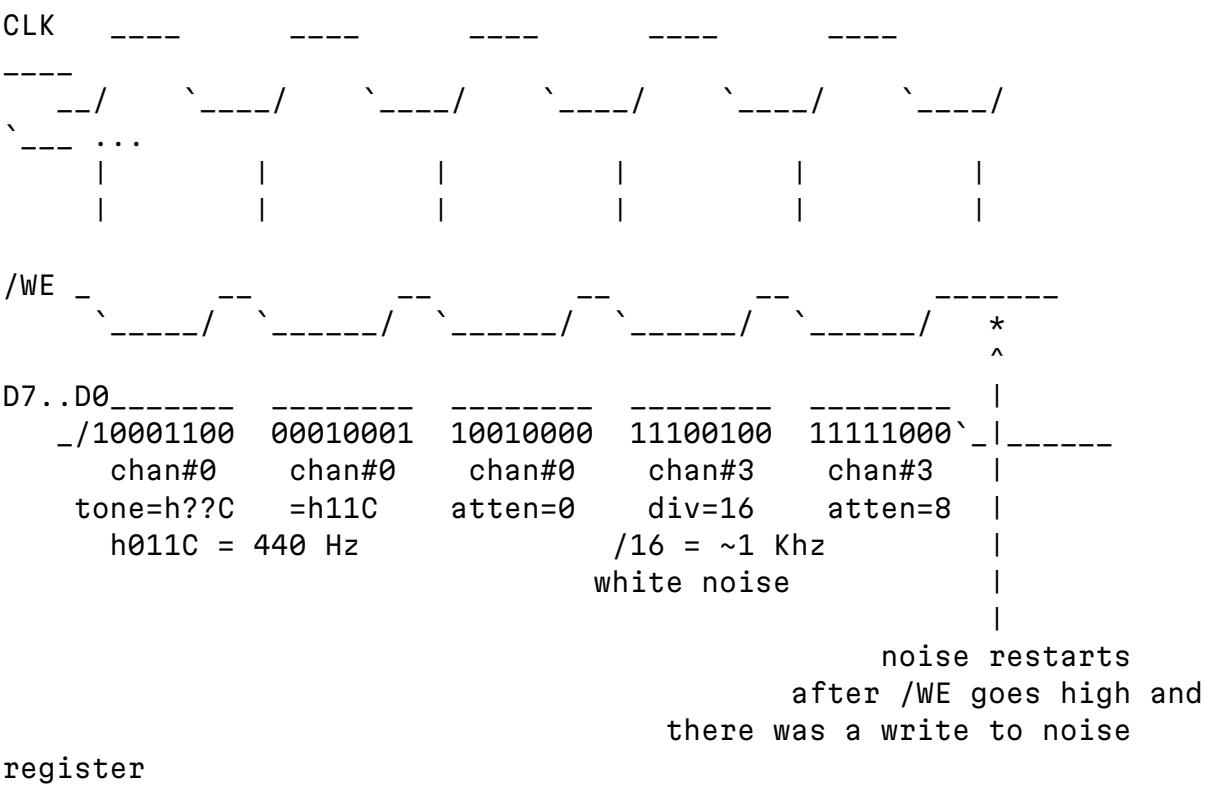
For example 10-bit value that plays 440 Hz note on a chip clocked at 4 MHz would be:

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

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

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

Timing diagram



Configurable clock divider

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

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

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

Some examples of music recorded from the chip simulation

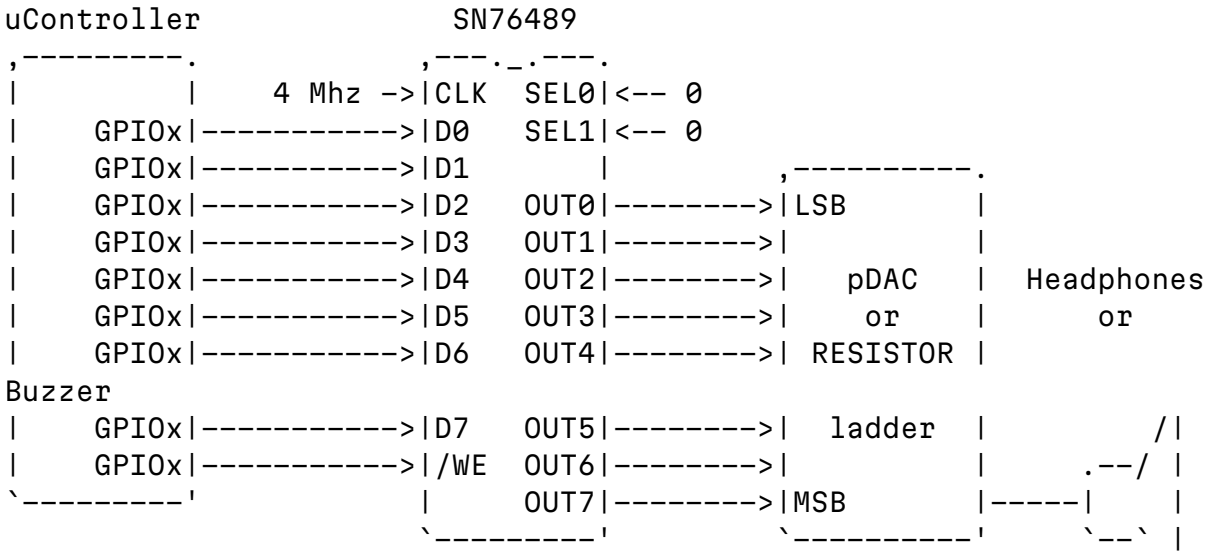
- Crazee Rider BBC Micro game
- MISSION76496 tune for Sega Master System

External hardware

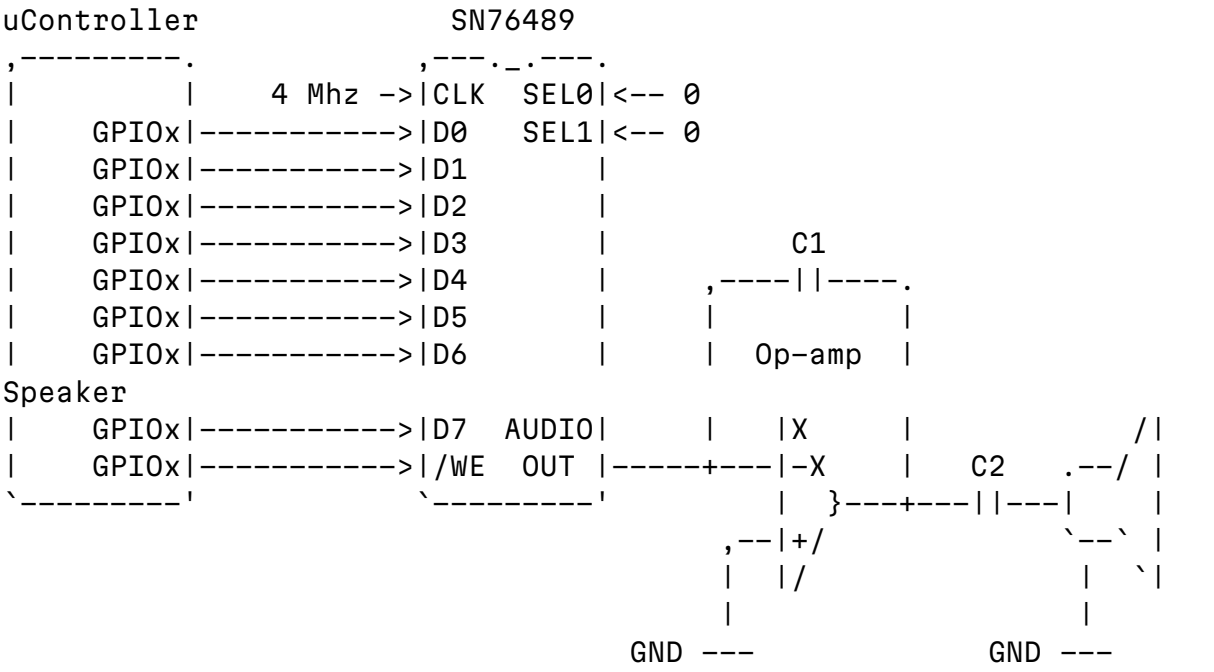
DAC (for ex. Digilent R2R PMOD), RC filter, amplifier, speaker.

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

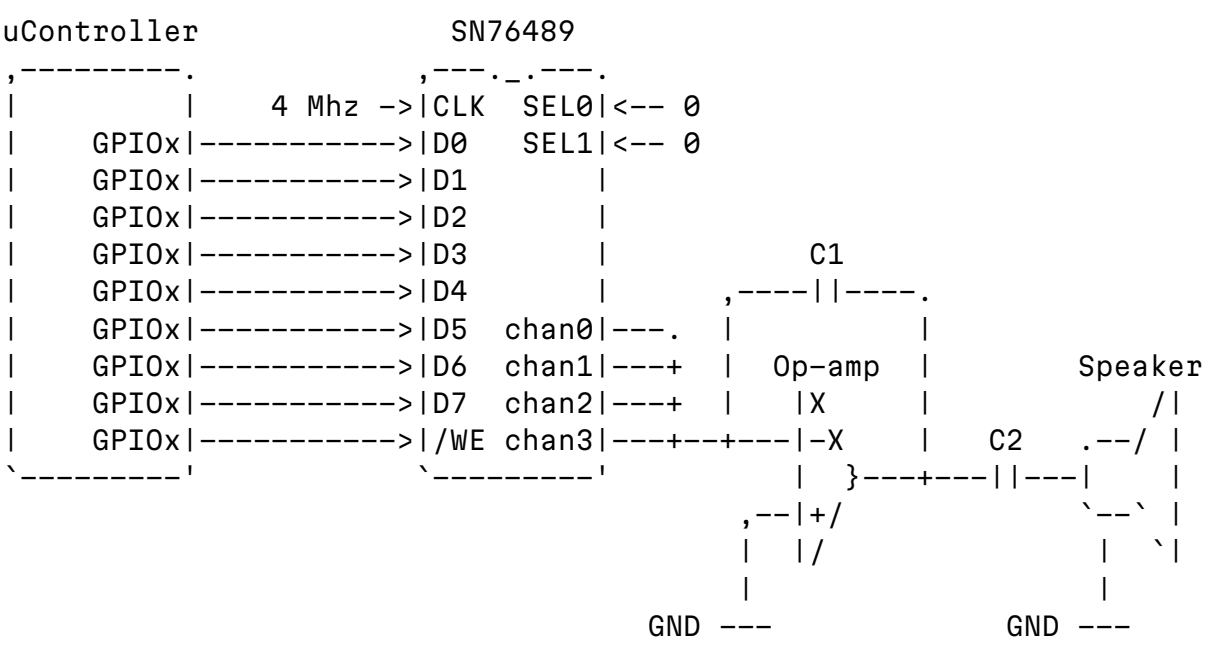
8-bit parallel output via DAC One option is to connect off the shelf data parallel Digital to Analog Converter (DAC) for example Digilent R2R Pmod to the output pins and route the resulting analog audio to piezo speaker or amplifier.



