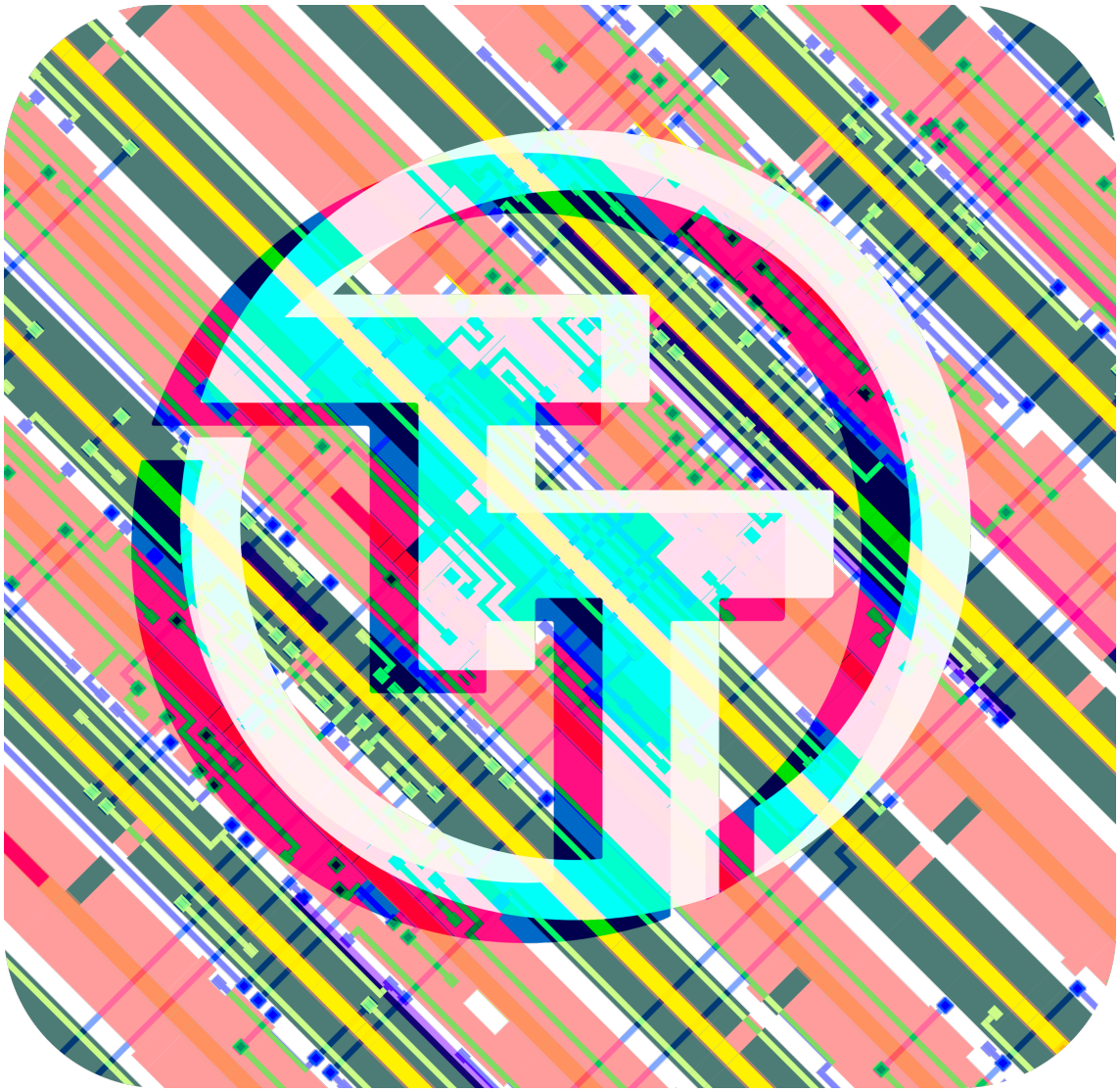


Tiny Tapeout IHP 25b Datasheet

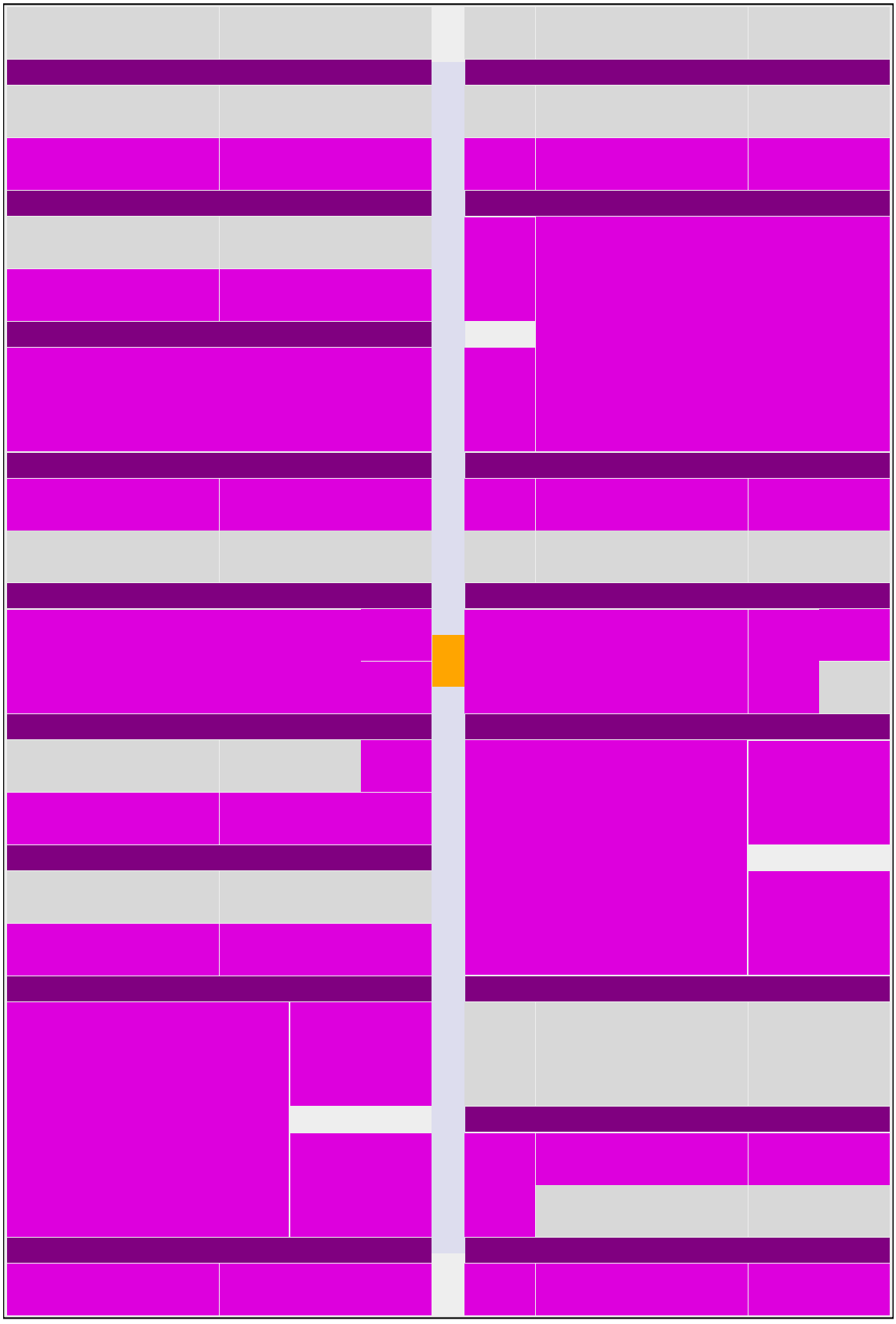
Project Repository
https://github.com/TinyTapeout/tinytapeout-ihp-25b

September 9, 2025

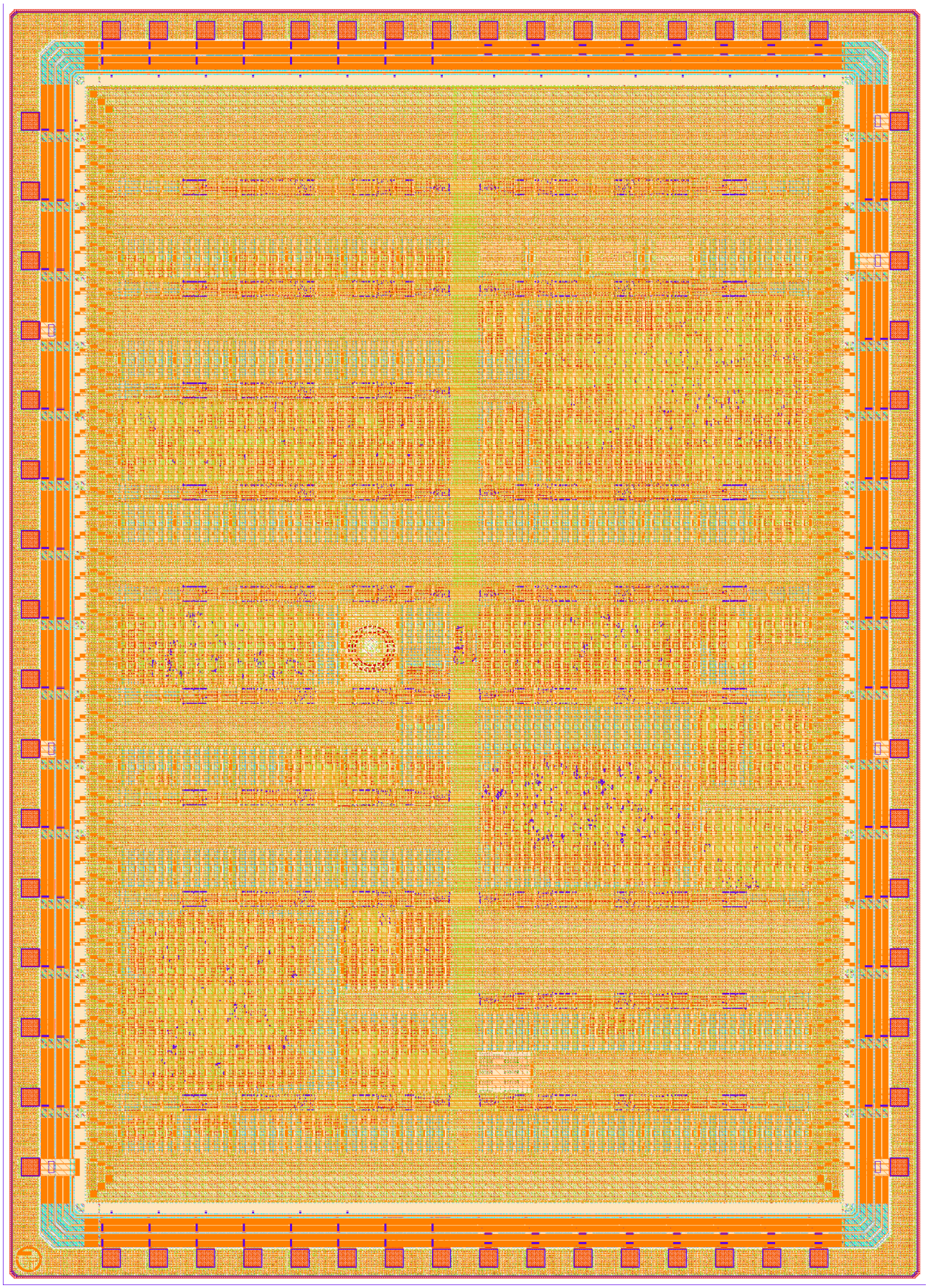
Contents

Chip renders

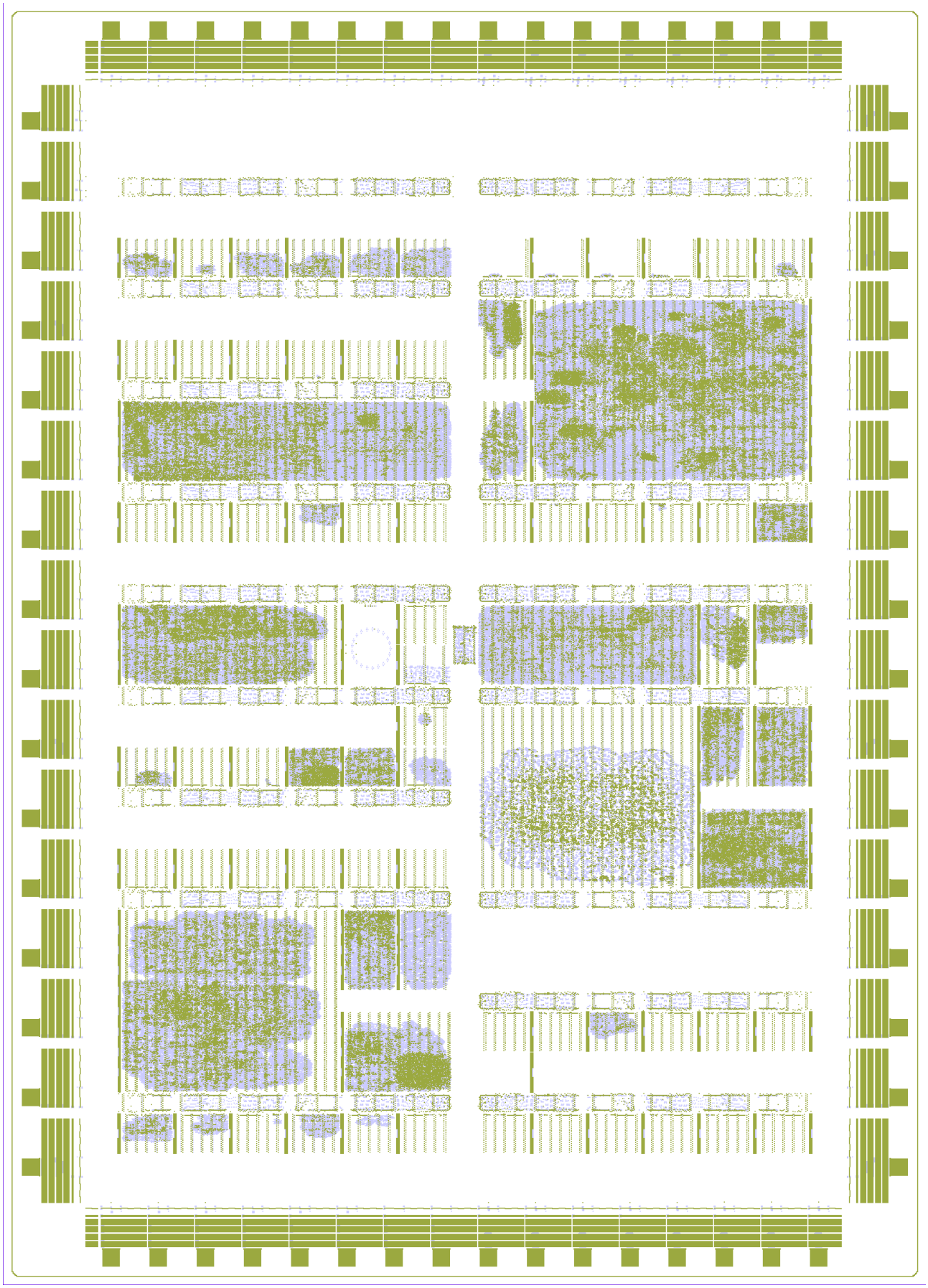
Chip layout map



Full chip render



Logic density view



Projects

Chip ROM [0]

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

How it works

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

The ROM layout The ROM layout is as follows:

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

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

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

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

Here is a complete example of a chip descriptor:

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

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

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

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

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

How to test

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

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

Pinout

#	Input	Output	Bidirectional
0	addr[0]	data[0]	
1	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 [1]

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

How it works

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

It has three modes of operation:

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

Pinout

#	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]

and gate [32]

- Author: mirosuv
- Description: a simple and gate
- GitHub repository
- Wokwi project
- Mux address: 32
- Extra docs
- Clock: 0 Hz

How it works

This design is a simple and gate to test how chip design works.

How to test

The design can be tested whit the first two switches of the test board. The output is visible on the 7-segment display.

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	and gate input pin a	and gate output	
1	and gate input pin b		
2			
3			
4			
5			
6			
7			

Oscillating Bones [34]

- Author: Uri Shaked
- Description: A stylish ring oscillator built from SkullFET transistors
- GitHub repository
- HDL project
- Mux address: 34
- Extra docs
- Clock: 0 Hz

How it works

A simple yet stylish ring oscillator that uses a chain of 21 SkullFET inverters to generate a square wave output. Based on simulation, the oscillator should have a frequency of around 148.6 MHz.

