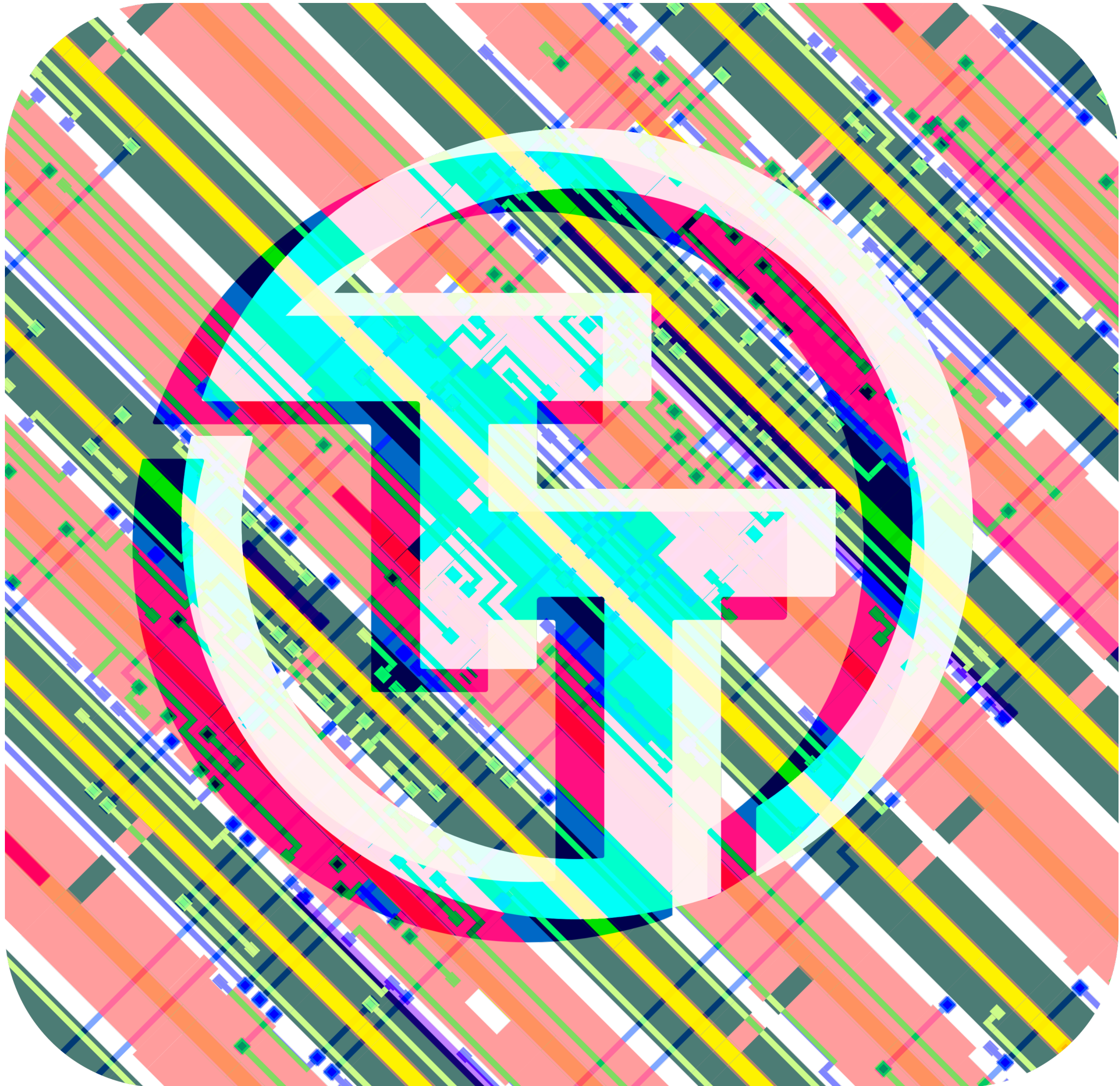


Tiny Tapeout 9 Datasheet

Project Repository
https://github.com/TinyTapeout/tinytapeout-09

November 12, 2024

Contents

Chip map 12

Projects 15

Chip ROM [0] . . . 15
TinyTapeout Factory Test 1 . . . 17
Trubick - Tiny Tapeout Logic Gate 2 . . . 19
Andrew Vo - Repository [3] . . . 20
tinytapeout [4] . . . 21
Half adder [5] . . . 22
Samson's Tiny Tapout Project [6] . . . 23
Jacks First Project [7] . . . 25
4 x 4 array multiplier NuKoP [8] . . . 26
MuxLED [9] . . . 28
Tiny Tapeout [10] . . . 29
halfadder+not [11] . . . 30
Yohan Tiny Tapeout Project [12] . . . 31
4-bit Multiplier [13] . . . 32
Yared Fente's Tiny Tapeout [14] . . . 35
Metastable Chip [15] . . . 36
Secret Initial [32] . . . 37
Binary to 7 Segment Display Decoder [33] . . . 38
Tahiti [34] . . . 39
Letter H [35] . . . 40
APTT [36] . . . 41
Two PFD [37] . . . 42
Zero to Nine Display Count [38] . . . 43
Redco [39] . . . 44
Light LED [40] . . . 46
Matmul System [41] . . . 48
Tiny Tapeout-Huerta [42] . . . 49
Light [43] . . . 50
TinyTapeOut [44] . . . 52
Nathan's chip [45] . . . 53
OR gate [46] . . . 54
project [47] . . . 55
D_flipflop_hold_test [64] . . . 56
Dipankar's first Wowki design [65] . . . 57
Bit Counter [66] . . . 58
Hamad's design [67] . . . 59
Full bit adder [68] . . . 60
Encoder [69] . . . 61
Encoder [70] . . . 62

GDS [71] 63
Big J's Big Circuit [72] 64
2 Bit Times 2 Bit Plus 4 Bit MAD and 5 Bit Binary to 7 Segment Display [73] 65
AndLogicPass [74] 67
Not Good BCD Decoder [75] 68
Half Adder [76] 69
tinytapeoutkr [77] 72
Jordan [78] 73
My First ASIC [79] 74
GJAA Design [96] 75
8b10b decoder and multiplier [97] 76
Logic Gates [98] 78
Test Design 1 [99] 79
My First TinyTapeout [100] 81
Decimation Filter for Incremental and Regular Delta-Sigma Modulators [101] 83
1st [102] 86
adder-accumulator [103] 87
JCB First WOKWI Design [104] 93
ECE 298A 8-Bit CPU Control Block [105] 94
Logic Gates 7-Segment Display [106] 98
LFSR Encrypter [107] 99
BadeTP [108] 100
SkyKing Demo [109] 102
Lynn's TinyTapeout Design [110] 103
Two LIF Neurons with STDP Learning [111] 104
4-bit-multiplier [128] 107
ece2204_4x4_mult [129] 109
my_4bit_multiplier [130] 111
T3 (Tiny Ternary Tapeout) [131] 113
Hybrid_Adder_8bit [132] 117
3 Neuron ALIF [133] 119
8-bit Carry Look-Ahead Adder [134] 121
2bit adder [135] 125
RISC-V Mini [136] 126
4-1 mux [137] 129
8-bit carry-skip [138] 130
STDP Circuit [139] 132
4 bit array multiplier [140] 134
instrumented_ring_oscillator [141] 135
Array Multiplier [142] 137
Linear Feedback Shift Register [143] 139
Frequency Encoder and Decoder [160] 140

TT Test [161] 142
carry skip adder [162] 143
4-bit up/down binary counter [163] 145
xor gate with registered output [164] 147
Team 17's 8 bit DAC [165] 149
Multi-LFSR [166] 150
ECE2204MultiplierProject [167] 152
Micro tile container [168] 155
4bit multiplier [169] 157
Forward Pass Network for Simple ANN [170] 158
Tiny Registers [171] 160
7-Segment Byte Display [172] 166
Leaky Integrate Fire Neuron [173] 169
znah_vga_ca [174] 171
Tiny Tapeout Group 7 Lab D [175] 172
4-bit Multiplier [192] 174
FIREngine [193] 176
4x4multiplier [194] 178
Lab B Group 1 Array Multiplier [195] 181
4-bit Multiplier [196] 182
Array Multiplier [197] 184
4x4 Multiplier [198] 187
4x4 Array Multiplier [199] 189
tt09 kathyhtt [200] 191
4x4 Array Multiplier [201] 193
TINY TAPE OUT [202] 196
ECE2204 4x4 Array Multiplier [203] 197
TinyTapeout1 [204] 200
comparator [205] 201
FB GDS [206] 202
4x4 Array Multiplier [207] 203
Semana UCU Verilog [224] 206
4 by 4 Array Multiplier [226] 212
4-bit multiplier [228] 215
OpenRAM SRAM macro [229] 218
Array Multiplier [230] 220
VGA Pride [231] 222
4-bit Array Multiplier [232] 226
Noise test for a CDAC capacitor chain [233] 229
ECE-UY 2204 4x4 Array Multiplier [234] 230
Analog Switch [235] 233
array_multiplier [236] 235

Digital OTA [237] 238
8-bit-CARRY_SKIP [238] 240
Telephone hybrid [239] 242
Array Multiplier [256] 244
Array multiplier [258] 247
Array Multiplier [260] 250
1bit_am_sdr [261] 253
Array Multiplier [262] 257
Time to Digital Converter [263] 260
Delta RNN and Leaky Integrate-and-Fire Nueron Circuit [264] 262
tt_um_tim2305_adc_dac [265] 264
Verilog ring oscillator [266] 266
2-bit Flash ADC [267] 267
Adaptive Leaky Integrate and Fire Neuron [268] 269
pll [269] 271
Matmul System [270] 273
Analog MUX module [271] 274
Steven's Wokwi Test [288] 276
2-Bit-Adder [289] 277
8-Bit CPU [290] 278
fulladder [291] 294
RLE Video Player [292] 295
Hopfield Network with Izhikevich-type RS and FS Neurons [293] 298
4-bit Multiplier [294] 299
Perceptron [295] 302
Histogramming [296] 303
test_friday2 [297] 306
Perceptron Neuron [298] 310
carry_select [299] 312
I2C and SPI [300] 314
Lab C 4x4 Mult-Array [301] 315
Configurable Logic Block [302] 318
Tiny RAM DFF 2r1w [303] 320
ECE-2204 4x4 Array Multiplier [320] 324
Senol Gulgonul tt09 [321] 326
ECE2204 4x4 Array Multiplier [322] 327
Space Detective Maze Explorer [323] 329
Array Multiplier [324] 332
Hamming Code (7,4) [325] 335
ece2204 project for tapeout [326] 341
tiny-tapeout-8bit-GPTPrefixCircuit [327] 344
4x4 array multiplier [328] 347

LIF on a Ring Topology [329] 350
4-bit-array-multiplier [330] 352
Delta-Sigma ADC Decimation Filter [331] 355
Array_Multiplier [332] 356
an lfsr with synaptic neurons (excitatory or inhibitatory) [333] 358
Generador PWM multiproposito con frecuencia y ciclo de trabajo modulable [334] 360
Perceptron [335] 362
2_bit_7seg [416] 363
Adbe_Project [417] 364
8 bit LFSR [418] 365
Odd or even [419] 366
Broken Two Bit Adder [420] 367
Manchester Encoder [421] 368
4 bit adder [422] 369
Tiny_Tapeout_Adder! [423] 370
TinyTapeout workshop - Wokwi 8 Bit LFSR [424] 371
Morse Code for J and R [425] 372
3bitFullAdder [426] 373
XorTree [427] 374
Sigma-Delta ADC [428] 375
tt09-4bit-adder-dhags [429] 377
Mini-Adder and Clock Divider [430] 378
7-seg display checker [431] 379
Drew's First Wokwi Design [448] 380
Shadoff Test [449] 381
Pseudo Random Generator Using 2 Ring Oscillators [450] 383
Tiny Tapeout Take 2 [451] 384
JonsFirstTapeout [452] 385
Speller [453] 386
And Gates that don't do much [454] 387
RAYS FIRST TAPEOUT rev 2 [455] 388
SimplePattern [456] 389
6 Bit shift register [457] 391
sphereinabox hello [458] 392
Duffy [459] 393
Input Counter [460] 394
Will It NAND? [461] 395
4 bit ALU [462] 396
Bad Logic [463] 398
Full Adder [481] 399
2048 sliding tile puzzle game (VGA) [482] 400

TT-Farhad [483] 402
Four Bit Adder [485] 403
SPI Logic Analyzer with Charlieplexed Display [486] 404
2 bit adder [487] 406
pio-ram-emulator example: Julia fractal [488] 407
AND and NOT gate testing [489] 410
Analog 8 bit 3.3v R2R DAC [490] 411
Kanoa's first Wokwi deseign Tinytapeout 2024 Nonsense [491] 413
Ring Oscillators [492] 414
add it [493] 416
AMS Chip ITS [494] 417
one [495] 419
SIC-1 8-bit SUBLEQ Single Instruction Computer [518] 420
4-bit R2R DAC [520] 423
Dickson Charge Pump [522] 425
Analog double inverter [524] 429
OpAmp 3stage [526] 431
Counter [544] 433
Shifter [545] 434
7-bit arbiter [546] 435
NAND Flip-Flop [547] 436
LCA's first Wokwi design [548] 437
chip [549] 438
Tinysynth [550] 439
rhTinyTapeout [551] 440
half adder [552] 441
rand [553] 442
Tiny Tapeout 9 Template [554] 443
Ripple counter [555] 444
four flip flops [556] 445
adder-tt09 [557] 446
Full Adder [558] 448
NAND-Equ [559] 449
Elevator Design [576] 450
L display [578] 451
S-R latch [580] 453
Gabe's Big AND [582] 454
Secret Code [584] 455
joes-first-tiny-tapeout [586] 457
Abey's 1st Chip Design [588] 458
patrick's project [590] 459
tt09-pettit-wokproc-trainer [591] 460

Full adder Design [608] 465
seven [609] 466
Vincent's First Design [610] 467
gatesoup [611] 468
A Tale of Two NCOs [612] 469
Tiny Tapeout 9 Template Version 1 Tata Luka [613] 471
Workshop demo [614] 472
UART TX [615] 473
LRC - Longitudinal Redundancy Check generator [616] 474
my First WokWi Design [617] 475
print [618] 476
Tiny Tapeout 9 [619] 477
hello [620] 478
tinydsp-lol [621] 479
Full Adder [622] 480
Leaky integrate and fire spiking neural network [623] 481
Stochastic Integrator [640] 483
E2M0 x INT8 Systolic Array [642] 485
VGA Nyan Cat [644] 487
15 channels emission counter [646] 489
Basic Oszilloscope and Signal Generator [648] 492
T3 (Tiny Ternary Tapeout) CSA [650] 496
CORA-16 [652] 500
ITS-RISCV [654] 506
16 Bit Izhikevich Neuron [672] 509
Giant Ring Oscillator (3853 inverters) [673] 512
dff_mem [674] 514
Lab B Group 10 Array Multiplier [675] 517
Verilog ring oscillator V2 [676] 519
TwoChannelSquareWaveGenerator [677] 521
Basic model for Systollic array implementation of LIF [678] 523
RGB Mixer demo [679] 525
mips.sv [680] 526
VGA clock [681] 527
gta6 [682] 529
8-bit CBILBO [683] 530
Name Speller [684] 532
Michaels Tiny Tapeout ALU [685] 533
2-bit Full Adder [686] 535
ovl abc chip [687] 536
Simon's Caterpillar [704] 537
tt6502 [706] 539

Oscillating Bones [708] 540
SoCET UART with FIFO buffers [710] 543
VGA Drop (audio/visual demo) [712] 545
Warp [714] 546
Sequential Shadows [TT08 demo competition] [716] 548
achasen workshop validation [718] 554
7-Segment Digital Desk Clock [736] 555
TinySnake [737] 557
Basic Perceptron + ReLU [738] 559
Classic 8-bit era Programmable Sound Generator SN76489 [739] 560
Basic Matrix-Vector Multiplication [740] 568
Classic 8-bit era Programmable Sound Generator AY-3-8913 [741] 570
8 bit MAC Unit [742] 579
Cgates [743] 581
Programmable PWM Generator [744] 583
eksdee [745] 585
Verilog test project [746] 586
ternary, E1M0, E2M0 decoders [747] 587
Basic LIF Neuron [748] 589
Dynamic Threshold Leaky Integrate-and-Fire [749] 591
Integrate-and-Fire Neuron Circuit [750] 592
tt09-C6-array-multiplier [751] 594
Zilog Z80 [770] 596
Spectrogram extractor, 2 channels [782] 600
Encoder [800] 603
chip_fab [801] 604
Clocked Display [802] 605
YoshiTP [803] 606
A simple leaky integrate and fire neuron [804] 608
Who knows what's happening Tiny Tapeout [805] 610
VGA Tiny Logo (1 tile) [806] 611
Tiniest of tapeouts [807] 612
SK Test Workshop [808] 613
Tian TT9 [809] 614
2-bit 2x2 Matrix Multiplier [810] 615
RISCV Processor Design [811] 617
Verilog ring oscillator V3 [812] 619
Test_project [813] 620
4-Bit Toy CPU [814] 621
RISCV Processor Design [815] 623
APA102 to WS2812 Translator [832] 625
Collatz conjecture brute-forcer [834] 627

TT09 SKY130 ROM Test [836] 629
TT09 SKY130 ROM Test (no LVT variant) [838] 631
PID Controller [840] 633
Frequency Counter SSD1306 OLED [842] 635
Basys 3 Over UART Link [844] 637
Tiny 1-bit AM Radio [846] 639
Encoder [864] 643
dummy [865] 644
First Tapeout Chip - OCR [866] 645
sarah's first chip [867] 647
Half Adder [868] 648
tiny cipher 4 bit key [869] 649
Kai's Death Adder [870] 650
2 input multiplexor [871] 651
Kevin Project [872] 652
Tutorial: Simple LIF Neuron [873] 653
Leaky Neuron Network [874] 655
Neuromorphic Hardware for SNN LSTM [876] 657
Project [878] 660
Hardware UTF Encoder/Decoder [897] 661
BINCounterAndGates [899] 667
Color Bars [901] 670
Fuzzy Search Engine [903] 672
TT09Ball GDS Art [905] 679
Simon Says memory game [907] 681
TT09Ball VGA Screensaver [909] 684
ChatGPT-generated Spiking Neural Network with Delays [910] 686
32x8 LED Matrix Animation [911] 688
8b10b decoder and multiplier [961] 690
Styler [963] 692
VGA Timing Experiments [965] 697
Universal Binary to Segment Decoder [967] 700
INTERCAL ALU [969] 709
Simple PWM Module [971] 714
freqSweep [973] 715
Atari 2600 [974] 720
LED Bitserial Cipher [975] 721