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



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



Project Pinout

Digital Pins

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

VGA Tiny Logo

by **Renaldas Zioma**

0515 25.175 MHz HDL Project

`github.com/rejunity/ttgf-vga-tiny-tapeout-animated-logo`

"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 and run it!

External hardware

TinyVGA PMOD, VGA monitor

Project Pinout

Digital Pins

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

Ring Oscillator (5 inverter)

by **Darryl Miles**

0517 HDL Project

`github.com/dlmiles/ttgf0p2-ringosc-5inv`

"*Ring Oscillator (5 inverter)*"

How it works

Set rst_n to 1 to make reset inactive. Set ui_in[0] to 1 to enable oscillator.

How to test

Measure frequency at outputs.

External hardware

Frequency counter.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	ring enable	divide-by-1	divide-by-256
1	—	divide-by-2	divide-by-512
2	—	divide-by-4	divide-by-1024
3	—	divide-by-8	divide-by-2048
4	—	divide-by-16	divide-by-4096
5	—	divide-by-32	divide-by-8192
6	—	divide-by-64	divide-by-16738
7	—	divide-by-128	divide-by-32768

Simon Says memory game

by **Uri Shaked**

519 | 50 kHz | HDL Project

github.com/urish/tt-simon-game

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

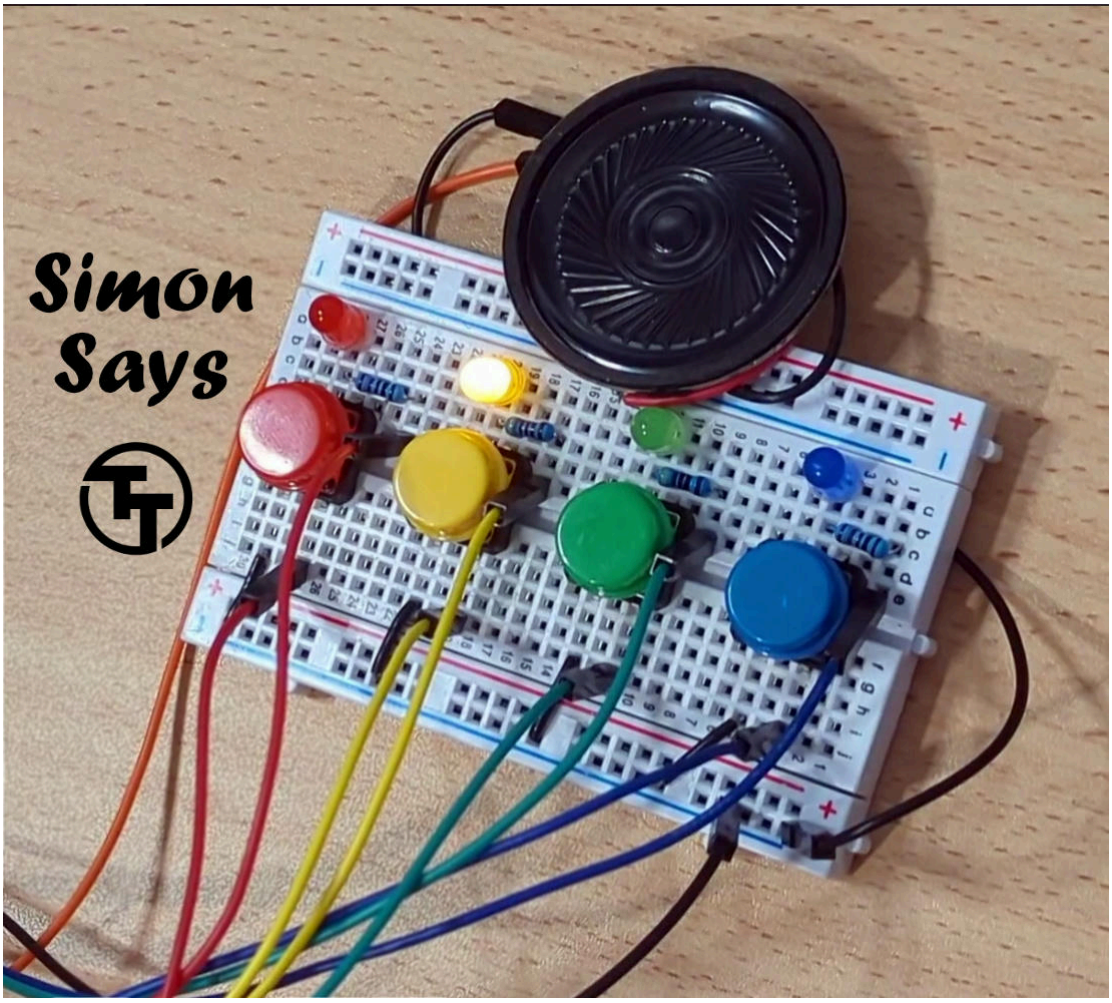


Figure 519.1: Simon Says Game

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 unknown frequency.

The internal clock is generated by a 9-stage ring oscillator, divided by 8192, to get a frequency of about 50.7 KHz (as measured in simulation).

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**

545 | 25 MHz | HDL Project

`github.com/kbeckmann/ttgf0p2-kbeckmann-flame`

"*Flame demo*"

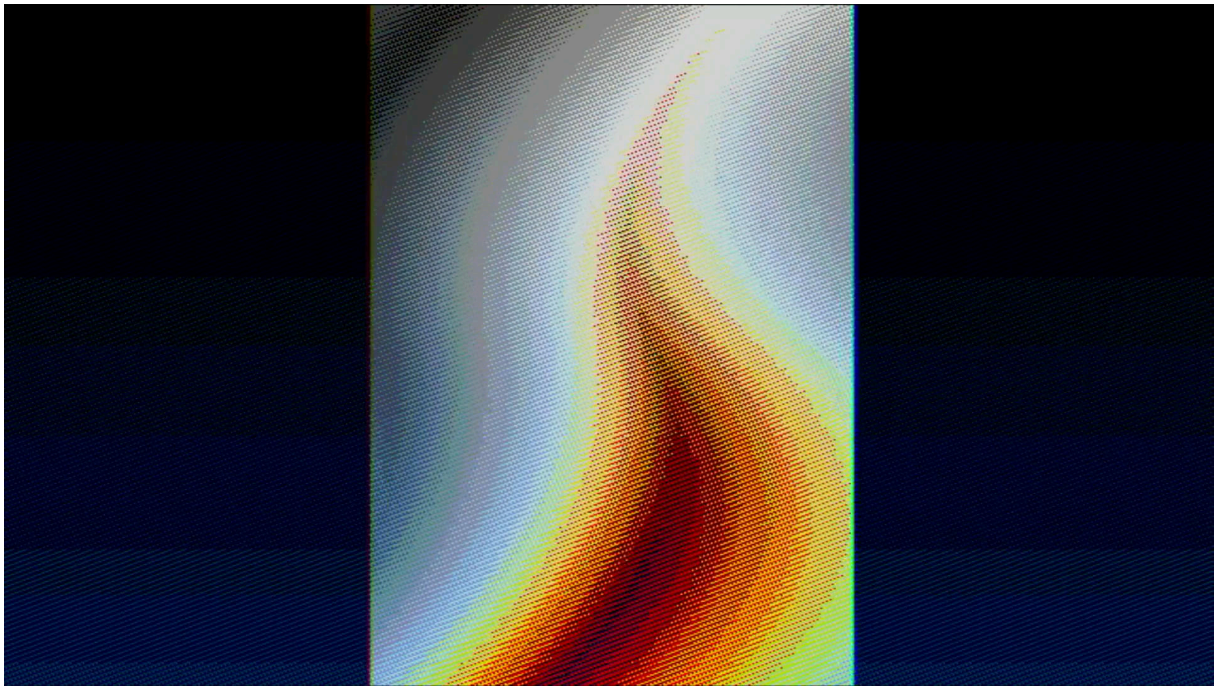


Figure 545.1: preview

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	—	R1	—
1	—	G1	—
2	—	B1	—
3	—	VSync	—
4	—	R0	—
5	—	G0	—
6	—	B0	—
7	—	HSync	AudioPWM