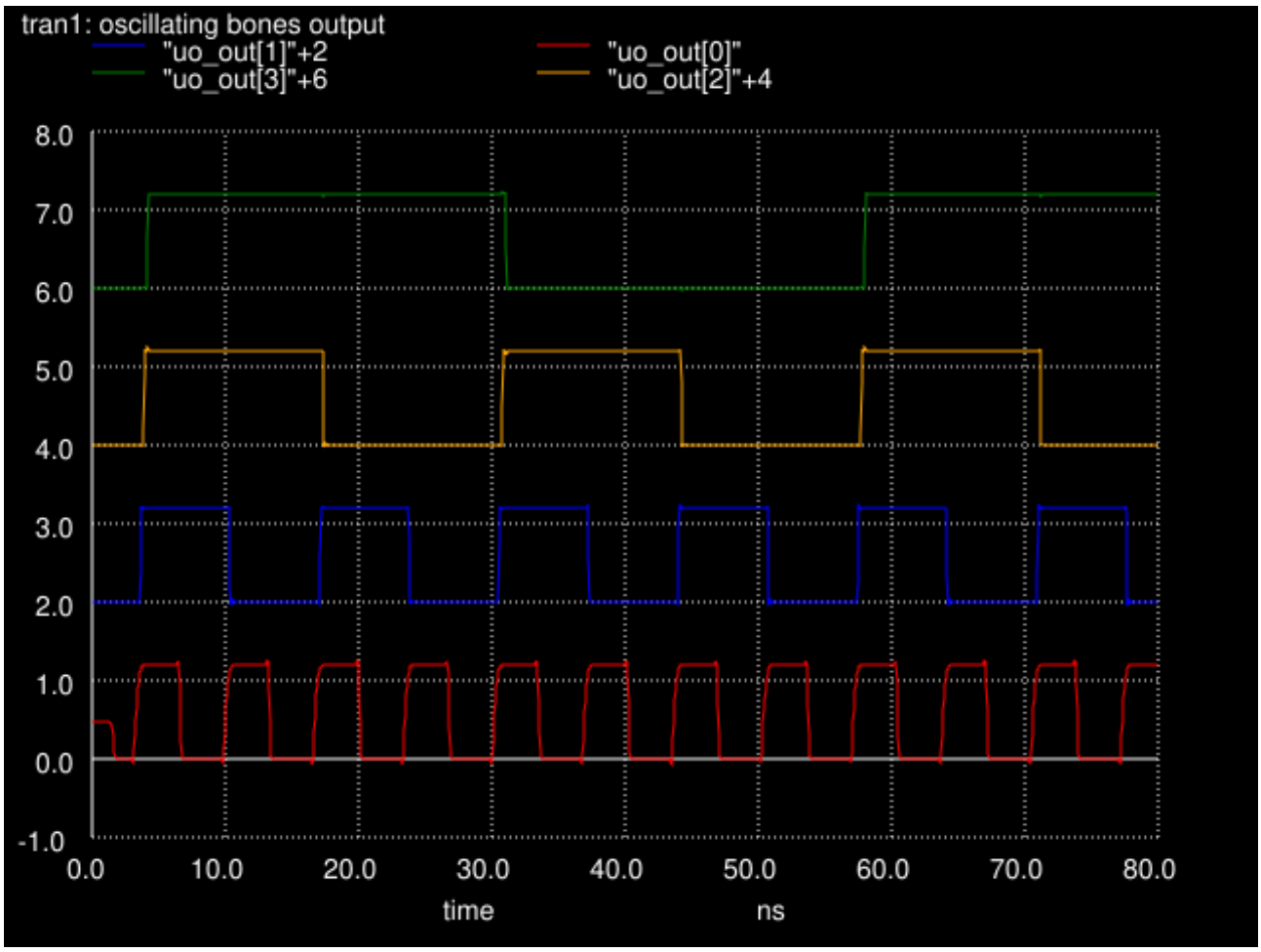
Pin	Expected frequency
osc_out	148.6 MHz
osc_div_2	74.3 MHz
osc_div_4	37.1 MHz
osc_div_8	18.6 MHz

How to test

Connect an oscilloscope to one of the output pins (eg. `osc_div_8` / `uo_out[3]`) and enjoy the show.

Simulation results

The following graph shows the output of the oscillator and the divided outputs. It was generated by running `make -C docs/layout_sim.png`:



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

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

Pinout

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

ENSEIRB-MATMECA RISC-V processor [42]

- Author: Mathieu Escouteloup
- Description: RISC-V core implementation (32-bit with 8 GPRs)
- GitHub repository
- HDL project
- Mux address: 42
- Extra docs
- Clock: 50000000 Hz

How it works

It is a simple RISC-V 32-bit implementation.

How to test

There is no implemented test yet.

External hardware

1 input to select boot loader, 1 input for UART Rx, 1 output for Uart Tx, 8 input/outputs for GPIOs

Pinout

#	Input	Output	Bidirectional
0	BOOT	OUT0_FREE	GPIO0
1	IN1_FREE	OUT1_FREE	GPIO1
2	IN2_FREE	OUT2_FREE	GPIO2
3	UART_RX	OUT3_FREE	GPIO3
4	IN4_FREE	UART_TX	GPIO4
5	IN5_FREE	OUT5_FREE	GPIO5
6	IN6_FREE	OUT6_FREE	GPIO6
7	IN7_FREE	OUT7_FREE	GPIO7

IZH Neuron [73]

- Author: Anwesha Panda
- Description: izh neuron implementation from Sohil Khan
- GitHub repository
- HDL project
- Mux address: 73
- Extra docs
- Clock: 0 Hz

How it works

bla bla lbla

How to test

bla bli bol oblr

External hardware

clfor fokr lpgrgl gt g5g g5

Pinout

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

PWM_SPI [75]

- Author: HES SO
- Description: A PWM module controllable from an spi interface.
- GitHub repository
- HDL project
- Mux address: 75
- Extra docs
- Clock: 50000000 Hz

How it works

PWM is a technique used to output analog results with digital means.
A digital control is used to generate digital signals that can have variable duty cycle.
The goal is to adjust the output pulse width in order to regulate the average output voltage. Our chip is a pwm module that is controlled over an SPI interface.

SPI Command Set These command values are sent over SPI to control the PWM peripheral:

Command (8-bit)	Operation
`8'd1`	Write Compare Value (CV)
`8'd2`	Write Prescaler value
`8'd3`	Write Duty Cycle 1
`8'd4`	Write Duty Cycle 2
`8'd5`	Write Duty Cycle 3
`8'd6`	Disable PWM output
`8'd7`	Enable PWM output

For operations other than `ENABLE_PWM` and `DISABLE_PWM`, the SPI command must be followed by **four data bytes**. These bytes represent the value to be written into the selected register (e.g., Compare Value, Prescaler, or Duty Cycle). The data is transmitted **least significant byte (LSB) first**, so the first byte on the SPI bus corresponds to bits `[7:0]` of the value, the second byte to bits `[15:8]`, and so on, up to the most significant byte.

The SPI slave has a clock polarity of zero and clock phase of zero, CPOL=0 and CPHA=0.

Override pins are present to test the functionality of the PWM module without having to use the SPI. When driven high, the pins set the counter value, prescaler, duty cycles,

and enable signal of the PWM directly to a preset value. When all override pins are driven the output signal should be 10kHz 50% duty cycle on all 3 PWM output pins when ran at 25MHz.

There are two input pins that set the output of a uio output pin to 0, 1, and Z. This is just to test functionality of tristating in tinytapeout.

How to test

Program counter value, and duty cycle registers over SPI by sending the proper command byte followed by 4 bytes to set the value of the register. Then send the enable pwm command byte to start the SPI. Outputs 3 to 7 indicate that the registers value is non zero. Bidirectional output 0 shows if the enable register is non zero. Or use the override pins present on the ui_in to test just the PWM.

External hardware

SPI master device and something to view the PWM signal with.

Pinout

#	Input	Output	Bidirectional
0	Sets Value of uio_out0	pwm_0	EN_nonZero
1	High Z on uio_out0 - 1 for Z	pwm_1	
2	CV_override	pwm_2	
3	PS_override	CV_nonZero	High Z test Pin
4	DC1_override	PS_nonZero	spi_clk
5	DC2_override	DC1_nonZero	miso
6	DC3_override	DC2_nonZero	mosi
7	EN_override	DC3_nonZero	cs_n

SIC-1 8-bit SUBLEQ Single Instruction Computer [102]

- Author: Uri Shaked
- Description: Hardware implementation of the 8-bit Single Instruction Computer
- GitHub repository
- HDL project
- Mux address: 102
- Extra docs
- Clock: 0 Hz

How it works

SIC-1 is an 8-bit Single Instruction computer. The only instruction it supports is SUBLEQ: Subtract and Branch if Less than or Equal to Zero. The instruction has three operands: A, B, and C. The instruction subtracts the value at address B from the value at address A and stores the result at address A. If the result is less than or equal to zero, the instruction jumps to address C. Otherwise, it proceeds to the next instruction.

Memory map The SIC-1 computer has an address space of 256 bytes, and and 8-bit program counter. The first 253 bytes are used for the program memory, and the last 3 bytes are used for input, output, and for halting the computer:

Address	Label	Read	Write
253	@IN	`ui` pins	Ignored
254	@OUT	Returns 0	`uo` pins
255	@HALT	Returns 0	Ignored

Setting the program counter to 253, 254, or 255 will halt the computer.

Each instruction is 3 bytes long, and the program counter is incremented by 3 after each instruction, except when a branch is taken.

For more information, check out the SIC-1 Assembly Language Reference.

Execution cycle Each instruction takes 6 cycles to execute, regardless of whether a branch is taken or not. The execution of an instruction is divided into the following stages:

1. Fetch A: Read the value at address PC

2. Fetch B: Read the value at address PC+1
3. Fetch C: Read the value at address PC+2
4. Read valA: Read the value at address A
5. Read valB: Read the value at address B
6. Store: Subtract valB from valA, store the result at A, and branch if the result is less than or equal to zero.

The pseudocode for the execution cycle is as follows:

```
(1) A <= memory[PC]
(2) B <= memory[PC+1]
(3) C <= memory[PC+2]
(4) valA <= memory[A]
(5) valB <= memory[B]
(6) result <= valA - valB
    memory[A] <= result
    if result <= 0:
      PC = C
    else:
      PC = PC + 3
```

Control signals The `uio` pins are used to load a program into the computer, and to control the computer:

uio pin	Name	Type	Description
0	run	input	Start the computer
1	halted	output	Computer has halted
2	set_pc	input	Set the program counter to the value on ui pins
3	load_data	input	Load the value from the ui pins into the memory at the PC
4	out_strobe	output	Pulsed for one clock cycle when the computer writes to @OUT (
5	dbg[0]	input	Debug select bit 0
6	dbg[1]	input	Debug select bit 1
7	dbg[2]	input	Debug select bit 2

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

dbg[2:0]	Signal
0	None
1	PC
2	A
3	B
4	C
5	valA
6	result (valA - valB)
7	state (3 bits)

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

Programming the SIC-1

You can use the online SIC-1 app to compile and simulate your SIC-1 programs. Click on "Run game" and then "Apply for the job", close the "Electronic mail" popup. Paste the code and click on "Compile" (on the bottom left). You'll see the compiled code in the "Memory" window on the right, and will be able to step through the code.

To load a program and run a program, follow this sequence:

1. Set the `ui` pins to 0 (target address)
2. Pulse the the `load_pc` pin
3. Set the `ui` pins to the value you want to load
4. Pulse the `load_data` pin
5. Repeat steps 3-4 until you have loaded the entire program
6. Set the `ui` pins to the address you want to start at (usually 0)
7. Pulse the `set_pc` pin
8. Set the `run` pin to 1. The computer will start running the program, and the `halted` pin will go high when the program is done.

If you want to step through the program, you can pulse the `run` pin to advance one instruction at a time.

Pinout

#	Input	Output	Bidirectional
0	in[0]	out[0]	run
1	in[1]	out[1]	halted
2	in[2]	out[2]	set_pc
3	in[3]	out[3]	load_data
4	in[4]	out[4]	out_strobe
5	in[5]	out[5]	dbg[0]
6	in[6]	out[6]	dbg[1]
7	in[7]	out[7]	dbg[2]

VGA Screensaver with Tiny Tapeout Logo [104]

- Author: Uri Shaked
- Description: Tiny Tapeout Logo bouncing around the screen (640x480, TinyVGA Pmod)
- GitHub repository
- HDL project
- Mux address: 104
- Extra docs
- Clock: 25175000 Hz

How it works

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

How to test

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

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

Pinout

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

LIF Neuron [106]

- Author: Sohil Khan
- Description: LIF Neuron implementation with editable configuration
- GitHub repository
- HDL project
- Mux address: 106
- Extra docs
- Clock: 0 Hz

How it works

This project implements a **hardware-optimized LIF (Leaky Integrate-and-Fire) neuron** in Verilog, designed for area efficiency while maintaining biological realism. The system consists of four main components:

Core LIF Neuron Engine The heart of the system implements authentic LIF dynamics:

- **Integration**: `V_mem = V_mem + weighted_input - leak_rate`
- **Spike Generation**: When `V_mem  threshold`, generate spike and reset `V_mem = 0`
- **Refractory Period**: 4-cycle no-spike period after each action potential
- **Variable Leak Rates**: 4 different membrane leak speeds (1-4 units/cycle)

The implementation uses **8-bit arithmetic** for hardware efficiency while maintaining biological accuracy across a 0-255 membrane potential range.

Enhanced Input Processing

- **3-bit Channel Precision**: Each synaptic input (Channel A, Channel B) accepts 0-7 stimulus levels
- **Weighted Integration**: `weighted_input = (chan_a × weight_a) + (chan_b × weight_b)`
- **Synaptic Depression**: Temporary weight reduction after spike generation for realistic synaptic fatigue
- **64 Input Combinations**: Full 8×8 stimulus space for fine-grained neural control

Serial Parameter Loader A dedicated state machine loads five key parameters via single-bit serial interface:

- **Weight A/B** (3 bits each): Synaptic strength for channels A and B
- **Leak Config** (2 bits): Membrane leak rate selection (1-4 units/cycle)
- **Threshold Min/Max** (8 bits each): Adaptive threshold bounds
- **40-bit Total**: Complete parameter set loaded serially for different neuron personalities

Advanced Biological Features

- **Adaptive Thresholds**: Increase by 4 units after spikes, decrease by 1 unit during silence
- **Synaptic Depression**: Weights temporarily reduced by 3 units after spiking
- **Multiple Neuron Types**: Configurable personalities (high/low sensitivity, balanced, custom)
- **Realistic Dynamics**: Proper integration, leak, refractory periods matching biological neurons

I/O Interface

- **6-bit stimulus input**: 3-bit Channel A + 3-bit Channel B with 2 pins reserved for expansion
- **8-bit neural output**: 7-bit membrane potential + 1-bit spike detection
- **Serial configuration**: Single-bit parameter loading with status monitoring
- **Debug outputs**: Parameter ready, spike monitor, activity indicators

How to test

Basic Operation Test

1. **System Reset**: Assert `rst_n` low, then release while keeping `ena` high
2. **Apply Stimulus**: Set `ui_in[2:0]` (Channel A) and `ui_in[5:3]` (Channel B) to desired values (0-7 each)
3. **Monitor Output**: Watch `uo_out` for spikes, `uo_out[6:0]` for real-time membrane potential
4. **Expected Behavior**: With default parameters, combined stimulus 4 should eventually generate spikes

Parameter Loading Test

1. **Enter Load Mode**: Set `uio_in` (load_mode) = 1
2. **Send Parameters**: Use `uio_in` (serial_data) to clock in 40 bits (5×8 bit parameters):
   - **Weight A**: 8 bits (try 0x04 for moderate synaptic strength)
   - **Weight B**: 8 bits (try 0x03 for balanced dual-channel response)
   - **Leak Config**: 8 bits (try 0x01 for slow leak, 0x03 for fast leak)
   - **Threshold Min**: 8 bits (try 0x20=32 for low sensitivity, 0x40=64 for high)
   - **Threshold Max**: 8 bits (try 0x60=96 for moderate, 0x80=128 for wide range)
3. **Monitor Status**: Watch `uio_out` (params_ready) transition from 1→0→1
4. **Exit Load Mode**: Set `uio_in` = 0
5. **Test New Behavior**: Apply stimuli and verify different firing patterns

Neuron Configuration Testing Load these parameter sets to test different neuron behaviors:

Configuration	Weight A	Weight B	Leak	Thr Min	Thr Max	Expected Behavior
High Sensitivity	0x06	0x05	0x01	0x19	0x50	Spikes with low inputs
Low Sensitivity	0x01	0x01	0x03	0x3C	0x78	Requires high inputs
Balanced	0x04	0x04	0x02	0x1E	0x5A	Moderate responses
Fast Dynamics	0x03	0x03	0x03	0x28	0x5A	Rapid leak, brief integra

Stimulus Response Testing

- **Chan A=0, Chan B=0**: Should remain at rest (membrane potential ~0-10)
- **Chan A=1, Chan B=1**: Subthreshold integration, gradual membrane rise
- **Chan A=2, Chan B=2**: Threshold region, occasional spikes depending on configuration
- **Chan A=3, Chan B=3**: Suprathreshold, regular spike generation
- **Chan A=7, Chan B=7**: Maximum input, high-frequency firing or rapid adaptation

Advanced Feature Testing

- **Adaptive Thresholds**: Apply repeated stimuli, observe increasing inter-spike intervals

- **Synaptic Depression**: High-frequency stimulation should show reduced response over time
- **Leak Rate Effects**: Compare integration speed with different leak configurations
- **Dual Channel**: Test various A/B combinations to verify independent channel processing

Debug Monitoring

- `uio_out`: Parameter loading status (1=ready, 0=loading)
- `uio_out`: Duplicate spike output for external monitoring
- `uio_out`: Membrane activity indicator (1=active, 0=quiet)
- `uio_out[5:7]`: Echo signals for load mode, serial data, and enable verification

External hardware

No external hardware required for basic operation:

- **Stimulus Input**: Connect DIP switches or digital signals to `ui_in[5:0]` for manual channel control
- **Spike Output**: Connect LED to `uo_out` for visual spike indication
- **Membrane Monitor**: Connect 7-segment display or LED bar to `uo_out[6:0]` for membrane voltage visualization

Pinout

#	Input	Output	Bidirectional
0	CHAN_A_BIT0	V_MEM_BIT0	LOAD_MODE
1	CHAN_A_BIT1	V_MEM_BIT1	SERIAL_DATA
2	CHAN_A_BIT2	V_MEM_BIT2	PARAMS_READY
3	CHAN_B_BIT0	V_MEM_BIT3	SPIKE_MONITOR
4	CHAN_B_BIT1	V_MEM_BIT4	MEM_ACTIVITY
5	CHAN_B_BIT2	V_MEM_BIT5	LOAD_MODE_ECHO
6	RESERVED_0	V_MEM_BIT6	SERIAL_DATA_ECHO
7	RESERVED_1	SPIKE_OUT	ENABLE_STATUS

RNG [128]

- Author: Felix N
- Description: Ring oscillator based random number generator
- GitHub repository
- HDL project
- Mux address: 128
- Extra docs
- Clock: 500000 Hz

How it works

This project is a true random number generator.