Pinout 724

The Tiny Tapeout Multiplexer 725
Overview 725

Operation 725
Pinout 728

Sponsored by **731**

Team **731**

Chip map

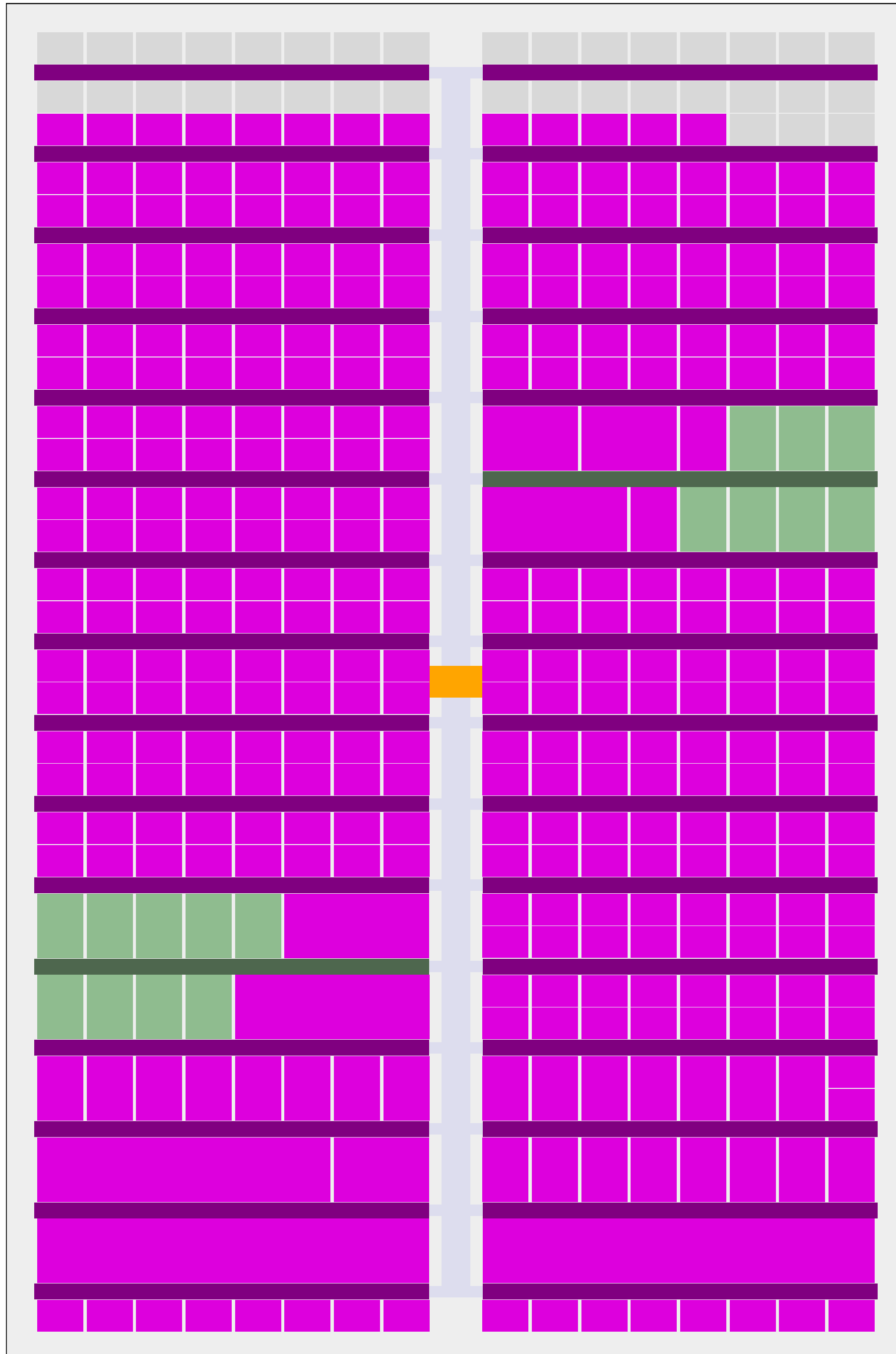


Figure 1: Full chip map

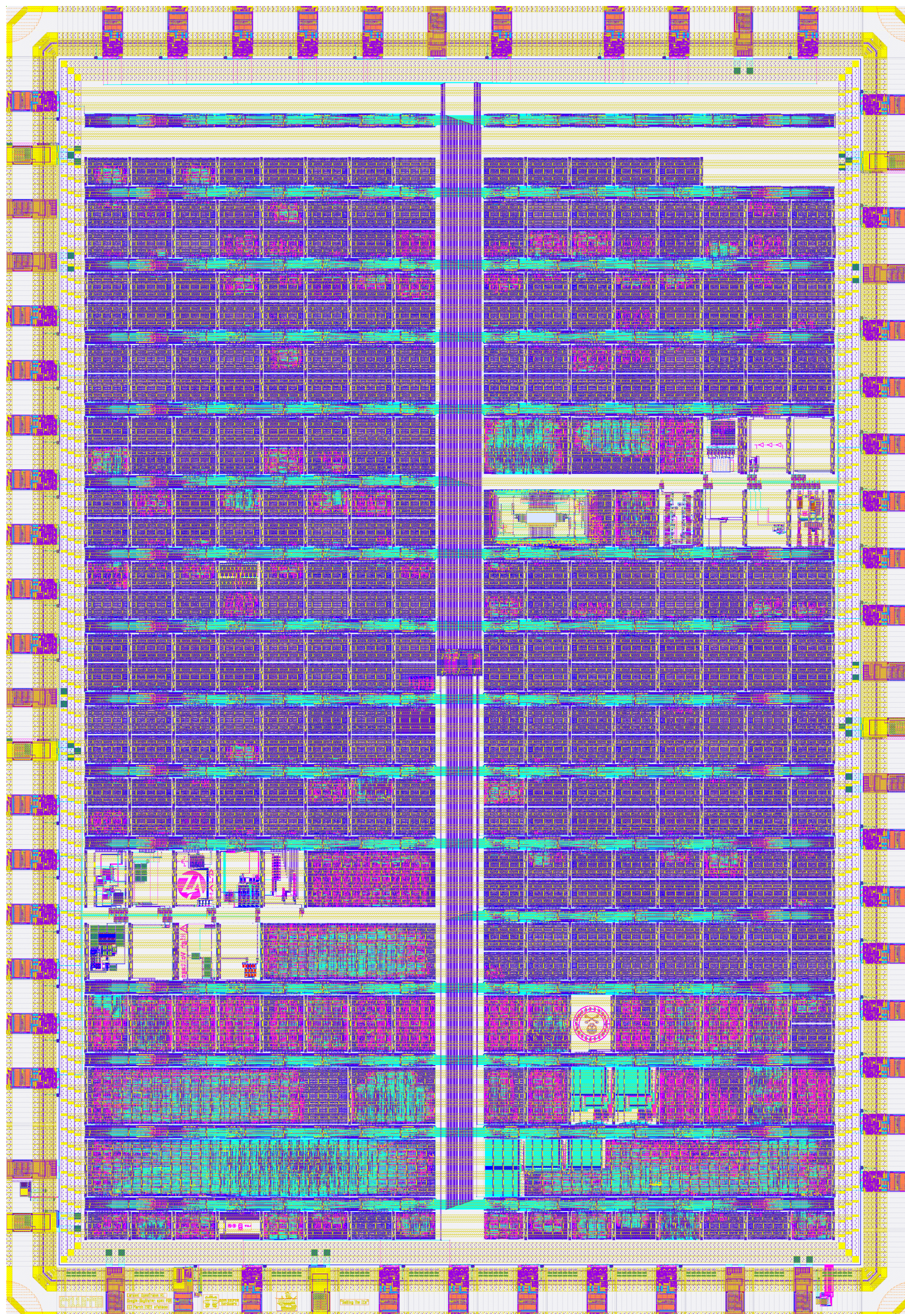


Figure 2: GDS render

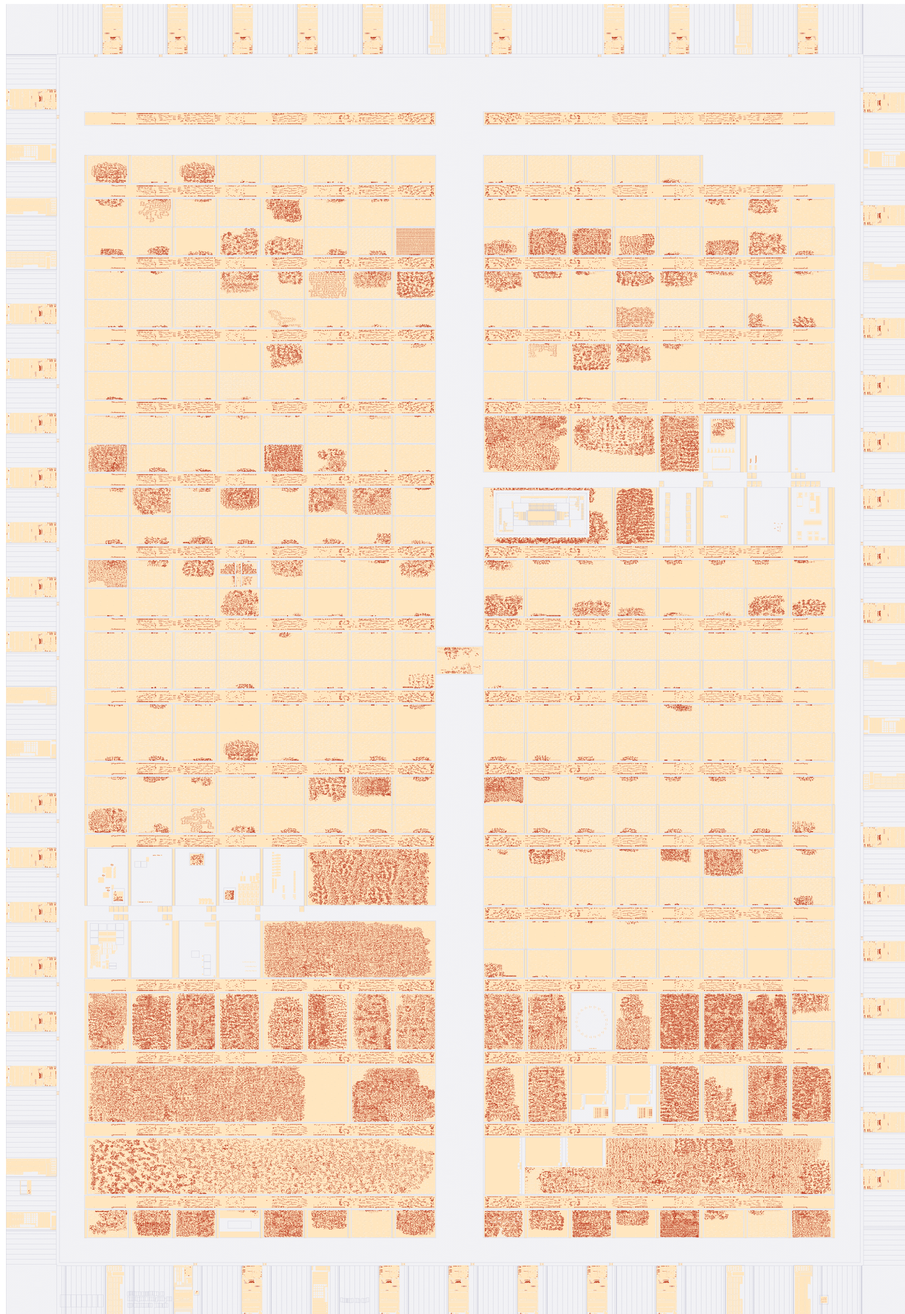


Figure 3: Logic density (local interconnect layer)

Projects

Chip ROM [0]

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

How it works

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

How to test

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

Pinout

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

TinyTapeout 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_a1	output1 / counter1	in_b1 / counter1
2	in_a2	output2 / counter2	in_b2 / counter2
3	in_a[3]	output[3] / counter[3]	in_b[3] / counter[3]
4	in_a[4]	output[4] / counter[4]	in_b[4] / counter[4]
5	in_a[5]	output[5] / counter[5]	in_b[5] / counter[5]
6	in_a[6]	output[6] / counter[6]	in_b[6] / counter[6]
7	in_a[7]	output[7] / counter[7]	in_b[7] / counter[7]

Trubick - Tiny Tapeout Logic Gate 2

- Author: Zane Trubick
- Description: Code for 7-Segment
- GitHub repository
- Wokwi project
- Mux address: 2
- Extra docs
- Clock: 0 Hz

How it works

This chip has a secret code. Figure out the code to activate the "lock." Success will be indicated by a light.

How to test

Troubleshoot until you get the code.

External hardware

7-Segment Display, LED

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	

Andrew Vo - Repository [3]

- Author: Andrew Vo
- Description: Repository WokWi
- GitHub repository
- Wokwi project
- Mux address: 3
- Extra docs
- Clock: 0 Hz

How it works

Using inverters to light up an 8 segment clock. Explain how your project works My chip uses a variety of 3 different inverters, which also involves a flip-flop inverter.

How to test

Flipping the switches will light up its corresponding segment, however the flip flop inverter has its energy stored with switch 3 and 4, which would alternate. Explain how to use your project

External hardware

LED 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	

tinytapeout [4]

- Author: Htun
- Description: Encoder
- GitHub repository
- Wokwi project
- Mux address: 4
- Extra docs
- Clock: 0 Hz

How it works

My project works by connecting the first few ins and outs using not gates. Following that there are a few connections that go straight to the output. There is an And gate and any not gates.

How to test

Press the run button and flip switches to see what lights light up.

External hardware

None.

Pinout

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

Half adder [5]

- Author: Keyshon Howard
- Description: 2x2 Half adder
- GitHub repository
- Wokwi project
- Mux address: 5
- Extra docs
- Clock: 0 Hz

How it works

"The Project is a half adder that uses an Xor gate for the Sum on the inputs and an and gate for the Carry bit"

How to test

"You change the inputs and see the output change based on the lights illuminating"

External hardware

"Two LED displays to see the output"

Pinout

#	Input	Output	Bidirectional
0		OUT0	
1	IN1		
2	IN2		
3			
4			
5		OUT5	
6			
7			

Samson's Tiny Tapout Project [6]

- Author: Samson
- Description: A game to get the number to show up as 0
- GitHub repository
- Wokwi project
- Mux address: 6
- Extra docs
- Clock: 0 Hz

How it works

The inputs 0-7 will change how the LED will work. Some inputs use AND statements and others use XOR. The user will try to find out how to get the number 0.

How to test

The user will input/guess to find the combination to get 0.

External hardware

7 segment display.

Pinout

#	Input	Output	Bidirectional
0	Connected to an AND statement with 1 to activate top and top right LEDs	Connected to an AND statement with 1 to activate top and top right LEDs	

#	Input	Output	Bidirectional
1	Connected to an AND statement with 0 to activate top and top right LEDs	Connected to an AND statement with 1 to activate top and top right LEDs	
2	Goes to input w/ the same number	Goes to input w/ the same number	
3	Goes to input w/ the same number	Goes to input w/ the same number	
4	Goes to input w/ the same number	Goes to input w/ the same number	
5	Goes to input w/ the same number	Goes to input w/ the same number	
6	Connected to an XOR statement with 7 to activate top and top right LEDs	Connected to an XOR statement with 7 to activate top and top right LEDs	
7	Connected to an XOR statement with 6 to activate top and top right LEDs	Connected to an XOR statement with 6 to activate top and top right LEDs	

Jacks First Project [7]

- Author: Jack B
- Description: Jacks Frist Wokwi template
- GitHub repository
- Wokwi project
- Mux address: 7
- Extra docs
- Clock: 0 Hz

How it works

This project is a full adder.

How to test

Test inputs 0 and 1 as input bits, and input 2 as Carry in. Output 0 is the first digit out, and output 1 as carry out.

External hardware

Used an LED to test the full adder.

Pinout

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

4 x 4 array multiplier NuKoP [8]

- Author: Aiden Li, Mahid Hosen
- Description: given two 4 bit unsigned binary numbers, outputs the product of the two numbers
- GitHub repository
- HDL project
- Mux address: 8
- Extra docs
- Clock: 0 Hz

How it works

Using the 4 by 4 multiplier from the previous Lab, we implemented the design so that it could be used by a TinyTapeout chip. The 4 by 4 multiplier uses a series of full adders and AND gates, in order to multiply two 3 bit numbers together. In order to do this, the module multiplies the top number by the bottom number, using the AND and Full Adders, and repeats this over and over for each digit in the second binary number. Each number created has a zero added as a least significant bit, and then when all the four numbers are made, they are all added together to find the total product. The Verilog design incorporates the design by using multiple modules to represent different parts of the design. A full adder module is made to add two one digit bits together. Using the full adder module, the part module uses this to multiply each digit of the first number to one digit in the bottom number. Finally, the array_mult_generate module uses the part module to repeat this process for every digit on the second number, using generate to loop through each one. This final module outputs the 8 bit product of the two 4 digit binary numbers.

How to test

The design works by having a set of 8 switches and 8 LEDs. The 8 switches represent the four bits for one input, and the 4 bits for another. The LEDs are for the output with it lighting up as a 1, and with it off being 0. Switching the switches changes the inputs for the two binary numbers you want and the LEDs will correspond with the 8 digit product. In order to test to see that the design works, you can choose two 4 bit numbers and see if the product displayed is correct. You can test for inputs such as 0000 and 0001 which should output 0000 and the other number respectively, because they are identities. We don't need to check for overflow because the largest product, 15 x 15 = 225 is able to be represented by the 8 digits. Other numbers can also be used in order to test the functionality of the design.

External hardware

The TinyTapeout chip has switches for the inputs and LEDs for the outputs.

Pinout

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

MuxLED [9]

- Author: Alex Moore
- Description: mulitplexor connected to LED
- GitHub repository
- Wokwi project
- Mux address: 9
- Extra docs
- Clock: 0 Hz

How it works

My project uses a mulitplexor to light up an LED as well as the display module.

How to test

Check if the LED and display both light up by switching input 7 on/off and inputs 4 and 6, if in7 is on then in 6 should turn the lights on. If in 7 is off then in4 will turn on the lights

External hardware

LED display & LED diode

Pinout

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

Tiny Tapeout [10]

- Author: Andy
- Description: Using logic gates to determine sections on a 7 segment display
- GitHub repository
- Wokwi project
- Mux address: 10
- Extra docs
- Clock: 0 Hz

How it works

Have several input values with respective output values while using gates to control different the inputs to have different outputs.

How to test

Switch on and off for the input gates, test around different value combinations and options to see what you can create on the number display.

External hardware

none.

Pinout

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

halfadder+not [11]

- Author: Vincent Phan
- Description: My project has a half adder and not gate which turns the lightoff and on for a clock.
- GitHub repository
- Wokwi project
- Mux address: 11
- Extra docs
- Clock: 0 Hz

How it works

My first two inputs go through an adder. The last input goes through an inverter. The rest pass through.

How to test

Flip the switches in order to light up the clock.

External hardware

This requires the default seven segment display and dip switches.

Pinout

#	Input	Output	Bidirectional
0	connected to adder	sum	
1	connected to adder	carry	
2	pass through	pass through	
3	pass through	pass through	
4	pass through	pass through	
5	pass through	pass through	
6	pass through	output to inverter	
7	input to inverter		

Yohan Tiny Tapeout Project [12]

- Author: Juan
- Description: Mixed Logic Gate to control 7 segment display
- GitHub repository
- Wokwi project
- Mux address: 12
- Extra docs
- Clock: 0 Hz

How it works

"This is a project in work. The gates makes signals do funky things."

How to test

"That is to be determined once the functionality is figured out."

External hardware

"Seven segment display, input board, output board, switch panel"

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 Multiplier [13]

- Author: Nick Pham, Nathan Macapinlac
- Description: 4-bit multiplier for NYU's digital logic course's Lab 4
- GitHub repository
- HDL project
- Mux address: 13
- Extra docs
- Clock: 0 Hz

How it works

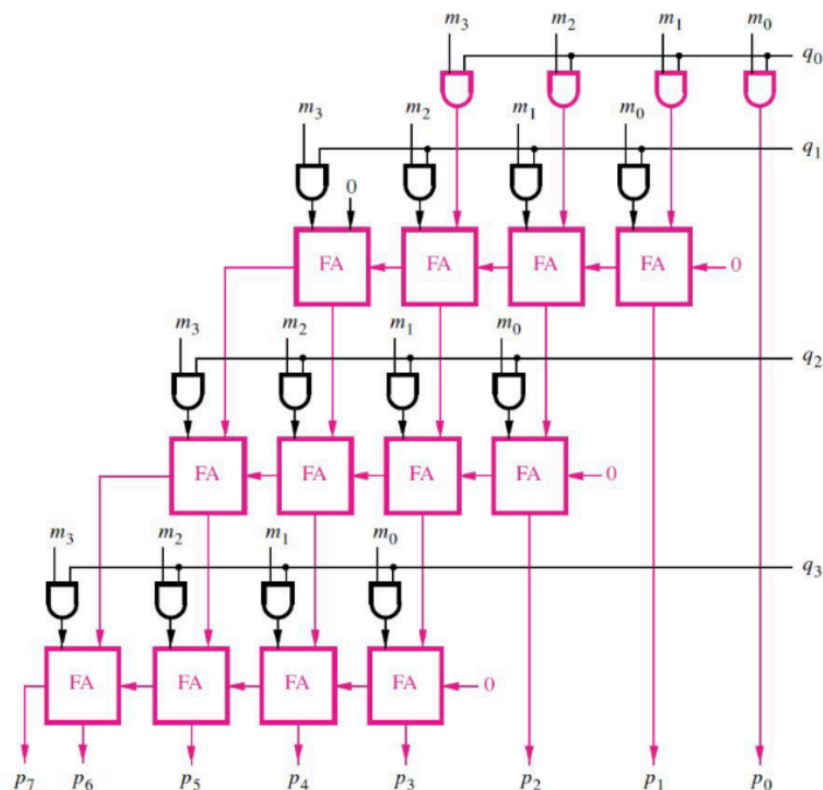


Figure 4: image

Above is a diagram that represents a 4-bit multiplier, which takes in two 4-bit integers and outputs a single 8-bit integer.

This was created using a manual structural design. We utilized a 1-bit full adder module in our implementation.

- AND-Gates are utilized to multiply each bit of input `m` with each bit of input `q`.

- We align partial products diagonally to mimic that of manual binary multiplication.
- We use 1-bit Full Adders to add products and handle carries.
- The outputs of the Full Adders eventually went to the bits of our output `p` which is an 8-bit integer.

How to test

Creating your own test cases:

- Go to the `test` folder and locate `test.py`.
- Edit `test.py` and add your own custom test cases.

```
# Example
# TEST CASE #0 -> 0 * 1
dut.ui_in.value = 0b00000001
await ClockCycles(dut.clk, 1)
assert dut.uo_out.value = 0b00000000
```

- Run the test with `make` and check the tests passed.

- If you've forked the repository
- Commit and push your changes to your forked repository
- Check Github Actions to check if your tests have passed

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	m[0]	p[0]	
1	m1	p1	
2	m2	p2	
3	m[3]	p[3]	
4	q[0]	p[4]	

#	Input	Output	Bidirectional
5	q1	p[5]	
6	q2	p[6]	
7	q[3]	p[7]	

Yared Fente's Tiny Tapeout [14]

- Author: Yared Fente
- Description: Adder Circuit
- GitHub repository
- Wokwi project
- Mux address: 14
- Extra docs
- Clock: 0 Hz

How it works

It performs addition of numbers.

How to test

Use 7 on-switch material to test.

External hardware

7-segment display.

Pinout

#	Input	Output	Bidirectional
0	Input to an XOR	Output from xor2	
1	Input to an XOR	Output from or1	
2	Input to an AND		
3			
4			
5			
6			
7			

Metastable Chip [15]

- Author: Patrick McDermott
- Description: Metastable Multiplier w/ 4 inputs
- GitHub repository
- Wokwi project
- Mux address: 15
- Extra docs
- Clock: 0 Hz

How it works

The metastable multiplier violates the timing delay needed by logic gates to accurately manipulate binary bits the way they're supposed to which causes cases of logic gates outputting a 1 and a 0 at the same time.

How to test

First set all inputs to logic 0 and then set all of them to 1. Observe the metastable of the circuit befure it resolves itself to either a logic 1 or 0.