WokiPWM

by **Ken Pettit**

0547 | 10 kHz | Wokwi Project

github.com/kdp1965/ttgf-um-wokwi-pwm

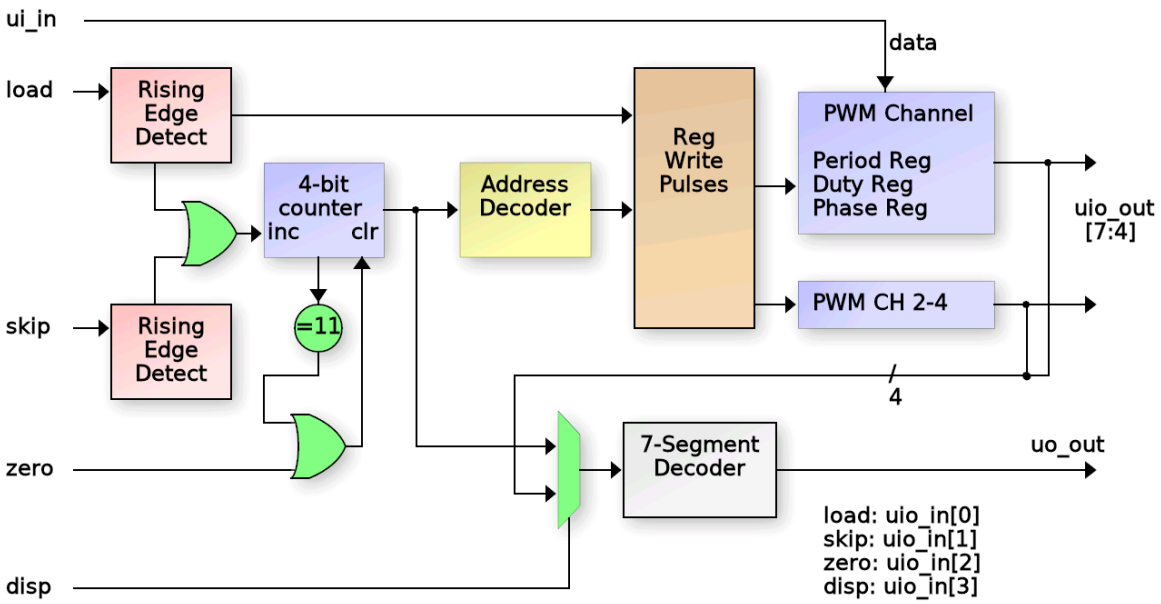
wokwi.com/projects/445338187869298689

"A 4-channel PWM impelemented in PWM gates"

Wokwi 4-Channel PWM

As it says, this is a 4-channel PWM coded in Wokwi. Each channel has 3 8-bit values to control their frequency, duty cycle and relative (starting) phase to each other. Values are loaded by setting 8-bit data in on ui_in[7:0] and loading to the current address which auto-increments.

Block diagram



How it works

After reset, there is an 4-bit counter that is the address where the next 8-bit input data at in[7:0] will be loaded. The 'load' signal loads the 8-bit input to the register at that current counter address and then increments the counter.

The 'skip' input will skip an address without loading it. And the 'zero' input will reset the counter back to zero.

None of the ‘load’, ‘skip’, ‘zero’ or ‘disp’ inputs are debounced, but they are riging edge detected (at least the load and skip are).

The current count (register address) is displayed on the 7-segment display via a 4-to-7 decoder. Each of 4 PWM channels use two 8-bit control registers to set frequency and duty cycle. The first (lowest address) register controls the duty cycle and the second (highest address) controls the period of that channel. Only channels with non-zero period value will operate (the counter does not count when period_val is zero). For PWM channels 2-4, there is an additional 8-bit register (address 8-10) to control the starting phase of that channel relative to channel 1. This means that if channels 2-4 are to be used, channel 1 must also be used.

Program PWM channel control registers to create independent PWMs:

Address Meaning 0x0 PWM1 duty 0x1 PWM1 period 0x2 PWM2 duty 0x3 PWM2 period 0x4 PWM3 duty 0x5 PWM3 period 0x5 PWM4 duty 0x7 PWM4 period 0x8 PWM2 phase 0x9 PWM3 phase 0xa PWM4 phase 0xb Control BIT0: BIT1: BIT2: BIT3: Write ‘1’ to clear and synchronize all PWM channels. Self clearing.

How to test

1. Start the clock and reset the circuit.
2. Set the DIP switches so only switch 7 is on (8’h40)
3. Press ‘load’. The display should now show ‘1’.
4. Set the DIP switches so only switch 8 is on (8’h80).
5. Press ‘load’. The display should show ‘2’ and the first LED should start blinking at about 50% rate.
6. Continue programming values for the other 3 PWMs by setting DIP switch value and pressing ‘load’ even values (duty cycle) must be less that the associated odd address value (period).
7. When the 7-segment display shows “8”, this is the phase offset value for PWM channel 2 relative to channel 1. Loading a value here will force a known phase for channel 2 WHEN THE CHANNELS ARE SYNCHRONIZED.
8. Load additional relative phase values for PWM channels 3 and 4 at the next two addresses (9 and A).
9. To synchronize the PWM channels, write a value of 4 to address B.

External hardware

LEDs should be connected to the PWM channel outputs. The 7-Segement display shows the current register address to be written.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	D0	seg_a	load
1	D1	seg_b	skip
2	D2	seg_c	zero
3	D3	seg_d	disp
4	D4	seg_e	pwm_0
5	D5	seg_f	pwm_1
6	D6	seg_g	pwm_2
7	D7	seg_dp	pwm_3

Dog Battle Game

by **Sophus Andreassen**

`549` `50 MHz` `HDL Project`

`github.com/jorgenkraghjakobsen/tt_gf_dog_fight`

> *"A simple VGA dog battle game with 2 moving boxes that collide."*

How it works

Dog Battle Game is a VGA-based game engine featuring 4 animated "dogs" (colored boxes) that bounce around the screen with physics simulation.

The design includes:

- **Physics engine**: Friction (0.99x decay per frame), elastic collisions, wall bouncing
- **4 dogs**: Each with individual position, velocity, mass, and color
- **Collision detection**: Hit counters track when dogs collide with each other
- **VGA output**: 640x480 @ 25MHz pixel clock with RGB color and sync signals

The game runs continuously, updating positions once per frame (60 FPS) with realistic physics including momentum conservation and energy loss on collisions.

How to test

Connect a VGA monitor to the output pins. The game will start automatically on power-up and run continuously.

Output pin mapping:

- Pins 0-1: VGA sync signals (HS, VS)
- Pins 2-3: VGA blue (2 bits)
- Pins 4-5: VGA green (2 bits MSB)
- Pins 6-7: VGA red (2 bits MSB)

External hardware

VGA monitor with 640x480 resolution support. Connect via standard VGA cable or appropriate PMOD adapter.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	vga_vs	—
1	—	vga_hs	—
2	—	vga_b0	—
3	—	vga_b1	—
4	—	vga_g1	—
5	—	vga_g2	—
6	—	vga_r1	—
7	—	vga_r2	—

PVTMonitorSuite

by **Susumu Yamazaki**

0551 | 50 MHz | HDL Project

github.com/zacky1972/ttgf0p2-PVTMonitorSuite

> "*PVTMonitorSuite: A fully digital, modular suite for measuring gate delays, flip-flop timings, and clock skew under varying process, voltage, and temperature conditions.*"

How it works

PVTMonitorSuite integrates ring oscillators and flip-flop-based measurement circuits to quantify the timing characteristics of digital logic.

The suite supports the following measurements:

- An inverter-based ring oscillator provides high-resolution estimates of gate propagation delay of an inverter ($t_{pd,INV}$).
- A NAND2-based ring oscillator provides high-resolution estimates of gate propagation delay of NAND2 ($t_{pd,NAND2}$).
- Standard, DICE, and LEAP-DICE D flip-flops measure clock-to-Q and setup timing ($t_{clkq} + t_{setup}$).
- Clock skew timing (t_{skew}) is determined using a time digitizer between two clock signals.

High-speed counters driven by the ring oscillators convert these delays into digital values, enabling precise evaluation of process, voltage, and temperature variations on a fully digital, open-access platform.

To meet TinyTapeout's requirements, no standard cells from the GF180MCU PDK process are used, and the design passes all strict error and warning checks.

How to test

Measure $t_{pd,INV}$

1. Connect `uo_out[0]` to your equipment to measure frequency.
2. Turn on `ui_in[0]`.
3. Measure the frequency.
4. Calculate $t_{pd,INV} = \frac{1}{32 \times 51 \times f}$.

Measure $t_{pd,NAND2}$

1. Connect `uo_out[1]` to your equipment to measure frequency.
2. Turn on `ui_in[0]`.
3. Measure the frequency.

4. Calculate $t_{pd,NAND2} = \frac{1}{32 \times 41 \times f}$.

Measure $t_{clkq} + t_{setup}$ of standard D-FF

1. Connect `ui_in[1]` to a device that generates exactly 50 MHz.
2. Set `ui_in[7:5]` to 3'b000, to choose measurement of $t_{clkq} + t_{setup}$.
3. Turn off `ui_in[3]`.
4. Turn off `ui_in[2]` to reset.
5. Turn on `ui_in[2]`.
6. Turn on `ui_in[3]` to start.
7. Read `uio`.
8. Calculate $t_{clkq} + t_{setup} = \frac{uio \times 20}{16} - 2 \times t_{pd,NAND2}$.

Measure $t_{clkq} + t_{setup}$ of DICE D-FF

1. Connect `ui_in[1]` to a device that generates exactly 50 MHz.
2. Set `ui_in[7:5]` to 3'b001, to choose measurement of $t_{clkq} + t_{setup}$.
3. Turn off `ui_in[3]`.
4. Turn off `ui_in[2]` to reset.
5. Turn on `ui_in[2]`.
6. Turn on `ui_in[3]` to start.
7. Read `uio`.
8. Calculate $t_{clkq} + t_{setup} = \frac{uio \times 20}{16}$.