The core of the TRNG is a set of three ring oscillators of different lengths (6, 12, and 24 inverters). These oscillators produce unstable, jittery signals. The outputs are combined using an XOR gate to create a chaotic bit stream. Here is the ring oscillator frequency estimates:

Ring Oscillator	Frequency Estimate	Period Estimate
6	~231 MHz	4.32 ns
12	~117 MHz	8.52 ns
24	~59 MHz	16.91 ns

The raw random bitstream may have a bias (more 1s than 0s). To correct this, a Von Neumann corrector is used. It takes pairs of bits from the stream:

- If the bits are 01, it outputs a 0.
- If the bits are 10, it outputs a 1.

If the bits are the same (00 or 11), it outputs nothing.

The debiased bits are collected one by one and shifted into a 32-bit register. Once a 32 bit number has been collected, it is output through the UART.

How to test

To test the design, you will need to monitor the UART output. Connect a UART-to-USB adapter to the uo_out[0] pin (which is the UART TX pin), the ground pin, and the power pin of your board.

Configure the serial terminal to match the UART settings: Baud Rate: 9600 Data Bits: 8 Parity: None Stop Bits: 1

Once connected, you should see a continuous stream of raw binary data appearing in your terminal. This is the 32-bit random numbers being sent from the chip.

The raw ring oscillator outputs can also be monitored on the uo_out[1], uo_out[2], and uo_out[3] pins, which correspond to the 6, 12, and 24 inverter ring oscillators respectively.

External hardware

A UART-to-USB adapter is required to connect the chip's output to a computer and view the generated random numbers.

Pinout

#	Input	Output	Bidirectional
0		Uart TX	
1		ring oscillator 6	
2		ring oscillator 12	
3		ring oscillator 24	
4			
5			
6			
7			

PQN Model with Verilog [130]

- Author: kinako71-2
- Description: ASIC implementation of a PQN model with two variations that can generate class 1 and class 2 firing patterns.
- GitHub repository
- HDL project
- Mux address: 130
- Extra docs
- Clock: 5000000 Hz

How it Works

This project is based on the PQN model [1], which is designed for the digital implementation of neuron circuits.
In particular, this work adopts a two-variation PQN model.
The parameters are configured to reproduce Class 1 and Class 2 neurons according to Hodgkin's classification [2].

Governing Equations Following [1], The neuron dynamics are defined as:

```
\frac{dv}{dt} = \frac{\phi}{\tau} \left( f(v) - n + I_0 + k I_{\text{stim
```

```
\frac{dn}{dt} = \frac{1}{\tau} \left( g(v) - n \right)
```

```
f(v) =
\begin{cases}
a_{fn}(v - b_{fn})^2 + c_{fn} & (v < 0) \\
a_{fp}(v - b_{fp})^2 + c_{fp} & (v \ge 0)
\end{cases}
```

```
g(v) =
\begin{cases}
a_{gn}(v - b_{gn})^2 + c_{gn} & (v < r_g) \\
a_{gp}(v - b_{gp})^2 + c_{gp} & (v \ge r_g)
\end{cases}
```

```
b_{fp} = \frac{a_{fn} b_{fn}}{a_{fp}}
```

```
c_{fp} = a_{fn} b_{fn}^2 + c_{fn} - a_{fp} b_{fp}^2
```

```
b_{gp} = r_g - \frac{a_{gn} (r_g - b_{gn})}{a_{gp}}
```

```
c_{gp} = a_{gn}(r_g - b_{gn})^2 + c_{gn} - a_{gp}(r_g - b_{gp})^2
```

To reduce the computational cost, each coefficient is expanded in the implementation. Here equations are expressed as follows:

```
\frac{dv}{dt} =
\begin{cases}
f_{vv_n} v^2 + f_{vv_n} v + f_{\text{const}_n} - f_{\text{coef}} n + I_{\
f_{vv_p} v^2 + f_{vv_p} p + f_{\text{const}_p} - f_{\text{coef}} n + I_{\
\end{cases}

\frac{dn}{dt} =
\begin{cases}
g_{vv_n} v^2 + g_{vv_n} v + g_{\text{const}_n} - g_{\text{coef}} n & (v <
g_{vv_p} v^2 + g_{vv_p} p + g_{\text{const}_p} - g_{\text{coef}} n & (v \
\end{cases}
```

Parameter Configuration Below are the detailed values of the parameters. For the expanded coefficients used in implementation, please refer to the module script for detailed values.

Parameter	class1	class2
dt	0.0001	0.0001
a_{fp}	-3.5	-4
a_{fn}	3.5	4
b_{fn}	-2	-2
c_{fn}	0.5	5.25
a_{gn}	-0.5	-3
a_{gp}	2.5	3
b_{gn}	-3	-2
c_{gn}	-16	-16
τ	0.0064	0.0064
I_0	-16	-16
k	8	8
ϕ	0.125	0.125
r_g	-2.5	-2.5

Module Interface The ports usage of the top module is as follows: | Pins | Bits | Direction | Description | | ——————— | — | ——— | ——————————— ——— | | `clk` | 1 | Input | Clock signal | | `rst_n` | 1 | Input | Active-low reset signal | | `uio_oe[7:0]` | 8 | Input | Set to 1 to enable outputs at all times | | `ui_in[7:1]` | 7 |

Input | Input current, converted to 16 bits | | `ui_in[0]` | 1 | Input | Mode select input | | `uo_out[7:0]`, `uio_out[7:0]` | 16 | Output | Signed 16-bit membrane voltage |

[1] Nanami, T., & Kohno, T. (2023). Piecewise quadratic neuron model: A tool for close-to-biology spiking neuronal network simulation on dedicated hardware. Frontiers in Neuroscience, 16, 1069133.c
[2] Hodgkin, A. L. (1948). The local electric changes associated with repetitive action in a non-medullated axon. The Journal of physiology, 107(2), 165.

How to Test

Simulation was originally conducted using Julia.
The given inputs and the corresponding ideal outputs are provided as text files (`ans_*` and `input_*`, where * = class1 or class2).
The test bench checks whether the circuit reproduces these results.
Please note that each output point is generated every 18 $\times$ 10 clock cycles.

External Hardware

A PCB board is sufficient.

Pinout

#	Input	Output	Bidirectional
0	Mode Select (0: Class 1, 1: Class 2)	Membrane Potential [0]	Membrane Potential [8]
1	Input Current [5]	Membrane Potential [1]	Membrane Potential [9]
2	Input Current [6]	Membrane Potential [2]	Membrane Potential [10]
3	Input Current [7]	Membrane Potential [3]	Membrane Potential [11]
4	Input Current [8]	Membrane Potential [4]	Membrane Potential [12]
5	Input Current [9]	Membrane Potential [5]	Membrane Potential [13]
6	Input Current [10]	Membrane Potential [6]	Membrane Potential [14]
7	Input Current [11]	Membrane Potential [7]	Membrane Potential [15]

VGA Screensaver with the IHP Logo [132]

- Author: Uri Shaked
- Description: IHP Logo bouncing around the screen (640x480, TinyVGA Pmod)
- GitHub repository
- HDL project
- Mux address: 132
- Extra docs
- Clock: 25175000 Hz

How it works

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

How to test

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

- `tile` (ui_in[0]) to repeat the logo and tile it across the screen,
- `solid_color` (ui_in[1]) to use a solid color instead of an animated gradient.
- `white_background` (ui_in[2]) to use a white background instead of a black one.

If you have a Gamepad Pmod connected, you can also use the following controls:

- Start button: start/pause bouncing
- Left/right/up/down: change the bouncing direction (if bouncing) or move the logo around the screen (if paused)

External hardware

- Tiny VGA Pmod
- Optional: Gamepad Pmod

Pinout

#	Input	Output	Bidirectional
0	tile	R1	
1	solid_color	G1	
2	white_bg	B1	
3		VSync	
4	gamepad_latch	R0	
5	gamepad_clk	G0	
6	gamepad_data	B0	
7		HSync	

4-Bit Adder [134]

- Author: Felix
- Description: 4-Bit Adder
- GitHub repository
- Wokwi project
- Mux address: 134
- Extra docs
- Clock: 0 Hz

How it works

This is a simple 4 Bit Adder.

How to test

Connect Buttons (or Dip Switches) to the inputs. 0 to 3 are for the first number, and 4 to 7 for the second number. Connect LEDs to the outputs.

External hardware

8 buttons and 8 leds

Pinout

#	Input	Output	Bidirectional
0	A0	LED0	
1	A1	LED1	
2	A2	LED2	
3	A3	LED3	
4	B0	LED4	
5	B1		
6	B2		
7	B3		

OCDCpro TT key lock test design IHP [136]

- Author: Johanna T. Wallenborn
- Description: key lock test chip ocdcpro
- GitHub repository
- Wokwi project
- Mux address: 136
- Extra docs
- Clock: 0 Hz

How it works

4-Input Digital Key Lock Circuit: This project implements a simple digital key lock using a combinational logic circuit with 4 input pins and 1 output pin. Only a specific combination of these inputs will unlock the circuit (output = 1). Any other combination keeps the output locked (output = 0).

How to test

The correct input number is 1001. Only this specific combination the output is set to high.

External hardware

Input: switches, output: LED

Pinout

#	Input	Output	Bidirectional
0	Pin B1	Pin A1	
1	Pin B2		
2	Pin B3		
3	Pin B4		
4			
5			
6			
7			

8-bit DDS sine wave generator [138]

- Author: Abhinav Prasad, Steven O'Shea
- Description: A simple direct digital synthesizer with 8-bit frequency control and external R-2R output
- GitHub repository
- HDL project
- Mux address: 138
- Extra docs
- Clock: 66000000 Hz

How it works

This project implements a basic 8-bit Direct Digital Synthesizer (DDS) that outputs a digitized sine wave with a user-defined frequency.

- The user provides an 8-bit phase increment (`phase_inc`) via the `ui_in[7:0]` pins
- A rising edge on `uio_in[0]` (`load_freq`) loads this value into DDS
- If no `load_freq` trigger is applied, the DDS runs at a default low frequency (corresponding to `phase_inc = 8'd1`)

Internally, a **phase accumulator** adds `phase_inc` to a phase register on each clock cycle. The 8-bit output of the accumulator directly indexes a sine look-up table (LUT) with 256 entries. Each LUT value corresponds to a digtized amplitude sample of a sine wave. These values are sent to `uo_out[7:0]`, which can be converted to an analog waveform using an external R-2R DAC.

The output frequency is given by the standard DDS formula:

f_out = (phase_inc / 2^N) * f_clk,

where,

- `phase_inc` is the user-defined tuning word (8 bits)
- `N = 8` is the size of the phase accumulator (for this simplified implementation)
- `f_clk = 66 MHz` is the system clock

This design is based on principles described in Analog Devices' application note:

> "Fundamentals of Direct Digital Synthesis (DDS)" — Analog Devices http s://www.analog.com/media/en/training-seminars/tutorials/MT-085.pdf

How to test

1. Connect an 8-bit DIP switch or microcontroller output to `ui_in[7:0]` to set the frequency
2. Pulse `uio_in[0]` high momentarily to load the new value
3. Connect `uo_out[7;0]` to an R-2R DAC to view the generated sine wave
4. You can low pass filter the DAC output to generate a smooth wave

Example:

- Phase increment = `8'b00000001` produces a wave of ~258 kHz
- Phase increment = `8'b10000000` produces a wave of ~33 MHz (Nyquist limit)

External hardware

- DIP switches or a microcontroller can be used to control the frequency (`ui_in`) and trigger (`uio_in[0]`)
- An **8-bit R-2R resistor ladder DAC** is required to convert `uo_out[7:0]` into an analog sine waveform

Pinout

#	Input	Output	Bidirectional
0	phase bit 0	sine amplitude bit 0	load_freq trigger (active high)
1	phase bit 1	sine amplitude bit 1	
2	phase bit 2	sine amplitude bit 2	
3	phase bit 3	sine amplitude bit 3	
4	phase bit 4	sine amplitude bit 4	
5	phase bit 5	sine amplitude bit 5	
6	phase bit 6	sine amplitude bit 6	
7	phase bit 7	sine amplitude bit 7	

Tapeout Test1 [160]

- Author: Marek
- Description:
- GitHub repository
- Wokwi project
- Mux address: 160
- Extra docs
- Clock: 0 Hz

How it works

Don't know

How to test

For decoration

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	in0	out0	
1	in1	out1	
2	in2	out2	
3	in3	out3	
4	in4	out4	
5	in5	out5	
6	in6	out6	
7	in7	out7	

Barans erster Template Design [162]

- Author: Baran Kabakcioglu
- Description: ein schwieriges Projekt mit der ich nicht so viel erfahrung habe
- GitHub repository
- Wokwi project
- Mux address: 162
- Extra docs
- Clock: 0 Hz

How it works

Es gibt Input und Output

How to test

Dafuer braucht man Skills

External hardware

Verwende einfach so zu sagen Elemente verbinde sie wie in Informatik aber 100-mal schwerer und hoffe das alle Hardwares miteinander in Harmonie funktionieren

Pinout

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

demo-tiny [164]

- Author: Nikhil Garg
- Description: test
- GitHub repository
- HDL project
- Mux address: 164
- Extra docs
- Clock: 0 Hz

How it works

Digital LIF neuron

How to test

Connecting to memristor

External hardware

TIA and DAC

Pinout

#	Input	Output	Bidirectional
0	IN	Dummy	MEM
1	OUT		
2			
3			
4			
5			
6			
7			

Random [166]

- Author: vans24
- Description: I have no idea.
- GitHub repository
- Wokwi project
- Mux address: 166
- Extra docs
- Clock: 0 Hz

How it works

It's very interesting how it works.

How to test

By testing.

External hardware

We shall see.

Pinout

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

noclue [168]

- Author: me_julian
- Description: boo
- GitHub repository
- Wokwi project
- Mux address: 168
- Extra docs
- Clock: 0 Hz

How it works

TBD

How to test

TBD

External hardware

TBD

Pinout

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

GG [170]

- Author: Vincent
- Description: not much
- GitHub repository
- Wokwi project
- Mux address: 170
- Extra docs
- Clock: 0 Hz

How it works

i DONT NOW

How to test

KLICK THE BOTTOM

External hardware

7 DIGTIS TEIL

Pinout

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

Tiny Tapeout Template Copy_Orion [224]

- Author: Caspar.Jakobi
- Description: Zählen, Rechnen, etc.
- GitHub repository
- Wokwi project
- Mux address: 224
- Extra docs
- Clock: 0 Hz

How it works

I cant explain

How to test

Klick on the Input board and the output we the result of the two numbers. The maximum is eight ond if you only click on inout on this nuber should be written.

External hardware

None

Pinout

#	Input	Output	Bidirectional
0	IN0_Orion	OUT0_Orion	
1	IN1_Orion	OUT1_Orion	
2	IN2_Orion	OUT2_Orion	
3	IN3_Orion	OUT3_Orion	
4	IN4_Orion	OUT4_Orion	
5	IN5_Orion	OUT5_Orion	
6	IN6_Orion	OUT6_Orion	
7	IN7_Orion	OUT7_Orion	

CRP - Custom Risc Processor [225]

- Author: David Polt
- Description: An 8-bit processor implementing a custom RISC architecture
- GitHub repository
- HDL project
- Mux address: 225
- Extra docs
- Clock: 50000000 Hz

How it works

The 8-Bit CRP CPU is a simple, custom-designed processor implemented in Verilog. It follows a classic **Von Neumann architecture**, where instructions and data share the same memory space. The CPU is based on a **multicycle design**, meaning that each instruction takes a variable number of clock cycles to complete.

The main components of the CPU include:

- **ALU (Arithmetic Logic Unit):** Performs arithmetic operations like addition and subtraction, as well as logical operations such as AND, OR, XOR, shifts (LSR, LSL/ASL, ASR), and comparison operations.
- **Registers:** 16 general-purpose 8-bit registers, with registers 14 and 15 reserved for memory addressing in `LD` and `ST` instructions (R14 = lower 8 bits, R15 = upper 7 bits). A dedicated **stack pointer** starts at `0x7FFF` and counts downward. The **program counter / instruction pointer (PC/IP)** starts at `0x0000` and increments according to the instruction.
- **Controller:** Decodes instructions and generates control signals to orchestrate data movement, ALU operations, memory access, and branching.
- **Datapath:** Connects all components and manages the flow of data between registers, ALU, and memory.
- **Multiplexers:** Select ALU and register file inputs depending on the current instruction.
- **State Counter:** Manages the instruction execution cycle, controlling fetch, decode, execute, and writeback stages.

The CPU supports **16-bit instruction width**. Instructions are categorized as R-type, I-type, and J-type:

- **R-type:** Register instructions (e.g., ADD, SUB, AND, OR). Includes a function field for the exact operation.
- **I-type:** Immediate operations, using an 8-bit immediate value (e.g., ADDI, SUBI, ANDI).

- **J-type:** Jump instructions, which are relative and may be signed or unsigned depending on the opcode.

The CPU communicates with external memory via:

- **8-bit data bus:** `DI0-DI7` for memory input, `DO0-DO7` for memory output.
- **15-bit address bus:** `A0-A14` for memory addressing.
- **Write Enable:** `WE` Signals a memory write request.