External hardware

4 switches are used to control the input along with an external clock and reset 4 Leds are used on the output to display which combination of binary you are using at the input

Pinout

#	Input	Output	Bidirectional
0	A0	Out0	
1	A1	Out1	
2	B0	Out2	
3	B1	Out3	
4			
5			
6			
7			

Secret Initial [32]

- Author: Kiarash
- Description: A certain set of inputs will display a secret initial.
- GitHub repository
- Wokwi project
- Mux address: 32
- Extra docs
- Clock: 0 Hz

How it works

Still to be decided, but I am planning on making a certain combination of inputs that will spell the letter F.

How to test

will fill in.

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	

Binary to 7 Segment Display Decoder [33]

- Author: Robert McLintock
- Description: This is a binary to 7 segment display decoder
- GitHub repository
- Wokwi project
- Mux address: 33
- Extra docs
- Clock: 0 Hz

How it works

This decoder uses 4 inputs that represent a binary number determined by the switches. In turn the number will then be decoded into decimal and displayed on the 7 segment display.

How to test

Play around, FLIP A SWITCH or 2, or 3, or 4:)

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		

Tahiti [34]

- Author: Harrison
- Description: Wokwi Design by Harrison
- GitHub repository
- Wokwi project
- Mux address: 34
- Extra docs
- Clock: 0 Hz

How it works

Wokwi design

How to test

Switch buttons on and off

External hardware

Wokwi

Pinout

#	Input	Output	Bidirectional
0	NOTin0	NOTout0	
1	ANDin1	ANDout1	
2	ANDin2		
3	NOT2in3	NOT2out3	
4	HALF ADDER in4		
5		SUMout5	
6		CARRYout6	
7	HALF ADDER in7		

Letter H [35]

- Author: Hannah Thoreson
- Description: Letter H
- GitHub repository
- Wokwi project
- Mux address: 35
- Extra docs
- Clock: 0 Hz

How it works

turning on all switches draws the letter H

How to test

turning on all switches draws the letter H

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	

APTT [36]

- Author: Andy
- Description: lights up depending on action
- GitHub repository
- Wokwi project
- Mux address: 36
- Extra docs
- Clock: 0 Hz

How it works

My project works by, allowing the user to turn on or off switches to turn on lights

How to test

To use my project,

External hardware

I used logic gates and LED'S

Pinout

#	Input	Output	Bidirectional
0			
1	IN1	Out1	
2	IN2		
3	IN3		
4	IN4	Out5	
5	IN5		
6	IN6		
7	IN7	Out7	

Two PFD [37]

- Author: Soumobrata Ghosh
- Description: A zero Dead zone PFD and a basic PFD
- GitHub repository
- Wokwi project
- Mux address: 37
- Extra docs
- Clock: 0 Hz

A zero blind spot phase frequency Detector

How it works

A basic phase frequency Dtector

How to test

If VCO is leading DN would be high and if REF is leading UP would be high

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	REF	UP	
1	VCO	DN	
2	Logic 1	UP2	
3	Ref	Reset	
4	VCO	DN2	
5			
6			
7			

Zero to Nine Display Count [38]

- Author: Mariano
- Description: First Design. Display numbers on seven segment display using flip flop counter.
- GitHub repository
- Wokwi project
- Mux address: 38
- Extra docs
- Clock: 0 Hz

How it works

Binary counter using flip flops connected to clock line. Displays numbers on the seven segment display.

How to test

Use the step button to count from zero to nine.

External hardware

number led array and step button

Pinout

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

Redco [39]

- Author: Shrikrishna Kaje
- Description: Reconfigurable DCO
- GitHub repository
- Wokwi project
- Mux address: 39
- Extra docs
- Clock: 0 Hz

How it works

This design is targetted to be used as

1. Ring oscillator
2. Clock divider
3. Digitally controlled oscillator

Basically it is a ring oscillator which is connected clock divider ckt. The clock divider can be muxed out for different frequencies

1. Ring oscillator Five inverters are used in chain. By shorting out0 pin to in0 and in4, the design can be configured as a ring oscillator Frequency = 1/(5*inverter cell delay)

2. Clock divider network Dff chain is used to introduce clock division. By using combination between s0(in1), s1(in2) , s2(in3) below we can different division at out1 fs - frequency of the clock signal

How to test

Pin description

1. clk (in4) - clock input, input the clock signal to this pin if the chip is to be used as a clock divider, out1 is the output of the clock divider. Short it to in0 and out0 if used as a Oscillator
2. in(in0) - Short it with clk pin and out0 if used as a ring oscillator
3. s0(in1) - LSB of the binary input (DCO input or clock divider select)
4. s1(in2) - second binary input (DCO input or clock divider select)
5. s2(in3) - MSB of the binary input (DCO input or clock divider select)
6. out0 - ring oscillator output
7. out1 - clock divider and DCO

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	in	out0	
1	s0	out1	
2	s1		
3	s2		
4	clk		
5			
6			
7			

Light LED [40]

- Author: Baruas
- Description: Set the switches to get the last led to light up
- GitHub repository
- Wokwi project
- Mux address: 40
- Extra docs
- Clock: 0 Hz

How it works

–> Tiny Tapeout Puzzles

In this puzzle you have to work out how to set the switches to get the last led to light up.

Your friend has been working hard to plan a vacation. (Un)fortunately, they love digital logic, so rather than print you an itinerary, they drew you the following digital circuit.

How to test

Two vacations are possible: can you use the switches to figure out where you'll go, what you'll eat, and what souvenir you'll return with?

External hardware

Switch #	
1	Beach
2	Mountains
3	Ski
4	Swim
5	Ice Cream
6	Fondue
7	Tacos
8	Sun Burn

Pinout

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

Matmul System [41]

- Author: Abarajithan
- Description: Matmul System
- GitHub repository
- HDL project
- Mux address: 41
- Extra docs
- Clock: 0 Hz

How it works

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

How to test

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

External hardware

None

Pinout

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

Tiny Tapeout-Huerta [42]

- Author: Fernando Huerta
- Description: My project displays my initials on a seven segment display
- GitHub repository
- Wokwi project
- Mux address: 42
- Extra docs
- Clock: 0 Hz

How it works

Create and edit a code for the seven segment dislay to create my initials

How to test

Trouble shoot on and of switches in order to utilize seven segment display

External hardware

seven segment display, AND/OR Gates, input an output connections

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	

Light [43]

- Author: Natnael Atnafu
- Description: Enables the light when the correct combination of switches are on
- GitHub repository
- Wokwi project
- Mux address: 43
- Extra docs
- Clock: 0 Hz

How it works

Put the correct combination of switches to make the light turn on

How to test

Turn some switches on or off to see which ones make the light turn on

External hardware

LED light, 7 segment display

Pinout

#	Input	Output	Bidirectional
0	goes to output w same number	comes from input w same number	
1	goes to output w same number	comes from input w same number	
2	goes to output w same number	comes from input w same number	
3	goes to or gate, goes to output 3 and 4	comes from or gate, from 3 or 4	

#	Input	Output	Bidirectional
4	goes to or gate, goes to output 3 and 4	comes from or gate, from 3 or 4	
5	goes to output w same number	comes from input w same number	
6	goes to not gate w same number	comes from not gate w same number	
7			

TinyTapeOut [44]

- Author: Siyem Russom
- Description: Tiny Tapeout
- GitHub repository
- Wokwi project
- Mux address: 44
- Extra docs
- Clock: 0 Hz

How it works

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

How to test

Set sel high and observe that the counter is output on the output pins (data_o) and the bidirectional pins (counter_o). Set sel low and observe that the bidirectional pins are mirrored to the output pins (data_o = data_i)

External hardware

No external hardware used in my project.

Pinout

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

Nathan's chip [45]

- Author: Nathineal
- Description: 1 0 outcome
- GitHub repository
- Wokwi project
- Mux address: 45
- Extra docs
- Clock: 0 Hz

How it works

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

How to test

Set sel high and observe that the counter is output on the output pins (data_o) and the bidirectional pins (counter_o). Set sel low and observe that the bidirectional pins are mirrored to the output pins (data_o = data_i).

External hardware

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

OR gate [46]

- Author: Joe Merriam
- Description: makes seven segment spell J
- GitHub repository
- Wokwi project
- Mux address: 46
- Extra docs
- Clock: 10000 Hz

How it works

We use the OR gate to generate the letter J on the seven segment LED.

How to test

Flip the switches to generate the letter J

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	

project [47]

- Author: ahmad
- Description: diagram
- GitHub repository
- Wokwi project
- Mux address: 47
- Extra docs
- Clock: 0 Hz

How it works

dsign contain four inverts .four inputs are directly connected to outputs and rest to outputs and rest of them are inverted.

How to test

just toggle the inputs.

External hardware

no 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	

D_flipflop_hold_test [64]

- Author: Nicole Ramirez
- Description: hold time violated for D Flip_flop NAND Logic
- GitHub repository
- Wokwi project
- Mux address: 64
- Extra docs
- Clock: 0 Hz

How it works

NAND logic circuit for flipflop violates hold time and set time (theortically) with the resistor

How to test

flipping switches

External hardware

Switches List external hardware used in your project (e.g. PMOD, LED display, etc), if any LEDs(6)

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		

Dipankar's first Wowki design [65]

- Author: Dipankar Shakya
- Description: Certain combination of switches produces a D or O
- GitHub repository
- Wokwi project
- Mux address: 65
- Extra docs
- Clock: 0 Hz

How it works

It uses 4 invertors, XOR, OR gates with 8 inputs, and 6 outputs, producing 6 out of 8 segments that can be lit up or not.

How to test

Switches 1 & 2 turn the top segment on or off depending on if only one switch is on. Switch 3 turns on or off the top right segment, switch 4 turns on or off the bottom right segment, switch 6 turns on or off the bottom left segment, switch 7 turns on or off the bottom segment, switch 8 turns on or off the top left segment.

External hardware

LED 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		
7	IN7		

Bit Counter [66]

- Author: Philip Measor
- Description: Three bit binary counter from 0-7 shown on a segment display
- GitHub repository
- Wokwi project
- Mux address: 66
- Extra docs
- Clock: 0 Hz

How it works

This is a three bit binary counter that is shown on the segment display. Counts from 0 to 8 with binary input.

How to test

Use the input pins (only pins 6-8) to count from 0 to 7.

External hardware

Used segment display and 8 pin switch.

Pinout

#	Input	Output	Bidirectional
0		top segment	
1		right top segment	
2		right bot segment	
3		bottom segment	
4		left bottom segment	
5	bit 3	left top segment	
6	bit 2	middle segment	
7	bit 1	dot segment	

Hamad's design [67]

- Author: Hamad Alwaqayan
- Description: microtapeout design
- GitHub repository
- Wokwi project
- Mux address: 67
- Extra docs
- Clock: 0 Hz

How it works

The project simply uses an AND gate to power A,B,C and D of the 7 segment display, and the other inputs power the remainder of the display.

How to test

Test each input and check which lights trigger on the 7-segment display

External hardware

Just the 7 segment display and switches

Pinout

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

Full bit adder [68]

- Author: Alan
- Description: 2 bit adder
- GitHub repository
- Wokwi project
- Mux address: 68
- Extra docs
- Clock: 1000 Hz

is

How it works

Input 0-1 are input A and B respectively, if A and B are one, the output should be reflected in Output 1 for 2, and if A or B are 1 while the other is, output is reflected in Output0

How to test

turn on A or B and cross check

External hardware

PMOD.LED 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	

Encoder [69]

- Author: Hoang Le
- Description: 8x3 Encoder
- GitHub repository
- Wokwi project
- Mux address: 69
- Extra docs
- Clock: 0 Hz

How it works

This works by utilizing only one AND gate and one OR gate.

How to test

To test, turn on each input except for input 6 and 7, for 6 and 7, only turn one of them on as they uilize an OR gate, meaning one of them have to be on to recieve an output, to ensure its working correctly, it should output the number 8 with a dot at the end.

External hardware

None

Pinout

#	Input	Output	Bidirectional
0			
1			
2			
3			
4	IN4	OUT4	
5	IN5	OUT5	
6	IN6	OUT6	
7	IN7	OUT7	

Encoder [70]

- Author: Mohammad Almutair
- Description: 8x3 decoder
- GitHub repository
- Wokwi project
- Mux address: 70
- Extra docs
- Clock: 0 Hz

How it works

This is a encoder design.

How to test

just toggle the design.

External hardware

‘no external hardware. Encoder

Pinout

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

GDS [71]

- Author: Ben
- Description: 8x3 encoder
- GitHub repository
- Wokwi project
- Mux address: 71
- Extra docs
- Clock: 0 Hz

How it works

My project changes the light of the LED when the switches are turned on and off.

How to test

My circuit has NOT, MUX, and AND gates that turn on the LED's. When you turn on the LED's correctly, it should form the letter A.

External hardware

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

Pinout

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

Big J's Big Circuit [72]

- Author: Jonathan Miller
- Description: 0 and 1: xor, or. 2 and 3: nand, and
- GitHub repository
- Wokwi project
- Mux address: 72
- Extra docs
- Clock: 0 Hz

How it works

my project shows the similarities and differences between nand and and, and also or and xor

How to test

in0 and in1 are connected to the or(out1) and xor(out0). in2 and in3 are connected to and(out3) and nand(out2)

External hardware

This project uses a 7 seg display, but any indicator will work

Pinout

#	Input	Output	Bidirectional
0	in0	xor(in0, in1)	
1	in1	or(in0, in1)	
2	in2	Nand(in2,in3)	
3	in3	And(in2,in3)	
4	in4	in4	
5	in5	in5	
6	in6	in6	
7	in7	in7	

2 Bit Times 2 Bit Plus 4 Bit MAD and 5 Bit Binary to 7 Segment Display [73]

- Author: Nathan
- Description: Bit 1 = A1, Bit 2 = A0, Bit 3 = B1, Bit 4 = B0, Bit 5 = C3, Bit 6 = C2, Bit 7 = C1, Bit 8 = C0
- GitHub repository
- Wokwi project
- Mux address: 73
- Extra docs
- Clock: 0 Hz

How it works

The first 2 bits represent A, the next 2 bits represent B, and the last 4 bits represent C.

Bit 1 = A1, Bit 2 = A0, Bit 3 = B1, Bit 4 = B0, Bit 5 = C3, Bit 6 = C2, Bit 7 = C1, Bit 8 = C0

A is multiplied by B and added to C. The output is shown on the 7 segment display, with the decimal representing "add 16".

How to test

- Set the switches to your desired numbers in binary
- For example A = 11b = 3d, B = 11b = 3d, C = 1111b = 15d
- The result will be shown on the 7 segment display
- For example 3 * 3 + 15 = 24 (showing an 8 and decimal adds 16 (8+16 = 24))

External hardware

There is no external hardware required or used.

#	Input	Output	Bidirectional

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	

AndLogicPass [74]

- Author: James Nguyen
- Description: Password to turn on light
- GitHub repository
- Wokwi project
- Mux address: 74
- Extra docs
- Clock: 0 Hz

How it works

My circuit uses And & NOT logic gates to make a password

How to test

You can test by turning on the 3145 levers

External hardware

No externals...

Pinout

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

Not Good BCD Decoder [75]

- Author: Erik Shimizu
- Description: Supposed to be a binary decoder
- GitHub repository
- Wokwi project
- Mux address: 75
- Extra docs
- Clock: 0 Hz

How it works

BCD Decoder that displays a 0 when pin 1 is off.

How to test

Plug in, and 7-segment should display 0.

External hardware

None.

Pinout

#	Input	Output	Bidirectional
0	IN0	OUT0	
1		OUT1	
2		OUT2	
3		OUT3	
4		OUT4	
5		OUT5	
6		OUT6	
7			

Half Adder [76]

- Author: Janani P Srinivasan
- Description: 1 bit Half Adder
- GitHub repository
- Wokwi project
- Mux address: 76
- Extra docs
- Clock: 0 Hz

How it works

A Half adder is used to perform a single bit addition where the sum and carry is displayed in the output.

The sum of the adder is given by XORing the inputs A and B

The carry of the adder is given by performing an AND operation between A and B

How to test

Half adder can be tested and expressed by

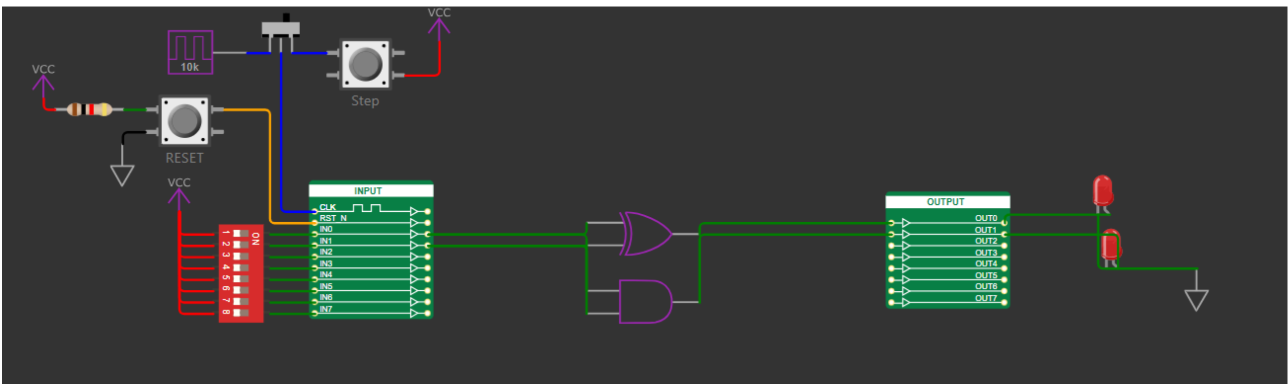
Sum (S) = A XOR B Carry (C) = A . B

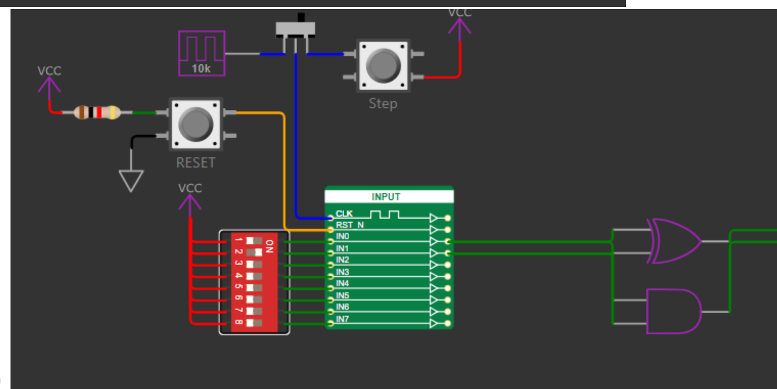
The Truth table is given by

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

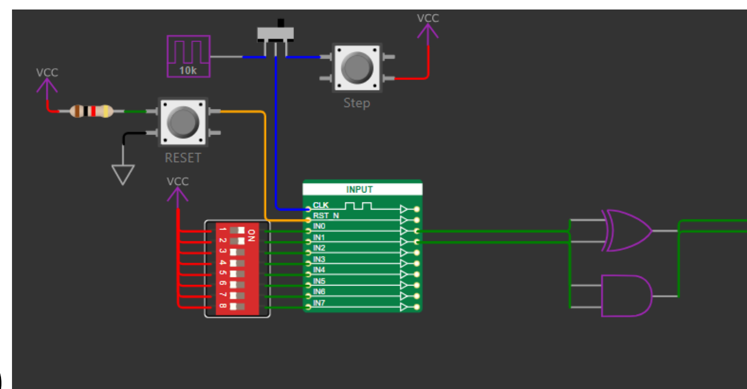
External hardware

Two LED bulbs are connected at the output of sum and carry The LED will blink when the respective values are high





(When A=0 and B=0, S=OFF and C= OFF)



(When A=0/1 or B=0/1, S=ON and C= OFF)
(When A=1 and B=1, S=OFF and C=ON)

Pinout

#	Input	Output	Bidirectional
0	IN0	OUT0	
1	IN1	OUT1	
2			
3			
4			
5			
6			
7			

tinytapeoutkr [77]

- Author: kamila ramirez
- Description: This is the tinytapeout chip project
- GitHub repository
- Wokwi project
- Mux address: 77
- Extra docs
- Clock: 0 Hz

How it works

It turns on a blue and purple light with different connecting factors.

How to test

Turn on the switch that is in the corner of the project

External hardware

LED Light

Pinout

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

Jordan [78]

- Author: Jordan Medina
- Description: 2 bit adder
- GitHub repository
- Wokwi project
- Mux address: 78
- Extra docs
- Clock: 0 Hz

How it works

ADDS TWO INPUTS AND OUTPUTS NUMBER TO TWO LED'S

How to test

INPUT TWO NUMBERS AND VERIFY OUTPUT IS CORRECT

External hardware

DIPSWITCH-8 2 LED'S

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	

My First ASIC [79]

- Author: Michael A. Enright
- Description: My very first ASIC design
- GitHub repository
- Wokwi project
- Mux address: 79
- Extra docs
- Clock: 0 Hz

How it works

This is a simple digital circuit that was designed beginning November 7, 2024 with TinyTapeout and UCSD in La Jolla.

How to test

The test process is very simple and TBD

External hardware

A logic analyzer is helpful.

Pinout

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

GJAA Design [96]

- Author: Guadalupe de Jesus Avelar Anguiano
- Description: To be determine
- GitHub repository
- Wokwi project
- Mux address: 96
- Extra docs
- Clock: 0 Hz

How it works

It uses 7 pins input and give 7 pin output to Dsiplay in a 7 display segement.