Measure $t_{clkq} + t_{setup}$ of Leap DICE D-FF

1. Connect `ui_in[1]` to a device that generates exactly 50 MHz.
2. Set `ui_in[7:5]` to 3'b011, to choose measurement of $t_{clkq} + t_{setup}$.
3. Turn off `ui_in[3]`.
4. Turn off `ui_in[2]` to reset.
5. Turn on `ui_in[2]`.
6. Turn on `ui_in[3]` to start.
7. Read `uio`.
8. Calculate $t_{clkq} + t_{setup} = \frac{uio \times 20}{16}$.

Measure t_{skew}

1. Connect `ui_in[1]` to a device that generates exactly 50 MHz.
2. Connect `ui_in[4]` to a device of the target clock.
3. Set `ui_in[7:5]` to 3'b100, to choose measurement of t_{skew}.
4. Turn off `ui_in[2]` to reset.
5. Turn on `ui_in[2]` to start.
6. Read `uio`.
7. Calculate $t_{skew} = uio \times t_{pd,INV}$.

External hardware

- An equipment to measure frequency.

- A device that generates a precise 50 MHz signal, such as a ceramic resonator.
- A device that generates the target clock.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	enable_ring_osc	inv_ring_osc_signal	counter_output[0]
1	measurement_clock	nand2_ring_osc_signal	counter_output[1]
2	reset_counter_n	—	counter_output[2]
3	d_input	—	counter_output[3]
4	target_clock	—	counter_output[4]
5	select[0]	—	counter_output[5]
6	select[1]	—	counter_output[6]
7	select[2]	—	counter_output[7]

PILIPINAS_IC

by **Alexander Co Abad** & **Dino Dominic Ligutan**

0577 1 Hz Wokwi Project

github.com/alexandercoabad/PILIPINAS_IC

wokwi.com/projects/392873974467527681

"7-seg Display for PILIPINASLASALLE"

How it works

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

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

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

How to test

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

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

External hardware

7-segment display

Project Pinout

Digital Pins

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

PRISM 8 with TinySnake

by **Ken Pettit**

0579 10 kHz Wokwi Project

github.com/kdp1965/ttgf-um-pettit-prism

wokwi.com/projects/442081253563458561

"*PRISM plus Tiny Snake implemented in Wowki*"

How it works

This is a very simple Wokwi example that incorporaates two different designs.

Design 1 - 7-Seg Snake

The first design uses the 7-Segment display to show a 3-segment "snake" as it moves around the display. It uses three 3-bit registers to store the current location of the "head", "body" and "tail", along with a register identifying the direction (0=clockwise, 1=counter clockwise).

There are two larger registers also, one a simple counter for speed control and the other a Linear Feedback Shift Register (LFSR) to randomize the direction of travel.

Design 2 - Mini PRISM

The second design is a small implementation of the Programmable Reconfigurable Indexed State Machine (PRISM). This is a Verilog programmable 8-state finite state machine that uses an 8-entry, 22 wide State Table Execution Word (STEW) to define state transitions and output values based on current state and input values. The PRISM includes a counter

How to test

Supply a 10 KHz clock. Then set the speed using the ui_in[7:0] pins. Larger binary values represent slower speed. Start off with something like 8'h20 (i.e. ui_in[5] HIGH, the rest LOW). Watch the snake move around. Try different speeds.

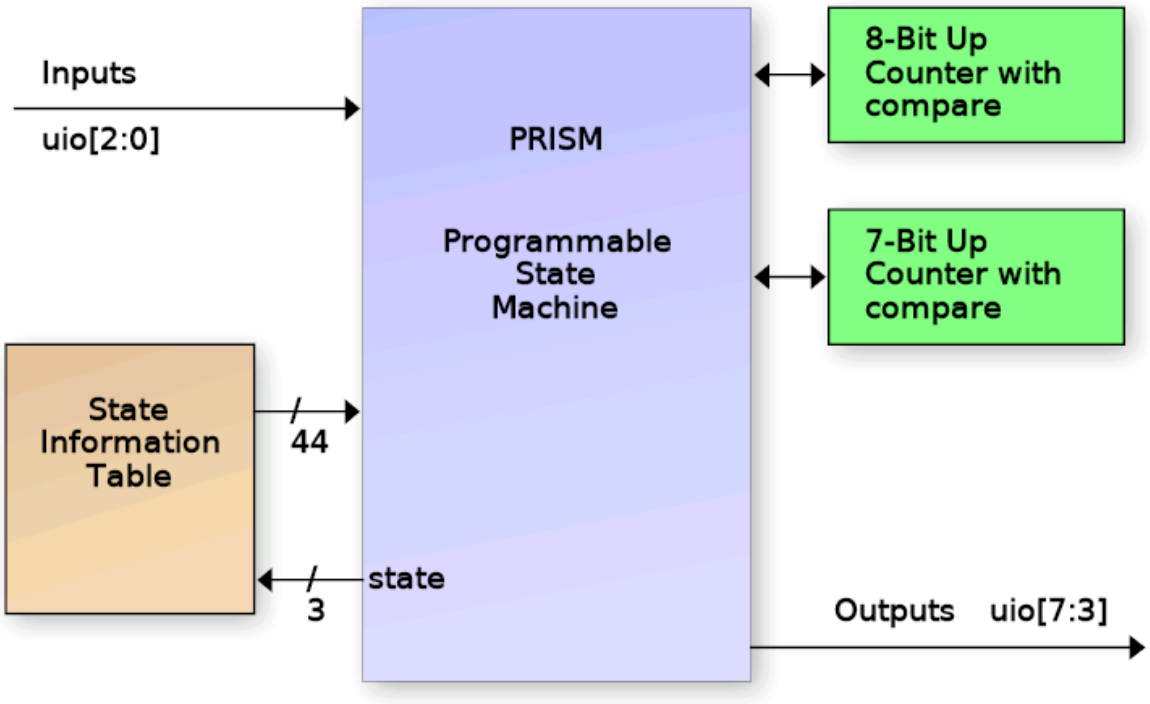
NOTE: When changing from a slower to a faster speed, the initial update may take a few seconds. This is because the This is because the counter may already be larger than the newly entered "speed" value, and therefore must count all the way up until it wraps to zero. The speed compare is a simple EQUAL circuit and doesn't check for GREATER-THAN-OR-EQUAL.

External hardware

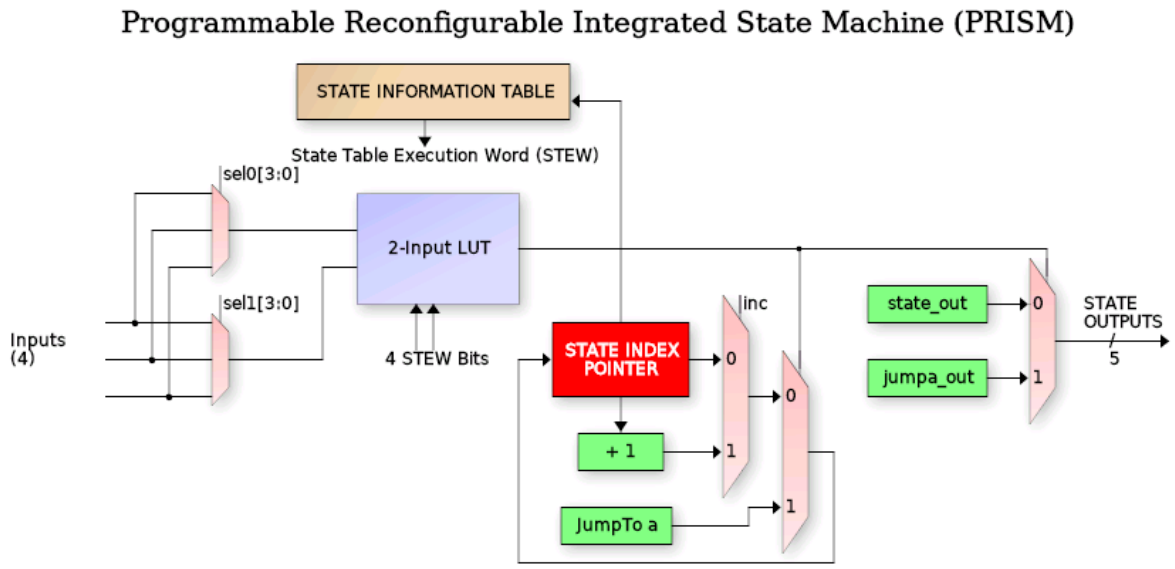
Only need the 7-Segment display on the demo board.

PRISM

PRISM (Programmable Reconfigurable Indexed State Machine) is a block that executes a Verilog-coded Mealy state machine loaded via a runtime loadable configuration bitstream generated by a custom branch of Yosys. PRISM includes it's own counter and compare sub-peripheral for performing timing operation as well.



The PRISM controller block is a programmable state machine that uses an N-bit (3 in this case) index register to track the current FSM state. That index is a pointer into the State Information Table (SIT) to request the State Execution Word (STEW).



What can it do?

Operating priciples of PRISM

PRISM supports FSM designs up to 8 states and includes controllable peripherals such as counters, communication shift register, FIFO and interrupt support. It also features an integrated debugger with 2 breakpoints, single-stepping and readback of state information. Due to long combinatorial delays, PRISM operates from a divide-by-two clock (32Mhz max). The following is a block diagram of the PRISM controller:

Each state is encoded with a 22-bit execution word that controls the FSM output values, input values and state transition decision tree for that state. The peripheral is operated by loading a “Chroma” (more on that below), or execution personality in the 192 bit configuration array as well as configuring the 3 bits of operational mode configurtaion. The 3-bits of operational configuration include:

- Debug output bit: Sends internal state operations to uo_out[7:0]
- Auto-clear bit: Auto clears the counter when compare match occurs
- Split-coiunter: Splits the 15-bit conter into an 8-bit plus 7-bit each with individual compare.