Memory write protocol:

1. **Clock cycle 1:** Place the 8-bit data to write on `DO0-DO7` and set `WE` high. The data is temporarily stored externally.
2. **Clock cycle 2:** Provide the 15-bit target address on `A0-A14`. Memory writes the previously buffered data to this address.

Instruction storage in memory:

- Each 16-bit instruction occupies **two consecutive addresses**:
  - Even addresses: lower 8 bits of the instruction
  - Odd addresses: upper 8 bits of the instruction

How to test

1. **Set up external memory**
   - Use a memory module with read/write capability and 15-bit address lines.
   - Connect CPU data input pins (`DI0-DI7`) to memory output pins.
   - Connect CPU data output pins (`DO0-DO7`) to memory input pins.
   - Connect CPU address pins (`A0-A14`) to memory address pins.
   - Connect CPU write enable pin (`WE`) to memory `WE` input via an external buffer.
2. **Load a program**
   - The complete instruction set for binary translation can be found in **Appendix A**.
   - Instructions are 16 bits wide and occupy two consecutive memory addresses:
     - Even addresses: lower 8 bits
     - Odd addresses: upper 8 bits

3. **Reset the CPU state**

    - Set `rst_n` **low** briefly while toggling the clock:
        1. `rst_n = 0`, `clk = 1`
        2. `clk = 0`
        3. `rst_n = 1`

4. **Provide a clock signal**

    - The CPU is multi-cycle; instructions take multiple clock cycles.
    - After reset, the program counter (PC/IP) begins at `0x000`.

External hardware

- **Memory module:** 8-bit data, 15-bit address, supports read/write operations.
- **Buffer register:** Controlled by `WE`, it captures the 8-bit data bus (`D00–D07`) for use in the next clock cycle.
- **Clock source:** Provides the clock signal for multicycle operation.

Appendix A: Instruction Set Overview

Mnemonic	Opcode	Operands	Description	Clock
MOV	0000	Rd, Rr, 0000	Copy from Rr to Rd	3
ADD	0000	Rd, Rr, 0001	Add Rr to Rd	3
SUB	0000	Rd, Rr, 0010	Subtract Rr from Rd	3
AND	0000	Rd, Rr, 0011	Bitwise AND	3
OR	0000	Rd, Rr, 0100	Bitwise OR	3
XOR	0000	Rd, Rr, 0101	Bitwise XOR	3
LD	0000	Rd, xxxx, 0110	Load from memory to Rd	4
ST	0000	Rd, xxxx, 0111	Store Rd to memory	4
PUSH	0000	Rd, xxxx, 1000	Push Register on Stack	4
POP	0000	Rd, xxxx, 1001	Pop Register from Stack	5
PUSHF	0000	xxxx, xxxx, 1010	Push Flags on Stack	4
POPF	0000	xxxx, xxxx, 1011	Pop Flags from Stack	4
LSR	0000	Rd, Rr, 1100	Logical Shift Right	3
LSL	0000	Rd, Rr, 1101	Logical Shift Left	3
ASR	0000	Rd, Rr, 1110	Arithmetic Shift Right	3
CMP	0000	Rd, Rr, 1111	Compare Rd with Rr	3

Mnemonic	Opcode	Operands	Description	Clock
CMPI	0001	Rd, immediate	Compare Rd with Immediate	3
ADDI	0010	Rd, immediate	Add Immediate to Rd	3
SUBI	0011	Rd, immediate	Subtract Immediate from Rd	3
ANDI	0100	Rd, immediate	Bitwise AND with Immediate	3
ORI	0101	Rd, immediate	Bitwise OR with Immediate	3
XORI	0110	Rd, immediate	Bitwise XOR with Immediate	3
MOV	0111	Rd, immediate	Load Immediate into Rd	3
RJMP	1000	address	Relative Jump	3
RET	1001	1101, 1101, xxxx	Subroutine Return[1]	7
RCALL	1010	address	Relative Subroutine Call	6
JE	1011	address	Jump If Equal	3
JNE	1100	address	Jump If Not Equal	3
JB	1101	address	Jump If Below, Unsigned	3
JAE	1110	address	Jump If Above Or Equal, Unsigned	3
JL	1111	address	Jump If Less, Signed	3

Legend

- **Rd** – Destination register (4-bit)
- **Rr** – Source register (4-bit)
- **Immediate** – 8-bit constant embedded in instruction
- **Address** – 12-bit relative address; MSB is sign-extended for overflow
- **x** – Ignored bit/operand
- **[1]RET** – Uses an internal temporary register for storing return address from stack. Here: `1101`

Pinout

#	Input	Output	Bidirectional
0	DI0	A0/DO0	A8
1	DI1	A1/DO1	A9
2	DI2	A2/DO2	A10
3	DI3	A3/DO3	A11
4	DI4	A4/DO4	A12
5	DI5	A5/DO5	A13
6	DI6	A6/DO6	A14

#	Input	Output	Bidirectional
7	DI7	A7/DO7	WE

Simple classification perceptron [226]

- Author: ChinZhe
- Description: simulation perceptron and display the result with 1 or 0
- GitHub repository
- Wokwi project
- Mux address: 226
- Extra docs
- Clock: 10000000 Hz

Perceptron with Tiny MAC

This project implements a perceptron that computes y = sign(3*x0* - *2*x1 + 1), where x0 and x1 are 4-bit signed inputs, which is ui_in[7:0]. The output is uo_out[7:0], and when the output is greater than 0, uo_out[0] is 1; otherwise, it is 0. It is given the name y_reg. The values will be saved to sum_reg which is uo_out[7:1] The design uses a sequential 4x4-bit MAC, which implements a simple FSM concept.

How it works

- Inputs are x0 (ui_in[3:0]) and x1 (ui_in[7:4]), which are multiplied by weights (3 and -2) and summed with a bias (1). Note that the weights are hard-coded, unlike a real "tunable" NN.
- The result is passed through a sign function to produce y_reg.
- The MAC operates in three cycles when ena=1, controlled by a finite state machine.

How to test

- Set ena=1 (ENA pin) –> It is automatically set to 1 when powering the chip
- Apply a reset pulse (RST_N=0 then 1). –> which we are unable to do as we don't have access to those control ports; it should be automatically set to 1.
- Set ui_in[7:0] using DIP switches (e.g., x0=1, x1=1 as 8'b00010001).
- Observe uo_out[0] (y_reg) for the result (1 if sum 0, else 0).
- Monitor uo_out[7:1] for the upper sum bits.

Pinout

#	Input	Output	Bidirectional
0	x0[0]	y_reg	
1	x0[1]	sum[1]	
2	x0[2]	sum[2]	
3	x0[3]	sum[3]	
4	x1[0]	sum[4]	
5	x1[1]	sum[5]	
6	x1[2]	sum[6]	
7	x1[3]	sum[7]	

Tiny Tapeout [228]

- Author: Lana
- Description: lol
- GitHub repository
- Wokwi project
- Mux address: 228
- Extra docs
- Clock: 0 Hz

How it works

How to test

External hardware

Pinout

#	Input	Output	Bidirectional
0	IN0	OUT0	
1	IN1	OUT1	
2	IN2	OUT2	
3	IN3	OUT3	
4	IN4	OUT4	
5	In5	OUT5	
6	IN6	OUT6	
7	IN7	OUT7	

example-verilog [230]

- Author: Daniele Parravicini
- Description: Exapmle verilog design
- GitHub repository
- HDL project
- Mux address: 230
- Extra docs
- Clock: 1000 Hz

How it works

Just an example

How to test

you will have to figure it out

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	IN0	OUT1	
1	IN1		
2			
3			
4			
5			
6			
7			

brostarscard [232]

- Author: brostar1962
- Description: tinytapeoutchip
- GitHub repository
- Wokwi project
- Mux address: 232
- Extra docs
- Clock: 0 Hz

How it works

vcdfv

How to test

fvfvvfdt

External hardware

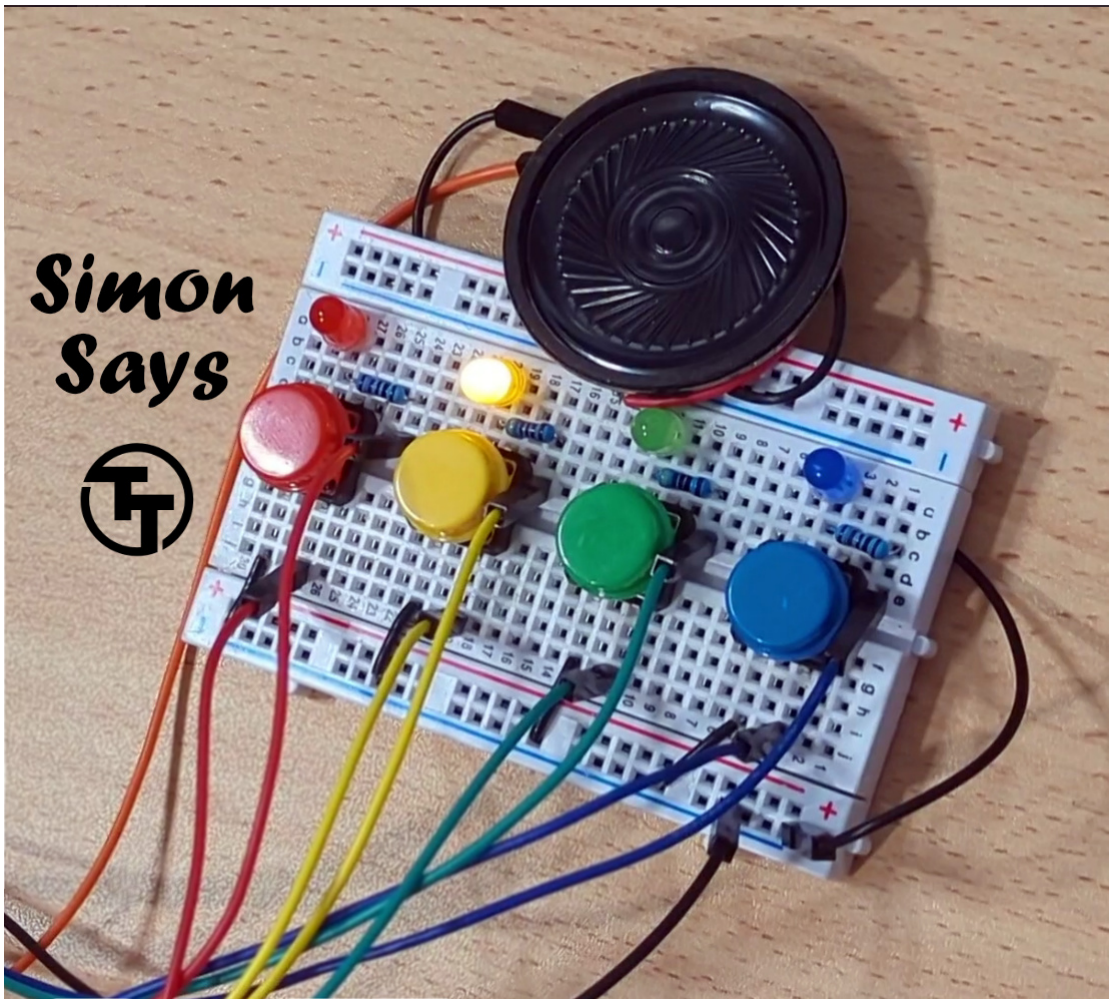
fdvdfv

Pinout

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

Simon Says memory game [234]

- Author: Uri Shaked
- Description: Repeat the sequence of colors and sounds to win the game
- GitHub repository
- HDL project
- Mux address: 234
- Extra docs
- Clock: 50000 Hz



How it works

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

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

the sequence correctly, the game will play a "leveling-up" sound, add a new color at the end of the sequence, and move to the next turn.

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

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

Clock settings

The `clk_sel` input selects the clock source:

- `0`: external 50 KHz clock, provided through the `clk` input.
- `1`: internal clock, generated by the `ring_osc` module, with a frequency of ~55 KHz.

The internal clock is generated by a 13-stage ring oscillator, divided by 16384 to get the desired frequency. The divider value was determined by running the ring oscillator simulation in <xschem/simulation/ring_osc.spice>.

When using the internal clock, its signal is also output on the `uo_out[7]` pin for debugging purposes.

How to test

Use a Simon Says Pmod to test the game.

Provide a 50 KHz clock input, reset the game, and enjoy!

If you don't have the Pmod, you can still connect the hardware manually as follows:

1. Connect the four push buttons to pins `btn1`, `btn2`, `btn3`, and `btn4`. Also connect each button to a pull down resistor.
2. Connect the LEDs to pins `led1`, `led2`, `led3`, and `led4`, matching the colors of the buttons (so `led1` and `btn1` have the same color, etc.). Don't forget current-limiting resistors!
3. Connect the speaker to the `speaker` pin (optional).
4. Connect the seven segment display as follows: `seg_a` through `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.

Pinout

#	Input	Output	Bidirectional
0	btn1	led1	seg_a
1	btn2	led2	seg_b
2	btn3	led3	seg_c
3	btn4	led4	seg_d
4	seginv	speaker	seg_e
5		dig1	seg_f
6		dig2	seg_g
7	clk_sel	clk_internal	

Timo 1 [256]

- Author: Timo
- Description: IDK
- GitHub repository
- Wokwi project
- Mux address: 256
- Extra docs
- Clock: 0 Hz

How it works

Normal just added an and and a nand

How to test

make it run

External hardware

Seven segment display

Pinout

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

Delta Sigma Comparator Based ADC [257]

- Author: Pawel Kozlowski
- Description: Evaluating the feasibility of implementing a delta-sigma analog-to-digital conversion using two switched-capacitor lines. One line serves as a reference, while the other is intended for current measurement, such as from a photodiode. The project also explores how process, voltage, and temperature (PVT) variations affect flip-flop behavior, whether these effects can be mitigated through the reference channel, and what additional insights or phenomena may emerge from this investigation.
- GitHub repository
- HDL project
- Mux address: 257
- Extra docs
- Clock: 10000000 Hz

How it works

This project explores the implementation of a delta-sigma ADC entirely in a digital environment. A flip-flop is used as the quantizer. Due to the strong DVT (device voltage threshold) dependency of flip-flops, a reference line is introduced. Assuming identical behavior across flip-flops, this helps mitigate vDVT variance.

Of course, in practice, flip-flops and capacitors are never perfectly matched. The goal is to evaluate how these mismatches affect ADC linearity, beyond the non-linearity introduced by the RC constant, and to observe the threshold behavior of the flip-flops. This is a quick, exploratory project, so some assumptions may be oversimplified or incorrect. However, in digital simulations, the concept appears to function as intended.

Implementing this on-chip (rather than on an FPGA) offers better control over parasitics at the silicon level, which could improve overall conversion accuracy.

The data output is 13 bits wide, but only 12 bits are effectively used. This is to account for potential overflow, especially in cases where simulation might not catch it. Additionally, since the system operates in bipolar mode, one more bit is reserved to represent the sign. As a result, the effective resolution of the ADC is 11 bits.

How to test

Start by defining the current range to be measured. For example, if the range is 0–100 μA, the baseline of the delta-sigma oscillation should be around 0.6 V (half of a 1.2 V supply). This gives:

R1 = Vc2 / I = 0.6 / (100e-6) = 6000 Ω

The current sink could a reverse-biased photodiode (as shown on the diagram below), but an SMU (source measure unit) can also be used to characterise the system. The circuit is designed to operate in bipolar mode, meaning it can measure both positive and negative currents.

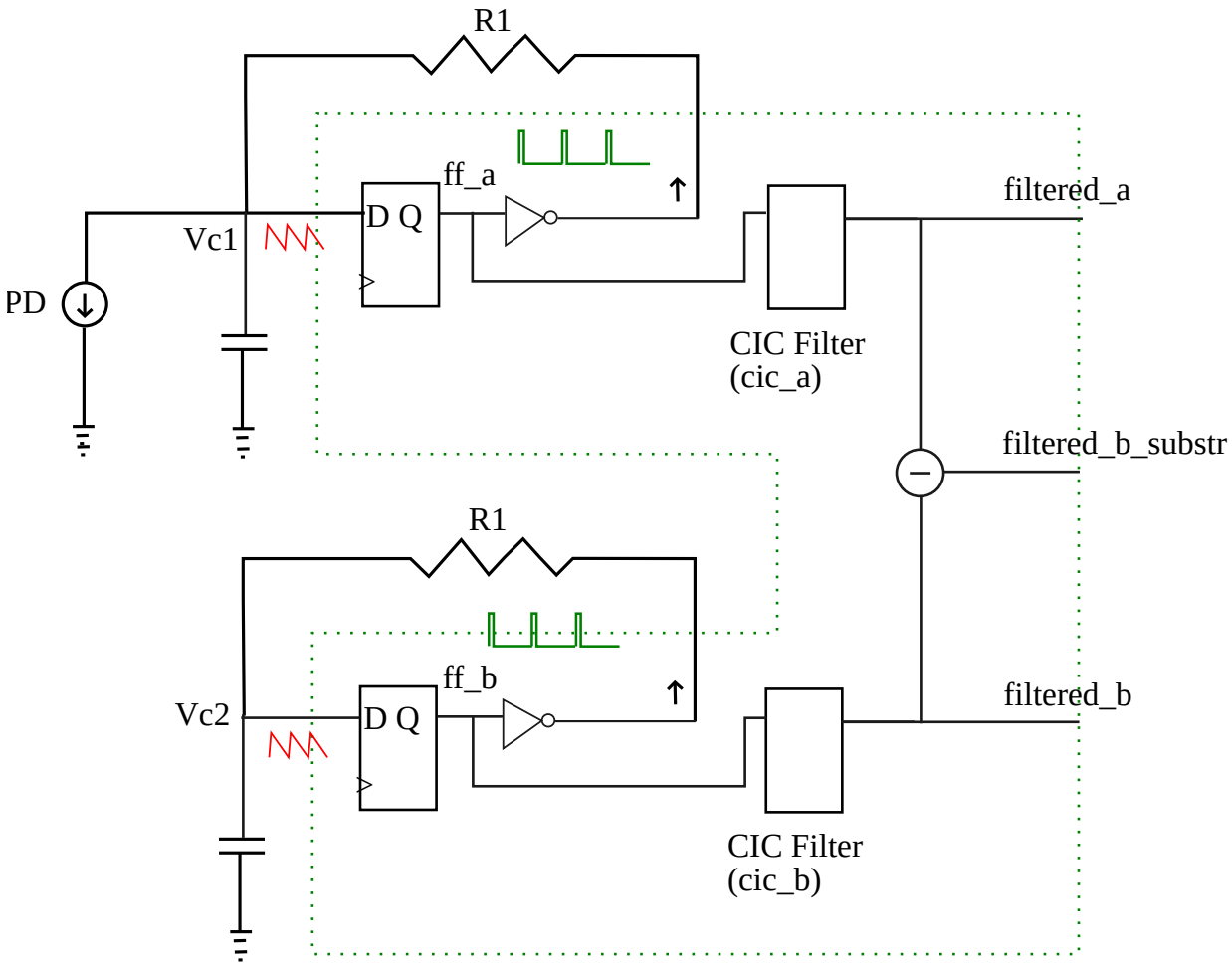
To begin operating the system, first connect the discrete components — resistors, capacitors, and a current source or sink. Once all components are connected, perform a system reset. After resetting, verify that both capacitor lines produce similar values (filtered_a and filtered_b) even without applying an external current source or sink. This confirms the system is functioning correctly. If the outputs are as expected, you can proceed to apply a current with your chosen polarity to begin active operation.