How to test

Set the outputs to 7 segment Display

External hardware

7 segemnt 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	

8b10b decoder and multiplier [97]

- Author: Mike Bell
- Description: 8b10b decoder and multiplier (HD version)
- GitHub repository
- HDL project
- Mux address: 97
- Extra docs
- Clock: 0 Hz

What is it?

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

How it works

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

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

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

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

How to test

Send 8b10b encoded data streams, check the outputs.

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

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

External hardware

None required

Pinout

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

Logic Gates [98]

- Author: Adonai Cruz
- Description: Very simple logic gates made with Wokwi
- GitHub repository
- Wokwi project
- Mux address: 98
- Extra docs
- Clock: 0 Hz

How it works

Simple 4 logic gates made with Wokwi

How to test

Use input pins 1-7 and see gates output on pins 1-4

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	IN1	OUT_INV	
1	IN2	OUT_AND	
2	IN3	OUT_OR	
3	IN4	OUT_XOR	
4	IN5		
5	IN6		
6	IN7		
7			

Test Design 1 [99]

- Author: Evan Armoogan
- Description: Test design, not sure what it does yet
- GitHub repository
- HDL project
- Mux address: 99
- Extra docs
- Clock: 0 Hz

How it works

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

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

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

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

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

How to test

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

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

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

Some example test waveforms are attached:

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

External hardware

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

Pinout

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

My First TinyTapeout [100]

- Author: Case Kirk
- Description: This is an 8:3 encoder with a bit to flip default low/high output
- GitHub repository
- Wokwi project
- Mux address: 100
- Extra docs
- Clock: 1 Hz

How it works

This tile design is an active high 7:3 encoder, capable of inverting its output. Reference both the gate diagram and logic table below.

IN 0	IN 1	IN 2	IN 3	IN 4	IN 5	IN 6	OUT 0	OUT 1	OUT 3
1	0	0	0	0	0	0	1	0	0
0	1	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	1	1	0
0	0	0	1	0	0	0	0	0	1
0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	1	0	0	1	1
0	0	0	0	0	0	1	1	1	1

How to test

Provide 7 togglable signals to the input lines (0/3.3V) and connect the first 3 output lines to LEDs. When you toggle each line, you should see the LED change and show its binary representation.

External hardware

TTBoard and LED's should do just fine.

Pinout

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

Decimation Filter for Incremental and Regular Delta-Sigma Modulators [101]

- Author: Andrea Murillo Martinez & Jaeden Chang
- Description: Decimation filter that efficiently reduces oversampled data from incremental and regular delta-sigma modulators, while preserving signal accuaracy.
- GitHub repository
- HDL project
- Mux address: 101
- Extra docs
- Clock: 50000000 Hz

Overview

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

Specifications

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

Mode Selection

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

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

How It Works

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

Operation

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

1. **Incremental DSM Mode (Input 2 Low):**
   - Use the ADC's oversampling frequency as the input clock for the filter.
   - Set the main reset signal to match the desired decimation rate.
   - For example, with a 50 MHz ADC frequency, setting the reset signal to 25 MHz achieves a decimation factor of 2.
2. **Regular DSM Mode (Input 2 High):**
   - The default decimation factor is set to 16.
   - For customized decimation factors, follow the configuration steps in Incremental DSM mode.

Testing Procedure

1. **Hardware Setup:**
   - Connect a 1-bit ADC output to Input 1.
   - Set Input 2 to low for Incremental DSM or high for Regular DSM.
2. **Verification:**
   - **Incremental DSM:** Set Input 2 low, connect a clock to the reset input, and observe decimated output changes.

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

Output Configuration

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

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

Compatibility

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

Pinout

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

1st [102]

- Author: HUSSAIN
- Description: circuit
- GitHub repository
- Wokwi project
- Mux address: 102
- Extra docs
- Clock: 0 Hz

How it works

Design contains four inverters. Four inputs are directly connected to outputs and rest of them are inverted.

How to test

Just toggle the inputs then it will be reflecgted in the output.

External hardware

Do not have any external harware.

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	

adder-accumulator [103]

- Author: Damir Gazizullin, Owen Golden
- Description: 8-bit ripple adder and the complementary accumulator register
- GitHub repository
- HDL project
- Mux address: 103
- Extra docs
- Clock: 50000000 Hz

How it works

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

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

IO Table: Accumulator (A) Register

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

IO Table: ALU (Adder/Subtractor)

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

Tests and Expected Functionality

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

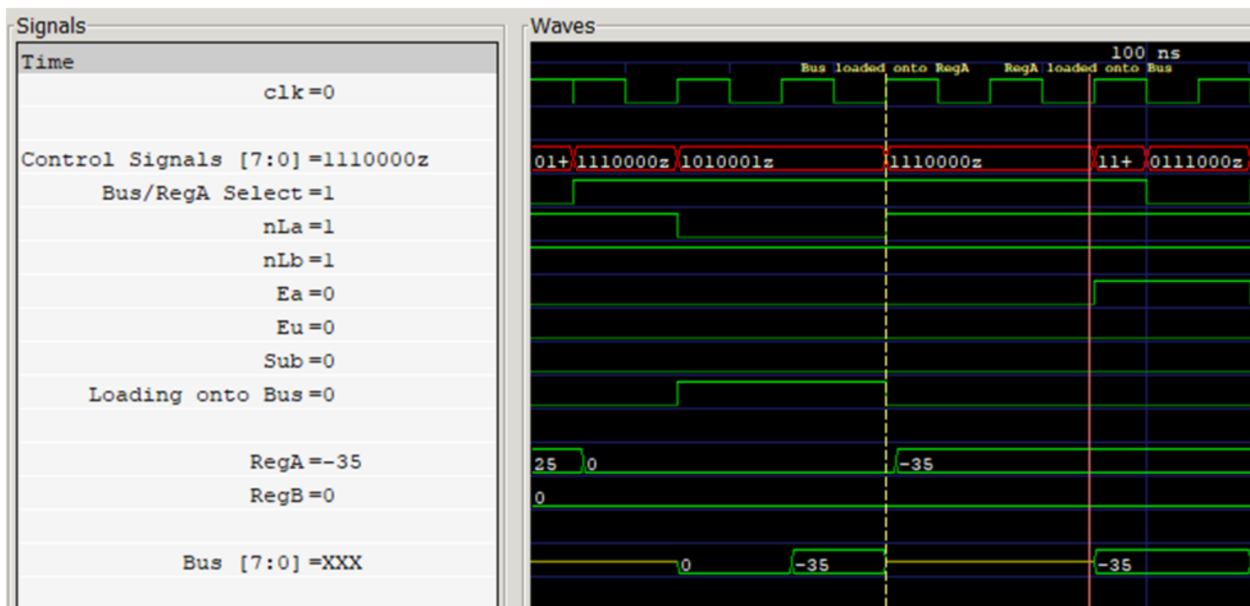


Figure 1: Accumulator Load onto bus and push onto bus

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

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

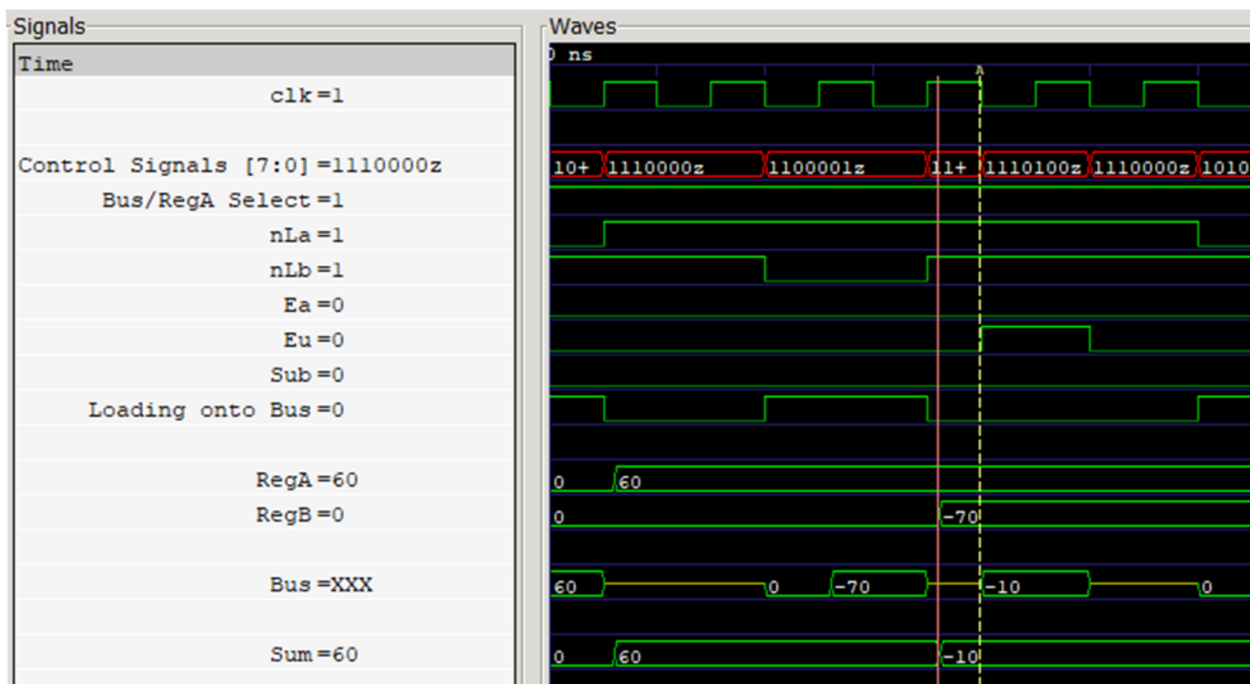


Figure 2: Addition and Output onto Bus

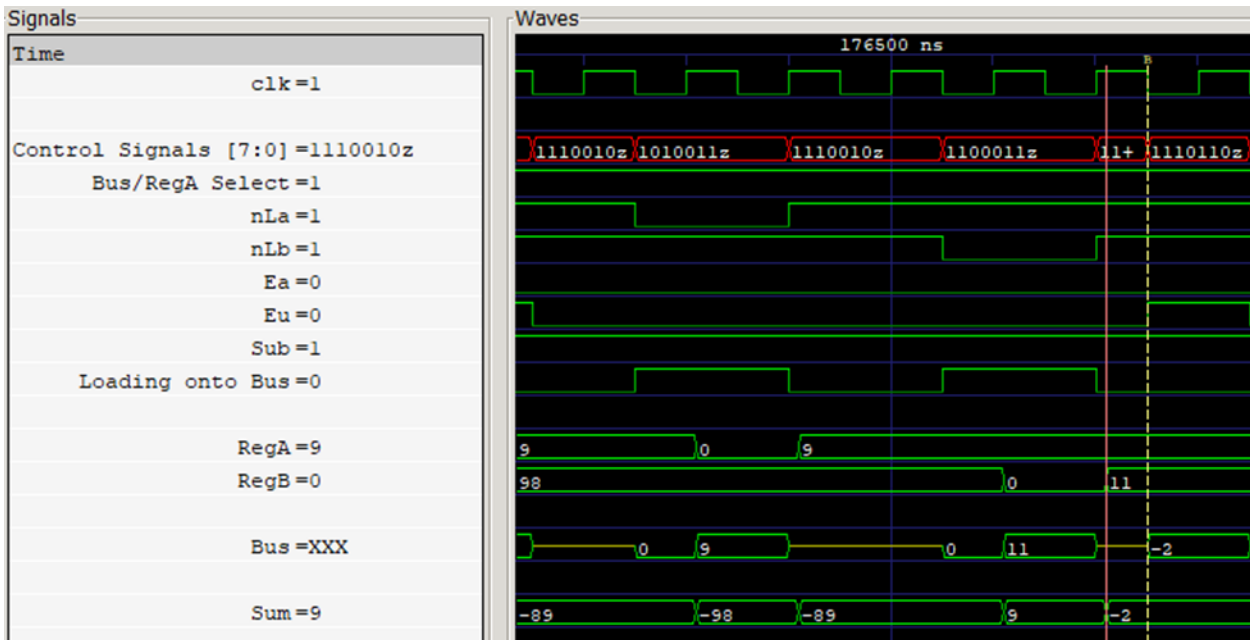


Figure 3: Subtraction and Output onto Bus

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

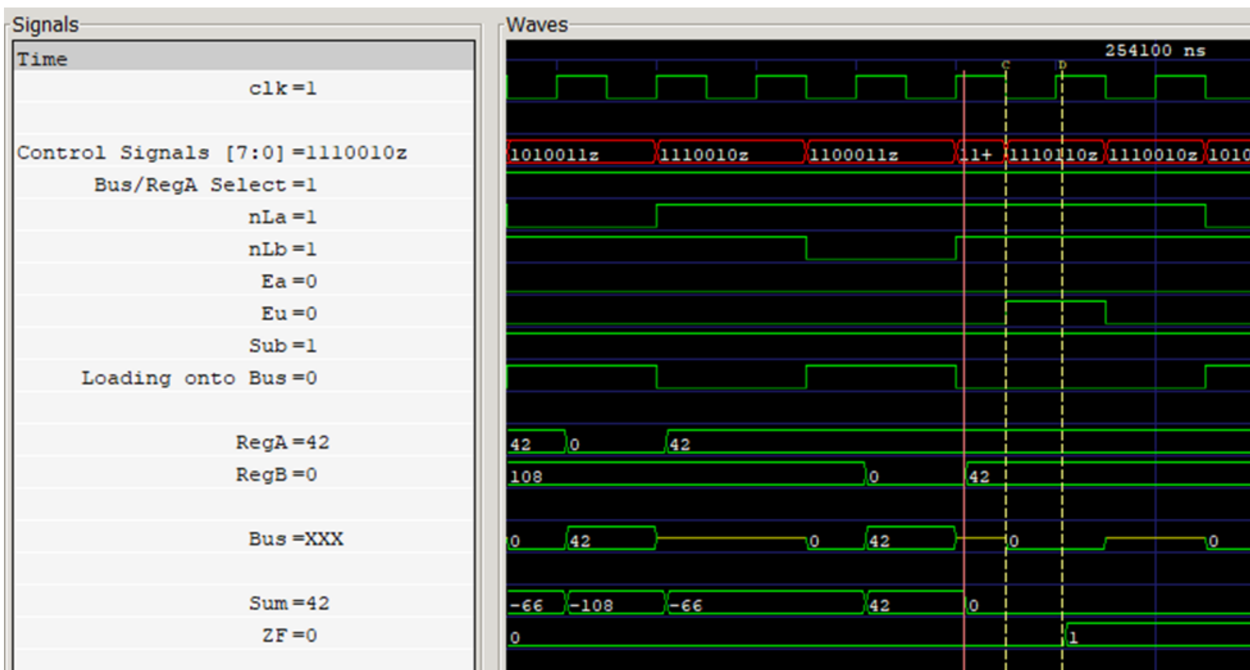


Figure 4: Zero Flag Functionality of Adder

Description of Testbenches

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

Adder Tests:

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

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

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

Accumulator Tests:

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

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

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

Pinout

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

JCB First WOKWI Design [104]

- Author: Jared Bruce
- Description: Display Letter When A Code Is Entered
- GitHub repository
- Wokwi project
- Mux address: 104
- Extra docs
- Clock: 0 Hz

How it works

It uses not/and gates to display the first character of my name.

How to test

It should display a J when entering the code 6531

External hardware

Requires Single digit Seven-Segment display for displaying output

Pinout

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

ECE 298A 8-Bit CPU Control Block [105]

- Author: Siddharth Nema & Gerry Chen
- Description: Generates the control signals required for other CPU sub blocks
- GitHub repository
- HDL project
- Mux address: 105
- Extra docs
- Clock: 50000000 Hz

How it works

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

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

Supported Instructions

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

Instruction Notes

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

Control Signal Descriptions

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

Sequencing Details

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

Instruction Micro-Operations

Stage	**HLT**	**NOP**	**STA**	**JMP**
T0	Ep, nLma	Ep, nLma	Ep, nLma	Ep, nLma

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

Stage	**LDA**	**ADD**	**SUB**	**OUT**
T0	Ep, nLma	Ep, nLma	Ep, nLma	Ep, nLma
T1	Cp	Cp	Cp	Cp
T2	nCE, nLi	nCE, nLi	nCE, nLi	nCE, nLi
T3	nEi, nLma	nEi, nLma	nEi, nLma	Ea, nLo
T4	nCE, nLa	nCE, nLb	nCE, nLb	-
T5	-	Eu, nLa	Su, Eu, nLa	-

Instruction Micro-Operations Notes

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

IO Table

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

IO Table Notes

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

How to test

The control block can be tested by:

- Providing an opcode through the `ui_in[3:0]` input pins.
- Monitoring the `uo_out[7:0]` and `uio_out[7:0]` output pins for the control signals and halt status
- For a given opcode, follow its Instruction Micro-Operation table to validate the control signal sequences
- Consider using a logic analyzer to generate a waveform and analyze the stages, or slow down the clock to manually observe the control signals at various times

Pinout

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

Logic Gates 7-Segment Display [106]

- Author: Abdul Karim Tamim
- Description: Logic Gates (AND, NAND, OR, XOR, NOT) Control the Output of a 7-Segment Display
- GitHub repository
- Wokwi project
- Mux address: 106
- Extra docs
- Clock: 0 Hz

How it works

The project uses Logic Gates (AND, NAND, OR, XOR, NOT) to Control the Output of a 7-Segment Display

How to test

The project uses a Switch containing 8 electrical switches that control the input depending on which switch is on or off. Then the input will go to the logic gate which will result in an output that will be displayed on the 7-segment display

External hardware

Switch, Buttons

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	

LFSR Encrypter [107]

- Author: Mitchell Tansey
- Description: Simple LFSR data encrypter. Takes data in and xor's it with an lfsr output to encrypt data.
- GitHub repository
- HDL project
- Mux address: 107
- Extra docs
- Clock: 0 Hz

How it works

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

How to test

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

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	ui_in[0]	ui_out[0]	uio_out[0]
1	ui_in1	ui_out1	uio_out1
2	ui_in2	ui_out2	uio_out2
3	ui_in[3]	ui_out[3]	uio_out[3]
4	ui_in[4]	ui_out[4]	uio_out[4]
5	ui_in[5]	ui_out[5]	uio_out[5]
6	ui_in[6]	ui_out[6]	uio_out[6]
7	ui_in[7]	ui_out[7]	uio_out[7]

BadeTP [108]

- Author: Brandon D
- Description: Makes the letter b with a period at the end.
- GitHub repository
- Wokwi project
- Mux address: 108
- Extra docs
- Clock: 0 Hz

How it works

How this work when the switches 1 or 2 either both on or just one the light on the bottom right will turn on but if both are off the light will turn off, swithes 3 and 4 both need to be on in order to turn on the light at the bottom if either are off then the light turns off. The rest of the switches control their own light. 5 controls the bottom left light, 6 controls top left, 7 controls the light in the middle, and 8 controls the period. If they are all on then it creates this "b."

How to test

In order to test it you can turn on and off each individual switch. As I said in the how it works the switches 1 or 2 control the bottom right light either or can be on and switches 3 and 4 both need to be on in order to turn on the light at the bottom.

External hardware

- Led display
- The "or" Switch
- The "and" Switch

Pinout

#	Input	Output	Bidirectional
0	IN0		
1	IN1		
2	IN2	OUT2	
3	IN3	OUT3	

#	Input	Output	Bidirectional
4	IN4	OUT4	
5	IN5	OUT5	
6	IN6	OUT6	
7	IN7	OUT7	

SkyKing Demo [109]

- Author: Nicklaus Thompson
- Description: Types some text over an image of a plane flying into the sunset
- GitHub repository
- HDL project
- Mux address: 109
- Extra docs
- Clock: 25200000 Hz

How it works

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

How to test

Runs automaticaly.

External hardware

VGA PMOD on UO.

Pinout

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

Lynn's TinyTapeout Design [110]

- Author: Lynn Francis
- Description: Blinking Letter E
- GitHub repository
- Wokwi project
- Mux address: 110
- Extra docs
- Clock: 2 Hz

How it works

If switch pins 1 and 7 are powered and the clock is on, the sevseg display will flash the letter E

How to test

Power switch pins 1 and 7 and the clock.

External hardware

On your signal outputs, hook up either a 7 segment display or leds.