Once a chroma has been loaded, the control register programmed and the PRISM enabled, the FSM will start at state 0. Eight of the bits in the State Execution Word (STEW) specify which of 16 inputs get routed to the 2-input Look Up Table (LUT) that makes the decision for jumping to the specified state (stored in 3 bits of the STEW). While in any state, there is a set of 11 (from the STEW) output bits that drive the PRISM block outputs when the

LUT output is zero (no jump) and 11 more that are output during a jump (transitional outputs).

Each state also has an independent 16-input mux (4-bits from STEW) driving a 1-input LUT to drive a “conditional output”. This is an output who’s value is not strictly depedent on the static values in the STEW for the current state, but rather depends on the selected input during that state.

State Looping (important)

In larger PRISM implementations, each state has “dual-compare” with two N-bit LUTs which allows jumping to one of two possible states. Due to size restrictions, this peripheral does not include dual-compare. Instead the PRISM implementation has (in each state’s STEW), a single “increment state” bit.

In any state where the ‘inc’ bit is set and the LUT output is FALSE (i.e no jump), then the state will increment to the next state, and the “starting state” of the first occurance of this will be saved (i.e. start of loop). Then each successive state can test a different set of inputs to jump to different states. When a state is encountered with the ‘inc’ bit NOT set, PRISM will loop back to the “starting state” and loop through that set of states until the first TRUE from a LUT causing a jump, clearing the loop.

TL/DR

1. Load a Chroma defining the FSM and enable PRISM.
2. State starts at zero.
3. Each state chooses up to 3 of 16 inputs via config bits.
4. 2-input LUT decides if “jump to defined state” occurs.
5. Increment bit decides if “state looping” is in effect.
6. State looping ends when first LUT jump occurs.
7. Outputs bits from STEW for “non-jump” and “transitional jump”.
8. One conditional output based on single selected input per state.

External PRISM Inputs

Inputs to the PRISM engine come from the uio_in[2:0] pins of the TinyTapeout ASIC (in 2-0) as well as the counter compare logic (in 3). The in3 compare logic has special modes as follows:

1. If not split-mode (i.e. 15-bit counter), in3 goes high when count[14:0] == compare.
2. If split-mode counter, in3 goes high when the 7-bit count compare matches, then again when 8-bit count compare matches.

External PRISM Outputs

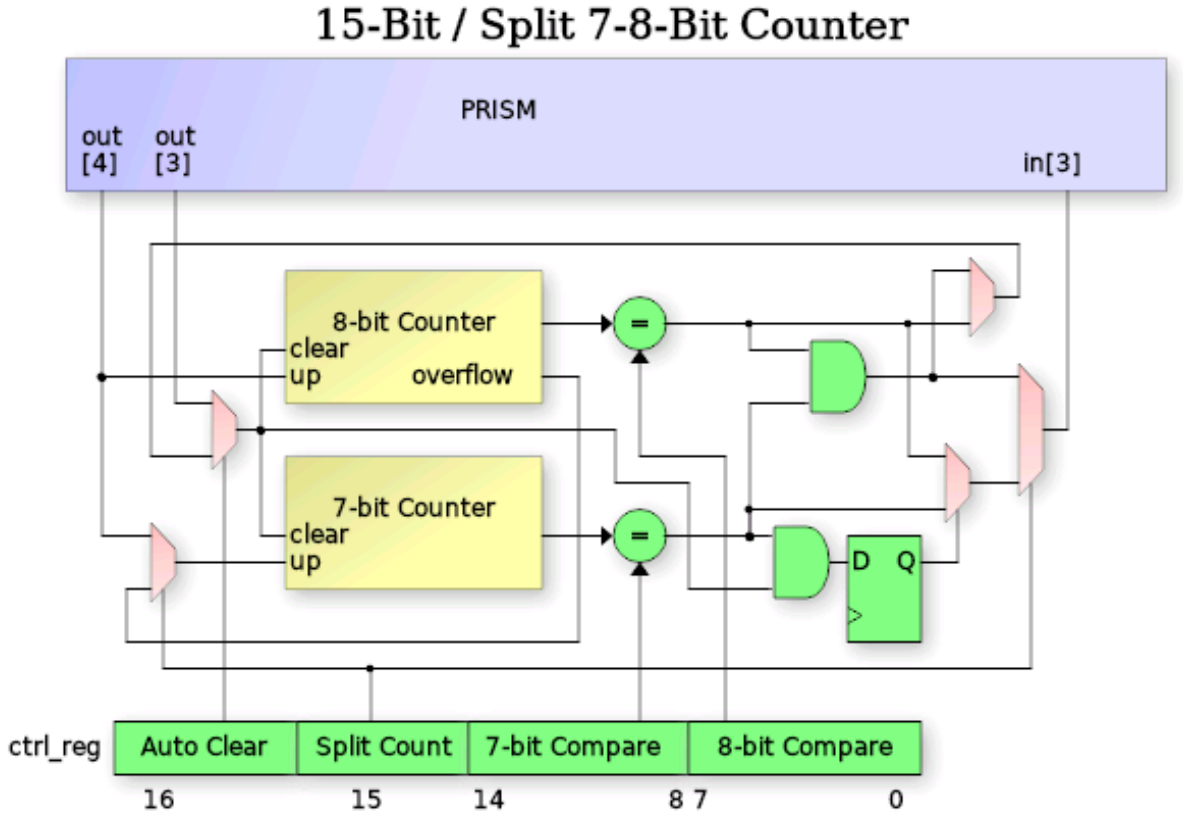
The PRISM has 5 outputs, all of which are visible on uio_out[7:3]. Outputs [5] and [4] also have special internal functions as follows:

out[5]: Counter increment out[4]: Counter clear (when not in auto-clear mode)

All outputs are registered via the main clock to prevent output glitching caused by combinatoril transitions during state decision switching.

15-Bit Counter

The 15-bit counter is an up/clear counter controllable from the PRISM chroma.



Chroma

Chroma are PRISM's version of "personalities". Each chroma is a unique hue of PRISM's spectrum of behavior. Chroma's are coded as Mealy state machines in Verilog to define FSM inputs, outputs and state transitions:

```
always @(posedge clk or negedge rst_n)
   if (~rst_n)
      curr_state <= 3'h0;
   else
      curr_state <= fsm_enable ? next_state : 'h0;

always_comb
begin
   pin_out[5:0]   = 6'h0;
   count1_dec     = 1'b0;
   etc.
```

```
   case (curr_state)
   STATE_IDLE:       // State 0
      begin
         // Detect I/O shift start
         if (host_in[HOST_START])
         begin
            // Load inputs
            pin_out[GPIO_LOAD] = 1'b0;

            // Load 24-bit shift register from preload (our
OUTPUTS)
            count1_load = 1'b1;
            next_state = STATE_LATCH_INPUTS;
         end
      end
   STATE_LATCH_INPUTS:  // State 1
      begin
         next_state = STATE_SHIFT_BITS;
      end
   etc.
end
```

Chroma are compiled into PRISM programmable bitstreams via a custom fork of Yosys (see link below) using a configuration file describing the PRISM architecture. In addition to bitstream generation, the Yosys PRISM backend also calculates the ctrl_reg value for configuring the PRISM peripheral muxes, etc. There are several output formats including C, Python and columnar list:

ST	Mux0	Mux1	Inc	JmpA	OutA	Out	CfgA	STEW
0	0	0	0	1	08	08	e	039082
1	3	0	0	2	11	11	a	0ea314
2	3	0	0	0	12	12	a	0ea520
3	0	0	0	0	00	00	f	03c000
4	0	0	0	0	00	00	f	03c000
5	0	0	0	0	00	00	f	03c000
6	0	0	0	0	00	00	f	03c000
7	0	0	0	0	00	00	f	03c000

The table has the following fields

- ST: the state (obvious)
- Mux0: Selects input for LUT2 input 0 (jump decision)
- Mux1: Selects input for LUT2 input 1
- Inc: Set when next state looping is requested (i.e. ‘else state <= ST_A’)
- JmpA: The “Jump to” state if LUT2 output is TRUE

- OutA: Outputs during “jump to” JmpA state (LSB is PRISM out[0])
- Out: Output during no-jump, steady-state dwelling
- CfgA: The LUT2 4-bit lookup table values
- STEW: The complete word aggregrated in proper bit order

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	speed[0]/prism_enable	seg_a	prism_in[0]
1	speed[1]/prism_cfg_data	seg_b	prism_in[1]
2	speed[2]/prism_array_clk	seg_c	prism_in[2]
3	speed[3]/prism_cfg_clk	seg_d	prism_out[0]
4	speed[4]	seg_e	prism_out[1]
5	speed[5]	seg_f	prism_out[2]
6	speed[6]	seg_g	prism_out[3]
7	speed[7]	prism_debug	prism_out[4]

Register bank accessible through SPI and I2C

by **Caio Alonso da Costa**

581 | 50 MHz | HDL Project

github.com/calonso88/tt_spi_i2c_reg_bank

"*Register bank accessible through SPI and I2C*"

How it works

Register bank accessible throught two different serial interfaces: SPI and I2C. Use digital input to select prefered interface.

There are 8 read/write 8 bit registers and 8 read only 8 bit registers.

Address 0 (first byte in read/write register space) drives the 7 segment display.

Digital input ui_in[7] = 0 selects SPI and ui_in[7] = 1 selects I2C.

SPI peripheral design based on https://github.com/calonso88/tt07_alu_74181

See that design's docs for information about the SPI peripheral.

Small improvement done on the spi_peripheral module. There used to be two buffer counters (one for RX and one for TX). Since the counters are not used together, it was possible to remove one of them and use a single buffer counter. This has reduced 4 flip flops in total and some combinatorial logic as well.