To start data acquisition, send a pulse in the clk clock domain to uio__in[7]. This will trigger the transmission of three filtered data signals in the next clock cycle: filtered_a, filtered_b, and filtered_ab_subtr. For synchronisation, a valid_out signal is sent during the transmission of the first bit of data. This simple approach was chosen to simplify the implementation.

If the internal logic for data transmission fails, components like the CIC filter can be implemented on an FPGA. This can be done by routing the inverter output not only to the capacitor but also to the FPGA. Be mindful of additional parasitics introduced in this setup.

External hardware

- **Two identical capacitors**: Test values ranging from picofarads to nanofarads. A similar analog-based design supported values from 30 pF (resulting in higher peaks between clock cycles) up to 1300 pF or more.
- **Two identical resistors**, e.g., 6000 Ω.
- **A current source/sink**



Pinout

#	Input
0	Capacitor a input port
1	Capacitor b input port
2	
3	
4	
5	
6	
7	Data triggger for adc data streaming. Asserting it causes filtered_a, filtered_b and filtered_

Tiny Tapeout Template Copy [258]

- Author: Katja Chu
- Description: This is a test
- GitHub repository
- Wokwi project
- Mux address: 258
- Extra docs
- Clock: 0 Hz

How it works

ToDo

How to test

ToDo

External hardware

ToDo

Pinout

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

AdExp DPI Neuron [259]

- Author: Saptarshi Ghosh
- Description: Adaptive Exponential Integrate-and-Fire Neuron
- GitHub repository
- HDL project
- Mux address: 259
- Extra docs
- Clock: 0 Hz

AdEx Spiking Neuron Core

This project is a digital hardware implementation of the Adaptive Exponential (AdEx) Integrate-and-Fire neuron model. It's designed to run on an ASIC, simulating the behavior of a biological neuron, including its membrane potential and adaptation mechanisms. The core is highly configurable, allowing it to model various neural firing patterns like regular spiking, bursting, and fast spiking.

How it works

The system operates based on two primary components: the **Neuron Core** and the **Parameter Loader**.

1. The Neuron Core The core solves two coupled differential equations in real-time using Q4.8 fixed-point arithmetic. These equations govern the neuron's two main state variables:

- **V:** The membrane potential, which simulates the voltage across the neuron's cell membrane.
- **w:** The adaptation current, which models cellular fatigue and is responsible for spike-frequency adaptation.

The behavior is defined by the following equations, which are a direct representation of the hardware's operation:

```
\frac{dV}{dt} = \frac{-g_L(V - E_L) + g_L\Delta_T\exp\left(\frac{V - V_T}
```

```
\frac{dw}{dt} = \frac{a(V - E_L) - w}{\tau_w}
```

When the membrane potential `V` crosses a threshold `VT`, the core outputs a digital **spike** . After a spike, `V` is reset to `Vreset` and the adaptation current `w` is increased by a value `b`.

2. The Parameter Loader The neuron's specific behavior is defined by 8 distinct user-configurable parameters. To configure the core, these parameters must be loaded serially via a simple 4-bit interface.

The parameters are loaded in the following order:

Index	Parameter	Symbol in Equation	Description
0	DeltaT	ΔT	Sharpness of the spike initiation
1	TauW	w	Adaptation time constant
2	a	a	Subthreshold adaptation level
3	b	b	Spike-triggered adaptation increment
4	Vreset	Vreset	Voltage to reset to after a spike
5	VT	VT	Firing threshold voltage
6	Ibias	I	Constant input current
7	C	C	Membrane capacitance

The loading process is controlled by the `ui_in` pins:

- `ui_in[4]` (`load_mode`): Must be high to enable loading.
- `ui_in[3]` (`load_enable`): A rising edge on this pin latches the 4-bit value present on `uio_in[3:0]`.

Each 8-bit parameter is sent as two 4-bit nibbles (high nibble first). After all **16 nibbles** have been sent, a special **footer nibble (`0xF`)** must be sent to commit the new parameters to the core.

Inputs and Outputs

- **Inputs**:
  - `clk`: Main clock signal.
  - `rst_n`: Active-low reset.
  - `ui_in[4]` (`load_mode`): Set to 1 to enable the parameter loader.
  - `ui_in[3]` (`load_enable`): Pulse high to load a 4-bit nibble from `uio_in`.
  - `ui_in[2]` (`enable_core`): Set to 1 to run the neuron simulation.
  - `ui_in[1]` (`debug_mode`): Selects the debug output on `uo_out[6:1]`.
  - `uio_in[3:0]`: 4-bit data bus for loading parameter nibbles.
- **Outputs**:
  - `uo_out[0]` (**`spike`**): The primary output. Pulses high for one clock cycle when the neuron fires.

- `uo_out[6:1]`: A 6-bit debug bus showing the most significant bits of either `V` (if `debug_mode=0`) or `w` (if `debug_mode=1`).

Firing Modes and How to Trigger Them

The AdEx model's strength is its ability to reproduce different neural behaviors. The firing pattern is primarily determined by the interplay between the adaptation parameters (`a`, `b`, `w`), input current (`I`), and membrane capacitance (`C`). By loading different parameter sets, you can make the neuron behave in specific ways.

Note: The 8-bit encoded value is what you need to send to the hardware. For signed values (`Vreset`, `VT`, `Ibias`), the encoding is `Real Value + 128`. For unsigned values, the encoding is just the `Real Value`. For the firing mode examples below, the `C` parameter should be loaded with its default value of 200 (Hex 0xC8).

Regular Spiking (Adapting) This is the "default" behavior for many excitatory neurons. The firing rate is initially high and then slows down as the adaptation current `w` builds up.

- **Mechanism**: A non-zero spike-triggered adaptation (`b`) increases `w` with every spike, making it harder for the neuron to reach its firing threshold again.
- **Parameter Values**: | Parameter | Real-World Value | 8-bit Encoded Value | Hex Value | | :— | :— | :— | :— | | `a` | 2 nS | 2 | `0x02` | | `b` | 40 pA | 40 | `0x28` | | `Vreset` | -65 mV | 63 | `0x3F` | | `Ibias` | 50 pA | 178 | `0xB2` |

Bursting This behavior is characterized by clusters of high-frequency spikes separated by periods of silence (hyperpolarization).

- **Mechanism**: Strong subthreshold adaptation (`a`) and a less-negative reset potential (`Vreset`) are key. The adaptation current `w` builds up slowly, eventually stopping the burst. As `w` decays, the membrane potential depolarizes again, initiating the next burst.
- **Parameter Values**: | Parameter | Real-World Value | 8-bit Encoded Value | Hex Value | | :— | :— | :— | :— | | `a` | 4 nS | 4 | `0x04` | | `b` | 0 pA | 0 | `0x00` | | `Vreset` | -50 mV | 78 | `0x4E` | | `Ibias` | 25 pA | 153 | `0x99` |

Fast Spiking Typical of inhibitory interneurons, this mode involves sustained high-frequency firing with little to no adaptation or slowdown.

- **Mechanism**: This is achieved by simply turning off all adaptation mechanisms (`a` and `b` are zero). The neuron behaves like a simple leaky integrate-and-fire model, with its firing rate determined solely by the input current.
- **Parameter Values**: | Parameter | Real-World Value | 8-bit Encoded Value | Hex Value | | :— | :— | :— | :— | | `a` | 0 nS | 0 | `0x00` | | `b` | 0 pA | 0 | `0x00` | | `Vreset` | -65 mV | 63 | `0x3F` | | `Ibias` | 80 pA | 208 | `0xD0` |

How to test

The recommended test procedure verifies the core's functionality by loading parameters to induce spiking and then observing the output.

The test procedure is as follows:

1. **Reset**: The chip is held in reset for 10 clock cycles to initialize all internal states.
2. **Load Parameters**: To provoke a spike, the test injects a strong, constant positive input current (`Ibias`) and sets the membrane capacitance (`C`).
    - The test enters `load_mode`.
    - It sends 12 dummy nibbles for the first 6 parameters.
    - It sends the two nibbles for `Ibias` (a value of 200, which is a strong supra-threshold current).
    - It sends the two nibbles for `C` (a value of 200, the default).
    - It sends the `0xF` footer nibble to commit all 8 parameters.
    - The test exits `load_mode`.
3. **Run and Verify**:
    - The test asserts `enable_core` to start the neuron simulation.
    - It then monitors the `uo_out[0]` (spike) pin on every clock cycle.
    - A successful test requires a spike to be detected within a set time limit (e.g., 1000 cycles).

External hardware

N/A. This project is a self-contained digital core and requires no external components.

Pinout

#	Input	Output	Bidirectional
0		spike	param_nibble_in[0]
1	debug_mode	debug_val[0]	param_nibble_in[1]
2	enable_core	debug_val[1]	param_nibble_in[2]
3	load_enable	debug_val[2]	param_nibble_in[3]
4	load_mode	debug_val[3]	
5		debug_val[4]	
6		debug_val[5]	
7			

dummy [260]

- Author: Joachim
- Description: cool stuff
- GitHub repository
- Wokwi project
- Mux address: 260
- Extra docs
- Clock: 0 Hz

How it works

This is the coolest chip ever, I just wish I knew what it did!

How to test

Hook it up to power to get started.

External hardware

You probablz want to hook it up to blinky lights, so you'll need some of those for sure.

Pinout

#	Input	Output	Bidirectional
0	I0	O0	IO0
1	I1	O1	IO1
2	I2	O2	IO2
3	I3	O3	IO3
4	I4	O4	IO4
5	I5	O5	IO5
6	I6	O6	IO6
7	I7	O7	IO7

Tapeout try [262]

- Author: Vinnie
- Description: default
- GitHub repository
- Wokwi project
- Mux address: 262
- Extra docs
- Clock: 0 Hz

How it works

i don't know either

How to test

ask chatgpt

External hardware

Tinytapeout

Pinout

#	Input	Output	Bidirectional
0	in0	out0	
1	in1	out1	
2	in2	out2	
3	in3	out3	
4	in4	out4	
5	in5	out5	
6	in6	out6	
7	in7	out7	

Chip design from Wokwi [264]

- Author: Gauransh
- Description: The project is about modelling chip designs
- GitHub repository
- Wokwi project
- Mux address: 264
- Extra docs
- Clock: 0 Hz

How it works

The project will work by modelling a chip design

How to test

The project will be used to see a model of the chip design in 3D

External hardware

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

Pinout

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

Encoder [266]

- Author: Skandha
- Description: 8*3 encoder
- GitHub repository
- Wokwi project
- Mux address: 266
- Extra docs
- Clock: 0 Hz

How it works

Just need to change that

How to test

Just need to change that for some reason

External hardware

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

Pinout

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

Nils Tinytapeout Proj [289]

- Author: Nils
- Description: I dont know
- GitHub repository
- Wokwi project
- Mux address: 289
- Extra docs
- Clock: 0 Hz

How it works

This is how it works.

How to test

This is how to use.

External hardware

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

Pinout

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

Projekt [291]

- Author: Timot
- Description: Programm
- GitHub repository
- Wokwi project
- Mux address: 291
- Extra docs
- Clock: 0 Hz

How it works

Dings

How to test

This is how to test

External hardware

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

Pinout

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

4 bit incrementer [293]

- Author: Ren
- Description: 4bit incrementer
- GitHub repository
- Wokwi project
- Mux address: 293
- Extra docs
- Clock: 0 Hz

How it works

4bit incrementer

How to test

just test duh

External hardware

VCC

Pinout

#	Input	Output	Bidirectional
0	IN0	OUT0	
1	IN1	OUT1	
2	IN2	OUT2	
3	IN3	OUT3	
4		OUT4	
5			
6			
7			

Tiny Tapeout Workshop Project by Nick Figner [295]

- Author: Nick Figner
- Description: Sth I made in the Tiny Tapeout Workshop @ ETH
- GitHub repository
- Wokwi project
- Mux address: 295
- Extra docs
- Clock: 0 Hz

How it works

Idk if it even works.....

How to test

Add cables and stuff probably

External hardware

- a seven digit display-thingy
- cables????

Pinout

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

Tiny Tapeout Chip [297]

- Author: Jasen Kouchev
- Description: IDK
- GitHub repository
- Wokwi project
- Mux address: 297
- Extra docs
- Clock: 0 Hz

How it works

Every secins input switch needs to be turned off, to work (Inverted)

How to test

Move the Input switches.

External hardware

Seven Digit Display

Pinout

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

Atari 2600 [298]

- Author: Renaldas Zioma
- Description: Replica of Atari 2600
- GitHub repository
- HDL project
- Mux address: 298
- Extra docs
- Clock: 25175000 Hz

How it works

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

How to test

Plug and play!

External hardware

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

Pinout

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

And Gate [299]

- Author: Tilman Kuttler
- Description: Not much honestly
- GitHub repository
- Wokwi project
- Mux address: 299
- Extra docs
- Clock: 0 Hz

How it works

My project uses an AND gate in order to display red lines on a seven segment display.

How to test

The project is used by switching on or off certain inputs.

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	IN0	OUT0	
1	IN1	OUT1	
2	IN3	OUT2	
3	IN4	OUT3	
4	IN5	OUT4	
5	IN6	OUT5	
6	IN7	OUT6	
7	IN8	OUT7	

LGN hand-written digit classifier (MNIST, 16x16 pixels) [326]

- Author: Renaldas Zioma, Jogundas Armaitis
- Description: Experiment with Deep Differential Logic Networks
- GitHub repository
- HDL project
- Mux address: 326
- Extra docs
- Clock: 1000000 Hz

How it works

Experiment with Deep Differential Logic Networks. Classifier network.

How to test

TO DO: Provide inputs. Read out the classifier results.

External hardware

MCU to provide input data and read out the response

Pinout

#	Input	Output	Bidirectional
0	Shift-in 8-bit input packet (LSB)	Digit (LSB)	(out) Confidence score (LSB)
1	Shift-in 8-bit input packet	Digit	(out) Confidence score
2	Shift-in 8-bit input packet	Digit	(out) Confidence score
3	Shift-in 8-bit input packet	Digit (MSB)	(out) Confidence score
4	Shift-in 8-bit input packet		(out) Confidence score
5	Shift-in 8-bit input packet		(out) Confidence score
6	Shift-in 8-bit input packet		(out) Confidence score (MSB)
7	Shift-in 8-bit input packet (MSB)		(in) /WE - Pause input

DUMBRV [330]

- Author: Yuanda Liu
- Description: A simple RISC-V processor operating on two SPI memory.
- GitHub repository
- HDL project
- Mux address: 330
- Extra docs
- Clock: 40 Hz

How it works

This is a small RISC-V processor supporting the base RV32E instruction set and the Zicond and Zbs instructions. This processor uses two SPI memories in two independent SPI ports, one is read-only and another is read-write. **Both SPI memory needs to use exactly 2-byte addresses.**

The address space is as below:

Start	End	Description
0x00000000	0x0000FFFF	Read only SPI port using `uio[3:0]`
0x01000000	0x0100FFFF	Read-write SPI port using `uio[3:0]`
0x02000000	0x02000000	One byte. `ui[7:0]` on read, `uo[7:0]` on write.

The processor begins execution immediately from address 0x0.

The SPI clock is half of that of the main clock. If you can only do 20MHz on the SPI bus, for example, put the design in 40MHz.

How to test

Hook a SPI flash or EEPROM on the `inst_spi` bus, and hook a SPI RAM on the `data_spi` bus. Program this ROM with instructions.

I have a test program in the `/program` directory of the repo. This program continuously records the input GPIO pins and outputs the recorded sequence on the output GPIO pins after some delay. If this program functions correctly, then it implies that the processor can use both SPI memories correctly.

External hardware

This processor requires one 512Kbit ROM and one 512Kbit RAM.