Pinout

#	Input	Output	Bidirectional
0	AND3:A	sevseg1:A	
1	OUT1	sevseg1:B	
2	OUT2	sevseg1:C	
3		sevseg1:D	
4		sevseg1:E	
5		sevseg1:F	
6	AND2:A	sevseg1:G	
7		sevseg1:DP	

Two LIF Neurons with STDP Learning [111]

- Author: Sebastian Hernandez
- Description: A compact spiking neural network implementation featuring: - Two Leaky Integrate-and-Fire (LIF) neurons connected via plastic synapse - Spike-timing-dependent plasticity (STDP) for dynamic weight adjustment - 8-bit fixed-point arithmetic for state and weight representation - Real-time monitoring of spikes and synaptic weight
- GitHub repository
- HDL project
- Mux address: 111
- Extra docs
- Clock: 50000000 Hz

How it works

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

Two LIF Neurons:

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

STDP Synapse:

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

Implementation Features:

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

How to test

he design can be tested in several ways:

Basic Functionality:

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

Spike Generation Test:

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

STDP Learning:

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

External hardware

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

Logic Analyzer:

Monitor spike timing Track synaptic weight changes Verify state transitions

Signal Generator (optional):

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

Target Performance

The design aims to achieve:

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

Resource Usage

The implementation utilizes:

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

Future Improvements

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

Pinout

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

4-bit-multiplier [128]

- Author: Eric Cheung, Bethel Sisay
- Description: 4X4 array multiplier
- GitHub repository
- HDL project
- Mux address: 128
- Extra docs
- Clock: 0 Hz

How it works

implements a 4x4 array multiplier, as shown in the schematic below

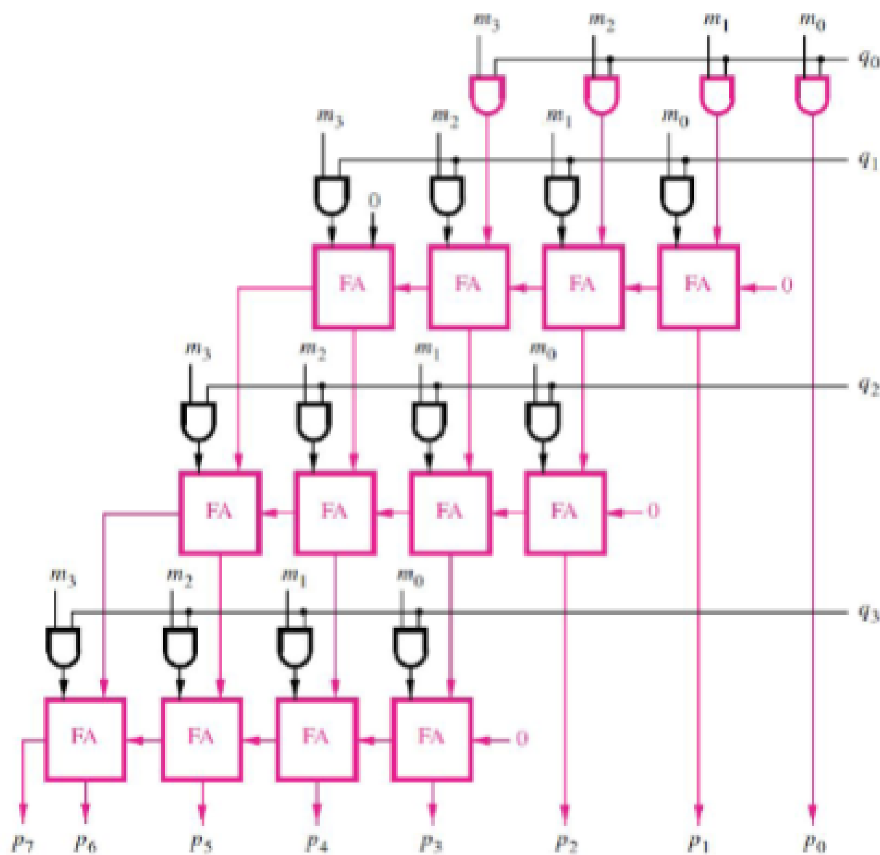


Figure 5: Block Diagram

How to test

follow test/README.md

use test/test.py to add test cases

External hardware

None

Pinout

#	Input	Output	Bidirectional
0	m[0]	p[0]	
1	m1	p1	
2	m2	p2	
3	m[3]	p[3]	
4	q[0]	p[4]	
5	q1	p[5]	
6	q2	p[6]	
7	q[3]	p[7]	

ece2204_4x4_mult [129]

- Author: Eric Wang, Alan Zhu
- Description: 4x4 structural array multiplier
- GitHub repository
- HDL project
- Mux address: 129
- Extra docs
- Clock: 0 Hz

How it works

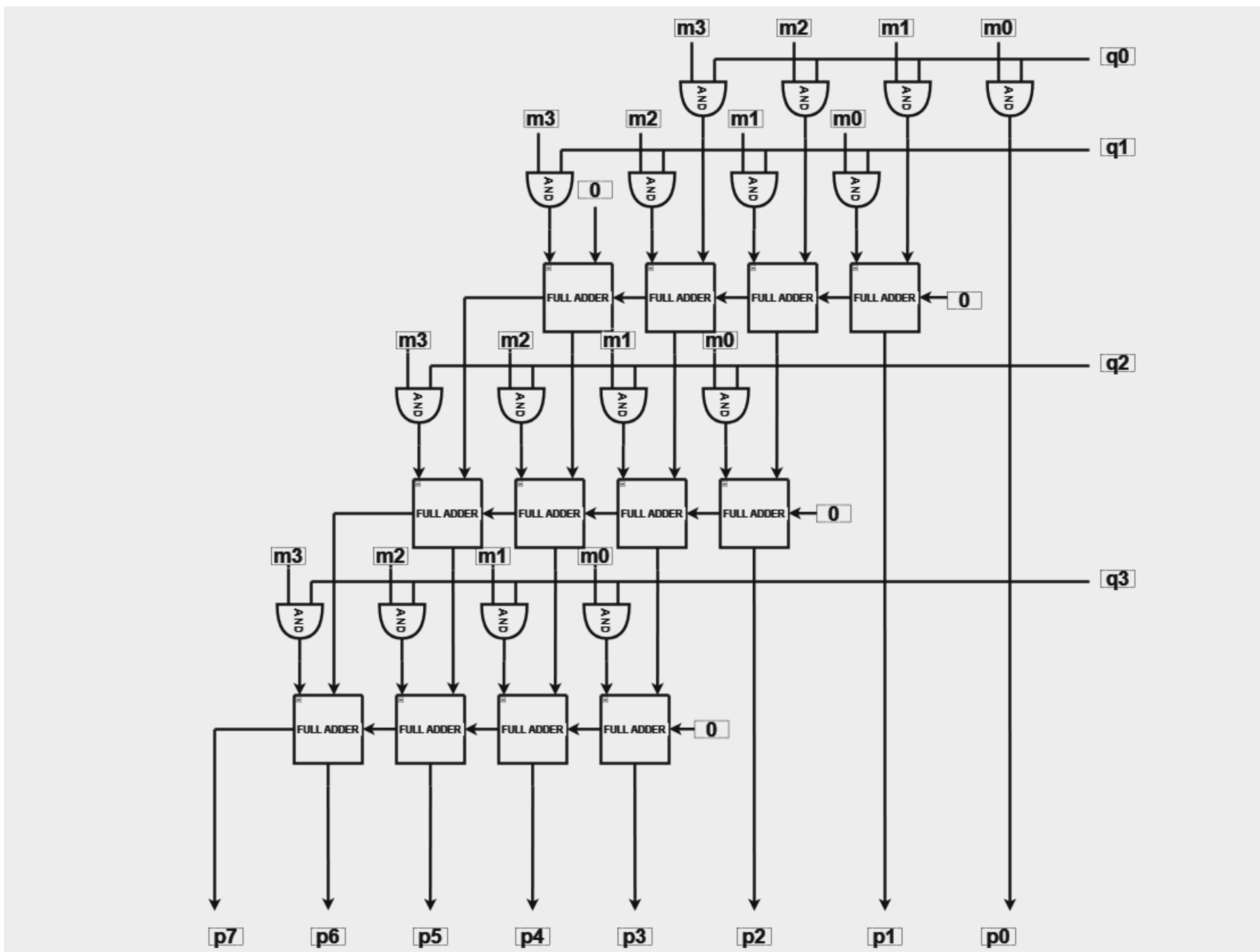


Figure 6: 4x4mult

A 4x4 array multiplier multiplies two 4-bit binary numbers by arranging AND gates and adders in a grid-like pattern. Each bit of the first 4-bit number (multiplicand) is ANDed with each bit of the second 4-bit number (multiplier), creating 16 "partial products." As shown in the image, each bit of "q" is ANDed with each bit of "m" for four rows. These products are organized in rows, with each row shifted to the left to

represent the binary place values for multiplication. Each column of partial products is then added vertically using full adders where columns without carries remain the same and others pass carry bits to the next column. The output for each column results in each bit of "p", which is the 8-bit product in this case.

How to test

Input a 4-bit number for the input "q" and a 4-bit number for the input "m". The outcome of the array will be an 8-bit binary product of the two input numbers.

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	q[0]	p[0]	
1	q1	p1	
2	q2	p2	
3	q[3]	p[3]	
4	m[0]	p[4]	
5	m1	p[5]	
6	m2	p[6]	
7	m[3]	p[7]	

my_4bit_multiplier [130]

- Author: Terry Mu, Omobolaji Alabi
- Description: This is our 4-bit multiplier
- GitHub repository
- HDL project
- Mux address: 130
- Extra docs
- Clock: 0 Hz

How it works

It accepts two 4-bit unsigned integer: m and q, and calculate the product, p.

How to test

Provide several pairs of m and q (for example, (0, 1), (12, 13), (3, 4), (0, 0), (15, 15)). Then check whether the output is equal to the product of the two numbers.

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	q[0]	p[0]	
1	q1	p1	
2	q2	p2	
3	q[3]	p[3]	
4	m[0]	p[4]	
5	m1	p[5]	
6	m2	p[6]	
7	m[3]	p[7]	

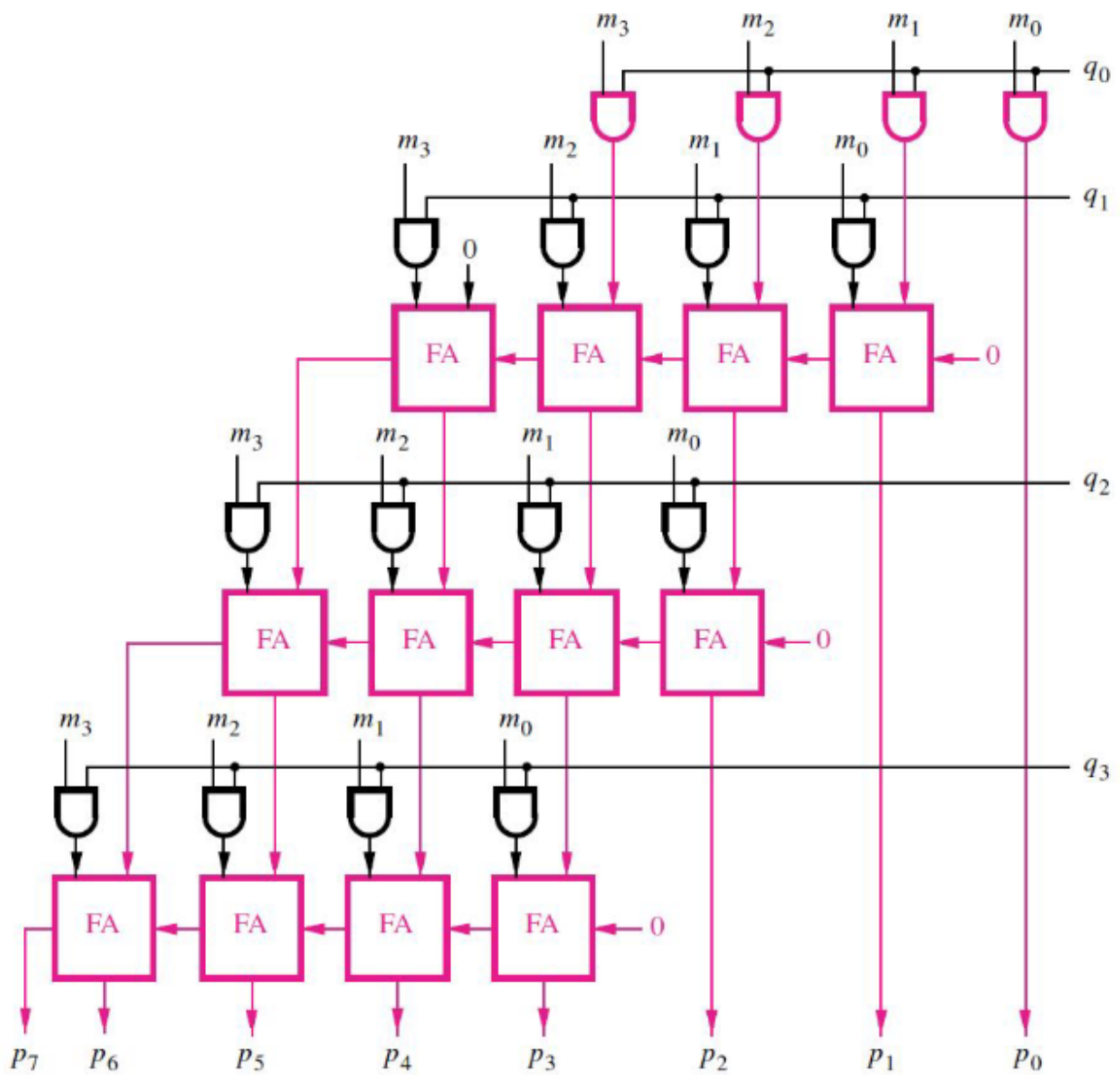


Figure 1: 4×4 Array Multiplier

Figure 7: alt text

T3 (Tiny Ternary Tapeout) [131]

- Author: Arnav Sacheti & Jack Adiletta
- Description: Ternary Matmul Processor
- GitHub repository
- HDL project
- Mux address: 131
- Extra docs
- Clock: 50000000 Hz

Tiny Ternary Tapeout Project Documentation

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

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

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

Loading the Weights (`tt_um_load.v`):

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

Matrix Multiplication (`tt_um_mult.v`):

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

Pipelined Architecture:

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

Outputting Results:

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

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

Step 1: Define a Ternary Array

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

Let's create a sample 4x2 ternary array:

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

Step 2: Define the Input Vector

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

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

Step 3: Compute the Output without Matrix Multiplication

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

For each column in the ternary array:

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

Step 4: Calculate Each Column's Output

Let's compute each column separately:

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

  Sum of Column 1: ($2 + 1 + 0 + 0 = 3$)

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

  Sum of Column 2: ($0 - 1 - 3 + 0 = -4$)

Final Output Vector

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

$$\text{Output} = \begin{bmatrix} 3 & -4 \end{bmatrix}$$

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

1. **Setup**:

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

Pinout

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

Hybrid_Adder_8bit [132]

- Author: James Xie, Cameron Bedard
- Description: 8-bit hybrid adder (using CLA and KSA)
- GitHub repository
- HDL project
- Mux address: 132
- Extra docs
- Clock: 0 Hz

How it works

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

How to test

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

External hardware

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

Pinout

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

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

3 Neuron ALIF [133]

- Author: Andrew Smith
- Description: TODO
- GitHub repository
- HDL project
- Mux address: 133
- Extra docs
- Clock: 0 Hz

How it works

3 Adaptive Leaky Integrate and Fire Neurons

1. Receives an 8-bit input signal (ui_in) with small offset variations

2. Processes the signal through the LIF model which simulates biological neuron behavior by:

- Integrating (accumulating) input current over time
- Applying a leak factor to gradually decrease membrane potential
- Generating a spike when membrane potential exceeds threshold
- Adjusting a moving threshold based on periods of past inputs

3. Outputs:

- Spike signals on uio_out[7:5]:

    - uio_out[7]: Neuron 1 spike output
    - uio_out[6]: Neuron 2 spike output
    - uio_out[5]: Neuron 3 spike output

- Internal state of Neuron 1 on uo_out[7:0] for debugging/testing

How to test

1. Basic Functionality Test:

- Apply a constant input value through ui_in
- Monitor uio_out[7:5] to observe spike patterns
- Check uo_out to monitor Neuron 1's internal state

2. Threshold Response Test:

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

3. Reset Test:

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

External hardware

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

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

Pinout

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

8-bit Carry Look-Ahead Adder [134]

- Author: Seongwan Jeon and Michael Zeng
- Description: Fast 8-bit adder
- GitHub repository
- HDL project
- Mux address: 134
- Extra docs
- Clock: 0 Hz

How it works

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

Figure 8: image

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

By calculating the carry bits by using combinatorial logic, a CLA is able to calculate all of the carry bits of the sum without relying on sequential operations, unlike a ripple carry adder. The main time complexity of the ripple carry adder is based on the

$$C_1 = G_0 + P_0 \cdot C_{in}$$
$$C_2 = G_1 + P_1 \cdot C_1 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_{in}$$
$$C_3 = G_2 + P_2 \cdot C_2 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_{in}$$
$$C_4 = G_3 + P_3 \cdot C_3 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_{in}$$

Figure 9: image

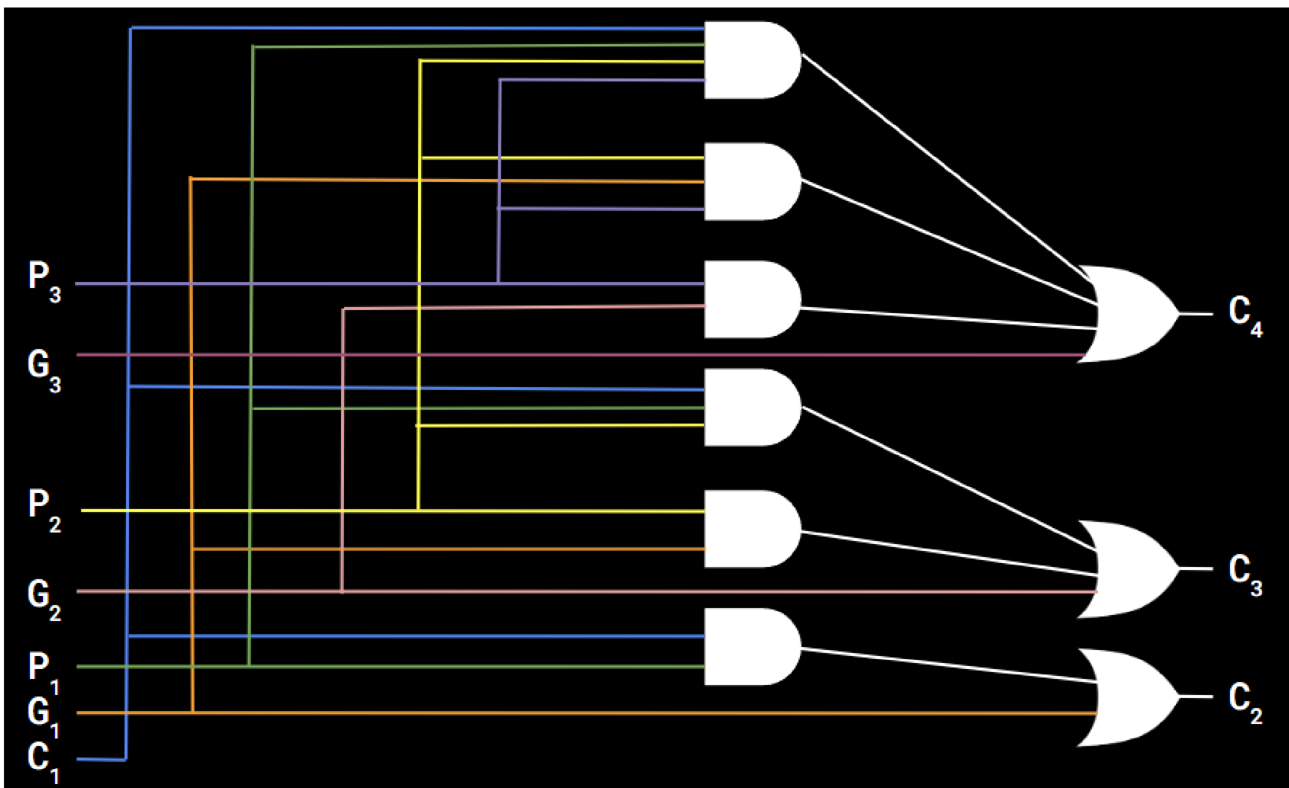


Figure 10: image

implementation of the last (and largest) AND gate of the most significant carry bit in the combinatorial equation. This AND gate has n+1 inputs, where n is the bits of the input. The implementation of multiple input AND gates in hardware consists of multiple smaller input AND gates organized in a tree structure, which inherently has a logarithmic time complexity. This logic extends to the CLA which possesses a logarithmic time complexity, and it makes CLAs viewed as one of the fastest implementations of digital adders due to its combinatorial nature. CLAs that calculate large bit-widths can also be designed by using multiple CLAs with smaller bit-widths in parallel to calculate intermediate values. This implementation using a tree structure of adders allows CLAs to also possess a modular design which can be scaled up to handle large bit-widths. However, this tree-like design is an implementation that other parallel prefix adders such as the Kogge-Stone adder utilize to a greater effect. Although CLAs are praised for their speed, it comes at the cost of a large area, as the components needed to calculate the carry bits for larger bit-widths become exponentially larger.

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

How to test

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

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

External hardware

No external hardware needed.

Pinout

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

#	Input	Output	Bidirectional
4	a[4]	sum[4]	b[4]
5	a[5]	sum[5]	b[5]
6	a[6]	sum[6]	b[6]
7	a[7]	sum[7]	b[7]

2bit adder [135]

- Author: Ya-Chin, Hu
- Description: shows the sum of in0 and in1
- GitHub repository
- Wokwi project
- Mux address: 135
- Extra docs
- Clock: 0 Hz

How it works

Takes input from in0 and in1. Assumes bit = 1 if input is ON, else bit = 0. Shows sum as decimal integer.

How to test

Connect in0 and in1 to ON-OFF switches.

External hardware

One 7 pad display, connected to output as in template.