Added logic to control driver for MISO. On previous submissions of this design, the MISO was always driven. Logic has been added to put MISO into high impedance when CS_N is driven high. Due to a 2-stage synchronizer, the MISO goes to high impedance after 2 clock cycles.

I2C peripheral design based on https://github.com/sanojn/tt06_ttrpg_dice

See that design's docs for information about the I2C peripheral.

How to test

Use SPI1 Master peripheral in RP2040 to start communication on SPI interface towards this design. Remember to configure the SPI mode using the switches in DIP switch (if you'd like to have CPOL=1 and CPHA=1). Alternatively, don't use the DIP switches and use the RP2040 GPIOs to configure the SPI mode in the desired mode.

Example code to initialize SPI in REPL:

```
spi_miso = tt.pins.pin_uio3
spi_cs = tt.pins.pin_uio4
spi_clk = tt.pins.pin_uio5
spi_mosi = tt.pins.pin_uio6
spi_miso.init(spi_miso.IN, spi_miso.PULL_DOWN)
spi_cs.init(spi_cs.OUT)
spi_clk.init(spi_clk.OUT)
spi_mosi.init(spi_mosi.OUT)
spi = machine.SoftSPI(baudrate=10000, polarity=0, phase=0, bits=8,
firstbit=machine.SPI.MSB, sck=spi_clk, mosi=spi_mosi,
miso=spi_miso)
spi_cs(1)
```

Example code to write 0xF8 to address[0]:

```
spi_cs(0); spi.write(b'\x80\xF8'); spi_cs(1)
```

This should set the 7 segment LED to 0xF8 which will display “t.”

Seg A - OFF, Seg B - OFF, Seg C - OFF, Seg D - ON, Seg E - ON, Seg F - ON, Seg G - ON, Seg DP - ON

Example code to read from address[0]:

```
spi_cs(0); spi.write(b'\x00'); spi.read(1); spi_cs(1)
```

The result should be 0xF8 or whatever you wrote to address[0].

TODO: I2C documentation.

External hardware

Not required. Write to the first register to set the LEDs on the demoboard.

External hardware

None.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	cpol	spare[0]	—
1	cpha	spare[1]	i2c_sda
2	—	spare[2]	i2c_scl
3	—	spare[3]	spi_miso
4	—	spare[4]	spi_cs_n
5	—	spare[5]	spi_clk
6	sel[1]	spare[6]	spi_mosi

#	Input	Output	Bidirectional
7	sel[0]	spare[7]	—

Cell mux

by **htfab**

0583 HDL Project

`github.com/htfab/ttgf0p2-cells`

"All the gf180mcu 7-track standard cells"

How it works

Instantiates one copy of each standard cell from the GF180mcu 7-track library, and multiplexes the project i/o pins so that the functional behaviour of each cell can be verified.

The instantiated cells have a total of 210 output pins, arranged into 27 pages of 8 pins each. Once a certain page is selected, those 8 outputs are mapped to the project output pins.

How to test

Select a page using `ui_in[4:0]`.

Set the cell inputs using `ui_in[7:5]` and `uio_in[2:0]`.

The cell outputs for the selected page should appear on `uo_out`.

External hardware

None

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	page bit 0	output bit 8*page	input bit 3
1	page bit 1	output bit 8*page+1	input bit 4
2	page bit 2	output bit 8*page+2	input bit 5
3	page bit 3	output bit 8*page+3	enable tristate cells
4	page bit 4	output bit 8*page+4	—
5	input bit 0	output bit 8*page+5	—
6	input bit 1	output bit 8*page+6	—
7	input bit 2	output bit 8*page+7	—

Super-Simple-SPI-CPU

by **James Ashie Kotey**

`0609` `30 MHz` `HDL Project`

`github.com/Sheffield-Chip-Design-Team/SPI-CPU-Final`

"A 4-bit CPU using the SPI Flash RAM from the QSPI PMOD to load programs."

How it works

The Super-Simple-SPI-CPU has uses the emulated RP40-RAM to make load and run programs.

How to test

Use the ram flasher app to load a program into the ram and then reset the CPU.

External hardware

Ram Flasher.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	DATA_A[0]	OUT_REG[0]	SPI_CS_N
1	DATA_A[1]	OUT_REG[1]	SPI_MOSI
2	DATA_A[2]	OUT_REG[2]	SPI_MISO
3	DATA_A[3]	OUT_REG[3]	SPI_SCK
4	DATA_B[0]	OUT_REG[4]	—
5	DATA_B[1]	OUT_REG[5]	—
6	DATA_B[2]	OUT_REG[6]	—
7	DATA_B[3]	OUT_REG[7]	CPU_VALID

Example of Bad Synchronizer

by **Darryl Miles project from Eric Smith**

0610 HDL Project

`github.com/dlmiles/ttgf25a-bad-synchronizer`

"*Figure 29.3 from Dally & Harting*"

How it works

Badly

This project is based on (https://github.com/ericsmi/tt07-bad-synchronizer) but for GF180MCU.

How to test

Align clocks, push them apart, look for bit errors

External hardware

A way to generate a clock

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	clk1	stage3[0]	stage2[0]
1	0=Base-2 1=GrayCode	stage3[1]	stage2[1]
2	counter enable	stage3[2]	stage2[2]
3	—	stage3[3]	stage2[3]
4	—	skew	stage1[0]
5	—	—	stage1[1]
6	—	—	stage1[2]
7	—	—	stage1[3]

Video mode tester

by **htfab**

0611 64 MHz HDL Project

`github.com/htfab/ttgf0p2-vga-tester`

"*Experiment with different VGA timing parameters*"

Author: htfab

Peripheral index: 28

What it does

There is quite some variability between screens (and VGA/HDMI adapters) in the set of VGA timing configurations they support.

Due to constraints and optimization pressures, VGA designs on Tiny Tapeout typically use a single resolution that cannot be changed without a respin. It would therefore be useful to gather some crowdsourced information on what VGA modes are well supported among the community.

This peripheral facilitates gathering that information.

It allows setting the horizontal and vertical timing parameters (visible pixels, front porch, sync pulse, back porch) and displays a simple test pattern on the screen. There is a thin white border along the screen edges to quickly check whether anything was cut off.

Each phase (visible pixels, front porch, sync pulse, back porch) is described by its length in pixels (a 13-bit integer) and 3 single-bit flags. Internally all 4 phases are identical and the flags are the mechanism to differentiate their behaviour:

- bit 15: keep hsync/vsync high during this phase
- bit 14: allow data on the r/g/b pins during this phase
- bit 13: advance to the next line/frame at the end of this phase

For instance, a video mode with positive hsync/vsync polarity could use flags 010 for the visible pixels, 000 for the front porch, 100 for the sync pulse and 001 for the back porch.

Register map

Address	Name	Access	Description
0x00	DATA	R/W	Horizontal visible pixels, high byte (incl. flags)
0x01	DATA	R/W	Horizontal visible pixels, low byte

0x02	DATA	R/W	Horizontal front porch, high byte (incl. flags)
0x03	DATA	R/W	Horizontal front porch, low byte
0x04	DATA	R/W	Horizontal sync pulse, high byte (incl. flags)
0x05	DATA	R/W	Horizontal sync pulse, low byte
0x06	DATA	R/W	Horizontal back porch, high byte (incl. flags)
0x07	DATA	R/W	Horizontal back porch, low byte
0x08	DATA	R/W	Vertical visible pixels, high byte (incl. flags)
0x09	DATA	R/W	Vertical visible pixels, low byte
0x0a	DATA	R/W	Vertical front porch, high byte (incl. flags)
0x0b	DATA	R/W	Vertical front porch, low byte
0x0c	DATA	R/W	Vertical sync pulse, high byte (incl. flags)
0x0d	DATA	R/W	Vertical sync pulse, low byte
0x0e	DATA	R/W	Vertical back porch, high byte (incl. flags)
0x0f	DATA	R/W	Vertical back porch, low byte

How to test

To use the universally supported 640x480 @ 60 Hz video mode, we would like to set

- Horizontal visible pixels: 640 (high byte 2, low byte 128)
  - 0x00: 66 (“visible” flag adds 64)
  - 0x01: 128
- Horizontal front porch: 16 (high byte 0, low byte 16)
  - 0x02: 0
  - 0x03: 16
- Horizontal sync pulse: 96 (high byte 0, low byte 96)
  - 0x04: 128 (“sync” flag adds 128)
  - 0x05: 96
- Horizontal back porch: 48 (high byte 0, low byte 48)
  - 0x06: 32 (“advance” flag adds 32)
  - 0x07: 48
- Vertical visible pixels: 480 (high byte 1, low byte 224)
  - 0x08: 65 (“visible” flag adds 64)
  - 0x09: 224
- Vertical front porch: 10 (high byte 0, low byte 10)
  - 0x0a: 0
  - 0x0b: 10
- Vertical sync pulse: 2 (high byte 0, low byte 2)
  - 0x0c: 128 (“sync” flag adds 128)

    - 0x0d: 2
- Vertical back porch: 33 (high byte 0, low byte 33)
    - 0x0e: 32 (“advance” flag adds 32)
    - 0x0f: 33

After setting the pixel clock to 25 MHz and writing these registers the test pattern should appear on the screen connected to the Tiny VGA PMOD.