Pinout

#	Input	Output	Bidirectional
0	gpio_out[0]	gpio_in[0]	inst_spi_cs
1	gpio_out[1]	gpio_in[1]	inst_spi_mosi
2	gpio_out[2]	gpio_in[2]	inst_spi_miso
3	gpio_out[3]	gpio_in[3]	inst_spi_sck
4	gpio_out[4]	gpio_in[4]	data_spi_cs
5	gpio_out[5]	gpio_in[5]	data_spi_mosi
6	gpio_out[6]	gpio_in[6]	data_spi_miso
7	gpio_out[7]	gpio_in[7]	data_spi_sck

Simple LIF Neuron [417]

- Author: Klemens Bauer
- Description: LIF Neuron implementation with editable configuration
- GitHub repository
- HDL project
- Mux address: 417
- Extra docs
- Clock: 0 Hz

How it works

This project implements a **hardware-optimized LIF (Leaky Integrate-and-Fire) neuron** in Verilog, designed for area efficiency while maintaining biological realism. The system consists of four main components:

Core LIF Neuron Engine The heart of the system implements authentic LIF dynamics:

- **Integration**: `V_mem = V_mem + weighted_input - leak_rate`
- **Spike Generation**: When `V_mem  threshold`, generate spike and reset `V_mem = 0`
- **Refractory Period**: 4-cycle no-spike period after each action potential
- **Variable Leak Rates**: 4 different membrane leak speeds (1-4 units/cycle)

The implementation uses **8-bit arithmetic** for hardware efficiency while maintaining biological accuracy across a 0-255 membrane potential range.

Enhanced Input Processing

- **3-bit Channel Precision**: Each synaptic input (Channel A, Channel B) accepts 0-7 stimulus levels
- **Weighted Integration**: `weighted_input = (chan_a × weight_a) + (chan_b × weight_b)`
- **Synaptic Depression**: Temporary weight reduction after spike generation for realistic synaptic fatigue
- **64 Input Combinations**: Full 8×8 stimulus space for fine-grained neural control

Serial Parameter Loader A dedicated state machine loads five key parameters via single-bit serial interface:

- **Weight A/B** (3 bits each): Synaptic strength for channels A and B
- **Leak Config** (2 bits): Membrane leak rate selection (1-4 units/cycle)
- **Threshold Min/Max** (8 bits each): Adaptive threshold bounds
- **40-bit Total**: Complete parameter set loaded serially for different neuron personalities

Advanced Biological Features

- **Adaptive Thresholds**: Increase by 4 units after spikes, decrease by 1 unit during silence
- **Synaptic Depression**: Weights temporarily reduced by 3 units after spiking
- **Multiple Neuron Types**: Configurable personalities (high/low sensitivity, balanced, custom)
- **Realistic Dynamics**: Proper integration, leak, refractory periods matching biological neurons

I/O Interface

- **6-bit stimulus input**: 3-bit Channel A + 3-bit Channel B with 2 pins reserved for expansion
- **8-bit neural output**: 7-bit membrane potential + 1-bit spike detection
- **Serial configuration**: Single-bit parameter loading with status monitoring
- **Debug outputs**: Parameter ready, spike monitor, activity indicators

How to test

Basic Operation Test

1. **System Reset**: Assert `rst_n` low, then release while keeping `ena` high
2. **Apply Stimulus**: Set `ui_in[2:0]` (Channel A) and `ui_in[5:3]` (Channel B) to desired values (0-7 each)
3. **Monitor Output**: Watch `uo_out` for spikes, `uo_out[6:0]` for real-time membrane potential
4. **Expected Behavior**: With default parameters, combined stimulus 4 should eventually generate spikes

Parameter Loading Test

1. **Enter Load Mode**: Set `uio_in` (load_mode) = 1
2. **Send Parameters**: Use `uio_in` (serial_data) to clock in 40 bits (5×8 bit parameters):
   - **Weight A**: 8 bits (try 0x04 for moderate synaptic strength)
   - **Weight B**: 8 bits (try 0x03 for balanced dual-channel response)
   - **Leak Config**: 8 bits (try 0x01 for slow leak, 0x03 for fast leak)
   - **Threshold Min**: 8 bits (try 0x20=32 for low sensitivity, 0x40=64 for high)
   - **Threshold Max**: 8 bits (try 0x60=96 for moderate, 0x80=128 for wide range)
3. **Monitor Status**: Watch `uio_out` (params_ready) transition from 1→0→1
4. **Exit Load Mode**: Set `uio_in` = 0
5. **Test New Behavior**: Apply stimuli and verify different firing patterns

Neuron Configuration Testing Load these parameter sets to test different neuron behaviors:

Configuration	Weight A	Weight B	Leak	Thr Min	Thr Max	Expected Behavior
High Sensitivity	0x06	0x05	0x01	0x19	0x50	Spikes with low inputs
Low Sensitivity	0x01	0x01	0x03	0x3C	0x78	Requires high inputs
Balanced	0x04	0x04	0x02	0x1E	0x5A	Moderate responses
Fast Dynamics	0x03	0x03	0x03	0x28	0x5A	Rapid leak, brief integra

Stimulus Response Testing

- **Chan A=0, Chan B=0**: Should remain at rest (membrane potential ~0-10)
- **Chan A=1, Chan B=1**: Subthreshold integration, gradual membrane rise
- **Chan A=2, Chan B=2**: Threshold region, occasional spikes depending on configuration
- **Chan A=3, Chan B=3**: Suprathreshold, regular spike generation
- **Chan A=7, Chan B=7**: Maximum input, high-frequency firing or rapid adaptation

Advanced Feature Testing

- **Adaptive Thresholds**: Apply repeated stimuli, observe increasing inter-spike intervals

- **Synaptic Depression**: High-frequency stimulation should show reduced response over time
- **Leak Rate Effects**: Compare integration speed with different leak configurations
- **Dual Channel**: Test various A/B combinations to verify independent channel processing

Debug Monitoring

- `uio_out`: Parameter loading status (1=ready, 0=loading)
- `uio_out`: Duplicate spike output for external monitoring
- `uio_out`: Membrane activity indicator (1=active, 0=quiet)
- `uio_out[5:7]`: Echo signals for load mode, serial data, and enable verification

External hardware

No external hardware required for basic operation:

- **Stimulus Input**: Connect DIP switches or digital signals to `ui_in[5:0]` for manual channel control
- **Spike Output**: Connect LED to `uo_out` for visual spike indication
- **Membrane Monitor**: Connect 7-segment display or LED bar to `uo_out[6:0]` for membrane voltage visualization

Pinout

#	Input	Output	Bidirectional
0	CHAN_A_BIT0	V_MEM_BIT0	LOAD_MODE
1	CHAN_A_BIT1	V_MEM_BIT1	SERIAL_DATA
2	CHAN_A_BIT2	V_MEM_BIT2	PARAMS_READY
3	CHAN_B_BIT0	V_MEM_BIT3	SPIKE_MONITOR
4	CHAN_B_BIT1	V_MEM_BIT4	MEM_ACTIVITY
5	CHAN_B_BIT2	V_MEM_BIT5	LOAD_MODE_ECHO
6	RESERVED_0	V_MEM_BIT6	SERIAL_DATA_ECHO
7	RESERVED_1	SPIKE_OUT	ENABLE_STATUS

test_design [419]

- Author: marc
- Description: test
- GitHub repository
- HDL project
- Mux address: 419
- Extra docs
- Clock: 0 Hz

How it works

it just works

How to test

TBD

External hardware

TBD

Pinout

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

RISC-V Mini IHP [421]

- Author: Thomas Uehlinger
- Description: RISC-V Mini 8 Bit on IHP
- GitHub repository
- HDL project
- Mux address: 421
- Extra docs
- Clock: 100000 Hz

How it works

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

Processor Components

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

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

How to test

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

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

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

Instructions List

R-Type

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

I-Type

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

L-Type

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

S-Type

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

B-Type

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

External hardware

No external hardware.

Pinout

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

LIF Neuron [423]

- Author: Alexandre Baigol
- Description: LIF Neuron implementation with editable configuration
- GitHub repository
- HDL project
- Mux address: 423
- Extra docs
- Clock: 0 Hz

How it works

This project implements a **hardware-optimized LIF (Leaky Integrate-and-Fire) neuron** in Verilog, designed for area efficiency while maintaining biological realism. The system consists of four main components:

Core LIF Neuron Engine The heart of the system implements authentic LIF dynamics:

- **Integration**: `V_mem = V_mem + weighted_input - leak_rate`
- **Spike Generation**: When `V_mem  threshold`, generate spike and reset `V_mem = 0`
- **Refractory Period**: 4-cycle no-spike period after each action potential
- **Variable Leak Rates**: 4 different membrane leak speeds (1-4 units/cycle)

The implementation uses **8-bit arithmetic** for hardware efficiency while maintaining biological accuracy across a 0-255 membrane potential range.

Enhanced Input Processing

- **3-bit Channel Precision**: Each synaptic input (Channel A, Channel B) accepts 0-7 stimulus levels
- **Weighted Integration**: `weighted_input = (chan_a × weight_a) + (chan_b × weight_b)`
- **Synaptic Depression**: Temporary weight reduction after spike generation for realistic synaptic fatigue
- **64 Input Combinations**: Full 8×8 stimulus space for fine-grained neural control

Serial Parameter Loader A dedicated state machine loads five key parameters via single-bit serial interface:

- **Weight A/B** (3 bits each): Synaptic strength for channels A and B
- **Leak Config** (2 bits): Membrane leak rate selection (1-4 units/cycle)
- **Threshold Min/Max** (8 bits each): Adaptive threshold bounds
- **40-bit Total**: Complete parameter set loaded serially for different neuron personalities

Advanced Biological Features

- **Adaptive Thresholds**: Increase by 4 units after spikes, decrease by 1 unit during silence
- **Synaptic Depression**: Weights temporarily reduced by 3 units after spiking
- **Multiple Neuron Types**: Configurable personalities (high/low sensitivity, balanced, custom)
- **Realistic Dynamics**: Proper integration, leak, refractory periods matching biological neurons

I/O Interface

- **6-bit stimulus input**: 3-bit Channel A + 3-bit Channel B with 2 pins reserved for expansion
- **8-bit neural output**: 7-bit membrane potential + 1-bit spike detection
- **Serial configuration**: Single-bit parameter loading with status monitoring
- **Debug outputs**: Parameter ready, spike monitor, activity indicators

How to test

Basic Operation Test

1. **System Reset**: Assert `rst_n` low, then release while keeping `ena` high
2. **Apply Stimulus**: Set `ui_in[2:0]` (Channel A) and `ui_in[5:3]` (Channel B) to desired values (0-7 each)
3. **Monitor Output**: Watch `uo_out` for spikes, `uo_out[6:0]` for real-time membrane potential
4. **Expected Behavior**: With default parameters, combined stimulus 4 should eventually generate spikes

Parameter Loading Test

1. **Enter Load Mode**: Set `uio_in` (load_mode) = 1
2. **Send Parameters**: Use `uio_in` (serial_data) to clock in 40 bits (5×8 bit parameters):
   - **Weight A**: 8 bits (try 0x04 for moderate synaptic strength)
   - **Weight B**: 8 bits (try 0x03 for balanced dual-channel response)
   - **Leak Config**: 8 bits (try 0x01 for slow leak, 0x03 for fast leak)
   - **Threshold Min**: 8 bits (try 0x20=32 for low sensitivity, 0x40=64 for high)
   - **Threshold Max**: 8 bits (try 0x60=96 for moderate, 0x80=128 for wide range)
3. **Monitor Status**: Watch `uio_out` (params_ready) transition from 1→0→1
4. **Exit Load Mode**: Set `uio_in` = 0
5. **Test New Behavior**: Apply stimuli and verify different firing patterns

Neuron Configuration Testing Load these parameter sets to test different neuron behaviors:

Configuration	Weight A	Weight B	Leak	Thr Min	Thr Max	Expected Behavior
High Sensitivity	0x06	0x05	0x01	0x19	0x50	Spikes with low inputs
Low Sensitivity	0x01	0x01	0x03	0x3C	0x78	Requires high inputs
Balanced	0x04	0x04	0x02	0x1E	0x5A	Moderate responses
Fast Dynamics	0x03	0x03	0x03	0x28	0x5A	Rapid leak, brief integra

Stimulus Response Testing

- **Chan A=0, Chan B=0**: Should remain at rest (membrane potential ~0-10)
- **Chan A=1, Chan B=1**: Subthreshold integration, gradual membrane rise
- **Chan A=2, Chan B=2**: Threshold region, occasional spikes depending on configuration
- **Chan A=3, Chan B=3**: Suprathreshold, regular spike generation
- **Chan A=7, Chan B=7**: Maximum input, high-frequency firing or rapid adaptation

Advanced Feature Testing

- **Adaptive Thresholds**: Apply repeated stimuli, observe increasing inter-spike intervals

- **Synaptic Depression**: High-frequency stimulation should show reduced response over time
- **Leak Rate Effects**: Compare integration speed with different leak configurations
- **Dual Channel**: Test various A/B combinations to verify independent channel processing

Debug Monitoring

- `uio_out`: Parameter loading status (1=ready, 0=loading)
- `uio_out`: Duplicate spike output for external monitoring
- `uio_out`: Membrane activity indicator (1=active, 0=quiet)
- `uio_out[5:7]`: Echo signals for load mode, serial data, and enable verification

External hardware

No external hardware required for basic operation:

- **Stimulus Input**: Connect DIP switches or digital signals to `ui_in[5:0]` for manual channel control
- **Spike Output**: Connect LED to `uo_out` for visual spike indication
- **Membrane Monitor**: Connect 7-segment display or LED bar to `uo_out[6:0]` for membrane voltage visualization

Pinout

#	Input	Output	Bidirectional
0	CHAN_A_BIT0	V_MEM_BIT0	LOAD_MODE
1	CHAN_A_BIT1	V_MEM_BIT1	SERIAL_DATA
2	CHAN_A_BIT2	V_MEM_BIT2	PARAMS_READY
3	CHAN_B_BIT0	V_MEM_BIT3	SPIKE_MONITOR
4	CHAN_B_BIT1	V_MEM_BIT4	MEM_ACTIVITY
5	CHAN_B_BIT2	V_MEM_BIT5	LOAD_MODE_ECHO
6	RESERVED_0	V_MEM_BIT6	SERIAL_DATA_ECHO
7	RESERVED_1	SPIKE_OUT	ENABLE_STATUS

Morse Code Trainer [425]

- Author: Angelo Nujic
- Description: Interactive Morse Code learning game with 7-segment display
- GitHub repository
- HDL project
- Mux address: 425
- Extra docs
- Clock: 100 Hz

Morse Code Trainer

An interactive educational game that teaches Morse code through hands-on practice with a 7-segment display and tactile input.

How it works

This Morse Code trainer presents letters on a 7-segment display and challenges users to input the correct Morse code pattern using a switch. The system provides immediate feedback and progresses through the english alphabet.

Game Flow

1. **Character Display**: When Start switch (sw0) is moved a letter is shown on the 7-segment display
2. **Input Phase**: User inputs Morse code using the switch 1
    - Short hold (~200ms) = Dot (.)
    - Long hold (~400ms) = Dash (-)
3. **Validation**: System checks input against expected pattern
4. **Feedback**: Shows . for correct, no dot for wrong
5. **Progress**: Move Start switch back, and into start position to start again.

How to test The design can be tested in simulation or on hardware:

Simulation Testing

1. Run the provided cocotb testbench
2. Observe state transitions and timing behavior
3. Verify correct morse pattern recognition
4. Test button debouncing and edge cases

Hardware Testing

1. Connect 7-segment display to `uo[6:0]`
2. Connect status LED to `uo[7]` (else dot is used on 7-seg display)
3. Connect telegraph key to `ui[0]`
4. Connect navigation buttons to `ui[1]`
5. Power on and follow the learning sequence

External hardware

Required Components

- **7-Segment Display**: Common cathode, connected to `uo[6:0]`
- **Status LED**: Connected to `uo[7]` with current limiting resistor
- **Start**: Momentary switch connected to `ui[0]`
- **Telegraph Key**: Momentary switch connected to `ui[1]`

Optional Enhancements

- **Buzzer**: For audio feedback (requires additional output pin)
- **Pull-up Resistors**: For reliable button operation (10kΩ recommended)
- **LED Indicators**: Additional status LEDs for game state visualization

Pin Configuration

```
ui[0] - Start switch
ui[1] - Morse Key Input (active high)
uo[6:0] - 7-Segment Display (A-G segments)
uo[7] - Status LED (correct/incorrect feedback)
```

This Morse Code Trainer combines historical significance with modern digital design, creating an engaging educational tool perfect for ham radio enthusiasts, educators, and anyone interested in classic communication methods!

Pinout

#	Input	Output	Bidirectional
0	start	seg_o[0]	
1	morse	seg_o[1]	
2		seg_o[2]	
3		seg_o[3]	
4		seg_o[4]	
5		seg_o[5]	
6		seg_o[6]	
7		correct_res	

Gamepad Pmod Demo [427]

- Author: Uri Shaked
- Description: Gamepad Pmod + Tiny VGA demo from VGA Playground
- GitHub repository
- HDL project
- Mux address: 427
- Extra docs
- Clock: 25175000 Hz

How it works

This project demonstrates how to use the Gamepad Pmod to get input from a gamepad and display it on a VGA monitor.

How to test

Connect the TinyVGA and Gamepad Pmods to the Tiny Tapeout board, activate the project, reset it, and start pressing buttons on the gamepad.

When you press a button on the gamepad, its corresponding symbol will appear in green on the VGA display.