Pinout

#	Input	Output	Bidirectional
0	in0	out0	
1	in1	out1	
2		out2	
3		out3	
4		out4	
5		out5	
6		out6	
7		out7	

RISC-V Mini [136]

- Author: RickGao
- Description: RISC-V Mini 8 Bit
- GitHub repository
- HDL project
- Mux address: 136
- Extra docs
- Clock: 100000 Hz

How it works

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

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

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

How to test

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

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

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

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

Instructions List

R-Type

Name | funct3 [15:13] | funct2 [12:11] | rs2 [10:8] | rs1 [7:5] | rd [4:2] | Opcode(00)

AND | 000 | 00 | XXX | XXX | XXX | Opcode(00)

OR | 001 | 00 | XXX | XXX | XXX | Opcode(00)

ADD | 010 | 00 | XXX | XXX | XXX | Opcode(00)

SUB | 011 | 00 | XXX | XXX | XXX | Opcode(00)

XOR | 001 | 01 | XXX | XXX | XXX | Opcode(00)

SLT | 111 | 00 | XXX | XXX | XXX | Opcode(00)

I-Type

Name | funct3 [15:13] | Imm [12:8] (5-bit unsigned) | rs1 [7:5] | rd [4:2] | Opcode(01)

SLL | 100 | XXXXX | XXX | XXX | Opcode(01)

SRL | 101 | XXXXX | XXX | XXX | Opcode(01)

SRA | 110 | XXXXX | XXX | XXX | Opcode(01)

ADDI | 010 | XXXXX | XXX | XXX | Opcode(01)

SUBI | 011 | XXXXX | XXX | XXX | Opcode(01)

L-Type

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

S-Type

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

B-Type

Name | funct3 [15:13] | funct2 [12:11] | rs2 [10:8] | rs1 [7:5] | 000 | Opcode(11)

BEQ | 011 | 00 | XXX | XXX | 000 | Opcode(11)

BNE | 011 | 10 | XXX | XXX | 000 | Opcode(11)

BLT | 111 | 00 | XXX | XXX | 000 | Opcode(11)

External hardware

No External Hardware

Pinout

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

4-1 mux [137]

- Author: zhengfeng wu
- Description: 4-1 mux
- GitHub repository
- Wokwi project
- Mux address: 137
- Extra docs
- Clock: 0 Hz

4-1 mux, a,b,c,d as four inputs, use s1,s2 to select one to output

Pinout

#	Input	Output	Bidirectional
0	a		
1	b		
2	c		
3	d	out	
4	s1		
5	s2		
6			
7			

8-bit carry-skip [138]

- Author: Dennis_Du
- Description: two 8-bit input adder
- GitHub repository
- HDL project
- Mux address: 138
- Extra docs
- Clock: 0 Hz

How it works

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

How to test

To test the carry-skip adder:

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

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

External hardware

No external hardware is required for this project.

Pinout

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

STDP Circuit [139]

- Author: Mariah Regalado
- Description: STDP Circuit using a trace to model exponential behavior
- GitHub repository
- HDL project
- Mux address: 139
- Extra docs
- Clock: 0 Hz

How it works

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

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

How to test

I am stil working on it.

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	Input Current Bit [0]	State Variable bit[0]	

#	Input	Output	Bidirectional
1	Input Current Bit 1	State Variable bit1	
2	Input Current Bit 2	State Variable bit2	
3	Input Current Bit [3]	State Variable bit[3]	
4	Input Current Bit [4]	State Variable bit[4]	
5	Input Current Bit [5]	State Variable bit[5]	
6	Input Current Bit [6]	State Variable bit[6]	
7	Input Current Bit [7]	State Variable bit[7]	Spike bit

4 bit array multiplier [140]

- Author: Abdulrahman Albaoud, Joe Leighthardt
- Description: Takes in 2 four-bit inputs and multiplies them into one eight-bit value
- GitHub repository
- HDL project
- Mux address: 140
- Extra docs
- Clock: 0 Hz

How it works

Takes in one 8bit binary array and breaks it into two 4bit arrays. It then multiplies these arrays by each other.

How to test

Use various numbers to test the multiplicative values.

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	q[0]	p[0]	
1	q1	p1	
2	q2	p2	
3	q[3]	p[3]	
4	m[0]	p[4]	
5	m1	p[5]	
6	m2	p[6]	
7	m[3]	p[7]	

instrumented_ring_oscillator [141]

- Author: Jeremy Mickelsen
- Description: A ring oscillator with a selectable number of stages and initial state.
- GitHub repository
- HDL project
- Mux address: 141
- Extra docs
- Clock: 0 Hz

How it works

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

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

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

How to test

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

External hardware

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

Pinout

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

Array Multiplier [142]

- Author: Noah Rivera & Filip Bukowski
- Description: 4x4 Structural Array Multiplier
- GitHub repository
- HDL project
- Mux address: 142
- Extra docs
- Clock: 0 Hz

4x4 Array Multiplier Block Diagram

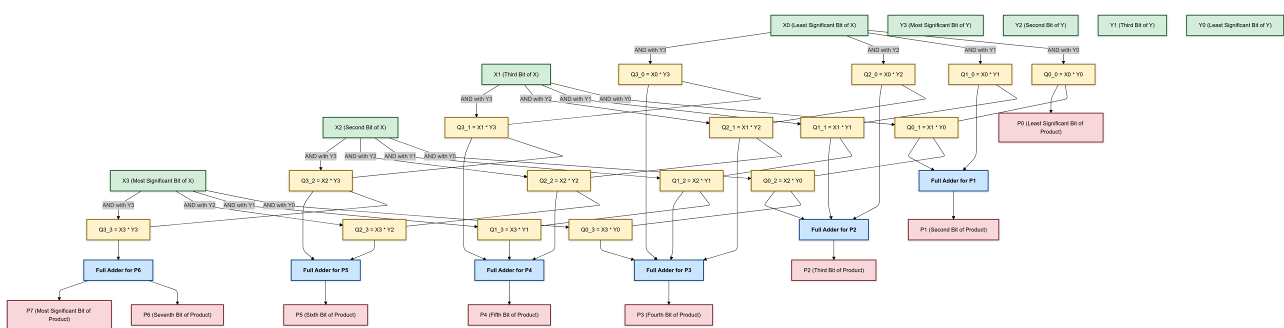


Figure 11: 4x4 Array Multiplier Block Diagram

How it works

The 4x4 array multiplier operates by multiplying two 4-bit binary numbers to produce an 8-bit number. The process is performed by generating partial products through a series of AND operations between each bit of the first 4-bit binary number (X) and the second 4-bit binary number (Y). This results in a total of 16 partial products, which correspond to the multiplicative contributions of each bit in X with each bit in Y. Once the partial products are generated, they are aligned according to their significance in the binary numeral system, to achieve the proper placement of each product. The result is achieved by summing up these aligned partial products by using a series of full adders to manage the addition and carry bits.

How to test

To test the 4x4 array multiplier, a variety of 4-bit binary inputs need to be created for both multiplicates X and Y. After establishing various proper inputs, the selected binary numbers can be entered into the multiplier using a proper simulation environment.

When the inputs are assigned, the simulation can be run, and the product of the two 4-bit binary inputs can be achieved.

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	q[0]	p[0]	
1	q1	p1	
2	q2	p2	
3	q[3]	p[3]	
4	m[0]	p[4]	
5	m1	p[5]	
6	m2	p[6]	
7	m[3]	p[7]	

Linear Feedback Shift Register [143]

- Author: Steve Jenson <stevej@gmail.com>
- Description: An implementation of a Linear Feedback Shift Register for TT09
- GitHub repository
- HDL project
- Mux address: 143
- Extra docs
- Clock: 0 Hz

How it works

Read the `ui_out` pins, each read should be different than the last. To reset the shift register, reset the chip, or set the 'write_enable' pin high after offering a value on `ui_in` as a seed.

How to test

Read several bytes from `ui_in`, they should each be different.

External hardware

No external hardware needed other than to read the pins.

Pinout

#	Input	Output	Bidirectional
0	Seed Bit 1	LFSR Bit 1	Write Enable
1	Seed Bit 2	LFSR Bit 2	
2	Seed Bit 3	LFSR Bit 3	
3	Seed Bit 4	LFSR Bit 4	
4	Seed Bit 5	LFSR Bit 5	
5	Seed Bit 6	LFSR Bit 6	
6	Seed Bit 7	LFSR Bit 7	
7	Seed Bit 8	LFSR Bit 8	

Frequency Encoder and Decoder [160]

- Author: Miguel Robles
- Description: Simple implementation of an 8-bit frequency encoder/decoder for a 1 bit frequency channel
- GitHub repository
- HDL project
- Mux address: 160
- Extra docs
- Clock: 10000000 Hz

How it works

Takes an 8-bit input voltage and treats it as a current injection to a LIF neuron

How to test

Do something

External hardware

NA

Pinout

#	Input	Output	Bidirectional
0	Input frequency channel for decoder OR input bit for encoder [0]	LSB output of decoder [0] OR output frequency channel for encoder	Input selector bit to choose between encoder or decoder
1	Input encoder bit 1	Output encoder bit 1	
2	Input encoder bit 2	Output encoder bit 2	

#	Input	Output	Bidirectional
3	Input encoder bit [3]	Output encoder bit [3]	
4	Input encoder bit [4]	Output encoder bit [4]	
5	Input encoder bit [5]	Output encoder bit [5]	
6	Input encoder bit [6]	Output encoder bit [6]	Input configuration bit for encoder sample rate [0]
7	Input encoder bit [7]	Output encoder bit [7]	Input configuration bit for encoder sample rate 1

TT Test [161]

- Author: Austin
- Description: 8-bit shift register.
- GitHub repository
- Wokwi project
- Mux address: 161
- Extra docs
- Clock: 0 Hz

How it works

This just works2

How to test

This just works2

External hardware

This just works

Pinout

#	Input	Output	Bidirectional
0	input	output0	
1	set	output1	
2		output2	
3		output3	
4		output4	
5		output5	
6		output6	
7		output7	

carry skip adder [162]

- Author: Dron Sankhala
- Description: two 8-bit input adder
- GitHub repository
- HDL project
- Mux address: 162
- Extra docs
- Clock: 0 Hz

How it works

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

How to test

To test the carry-skip adder:

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

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

External hardware

No external hardware is required for this project.

Pinout

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

4-bit up/down binary counter [163]

- Author: claudiotalarico
- Description: 4-bit up/down binary counter with enable and test mode
- GitHub repository
- HDL project
- Mux address: 163
- Extra docs
- Clock: 50000000 Hz

How it works

4 bit up/down binary counter with enable

Pin Mapping

```
direction | pin name      | function
----------+--------------+-------------------
in        | clk           | clk
in        | rst_n         | rst_n
in        | ui_in[0]      | test      (test mode)
in        | ui_in[1]      | ud        (up/down)
in        | ui_in[2]      | en        (enable)
out       | ui_out[3:0]   | cnt[3:0]  (count)
```

How to test

```
Connect input pin EN to VDD
Connect input pin TEST to GND
Connect input pin UD to VDD or GND through a switch
Connect input pin RST_N to an R-C startup circuit
Connect input pin CLK to a 50 MHz square waveform
Connect the output pins CNT[3:0] to 4 LEDs
```

External hardware

switch 4 LEDs R-C startup circuit

Pinout

#	Input	Output	Bidirectional
0	test	cnt[0]	
1	ud	cnt1	
2	en	cnt2	
3		cnt[3]	
4			
5			
6			
7			

xor gate with registered output [164]

- Author: claudiotalarico
- Description: xor gate
- GitHub repository
- Wokwi project
- Mux address: 164
- Extra docs
- Clock: 50000000 Hz

How it works

XOR gate with output registered by a FF. The FF has active high Set and Reset. The Set is unused (stuck at GND).

How to test

```
IN0 IN1    OUT0
---------------
 0   0      0
 0   1      1
 1   0      1
 1   1      0
```

External hardware

push button (for Reset) DIP switch 8 LED

Pinout

#	Input	Output	Bidirectional
0	CLK	OUT0	
1	RST_N		
2	IN0		
3	IN1		
4			
5			
6			

#	Input	Output	Bidirectional
7			

Team 17's 8 bit DAC [165]

- Author: Vance Wiberg
- Description: This 8 bit digital to analogue converter uses a SAR to convert signals from Digial into Analoge
- GitHub repository
- HDL project
- Mux address: 165
- Extra docs
- Clock: 0 Hz

How it works

Uses nonlinear sampling to convert a input coming from a comparator to a digital signal

How to test

In put comparator values, check for desired digital outputs

External hardware

Analog comparator and resistor array

Pinout

#	Input	Output	Bidirectional
0	A[0]	Z[0]	O[0]
1	A1	Z1	
2	A2	Z2	
3	A[3]	Z[3]	
4	A[4]	Z[4]	
5	A[5]	Z[5]	
6	A[6]	Z[6]	
7	A[7]	Z[7]	

Multi-LFSR [166]

- Author: Kevin W. Rudd
- Description: variable-length 2-tap and 4-tap LFSR with hold & step
- GitHub repository
- HDL project
- Mux address: 166
- Extra docs
- Clock: 1000 Hz

How it works

The LFSR taps are produced via a `length` => mask table which is selected by `n_taps`; `valid` indicates that there is an LFSR as confibured.. Each clock cycle produces a new LFSR `value`. `hold` prevents the LFSR from generating a new cycle and every `step` cycle produces a new value while holding.

How to test

There is no included test (yet). The design was tested using hand-generated top-level test modules (lfsr and logic) and both embedced $display invocations and GTK signal evaluation.

External hardware

This circuit can be run (input) by setting `length` and `n_taps` for the desired configuration and by using `hold` and `step` as desired to control LFSR `value` generation. LFSR state is driven by the clock (internal) and exposed (output and bidirectional output) by observing `valid` to see if there is an LFSR for the specified configuration and `value` provides the low-order 15b of the LFSR; invalid LFSR configurations produce no output.

Pinout

#	Input	Output	Bidirectional
0	length_0	value_00	value_08
1	length_1	value_01	value_09
2	length_2	value_02	value_10

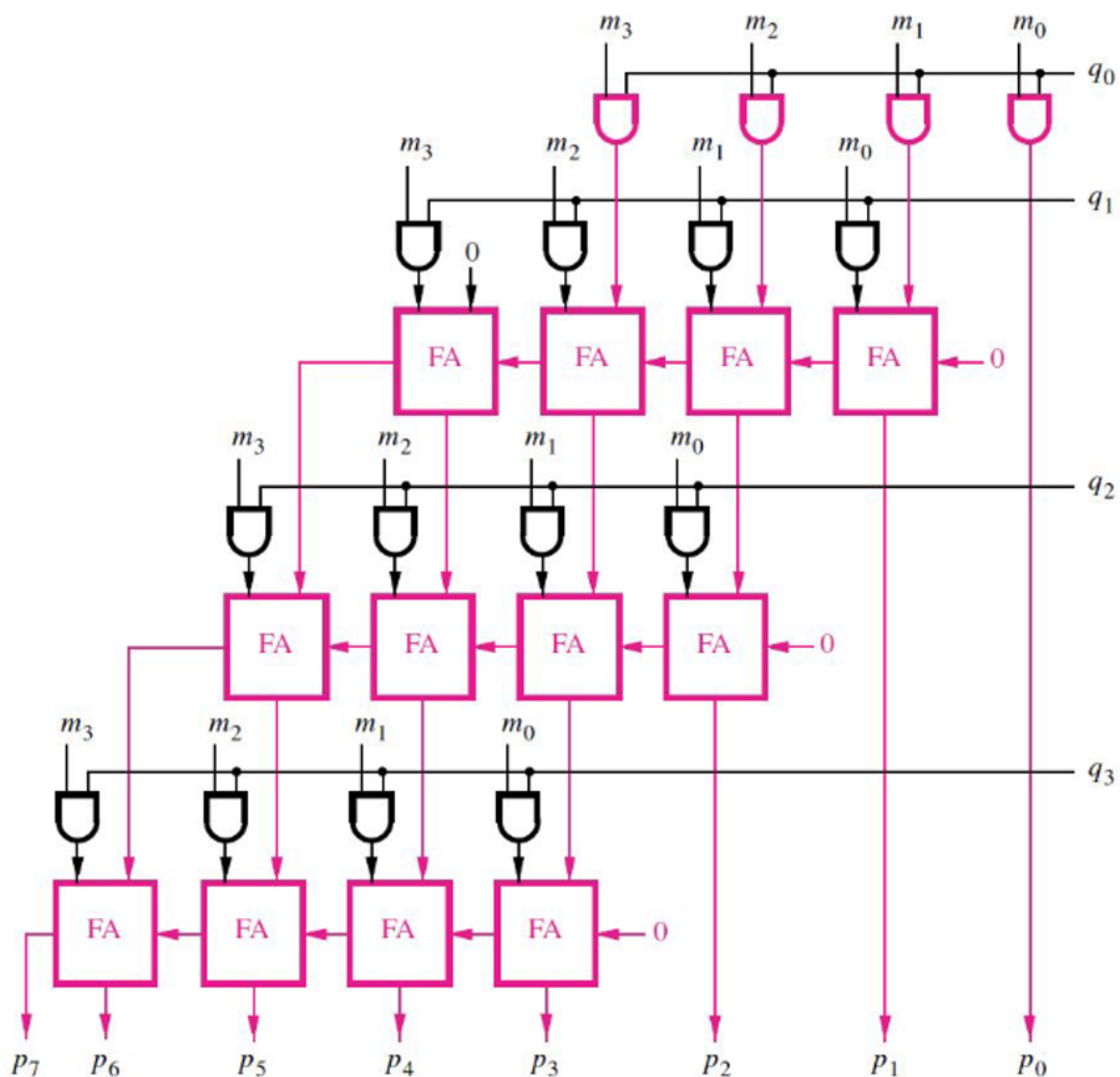
#	Input	Output	Bidirectional
3	length_3	value_03	value_11
4	length_4	value_04	value_12
5	n_taps	value_05	value_13
6	step	value_06	value_14
7	hold	value_07	valid

ECE2204MultiplierProject [167]

- Author: CaoKeHanMax
- Description: ECE2204MultiplierProject
- GitHub repository
- HDL project
- Mux address: 167
- Extra docs
- Clock: 0 Hz

How it works

4 by 4 Arrary Structural Multiplier This is a class project designated to design a 4 by 4 arrary multiplier using logic gates and 1 bit full adders. Verilog codes were used in this projec t to implement this multiplier. The structural of the design is shown below.



The project fully implements this structure by replicating the logic gates and the connections to the repective components one by one. The adder module is included in the "project.v" file in the "src" folder. It is named as "black_box", because we used it for our first lab experiment and it was provided to us to see what it does and how it is implemented. The idea used in making this multiplier is partial product and adding together the products. This means that the product of each digits of the binary number is multiplied, shifted, and added with next line just like the way we did multiplication in decimal. Here is an example of binary product, and you can see how the idea of multipling (partial produuct), shifting, and adding is done.

For example:

```
    1011   (this is binary for decimal 11)
  × 1110   (this is binary for decimal 14)
  ======
    0000   (this is 1011 × 0)
   1011    (this is 1011 × 1, shifted one position to the left)
  1011     (this is 1011 × 1, shifted two positions to the left)
+ 1011     (this is 1011 × 1, shifted three positions to the left)
=========
10011010   (this is binary for decimal 154)
```

The Credit of this picture to Wikipedia, and you can read more about this idea here: Binary multiplier

This means that the multiplier support only unsigned binary numbers, so you should not expect to multiply signed decimal, 1's complemet, or 2's complement to work with this multiplier. Overall, this is a simple 4 by 4 multiplier.

How to test

To test if this project works, two ways are presented. The first way is to check the automatic test of the project, which is shown in the "Actions" bar. A green checkmark will show saying "test" if the project is working properly. You can modify the values of the tests by changing the "test.py" code. Here is how you can change to different values: Find the line that says "assert dut.uo_out.value", and change the value of the designated test value after the two equal sign. Then change the "dut.ui_in.value" value below the variable you just modified before this to the two binary numbers that you want to multiply. This variuable has eight bits, so that means the first 4 and last 4 bits each contributes to unsigned binary numbers respectively. Change this value so that when you calculate yourself, it matches with the valuee you entered above. The numbers you entered in "dut.uo_out.value" can be in decimal.

Another way to test this is to make the circuit for this structure according to the pictures provided and the codes in this project. You should get the same answer with the output of this project.

External hardware

No External Hardware is used in this project, and it does not support external hardware for now.

Pinout

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

Micro tile container [168]

- Author: Arna
- Description: Example Experimental microtile TDC container
- GitHub repository
- HDL project
- Mux address: 168
- Extra docs
- Clock: 20000000 Hz

How it works

Four micro tiles were combined into a single Tiny Tapeout tile to analyze power coupling within the Power Distribution Network.

Selecting the active project Use uio[1:0] to see the output of the micro-tile projects.

Project 1 - Sensor

- Repo:https://github.com/Secure-Embedded-Systems/tt09-microtile-sensor-example
- Author: Arna Roy
- Description: A sensor

How it works The project generates a delayed clock signal utilizing eight distinct delay lines

How to test Nothing to test here.

Project 2 - Time to Delay Converters (TDC)

- Repo:https://github.com/Secure-Embedded-Systems/tt09-microtile-tdc-example
- Author: Arna Roy
- Description:Delay line based TDC to measure timing effects

How it works A Time-to-Digital Converter (TDC) is embedded in one tile and interfaced with the sensor. It measures the time interval between the clock signal from the sensor and the delayed clock signal generated by the sensor.

How to test Nothing to test here, an experimental basis.

Project 3 - Ring Oscillator (RO)

- Repo:https://github.com/Secure-Embedded-Systems/tt09-microtile-RO-tile1
- Author: Arna Roy
- Description: 32 ROs in one tile to add power stress

How it works It includes an activation signal capable of enabling 16 ring oscillators simultaneously, primarily to induce power stress for monitoring the Power Distribution Network (PDN)."