External hardware

Tiny VGA PMOD

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	TinyVGA red 1	—
1	—	TinyVGA green 1	—
2	—	TinyVGA blue 1	—
3	—	TinyVGA vsync	spi_miso
4	—	TinyVGA red 0	spi_cs_n
5	—	TinyVGA green 0	spi_clk
6	—	TinyVGA blue 0	spi_mosi
7	—	TinyVGA hsync	—

One Bit PUF

by **Yimin Gao** & **Ceylan Morgul**

0612 HDL Project

github.com/Capulus123/ttihp03-multi-bit-puf

"It is a PUF based on a difference of two registers"

How it works

This is a PUF design that includese 2**ADDR_BITS x OUT_BITS one_bit_pufs The addr is the address to read OUT_bits of the PUF bits For instance if ADDR_BITS = 2, OUT_BITS = 2 The design will include 8 one_bit_pufs, addr = 2'b10 will read 2 puf bits (OUT[5:4])

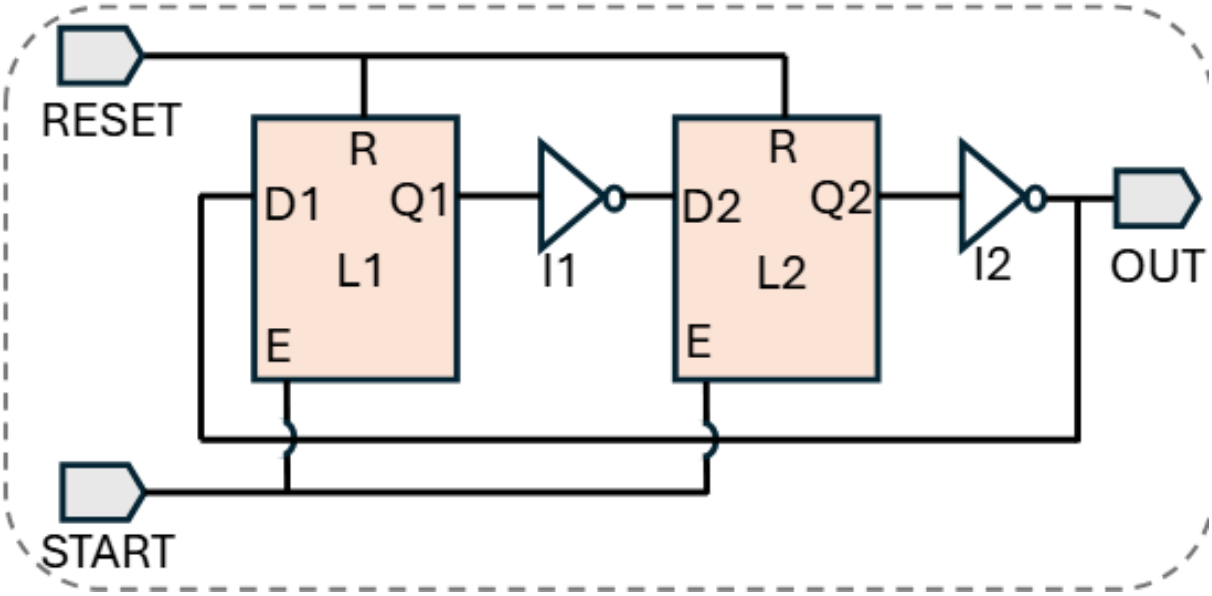


Figure 612.1: PUF block diagram

How to test

The output is 0 in the reset condition.

External hardware

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	addr[0]	puf_out[0]	—

#	Input	Output	Bidirectional
1	addr[1]	puf_out[1]	—
2	addr[2]	puf_out[2]	—
3	addr[3]	puf_out[3]	—
4	—	puf_out[4]	—
5	—	puf_out[5]	—
6	—	puf_out[6]	—
7	start_signal	puf_out[7]	—

DDR throughput and flop aperature test

by **Darryl Miles project from Eric Smith**

613 HDL Project

`github.com/dlmiles/ttgf25a-ddr-throughput-test`

> *"Grab data on every edge of clock with varying pos pulse width"*

How it works

Badly probably.

Use a positive edge detector on the clock and its compliment. Or together those dectors to get 2 positive pulses per period or a 2x clock. Vary clk 2x pos pulse width by varying number of inv per detect.

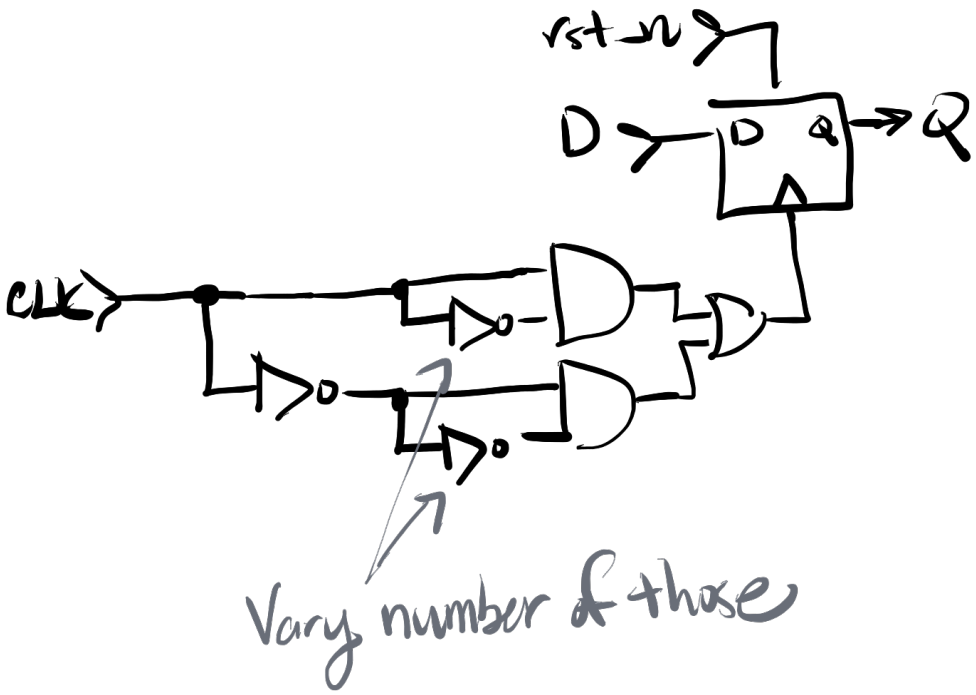


Figure 613.1: Concept Diagram

How to test

Carefully.

External hardware

Analog Discovery 3

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	pulse = 1 inv	q for pulse = 1 inv	—
1	pulse = 3 inv	q for pulse = 3 inv	—
2	pulse = 5 inv	q for pulse = 5 inv	—
3	pulse = 7 inv	q for pulse = 7 inv	—
4	—	q for normal flop	—
5	—	1	—
6	—	1	—
7	—	clk	—

Ring osc on VGA

by **algofoogle (Anton Maurovic)**

`0614` `25.175 MHz` `HDL Project`

`github.com/algofoogle/ttgf0p2-vga-ring-osc`

"VGA display visualisation of a ring oscillator doing work"

How it works

Manually-instantiated gf180mcuD inverter cells form a chain out of chain segments of varying lengths, allowing the user to select given points in the overall chain to loop back to produce a ring oscillator. This makes a configurable ring oscillator that is expected to be able to oscillate from about 15MHz up to 350MHz.

This (or an external clock) then can be selected to drive a "worker" module: a counter which counts up to 3000.

Alongside this is a VGA sync generator which takes its pixel colour from whatever is in the upper 6 bits of the worker's counter at the time. The worker is reset during HBLANK of each VGA line.

It's expected that at the faster ring oscillator speeds, the counter will reach its target of 3000 sooner than the width of the VGA line but with some jitter... or the counter/compare logic will break down because it's too fast.

How to test

Set `clksel2[1:0]` to 0.

Set `clksel[3:0]` to (say) 10, or anything greater than 1.

Set `mode[1:0]` to 0 (though these are unused at the time of writing; TBA).

Set `vga_mode` to 0.

Attach a Tiny VGA PMOD to `uo_out`.

Supply a 25MHz clock to the system `clk`, and assert reset for at least 2 clocks.

Expect to see vertical coloured bars on screen, but expect some jitter. Their width should increase as you increase `clksel`.

Measure the ring oscillator (or rather, the selected clock source) on `uio_out[7:4]`: `uio_out[4]` is the raw oscillator output, and the higher bits are the oscillator divided by powers of 2.

More testing notes:

- When `vga_mode==1`, `clk` should be 26.6175MHz (106.47 MHz ÷ 4) to drive a 1440x900 60Hz VGA display.
- When `clksel2` is:
  - 0: Just rely on `clksel`.
  - 1: Use fixed 5-deep inv_1 ring oscillator.
  - 2: Use fixed 5-deep inv_4 ring oscillator.
  - 3: Use inverted `clk`.
  - NOTE: options 1 and 2 require `clksel > 1` (any value will do) to enable the rings.
- When `clksel2==0` and `clksel` is:
  - 0: Use `clk`.
  - 1: Use `altclk`.
  - For values 2 and above, use an inv_2-based ring oscillator tapped at...
  - 2: => 3
  - 3: => 5
  - 4: => 7
  - 5: => 9
  - 6: => 13
  - 7: => 17
  - 8: => 21
  - 9: => 25
  - 10: => 33
  - 11: => 41
  - 12: => 49
  - 13: => 65
  - 14: => 97
  - 15: => 161

External hardware

Tiny VGA PMOD and a VGA monitor.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	clksel[0]	r7	IN: clksel2[0]
1	clksel[1]	g7	IN: clksel2[1]
2	clksel[2]	b7	—
3	clksel[3]	vsync	—
4	altclk	r6	OUT: osc

#	Input	Output	Bidirectional
5	mode[0]	g6	OUT: div2
6	mode[1]	b6	OUT: div4
7	vga_mode	hsync	OUT: div8

GF180MCU loopback tile with input skew measurement

by **Darryl Miles project from Eric Smith**

0615 | 10 MHz | HDL Project

github.com/dlmiles/ttgf25a-loopback-with-skew

"Count up to 10, one second at a time."

How it works

This project is based on (https://github.com/ericsmi/tt05-loopback-with-skew) but for GF180MCU.

How to test

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

External hardware

Skewable clock and data source.