External hardware

- TinyVGA Pmod
- Gamepad Pmod

Pinout

#	Input	Output	Bidirectional
0		R1	
1		G1	
2		B1	
3		VSync	
4	gamepad_latch	R0	
5	gamepad_clk	G0	
6	gamepad_data	B0	

tinytapeoutchip [449]

- Author: elena
- Description: chip stuff
- GitHub repository
- Wokwi project
- Mux address: 449
- Extra docs
- Clock: 0 Hz

How it works

If you turn on the correct switches, it will turn on a light and it show the letter E.

How to test

Switch on the correct switches.

External hardware

Most likely none, I think.

Pinout

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

tiny tapeout chip [451]

- Author: Mayra
- Description: lol
- GitHub repository
- Wokwi project
- Mux address: 451
- Extra docs
- Clock: 0 Hz

How it works

Chip

How to test

Chip

External hardware

Pinout

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

ToDo [453]

- Author: Alexander Flasby
- Description: Nothing yet
- GitHub repository
- HDL project
- Mux address: 453
- Extra docs
- Clock: 0 Hz

How it works

It does not. Do not use this under any conditions.

How to test

Please don't test anything. You will be disappointed.

External hardware

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

Pinout

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

in progress [455]

- Author: mue
- Description: tiny chip design...
- GitHub repository
- Wokwi project
- Mux address: 455
- Extra docs
- Clock: 0 Hz

How it works

Does something....

How to test

This is how you would test it: ...

External hardware

external HW will be described later...

Pinout

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

numbers [457]

- Author: Majla Uehla
- Description: nothing so far
- GitHub repository
- Wokwi project
- Mux address: 457
- Extra docs
- Clock: 0 Hz

How it works

i do not know yet

How to test

i do not know yet

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	no.1	top	1
1	no.2	topr	2
2	no.3	botr	3
3	no.4	bot	4
4	no.5	botl	5
5	no.6	topl	6
6	no.7	mid	7
7	no.8	dot	8

number display [459]

- Author: Lena Stadel
- Description: displays the number that is on (in the input)
- GitHub repository
- Wokwi project
- Mux address: 459
- Extra docs
- Clock: 0 Hz

How it works

not finished yet

How to test

enter a number (a input) and it should show that nuber on the display

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	no1	top	1
1	no2	topr	2
2	no3	botr	3
3	no4	bot	4
4	no5	botl	5
5	no6	topl	6
6	no7	middle	7
7	no8	dot	8

VGA Screensaver with Zero to ASIC Logo [480]

- Author: Matt Venn
- Description: Zero to ASIC Logo bouncing around the screen (640x480, TinyVGA Pmod)
- GitHub repository
- HDL project
- Mux address: 480
- Extra docs
- Clock: 25175000 Hz

How it works

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

How to test

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

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

Pinout

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

weaving in silicon #1 [481]

- Author: bleeptrack
- Description: Imagine the norns would weave in silicon
- GitHub repository
- HDL project
- Mux address: 481
- Extra docs
- Clock: 0 Hz

How it works

This tile is an experimental silicon art tile. It plays with the idea of crossing traditional crafts like weaving with the weaving-like work of entangled transistor lines.

How to test

No test - just looks

External hardware

The artwork will be unveiled when decapping the chip

Pinout

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

weaving in silicon #2 [483]

- Author: bleeptrack
- Description: the randomness in life
- GitHub repository
- HDL project
- Mux address: 483
- Extra docs
- Clock: 0 Hz

How it works

This tile is an experimental silicon art tile. It plays with the idea of crossing traditional crafts like weaving with the weaving-like work of entangled transistor lines.

How to test

No test - just looks

External hardware

The artwork will be unveiled when decapping the chip

Pinout

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

weaving in silicon #3 [485]

- Author: bleeptrack
- Description: Life comes in cycles
- GitHub repository
- HDL project
- Mux address: 485
- Extra docs
- Clock: 0 Hz

How it works

This tile is an experimental silicon art tile. It plays with the idea of crossing traditional crafts like weaving with the weaving-like work of entangled transistor lines.

How to test

No test - just looks

External hardware

The artwork will be unveiled when decapping the chip

Pinout

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

weaving in silicon #4 [487]

- Author: bleeptrack
- Description: Alpha to Omega
- GitHub repository
- HDL project
- Mux address: 487
- Extra docs
- Clock: 0 Hz

How it works

This tile is an experimental silicon art tile. It plays with the idea of crossing traditional crafts like weaving with the weaving-like work of entangled transistor lines.

How to test

No test - just looks

External hardware

The artwork will be unveiled when decapping the chip

Pinout

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

Verilog OR-Gate [489]

- Author: Pius Sieber
- Description: Simple OR-Gate
- GitHub repository
- HDL project
- Mux address: 489
- Extra docs
- Clock: 0 Hz

How it works

Simple Test of Verilog Chip creation, only includes one or gate currently

How to test

Test with the first two inputs and the first output.

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	OR-Gate Input A	OR-Gate Output	
1	OR-Gate Input B		
2			
3			
4			
5			
6			
7			

TinyQV - Crowdsourced Risc-V SoC [490]

- Author: Michael Bell, et al
- Description: A Risc-V SoC with peripherals from the Tiny Tapeout Risc-V challenge
- GitHub repository
- HDL project
- Mux address: 490
- Extra docs
- Clock: 64000000 Hz

How it works

This is an early version of the Tiny Tapeout collaborative competition Risc-V SoC.

The CPU is a small Risc-V CPU called TinyQV, designed with the constraints of Tiny Tapeout in mind. It implements the RV32EC instruction set plus the Zcb and Zicond extensions, with a couple of caveats:

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

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

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

The peripherals making up the SoC are contributed by the Tiny Tapeout community, with prizes going to the best designs!

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

Address range	Device
0x8000100 - 0x80003FF	User peripherals 4-15
0x8000400 - 0x80004FF	Simple user peripherals 0-15
0xFFFFF00 - 0xFFFFF07	TIME

DEBUG

Register	Address	Description
SEL	0x800000C (R/W)	Bits 6-7 enable peripheral output on the correspo
DEBUG_UART_DATA	0x8000018 (W)	Transmits the byte on the debug UART
STATUS	0x800001C (R)	Bit 0 indicates whether the debug UART TX is b

See also debug docs

TIME

Register	Address	Description
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 selec
IN	0x8000044 (R)	Reads the current state of in0-7
FUNC_SEL	0x8000060 - 0x800007F (RW)	Function select for out0-7

Function Select	Peripheral
0	Disabled

Function Select	Peripheral
1	GPIO
2	UART
3	Disabled
4 - 15	User peripheral 4-15
16 - 31	User byte peripheral 0-15

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
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),

How to test

Load an image into flash and then select the design.

Reset the design as follows:

- Set rst_n high and then low to ensure the design sees a falling edge of rst_n. The bidirectional IOs are all set to inputs while rst_n is low.
- Program the flash and leave flash in continuous read mode, and the PSRAMs in QPI mode
- Drive all the QSPI CS high and set SD1:SD0 to the read latency of the QSPI flash and PSRAM in cycles.
- Clock at least 8 times and stop with clock high
- Release all the QSPI lines
- Set rst_n high
- Set clock low
- Start clocking normally

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

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

Pinout

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

SAR ADC Controller [491]

- Author: Cédric Cyril Hirschi
- Description: Simple SAR ADC Controller with SPI interface
- GitHub repository
- HDL project
- Mux address: 491
- Extra docs
- Clock: 0 Hz

How it works

This project implements a simple SAR ADC controller designed to work with external analog circuitry.

How to test

Attach the external hardware as described below.

To start a conversion, toggle the `start_i` signal to high. As long as this signal is high, the ADC performs conversions.

There is no end of conversion output signal!

External hardware

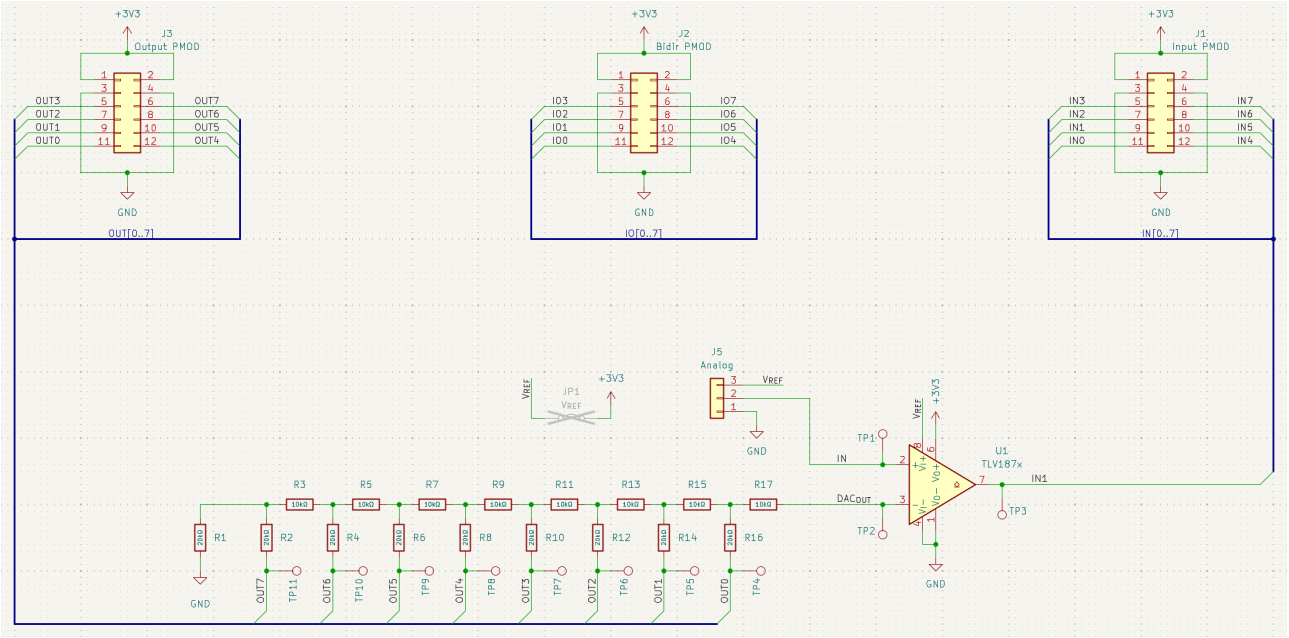
This project needs external analog circuitry to test the ADC functionality.

You need to connect a R-2R ladder network to the DAC outputs (`uo0` to `uo7`) of the design.

The output of this DAC has to be input into the negative comparator input, where the positive comparator input would be the input voltage.

The output of this comparator would then connect to the `comp_i` input (`ui1`) of the design.

The circuit could look something like this:



Pinout

#	Input	Output	Bidirectional
0	start_i	dac0_o	result0_o
1	comp_i	dac1_o	result1_o
2		dac2_o	result2_o
3		dac3_o	result3_o
4		dac4_o	result4_o
5		dac5_o	result5_o
6		dac6_o	result6_o
7		dac7_o	result7_o

MC first Wokwi [513]

- Author: Marko
- Description: learning how to use Wokwi
- GitHub repository
- Wokwi project
- Mux address: 513
- Extra docs
- Clock: 0 Hz

How it works

Inputs to LED number display experiment

How to test

dont use it. not good

External hardware

yes

Pinout

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

2048 sliding tile puzzle game (VGA) [514]

- Author: Uri Shaked
- Description: Slide numbered tiles on a grid to combine them to create a tile with the number 2048.
- GitHub repository
- HDL project
- Mux address: 514
- Extra docs
- Clock: 25175000 Hz

How it works

2048 is a single-player sliding tile puzzle video game. Your goal is to slide numbered tiles on a grid to combine them and create a tile with the number 2048. The game is won when a tile with the number 2048 appears on the board, hence the name of the game. The game is lost when the board is full and no more moves can be made.

The game is played on a 4x4 grid, with numbered tiles that slide when a player moves them using `ui_in` pins or using a SNES compatible controller along with the Gamepad Pmod.

The game starts with two tiles with the number 2 on the board. The player can move the tiles in four directions: up, down, left, and right. When the player moves the tiles in a direction, the tiles slide as far as they can in that direction until they hit the edge of the board or another tile. If two tiles with the same number collide, they merge into a single tile with the sum of the two numbers. The resulting tile cannot merge with another tile again in the same move.

How to test

Use the `ui_in` pins to move the tiles on the board:

`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

Pinout

#	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

TinyTapeout 2025 [515]

- Author: SG Nanovolt
- Description: 8 bit Pseudo-Random Number Generator
- GitHub repository
- Wokwi project
- Mux address: 515
- Extra docs
- Clock: 10000 Hz

How it works

This is a 8 bit pseudo-random number generator. First, the chip clock is divided down by 14 cascaded divide-by-two stages. This chain results in a total division factor of 16'384. For a chip clock of 10 kHz, a frequency of 0.61 Hz results. This local slow clock drives four Fibonacci maximum-length linear shift registers (LFSRs) with 9, 10, 11, and 13 bits. To minimize correlations, these LFSRs are selected such that the sequence lengths are relative prime. For the final output, these four LFSR sequences are combined with XOR operations to generate 8 bits.

How to test

The pseudo-random output bits can be observed with a 7-segment LED display, an oscilloscope, or a logic analyzer.

External hardware

A 7-segment LED display is recommended

Pinout

#	Input	Output	Bidirectional
0		Random bit output 0	
1		Random bit output 1	
2		Random bit output 2	
3		Random bit output 3	
4		Random bit output 4	
5		Random bit output 5	

#	Input	Output	Bidirectional
6		Random bit output 6	
7		Random bit output 7	

LIF neuron [517]

- Author: Lara Pregej
- Description: Low power LIF neuron
- GitHub repository
- HDL project
- Mux address: 517
- Extra docs
- Clock: 0 Hz

How it works

It just works

How to test

Just do it.

External hardware

None.

Pinout

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

3-bit up-down counter [519]

- Author: Nefeli Metallidou
- Description: A 3-bit up-down counter
- GitHub repository
- HDL project
- Mux address: 519
- Extra docs
- Clock: 50000000 Hz

How it works

A 3-bit up/down counter with a reset signal, enable signal, load signal, and up/down signal.

When the rst is low, the output is set to 0. When the load_cnt signal is low, the input data is assigned to the output. When count_enb is high, counting occurs at every positive edge of the clock. updn_cnt controls whether the counter counts up or down.

How to test

Set signals and confirm counting.

External hardware

7-segment disply and driver, resistors.

Pinout

#	Input	Output	Bidirectional
0	data_in[0]	data_out[0]	
1	data_in[1]	data_out[1]	
2	data_in[2]	data_out[2]	
3	rst_		
4	ld_cnt		
5	updn_cnt		
6	count_enb		

Prism [521]

- Author: bleeptrack
- Description: a hypnotic prism floating in a starry night
- GitHub repository
- HDL project
- Mux address: 521
- Extra docs
- Clock: 25175000 Hz

How it works

It's a visual. not much to do currently :)

How to test

Hook it up into the VGA dongle and turn it on!

External hardware

Mole99

Pinout

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

KianV uLinux SoC [522]

- Author: Hirosh Dabui
- Description: A RISC-V ASIC that can boot Linux.
- GitHub repository
- HDL project
- Mux address: 522
- Extra docs
- Clock: 0 Hz

How it works

32-bit RISC-V IMA processor, capable of booting Linux. Features 16 MiB of external SPI flash memory, 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.

Pinout

#	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
7	gpio[7]	gpio[7]	cs2 psram

E-ink display driver [523]

- Author: Tim Edwards
- Description: Test driver for Adafruit 2.13 inch e-ink display
- GitHub repository
- HDL project
- Mux address: 523
- Extra docs
- Clock: 50 Hz

How it works

This is an example hardware driver for an e-ink display. Adafruit makes a nice series of small e-ink displays, but they are designed for an Arduino and driven by software. This project shows how to build a display driver in verilog. To keep memory overhead to a minimum, it operates like a VGA screen saver, displaying simple patterns that can be computed in real time as the pixel positions are counted and transmitted to the driver.

The driver instantiates an SPI master which communicates with the SSD1680 chipset on the e-ink display. Whenever a bit from the input PMOD is set to "1", and initialization sequence is send to the display, followed by a transmission of the display image, followed by a deep sleep power-down. Once in deep sleep mode, the displayed image will remain indefinitely, even if the display is disconnected from the development board.

How to test

The input/output PMOD is used to connect to the e-ink display pins. Since the e-ink display is not PMOD-compatible, it is necessary to install a header onto the e-ink display and create a bundle of jumper wires to connect to the PMOD as follows:

pin signal direction PMOD pin

ECS: uio[0] output 1

MOSI: uio[1] output 2 MISO: uio[2] input 3 SCK: uio[3] output 4 SRCS: uio[4] output 7 RST: uio[5] output 8 BUSY: uio[6] input 9 D/C: uio[7] output 10 GND: 11 or 5 VIN: 12 or 6

To test the eight example patterns, raise one of the input pins to value "1". This can be done with a set of external buttons on the input PMOD, or the input PMOD value can be set from software.

ui[5] is a special case in which the contents of the display board's SRAM are copied directly to the e-ink display. This uses an unusual method in which the SRAM is set to a sequential read mode and then is left enabled while the e-ink display is initialized. Commands being sent to the display are ignored by the SRAM, which outputs one bit on every clock cycle. The SRAM contents are then copied into the display starting at offset address 30 (which is the number of SPI bytes clocked while initializing the display). The SRAM is volatile and so unprogrammed at power-up. It can be programmed using the "pass-through" mode, in which the SRAM's SPI can be bit-banged from the ui[] port using software. Enable "pass-through" mode by setting ui[7:4] to 0xf, then bit-bang using ui[0] for clock and ui[1] for data (if the SRAM is given a READ command, then output from the SRAM can be read from uo[0]). First put the SRAM into sequential mode with command 0x01 0x40. End pass-through mode with ui = 0x00, then re-enter pass-through mode with ui = 0xf0. Continue with the command 0x02 0x00 0x1e and then write 3904 bytes of image data (32 bytes x 122 lines). End pass-through mode again with ui = 0x00, then display the image data with ui = 0x20.