Project 4 - Ring Oscillator (RO)

- Repo: https://github.com/Secure-Embedded-Systems/tt09-microtile-RO-tile2
- Author: Arna Roy
- Description: 32 ROs in one tile to add power stress

How it works Same as the previous design of ROs, just placed in another tile to add power stress

Pinout

#	Input	Output	Bidirectional
0	in[0]	out[0]	sel[0]
1	in1	out1	sel1
2	in2	out2	
3	in[3]	out[3]	
4	in[4]	out[4]	
5	in[5]	out[5]	
6	in[6]	out[6]	
7	in[7]	out[7]	

4bit multiplier [169]

- Author: Kylian Yan
- Description: tiny tapeout
- GitHub repository
- HDL project
- Mux address: 169
- Extra docs
- Clock: 0 Hz

a 4 bit adder

How it works

we split the input into two portions and assign them to be q and m. The output is p, we then use full adder to achieve the purpose of multiplcaton

How to test

we use idle to tst our cases

External hardware

None

Pinout

#	Input	Output	Bidirectional
0	q[0]	p[0]	
1	q1	p1	
2	q2	p2	
3	q[3]	p[3]	
4	m[0]	p[4]	
5	m1	p[5]	
6	m2	p[6]	
7	m[3]	p[7]	

Forward Pass Network for Simple ANN [170]

- Author: Arian Heidari
- Description: ANN that takes in a 4-bit value, and completes a forward pass.
- GitHub repository
- HDL project
- Mux address: 170
- Extra docs
- Clock: 50000000 Hz

How it works

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

How to test

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

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

External hardware

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

Pinout

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

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

Tiny Registers [171]

- Author: Roni Kant, Jeremy Kam
- Description: Various Registers for 8-bit CPU
- GitHub repository
- HDL project
- Mux address: 171
- Extra docs
- Clock: 50000000 Hz

How it works

The various registers used for a basic 8-bit CPU design. Consists of a simple general purpose register, a memory address register, and an instruction register. The 3 registers are selected using the 6th and 7th uio pins. | uio[7] | uio[6] | Selected Register | |——————|——————|———————-| | 0 | 0 | General Purpose Register | | 0 | 1 | Memory Address Register | | 1 | 0 | Instruction Register |

Design Specifications

Instruction Register

Label	Input/Output	Description
CLK [1 bit]	Input	Clock signal. Executes actions on rising edges.
W bus [8 bit]	Input	Takes 8 bits with the most significant 4 bits representing the opcode and the least significant 4 bits representing any other necessary value. Write them to the instruction register.
I [1 bit]	Input	Control signal that decides whether to read from the bus.
I [1 bit]	Input	Control signal that decides tri-state buffer output to bus (drive register value if enabled, Z if disabled).
CLR [1 bit]	Input	Clears the instruction register's data.
Instruction register[3:0] [4 bit]	Output	Output to W bus
Instruction register[7:4] [4 bit]	Output	Output to controller/sequences

Pinouts when instruction register is selected

Test Input Name	Description
clk	CLK
ui_in[7:0]	W bus
uio_in [4]	I
uio_in [5]	I
rst_n	CLR
uio_out[3:0]	Instruction register[7:4]
uo_out[3:0]	Instruction register[3:0]

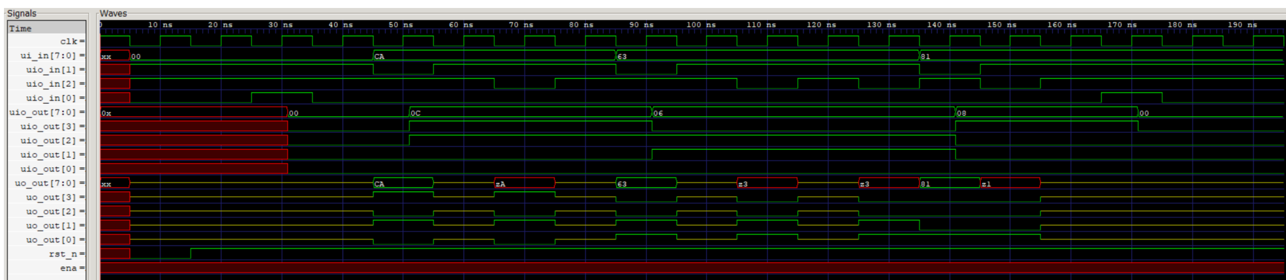


Figure 12: instruction_register

- **Note**: All simulations pictured in this document were run using a 10 ns clock. The actual design will have a 100 ns clock.

Test Input Connections (as seen in waveform)

Test Input Name	Description
clk	CLK
ui_in[7:0]	W bus
uio_in 1	I
uio_in 2	I
uio_in [0]	CLR
uio_out[3:0]	Instruction register[7:4]
uo_out[3:0]	Instruction register[3:0]

Output Register

Label	Input/Output	Description
CLK [1 bit]	Input	Clock signal. Executes actions on rising edges.
W bus [8 bit]	Input	Data from the bus lines that are to be written to the Output register.
O [1 bit]	Input	Control signal that decides whether to read from the bus and load onto the output register.
Output register [8 bit]	Output	Register data that will be written to the binary display.

Pinouts when output register is selected

Test Input Name	Description
clk	CLK
ui_in[7:0]	W bus
uio_in [4]	O
uo_out[7:0]	Output register

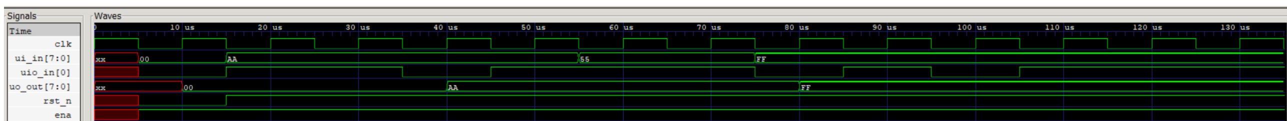


Figure 13: register

Test Input Connections (as seen in waveform)

Test Input Name	Description
clk	CLK
ui_in[7:0]	W bus
uio_in [0]	O
uo_out[7:0]	Output register

B Register

Label	Input/Output	Description
CLK [1 bit]	Input	Clock signal. Executes actions on rising edges.

Label	Input/Output	Description
W bus [8 bit]	Input	Data from the bus lines that are to be written to the B register.
B [1 bit]	Input	Control signal that decides whether to read from the bus and load onto the B register.
B register [8 bit]	Output	Register data that will be written to adder/subtractor.

Pinouts when b register is selected

Test Input Name	Description
clk	CLK
ui_in[7:0]	W bus
uio_in [4]	B
uo_out[7:0]	B register

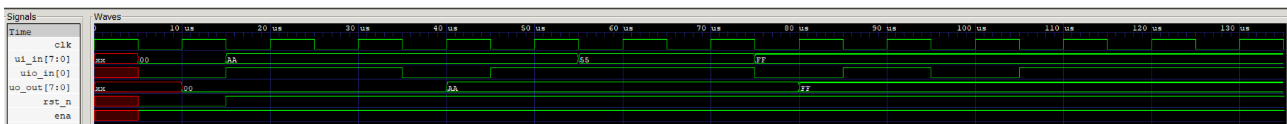


Figure 14: register

Test Input Connections (as seen in waveform)

Test Input Name	Description
clk	CLK
ui_in[7:0]	W bus
uio_in [0]	B
uo_out[7:0]	B register

Input and MAR

Label	Input/Output	Description
CLK [1 bit]	Input	Clock signal. Executes actions on rising edges.
W bus [8 bit]	Input	Data from the bus lines that are to be written either Input or MAR register.

Label	Input/Output	Description
MD [1 bit]	Input	Control signal that decides if W bus data is to be written to the Input register. Should not be active at the same time as the MA control signal.
MA [1 bit]	Input	Control signal that decides if W bus data is to be written to the MAR register. Should not be active at the same time as the MD control signal.
Input register [8 bit]	Output	Register data to be written to memory.
MAR [4 bit]	Output	Register data taken by RAM that controls where the data is to be written.

Pinouts when input and mar register is selected

Test Input Name	Description
clk	CLK
ui_in[7:0]	W bus
uio_in [4]	MD
uio_in [5]	MA
uo_out[7:0]	Input register
uio_out[3:0]	MAR

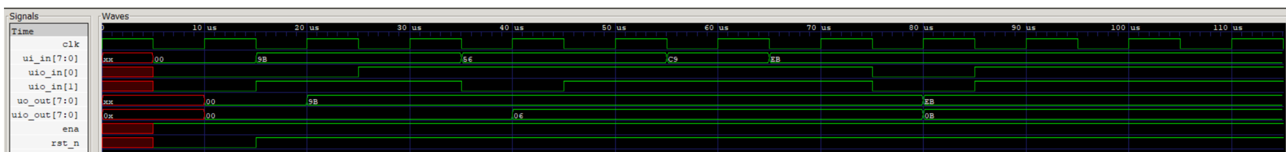


Figure 15: input_mar_register

Test Input Connections (as seen in waveform)

Test Input Name	Description
clk	CLK
ui_in[7:0]	W bus
uio_in [0]	MD
uio_in 1	MA
uo_out[7:0]	Input register
uio_out[3:0]	MAR

Pinout

#	Input	Output	Bidirectional
0	in_0	out_0	extra_output_0
1	in_1	out_1	extra_output_1
2	in_2	out_2	extra_output_2
3	in_3	out_3	extra_output_3
4	in_4	out_4	extra_input_0
5	in_5	out_5	extra_input_1
6	in_6	out_6	register_select_0
7	in_7	out_7	register_select_1

7-Segment Byte Display [172]

- Author: Mike Goelzer
- Description: Drives a single hex digit 7-segment display based on the value of a 1-byte input
- GitHub repository
- HDL project
- Mux address: 172
- Extra docs
- Clock: 0 Hz

How it works

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

How to test

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

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

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

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	Byte to display on 7-segment display (rightmost / low order bit)	7-segment display (segment a)	write enable (1=latch byte value on ui[7:0] and display it, 0=ignore ui[7:0] and keep displaying the current value)
1	Byte to display on 7-segment display (next bit)	7-segment display (segment b)	
2	Byte to display on 7-segment display (next bit)	7-segment display (segment c)	
3	Byte to display on 7-segment display (next bit)	7-segment display (segment d)	
4	Byte to display on 7-segment display (next bit)	7-segment display (segment e)	
5	Byte to display on 7-segment display (next bit)	7-segment display (segment f)	
6	Byte to display on 7-segment display (next bit)	7-segment display (segment g)	

#	Input	Output	Bidirectional
7	Byte to display on 7-segment display (leftmost / high order bit)		

Leaky Integrate Fire Neuron [173]

- Author: Rocky Lim
- Description: Simulates a Leaky Integrate Fire Neuron based on snnTorch's implementation
- GitHub repository
- HDL project
- Mux address: 173
- Extra docs
- Clock: 0 Hz

How it works

This chip takes in an 8-bit number voltage to simulate a Leaky Fire Integrate (LIF) Network. The 8-bit number is split into two different neurons in which they have their respective layers, and it takes that voltage to act as an input current to the LIF neurons. Each neuron generates a spike when the threshold, defined to be 8, is reached or surpassed. Once an input current is passed through, each neuron will decay the value over each clock cycle by shifting the bits of the current state once as it constantly takes the input current. The idea behind the layers is for more significant spikes to be able to reach the output states while less significant events would not affect the output.

How to test

N/A

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	Input Current Bit [0] (Input Neuron 1)	State Variable Bit [0] (Output Neuron 1)	Spike Bit (Output Layer, Neuron 2)

#	Input	Output	Bidirectional
1	Input Current Bit 1 (Input Neuron 1)	State Variable Bit 1 (Output Neuron 1)	Spike Bit (Output Layer, Neuron 1)
2	Input Current Bit 2 (Input Neuron 1)	State Variable Bit 2 (Output Neuron 1)	Spike Bit (Inner Layer 2, Neuron 2)
3	Input Current Bit [3] (Input Neuron 1)	State Variable Bit [3] (Output Neuron 1)	Spike Bit (Inner Layer 2, Neuron 1)
4	Input Current Bit [4] (Input Neuron 2)	State Variable Bit [4] (Output Neuron 2)	Spike Bit (Inner Layer 1, Neuron 2)
5	Input Current Bit [5] (Input Neuron 2)	State Variable Bit [5] (Output Neuron 2)	Spike Bit (Inner Layer 1, Neuron 1)
6	Input Current Bit [6] (Input Neuron 2)	State Variable Bit [6] (Output Neuron 2)	Spike Bit (Input Layer, Neuron 2)
7	Input Current Bit [7] (Input Neuron 2)	State Variable Bit [7] (Output Neuron 2)	Spike Bit (Input Layer, Neuron 1)

znah_vga_ca [174]

- Author: Alexander Mordvintsev
- Description: Simple VGA 1D cellular automata generator
- GitHub repository
- HDL project
- Mux address: 174
- Extra docs
- Clock: 25175000 Hz

How it works

VGA signal generator iterates through a number of 1D elementary cellular automata

How to test

Plug and play

External hardware

VGA PMOD

Pinout

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

Tiny Tapeout Group 7 Lab D [175]

- Author: Will and Andrea
- Description: Our project implements a 4x4 array multiplier
- GitHub repository
- HDL project
- Mux address: 175
- Extra docs
- Clock: 0 Hz

How it works

Our program works by using a 4x4 array multiplier computes the product of two 4-bit binary numbers, m and q, through bitwise multiplication and summing partial products. Each bit of q is multiplied by every bit of m, generating partial products that are shifted based on their significance. Full adders (FA) then sum these partial products. At each stage, the full adders combine two partial product bits and any carry from the previous stage. As the process progresses through the rows, the number of bits to sum increases, which is managed by additional full adders. The final output is an 8-bit product p, with the least significant bit produced by the sum of the first row and the most significant bit formed by the final carry after all additions.

How to test

To test the 4x4 multipler feed the multiplier two 4 bit inputs. From here the partial products will be calculated and the remaining product should be a binary representation of the decimal product. To verify you can convert final products between binary and decimal and compare expected values.

External hardware

Tiny Tapeout design

Pinout

#	Input	Output	Bidirectional
0	m[0]	p[0]	
1	m1	p1	

#	Input	Output	Bidirectional
2	m2	p2	
3	m[3]	p[3]	
4	q[0]	p[4]	
5	q1	p[5]	
6	q2	p[6]	
7	q[3]	p[7]	

4-bit Multiplier [192]

- Author: Asfaq Fahim & Sreeja Ghose
- Description: Multiplies 2 4-bit binary numbers.
- GitHub repository
- HDL project
- Mux address: 192
- Extra docs
- Clock: 0 Hz

How it works

It works by multiplying two 4-bit binary numbers using full adders and outputing the product.

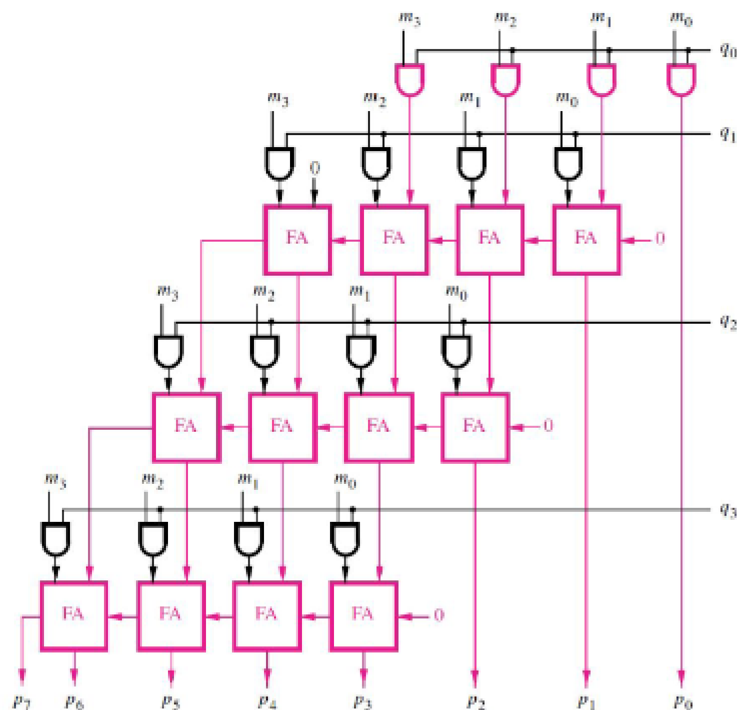


Figure 16: Block Diagram

How to test

To test, you open the test.py file and input two 4-bit binary numbers. The first 4 are the first number and the last 4 are the second number. The expected value should be the product of the two numbers.

External hardware

None

Pinout

#	Input	Output	Bidirectional
0	n[0]	p[0]	
1	n1	p1	
2	n2	p2	
3	n[3]	p[3]	
4	m[0]	p[4]	
5	m1	p[5]	
6	m2	p[6]	
7	m[3]	p[7]	

FIREngine [193]

- Author: Hao Wang, Andrew Malnicof
- Description: FIR Filter for Audio PMOD
- GitHub repository
- HDL project
- Mux address: 193
- Extra docs
- Clock: 50000000 Hz

How it works

FIREngine is a Digital FIR filter that filters inputs from an I2S2 PMOD ADC and DAC module. The purpose of this design is to filter audio from an I2S2 PMOD device found here: https://digilent.com/shop/pmod-i2s2-stereo-audio-input-and-output/. Although the number of taps the filter is not adjustable and must be determined before synthesis, the coefficients of each tap are programmable. This allows for different low, band, and high pass filters to be constructed for multiple audio filtering configurations. If is a parametrizable filter with symmetric or antisymmetric coefficients, odd number of taps. Uses 2s complement and fixed-point data. Coefficients are set via an SDI Interface.

How to test

Use TinyTapeout Demo board to connect PMOD to Tiny Tapeout project, program filter coefficients serially, and experience the results!

External hardware

- I2S2 PMOD device: https://digilent.com/shop/pmod-i2s2-stereo-audio-input-and-output/
- Serial programmer

Pinout

#	Input	Output	Bidirectional
0	SPI CS		DAC MCLK
1	SPI MOSI		DAC LRCK

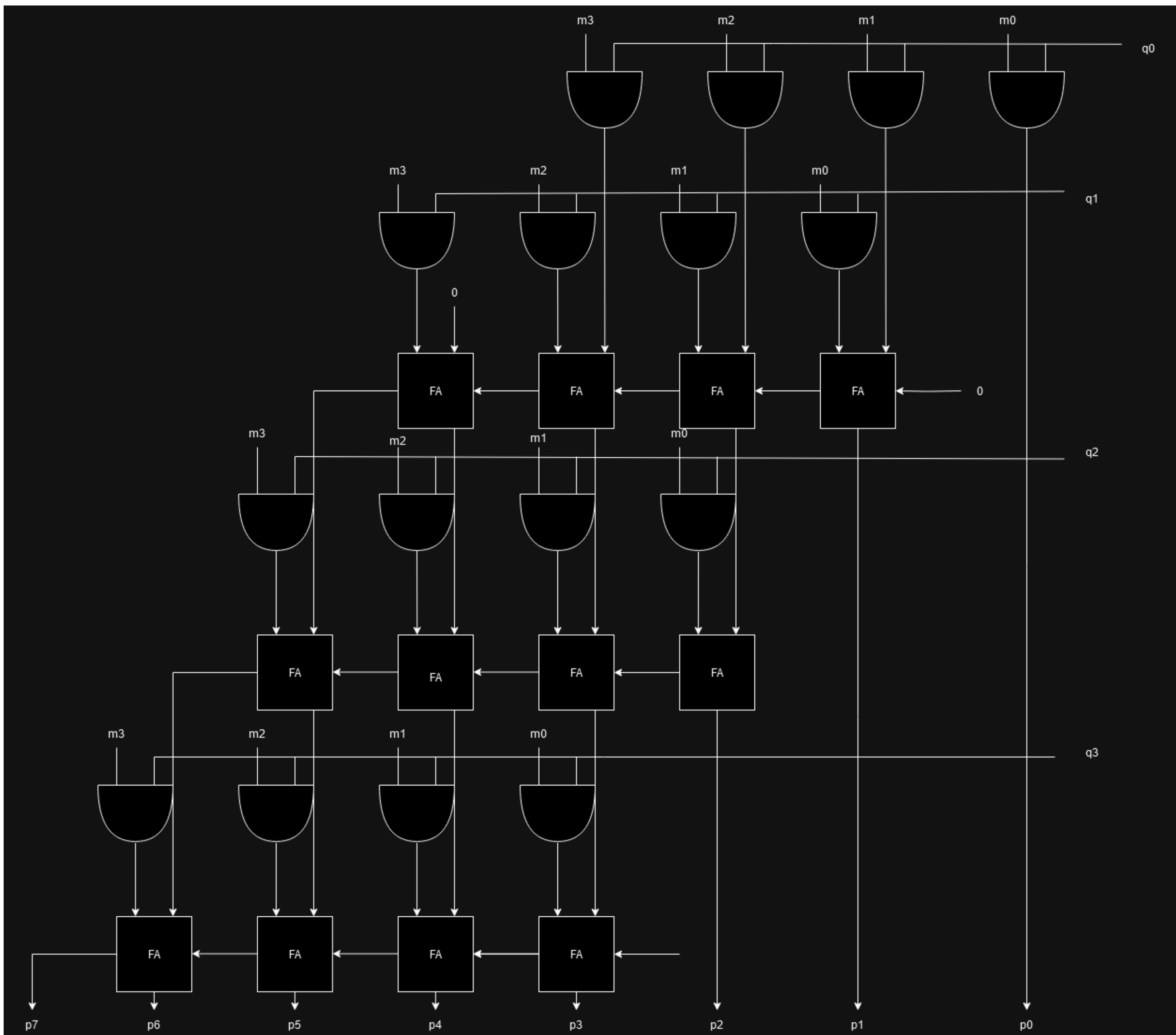
#	Input	Output	Bidirectional
2			DAC SCLK
3	SPI SCLK		DAC Data
4			ADC MCLK
5			ADC LRCK
6			ADC SCLK
7			ADC Data

4x4multiplier [194]

- Author: hirod nazari, samarth pusegaonkar
- Description: A multiplier that takes in 2 4-bit inputs and outputs a 8-bit result
- GitHub repository
- HDL project
- Mux address: 194
- Extra docs
- Clock: 0 Hz

How it works

The code takes in two 4-bit inputs, and multiplies them, outputting the 8-bit result. The way this is done is by utilizing multiple layers of full adders, with each layer of full adders acting as a multiplication. Four layers indicate the four multiplications, each bit of the second input being multiplied by the first input. Each layer is additionally offset, to correct for bit placement.