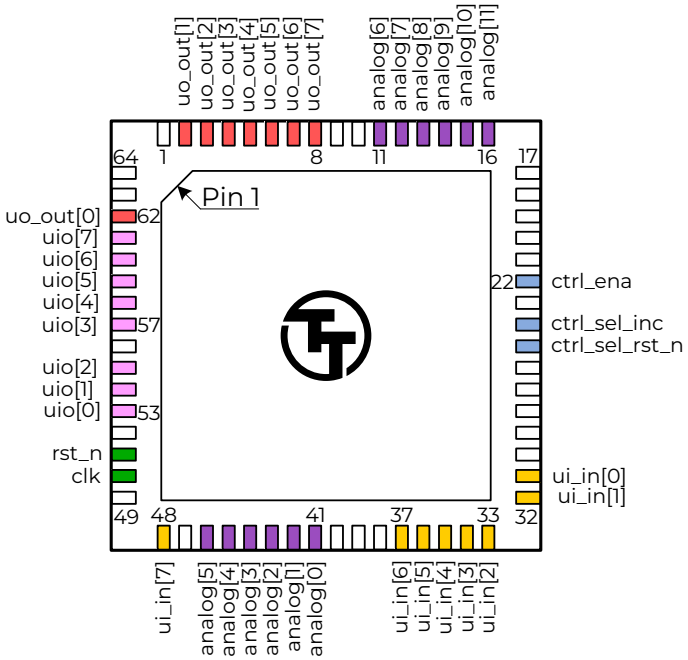
Project Pinout

Digital Pins

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

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:

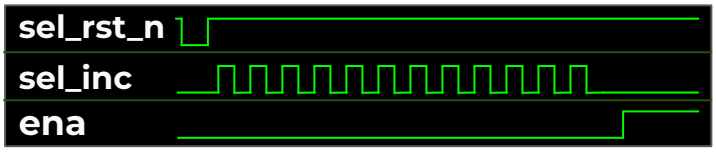


Figure 1: Mux signals for activating the design at address 12

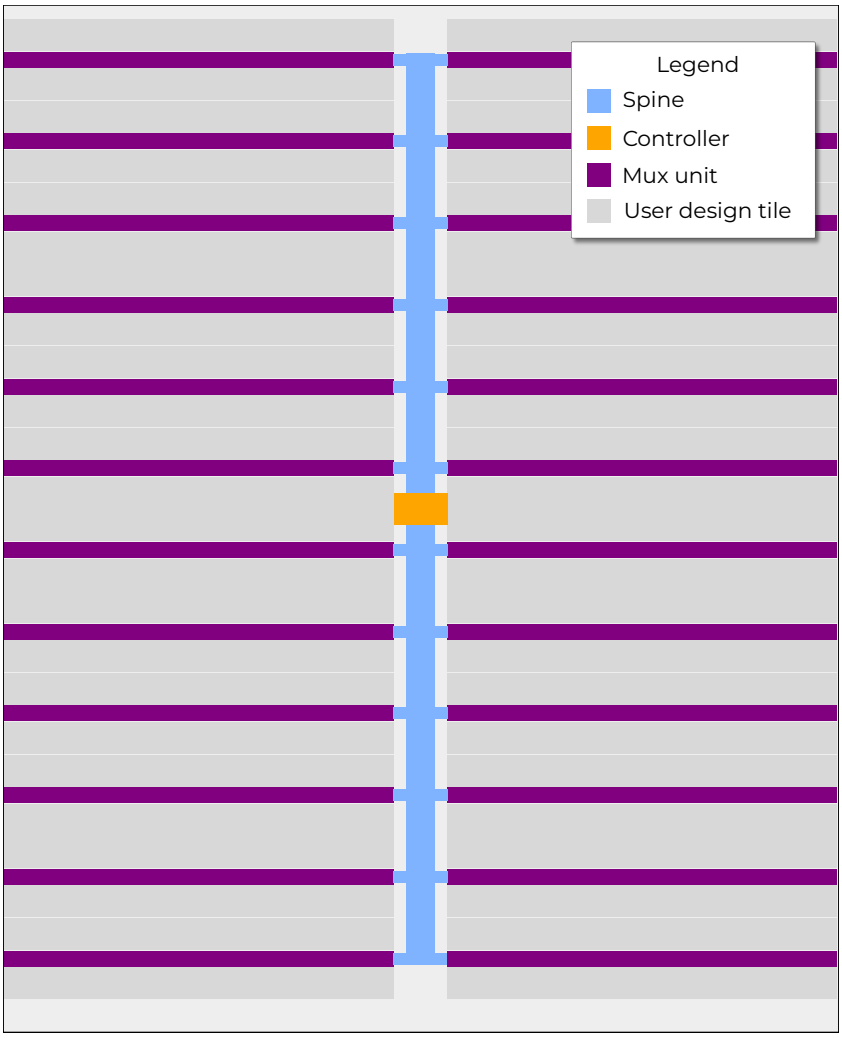


Figure 2: Mux Diagram

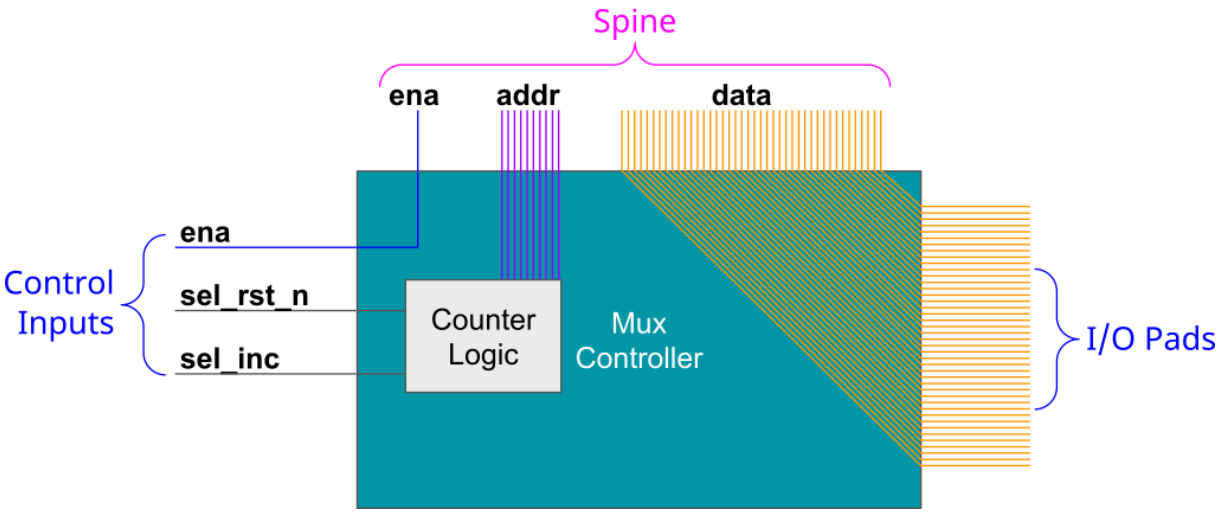


Figure 3: Mux Controller Diagram

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	Function	Signal
1	Mux Control	ctrl_ena
2	Mux Control	ctrl_sel_inc
3	Mux Control	ctrl_sel_rst_n
4	Reserved	(none)
5	Reserved	(none)
6	Reserved	(none)
7	Reserved	(none)
8	Reserved	(none)
9	Output	uo_out[0]
10	Output	uo_out[1]
11	Output	uo_out[2]
12	Output	uo_out[3]
13	Output	uo_out[4]
14	Output	uo_out[5]
15	Output	uo_out[6]
16	Output	uo_out[7]
17	Power	VDD IO
18	Ground	GND IO
19	Analog	analog[0]
20	Analog	analog[1]
21	Analog	analog[2]
22	Analog	analog[3]
23	Power	VAA Analog
24	Ground	GND Analog
25	Analog	analog[4]
26	Analog	analog[5]
27	Analog	analog[6]
28	Analog	analog[7]
29	Ground	VDD Core
30	Power	VDD Core
31	Ground	GND IO
32	Power	VDD IO
33	Bidirectional	uio[0]
34	Bidirectional	uio[1]
35	Bidirectional	uio[2]

QFN64 Pin	Function	Signal
36	Bidirectional	uio[3]
37	Bidirectional	uio[4]
38	Bidirectional	uio[5]
39	Bidirectional	uio[6]
40	Bidirectional	uio[7]
41	Input	ui_in[0]
42	Input	ui_in[1]
43	Input	ui_in[2]
44	Input	ui_in[3]
45	Input	ui_in[4]
46	Input	ui_in[5]
47	Input	ui_in[6]
48	Input	ui_in[7]
49	Input	rst_n †
50	Input	clk †
51	Ground	GND IO
52	Power	VDD IO
53	Analog	analog[8]
54	Analog	analog[9]
55	Analog	analog[10]
56	Analog	analog[11]
57	Ground	GND Analog
58	Power	VDD Analog
59	Analog	analog[12]
60	Analog	analog[13]
61	Analog	analog[14]
62	Analog	analog[15]
63	Ground	GND Core
64	Power	VDD Core

† Internally, there's no difference between `clk`, `rst_n` and `ui_in` pins. They are all just bits in the `pad_ui_in` bus. However, we use different names to make it easier to understand the purpose of each signal.

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
Artwork	Used to showcase artwork from our community.
#123 #423/2	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.
25.175 MHz	Clock frequency of the project. May be truncated from actual value or omitted completely.
HDL Project Wokwi Project Analog Project	Project type, indicating if it was made with a HDL, Wokwi, or if it is analog.
Medium Danger High Danger	Indicates the risk that the project presents to the ASIC. Medium danger projects can damage the ASIC under certain conditions, whilst high danger projects *will* 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	Digital design guide	Discord server
tinytapeout.com	tinytapeout.com/digital_design	tinytapeout.com/discord