External hardware

Every e-ink display has a very specific driver, and making a general-purpose driver is prohibitive for Tiny Tapeout. The project is designed to drive the Adafruit 2.13" e-ink display, Product ID: 4197, URL https://www.adafruit.com/product/4197 (as of this writing, cost is $22.50).

Pinout

#	Input	Output	Bidirectional
0	All white	Bitbang SCK	SRAM MISO (out)
1	All black	Bitbang MOSI	
2	Vertical stripes		MISO (in, unused)
3	Horizontal stripes		SCK (out)
4	Small checkerboard		SRCS (out, =1)
5	User SRAM contents		RSTB (out)
6	Large checkerboard		BUSY (in)
7	Low-res smiley face		D/C (out)

Yet Another Diffraction Grating Experiment [576]

- Author: htfab
- Description: A remix of Uri's Colorful Stripes
- GitHub repository
- HDL project
- Mux address: 576
- Extra docs
- Clock: 0 Hz

How it works

Should hopefully generate a colorful pattern when viewed under the microscope.

How to test

View under the microscope or smartphone camera.

External hardware

Microscope.

Pinout

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

'' [577]

- Author: Lae
- Description: Calculation
- GitHub repository
- Wokwi project
- Mux address: 577
- Extra docs
- Clock: 0 Hz

How it works

Not desided yet

How to test

Not desided yet

External hardware

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

Pinout

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

WowkiProject [579]

- Author: Kevin
- Description: TestDescription
- GitHub repository
- Wokwi project
- Mux address: 579
- Extra docs
- Clock: 1000000 Hz

How it works

XOR outputs on first four pins, selectable with multiplexer from IN4

How to test

Change toggles from IN0 to IN4

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	IN0	OUT0	
1	IN1	OUT1	
2	IN2	OUT2	
3	IN3	OUT3	
4	IN4		
5			
6			
7			

Mini Calculator v1 [581]

- Author: Luis
- Description: adds up 1 and 2
- GitHub repository
- Wokwi project
- Mux address: 581
- Extra docs
- Clock: 0 Hz

How it works

TBD

How to test

TBD

External hardware

TBD

Pinout

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

and [583]

- Author: eric
- Description: and
- GitHub repository
- Wokwi project
- Mux address: 583
- Extra docs
- Clock: 0 Hz

How it works

it works

How to test

TBD

External hardware

TBD

Pinout

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

DigOTA [585]

- Author: Ali Meimandi
- Description: It works as a digital based OTA
- GitHub repository
- Wokwi project
- Mux address: 585
- Extra docs
- Clock: 50000 Hz

How it works

It is a digital OTA that needs 50kHz clock and two inputs and it has three outputs

How to test

you connect two input signals of OTA and the output current is proportional to number of pulses in the output stages

External hardware

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

Pinout

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

TinyTapeoutWorkshop [587]

- Author: Robin LEPLAE
- Description: Logic gates
- GitHub repository
- Wokwi project
- Mux address: 587
- Extra docs
- Clock: 0 Hz

How it works

Bunch of logic gates

How to test

Test truth tables

External hardware

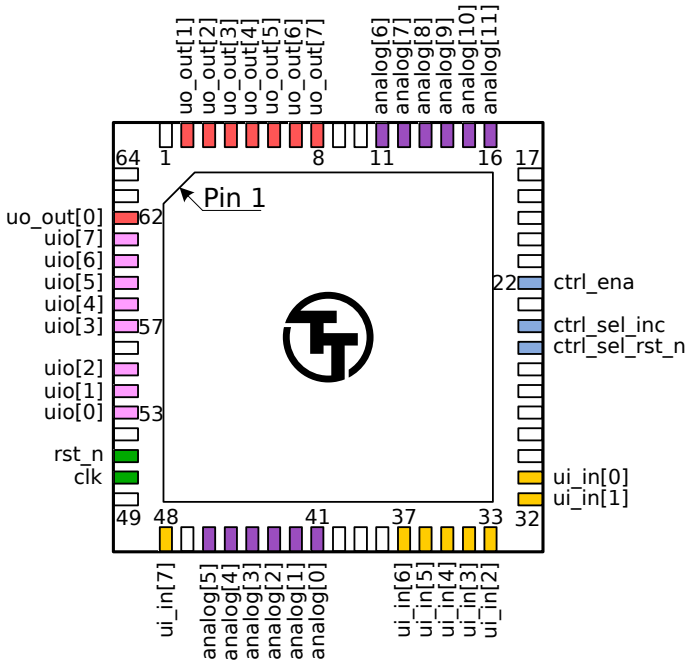
None

Pinout

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

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. The pinout is provided for advanced users, as most users will not need to solder the chip directly.

The Tiny Tapeout Multiplexer

Overview

The Tiny Tapeout Multiplexer distributes a single set of user IOs to multiple user designs. It is the backbone of the Tiny Tapeout chip.

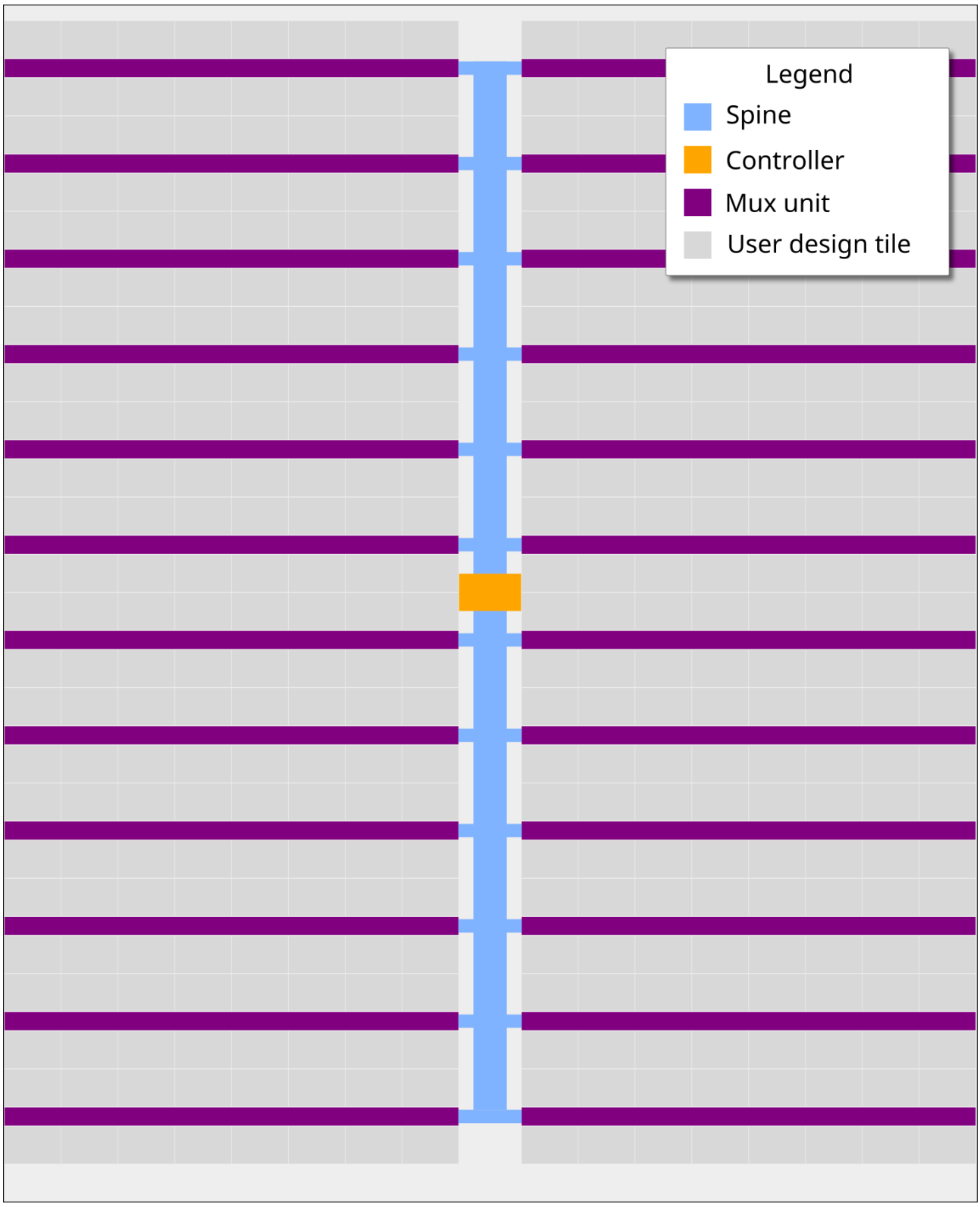
It has the following features:

- 10 dedicated inputs
- 8 dedicated outputs
- 8 bidirectional IOs
- Supports up to 512 user designs (32 mux units, each with up to 16 designs)
- Designs can have different sizes. The basic unit is a called a tile, and each design can occupy up to 16 tiles.

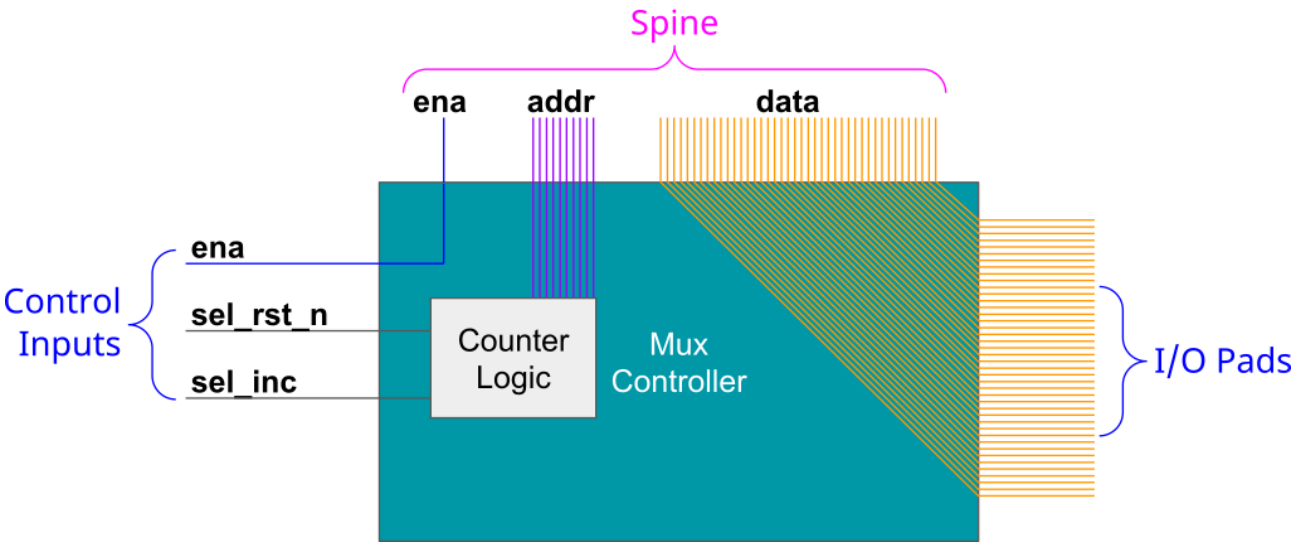
Operation

The multiplexer consists of three main units:

1. The controller - used to set the address of the active design
2. The spine - a bus that connects the controller with all the mux units
3. Mux units - connect the spine to individual user designs


Legend
Spine
Controller
Mux unit
User design tile

The Controller

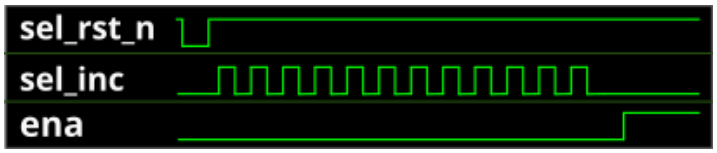


The mux controller has 3 inputs lines:

Input	Description
`ena`	Sent as-is (buffered) to the downstream mux units
`sel_rst_n`	Resets the internal address counter to 0 (active low)
`sel_inc`	Increments the internal address counter by 1

It outputs the address of the currently selected design on the `si_sel` port of the spine (see below).

For instance, to select the design at address 12, you need to pulse `sel_rst_n` low, and then pulse `sel_inc` 12 times:



Internally, the controller is just a chain of 10 D flip-flops. The `sel_inc` signal is connected to the clock of the first flip-flop, and the output of each flip-flop is connected to the clock of the next flip-flop. The `sel_rst_n` signal is connected to the reset of all flip-flops.

The following Wokwi projects demonstrates this setup: https://wokwi.com/projects/364347807664031745. It contains an Arduino Nano that decodes the currently selected mux address and displays it on a 7-segment display. Click on the button labeled `RST_N` to reset the counter, and click on the button labeled `INC` to increment the counter.

The Spine

The controller and all the muxes are connected together through the spine. The spine has the following signals going on it:

From controller to mux:

- `si_ena` - the `ena` input
- `si_sel` - selected design address (10 bits)
- `ui_in` - user clock, user `rst_n`, user inputs (10 bits)
- `uio_in` - bidirectional I/O inputs (8 bits)

From mux to controller:

- `uo_out` - User outputs (8 bits)
- `uio_oe` - Bidirectional I/O output enable (8 bits)
- `uio_out` - Bidirectional I/O outputs (8 bits)

The only signal which is actually generated by the controller is `si_sel` (using `sel_rst_n` and `sel_inc`, as explained above). The other signals are just going through from/to the chip IO pads.

The Multiplexer (The Mux)

Each mux branch is connected to up to 16 designs. It also has 5 bits of hard-coded address (each unit gets assigned a different address, based on its position on the die).

The mux implements the following logic:

If `si_ena` is 1, and `si_sel` matches the mux address, we know the mux is active. Then, it activates the specific user design port that matches the remaining bits of `si_sel`.

For the active design:

- `clk`, `rst_n`, `ui_in`, `uio_in` are connected to the respective pins coming from the spine (through a buffer)
- `uo_out`, `uio_oe`, `uio_out` are connected to the respective pins going out to the spine (through a tristate buffer)

For all others, inactive designs (including all designs in inactive muxes):

- `clk`, `rst_n`, `ui_in`, `uio_in` are all tied to zero
- `uo_out`, `uio_oe`, `uio_out` are disconnected from the spine (the tristate buffer output enable is disabled)

Pinout

SKY130 Open Frame chips

mprj_io pin	Function	Signal	QFN64 pin
0	Input	ui_in[0]	31
1	Input	ui_in[1]	32
2	Input	ui_in[2]	33
3	Input	ui_in[3]	34
4	Input	ui_in[4]	35
5	Input	ui_in[5]	36
6	Input	ui_in[6]	37
7	Analog	analog[0]	41
8	Analog	analog[1]	42
9	Analog	analog[2]	43
10	Analog	analog[3]	44
11	Analog	analog[4]	45
12	Analog	analog[5]	46
13	Input	ui_in[7]	48
14	Input	clk †	50
15	Input	rst_n †	51
16	Bidirectional	uio[0]	53
17	Bidirectional	uio[1]	54
18	Bidirectional	uio[2]	55
19	Bidirectional	uio[3]	57
20	Bidirectional	uio[4]	58
21	Bidirectional	uio[5]	59
22	Bidirectional	uio[6]	60
23	Bidirectional	uio[7]	61
24	Output	uo_out[0]	62
25	Output	uo_out[1]	2
26	Output	uo_out[2]	3
27	Output	uo_out[3]	4
28	Output	uo_out[4]	5
29	Output	uo_out[5]	6
30	Output	uo_out[6]	7
31	Output	uo_out[7]	8
32	Analog	analog[6]	11
33	Analog	analog[7]	12
34	Analog	analog[8]	13
35	Analog	analog[9]	14

mprj_io pin	Function	Signal	QFN64 pin
36	Analog	analog[10]	15
37	Analog	analog[11]	16
38	Mux Control	ctrl_ena	22
39	Mux Control	ctrl_sel_inc	24
40	Mux Control	ctrl_sel_rst_n	25
41	Reserved	(none)	26
42	Reserved	(none)	27
43	Reserved	(none)	28

IHP SG13G2 custom pad frame

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]

QFN64 pin	Function	Signal
27	Analog	analog[6]
28	Analog	analog[7]
29	Ground	GND Core
30	Power	VDD Core
31	Ground	GND IO
32	Power	VDD IO
33	Bidirectional	uio[0]
34	Bidirectional	uio[1]
35	Bidirectional	uio[2]
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

Notes

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

Funding

IHP PDK support for Tiny Tapeout was funded by The SwissChips Initiative.

The manufacturing of Tiny Tapeout IHP 0p2 silicon was funded by the German BMBF project FMD-QNC (16ME0831).

Team

Tiny Tapeout would not be possible without a lot of people helping. We would especially like to thank:

- Uri Shaked for wokwi development and lots more
- Patrick Deegan for PCBs, software, documentation and lots more
- Sylvain Munaut for help with scan chain improvements
- Mike Thompson and Mitch Bailey for verification expertise
- Tim Edwards and Harald Pretl for ASIC expertise
- Jix for formal verification support
- Proppy for help with GitHub actions
- Maximo Balestrini for all the amazing renders and the interactive GDS viewer
- James Rosenthal for coming up with digital design examples
- All the people who took part in Tiny Tapeout 01 and volunteered time to improve docs and test the flow
- The team at YosysHQ and all the other open source EDA tool makers
- Jeff and the Efabless Team for running the shuttles and providing OpenLane and sponsorship
- Tim Ansell and Google for supporting the open source silicon movement
- Zero to ASIC course community for all your support
- Jeremy Birch for help with STA