How to test

Put inputs in test.py, with input 1 being the left 4 bits of dut.ui_in.value, and input 2 being the right 4 bits of dut.ui_in.value. The corresponding output is compared with dut.uo_out.value, which should be the correct result of the multiplication.

External hardware

N/A

Pinout

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

Lab B Group 1 Array Multiplier [195]

- Author: MarcAnthony Williams & Ivy Zheng
- Description: 4x4 Structural Array Multiplier
- GitHub repository
- HDL project
- Mux address: 195
- Extra docs
- Clock: 0 Hz

How it works

There will be two 4-bit inputs that represents the binary factors and an 8-bit output for the product. With the use of Full Adders, it combines the partial products and produces the final binary multiplication result.

How to test

To test this project, there are two inputs. You would set one of them to a negative in 2's complement form, in binary. You would then multiply both these inputs in binary and should have a signed integer that accurately represents the multiplication.

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	q[0]	p[0]	
1	q1	p1	
2	q2	p2	
3	q[3]	p[3]	
4	m[0]	p[4]	
5	m1	p[5]	
6	m2	p[6]	
7	m[3]	p[7]	

4-bit Multiplier [196]

- Author: Jeremy Kang, Idris Al-Wazani
- Description: 4x4 Multiplier using structural verilog
- GitHub repository
- HDL project
- Mux address: 196
- Extra docs
- Clock: 0 Hz

How it works

4x4 multiplier using structural verilog. The structure of the 4x4 multiplier array that the exercise should emulate is shown below in the diagram. This multiplies two 4-bit inputs, 'm' and 'q' in this case, to an 8-bit product, 'p'. The code takes each consecutive bit of q and cascades it along the first bit of m. From there, full adders are used to combine each bit's value.

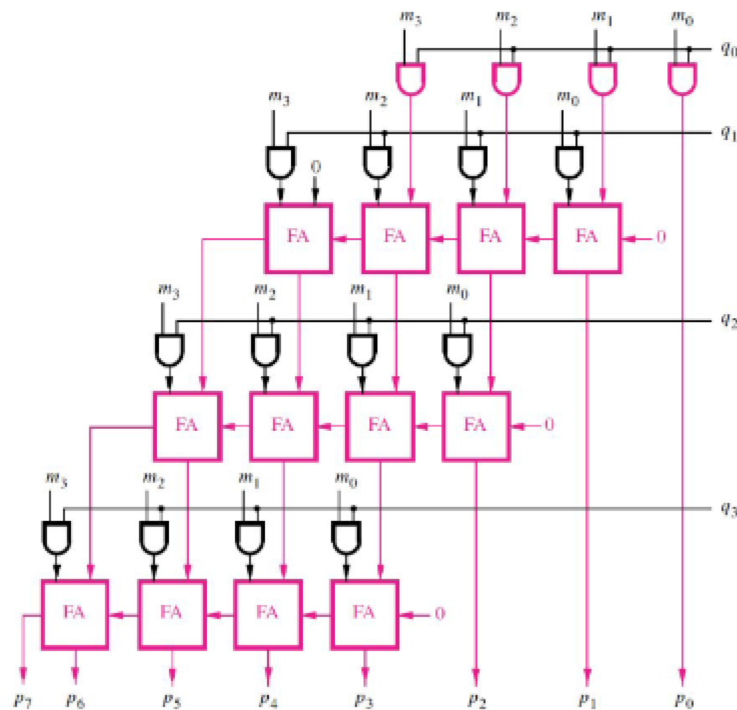


Figure 1: 4 × 4 Array Multiplier

Figure 17: Block Diagram

How to test

Test cases for inputs m and q should result in the expected product value of p. The test cases in the test python compilation essentially declare two values that should be multiplied and the predicted correct output of the multiplication. To test the functionality, access the test/README.md file.

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	q[0]	p[0]	
1	q1	p1	
2	q2	p2	
3	q[3]	p[3]	
4	m[0]	p[4]	
5	m1	p[5]	
6	m2	p[6]	
7	m[3]	p[7]	

Array Multiplier [197]

- Author: Jaden Daily
- Description: 4x4 Structural Array Multiplier
- GitHub repository
- HDL project
- Mux address: 197
- Extra docs
- Clock: 0 Hz

How it works

Explain how an array multiplier works, as well as partial products:

An array multiplier is a combinational circuit that multiplies two binary numbers together. The partial products are generated for each bit of the second operand and then adding them together using full adders. Each bit of the multiplier, in this case Q, produces partial products by using an AND-gate with every bit of M. Since this is a 4x4 multiplier, there will be 4 rows of 4-bit partial products. The partial products are then added together using full adders, with carry bits being moved to the next column as needed.

Block diagram

```
graph TD;
    M["Input M (4 bits)"] -->|"Partial Products"| PP["Partial Product Cre
    Q["Input Q (4 bits)"] -->|"Partial Products"| PP
    PP -->|"Partial Products"| FA["Full Adders"]
    FA -->|"Product (8 bits)"| P["Output Product (p)"]
    CL["Control Logic"] -->|"Control signals"| PP
    CL -->|"Control signals"| FA
    FA -->|"Cout signals"| C["Carry Outputs (c1, c2, c3, c4, c5, c6, c7,
    C -->|"Final Carry"| P
    E["Enable Signal (ena)"] -->|"Active High"| OE["Output Enable (uio_oe
    OE -->|"Enable Control"| P
    U["Unused inputs (ena, clk, rst_n)"] -->|"Handles unused"| W["Rest of
    W -->|"Included for completeness"| OE
```

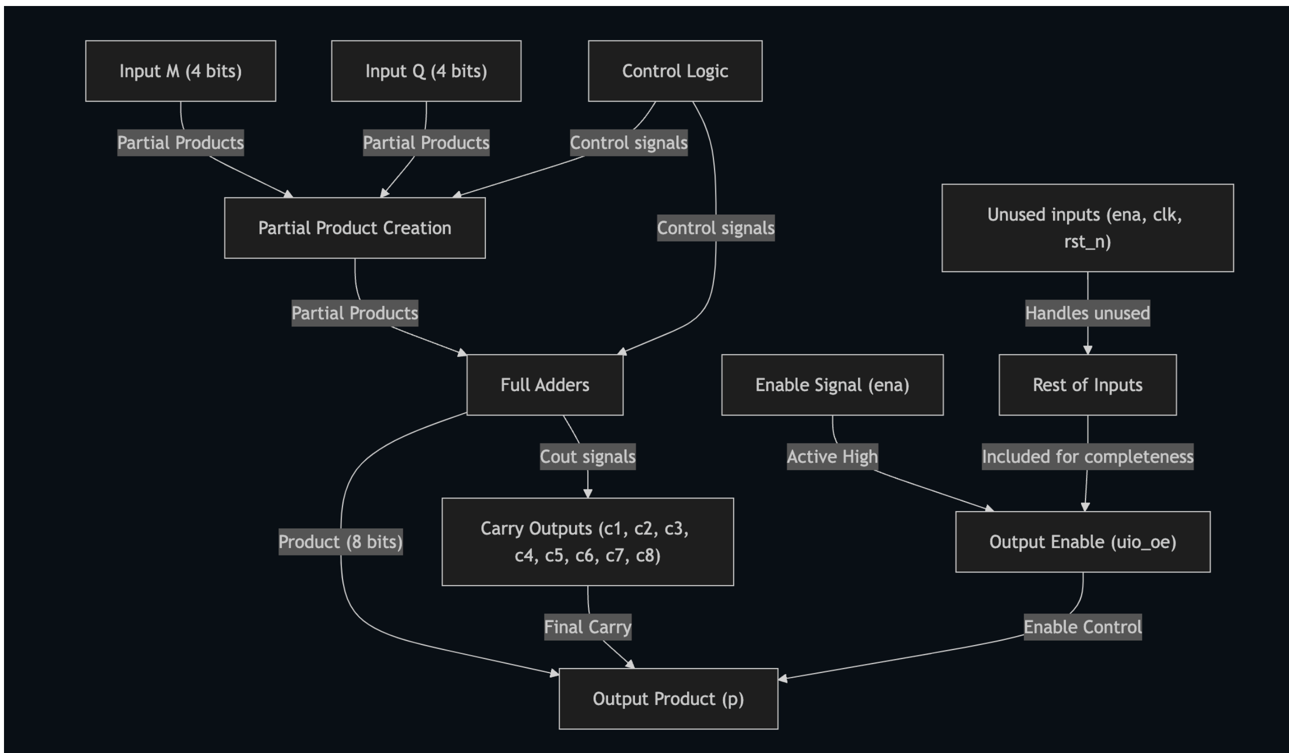


Figure 18: 4x4 Array Multiplier Block Diagram

Block diagram PDF

How to test

Explain how you know that your hardware is working when you get it:

There are numerous ways of testing the hardware. One method is to use a simulation window with a provided testbench to verify that the output matches the inputs for each of the test cases. In this scenario, we have five separate test cases to ensure the hardware functions as needed. A separate check for minimum and maximum values to guarantee correct carry propagation is also required to ensure proper functionality.

External hardware

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

No external hardware was used in this project. The design functions purely on the FPGA.

Pinout

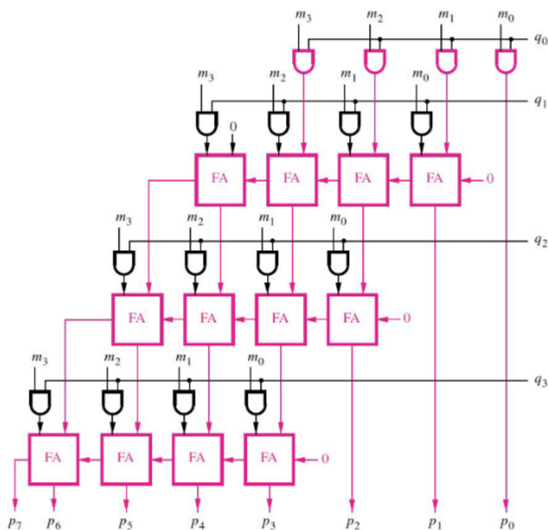
#	Input	Output	Bidirectional
0	m[0]	p[0]	
1	m1	p1	
2	m2	p2	
3	m[3]	p[3]	
4	q[0]	p[4]	
5	q1	p[5]	
6	q2	p[6]	
7	q[3]	p[7]	

4x4 Multiplier [198]

- Author: Fajr Baig, Sahana Long
- Description: 4-by-4 Bit Multiplier, Lab 3
- GitHub repository
- HDL project
- Mux address: 198
- Extra docs
- Clock: 0 Hz

How it works

This is a 4 by 4 bit multiplier designed in Verilog using structural designs.



How to test

To run test, refer to test/README.md. To add new test, modify test/test.py.

External hardware

None.

Pinout

#	Input	Output	Bidirectional
0	q[0]	p[0]	
1	q1	p1	
2	q2	p2	
3	q[3]	p[3]	
4	m[0]	p[4]	
5	m1	p[5]	
6	m2	p[6]	
7	m[3]	p[7]	

4x4 Array Multiplier [199]

- Author: Marisol and Shahran
- Description: 4x4 structural Array Multiplier
- GitHub repository
- HDL project
- Mux address: 199
- Extra docs
- Clock: 0 Hz

How it works

Our implementation of the multiplier took in two 4-inputs, m and q, and produced an 8-bit output, p, representing their product. We then genereated partial products by ANDing each bit of m with all bits of q and used a series of full adders, to sum these partial products. Each adder handled the addition of bits and carry-out/carry-in signals from the previous stage, ensuring proper alignment of the products.

How to test

We parse in an 8-bit value which is split into 2 4-bit values to generate m(bits 8 to 5), and q(bits 4 to 1). These two values are then multiplied to produce our product p.

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	q[0]	p[0]	
1	q1	p1	
2	q2	p2	
3	q[3]	p[3]	
4	m[0]	p[4]	
5	m1	p[5]	
6	m2	p[6]	
7	m[3]	p[7]	

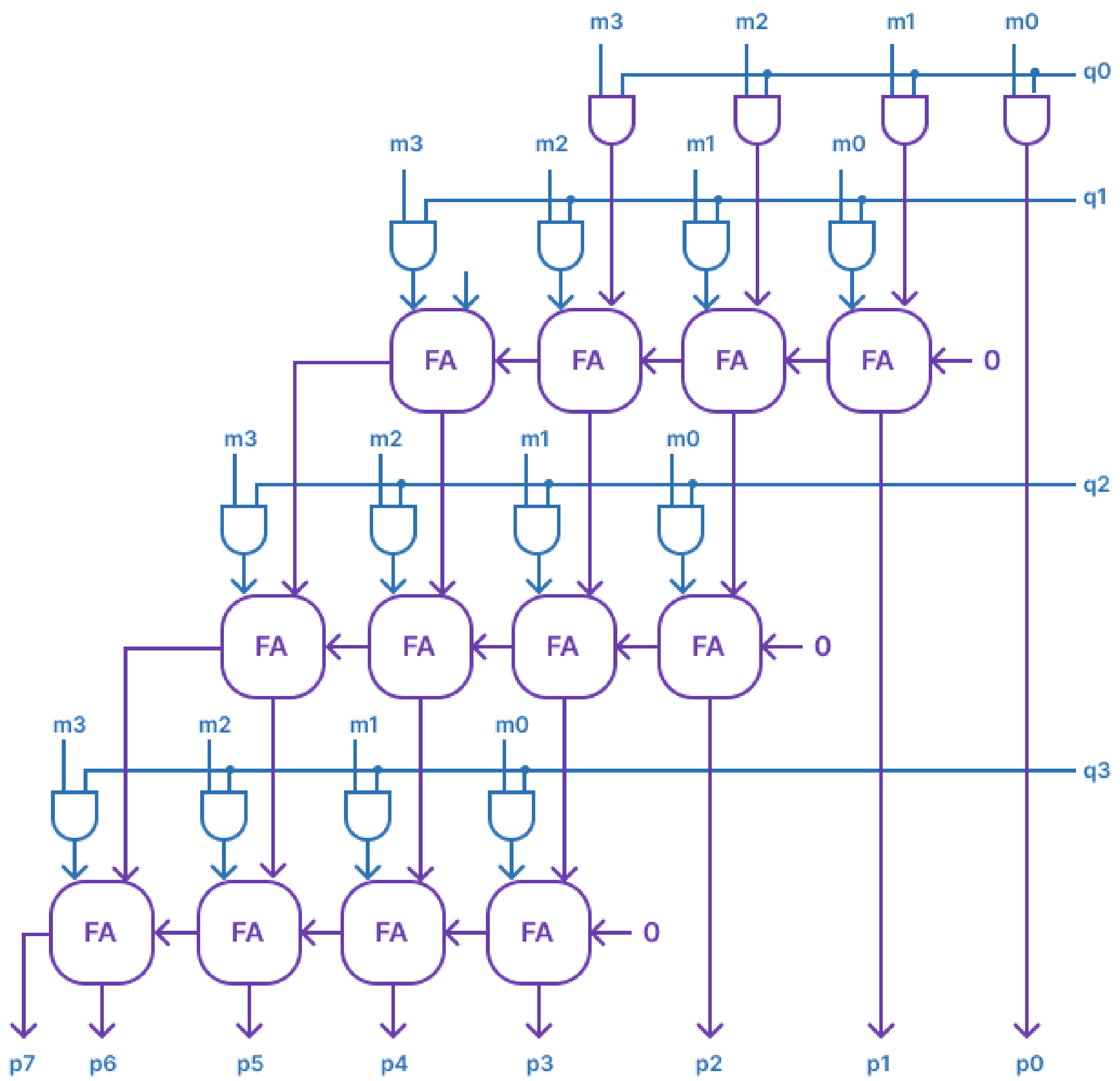


Figure 19: 4x4 array multiplier

tt09 kathyhtt [200]

- Author: kathyh
- Description: sample design from tiny tapeout
- GitHub repository
- Wokwi project
- Mux address: 200
- Extra docs
- Clock: 0 Hz

How it works

IEEE is proud to sponsor the Tiny Tapeout Workshop in San Diego this November 2024, in collaboration with UCSD and Tiny Tapeout, founded by Matt Venn. This workshop offers students worldwide a hands-on experience in semiconductor design, guiding them through the complete workflow all the way to tapeout. Making this process accessible empowers the next generation of engineers and technologists to understand and participate in semiconductor innovation. We extend our gratitude to UCSD and Tiny Tapeout for creating this incredible learning opportunity.

This tile will turn the various line segments of a a digital number, including a period. By toggling the inputs, the segments will turn on and off.

How to test

Toggle the inputs and see the segments turn on and off. This can be extended to larger displays and uses.

External hardware

Digital Display - segmants plus a period

Pinout

#	Input	Output	Bidirectional
0	in0	out0	ui0
1	in1	out1	ui1
2	in2	out2	ui2
3	in3	out3	ui3

#	Input	Output	Bidirectional
4	in4	out4	ui4
5	in5	out5	ui5
6	in6	out6	ui6
7	in7	out7	ui7

4x4 Array Multiplier [201]

- Author: Dominic Iafrate
- Description: Multiplies 2 4bit Arrays
- GitHub repository
- HDL project
- Mux address: 201
- Extra docs
- Clock: 0 Hz

How it works

This project utilizes full adders to create a 4x4 array multiplier. It takes an input of 2 4-bit signals (call them m and q) and multiplies them together to produce an 8-bit product (call it p). Firstly, each bit of m is ANDed with each bit of q, which creates a set of partial products. These partial products are then grouped into rows for each bit of q, making the diagram far more readable and organized. The partial products are aligned based on their binary place values, and each column corresponds to a bit position in the final 8-bit product, with the columns further to the left representing more significant bits. Then, the full adders are used to sum the bits in each column along with any carry-in from the column before. Logically, the adding of the partial products begins with the rightmost column, allowing for any carry to be passed up to a more significant bit. Because each column represents a bit in the final product, the sum of each column is simply the bit in the product, and once all columns are added, the final 8-bit product is obtained. The schematic diagram is shown below.

How to test

The easiest way to test this design would be to input binary test values and compare them to their known product. Using edge cases and arbitrary values would ensure that all areas are addressed in the testing process, and once the module outputs a value, comparing it to the known product of the two inputted values would ensure that the module is working properly. This can be done by cd'ing into the test folder on Git, opening test.py, and editing the test values.

As for internal testing, changes to the Verilog module can be made such that internal values can be outputted such as the partial products or the carry-overs. This is similar to debugging in, say, a Python script, as it ensures that each value at each step of the process is as it should be, not just the output. If the test value ouptuts are not equivalent to their known product, this step should be performed to find the logical

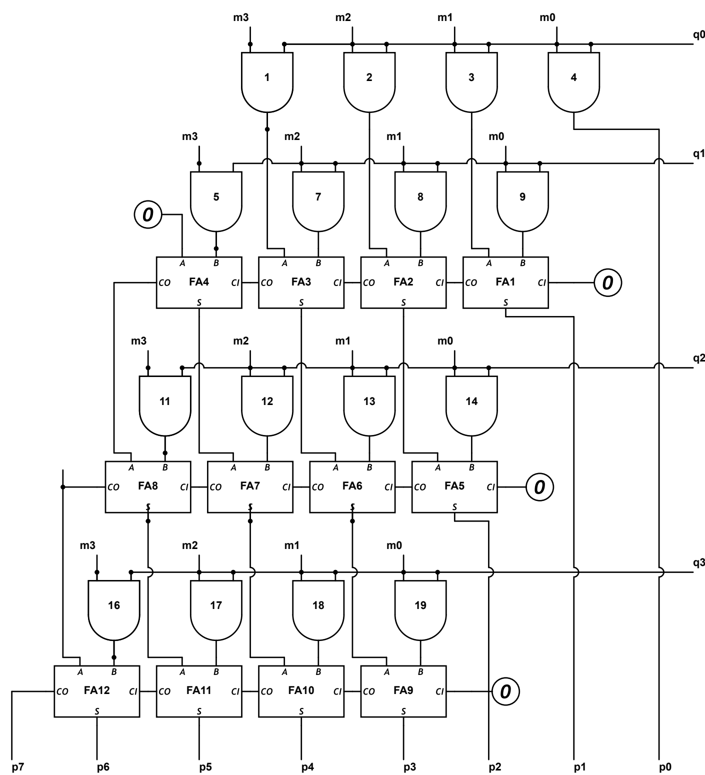


Figure 20: Array Multiplier

error in the Verilog module. Using these testing methods will make sure that the multiplier circuit works as expected.

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	m[0]	p[0]	
1	m1	p1	
2	m2	p2	
3	m[3]	p[3]	
4	q[0]	p[4]	
5	q1	p[5]	
6	q2	p[6]	
7	q[3]	p[7]	

TINY TAPE OUT [202]

- Author: Slaiman
- Description: Xand gate
- GitHub repository
- Wokwi project
- Mux address: 202
- Extra docs
- Clock: 0 Hz

How it works

BY USING INPUT AND OUTPUTS.

How to test

BY RUNNING THE SIMULATION

External hardware

LOGIC GATES

Pinout

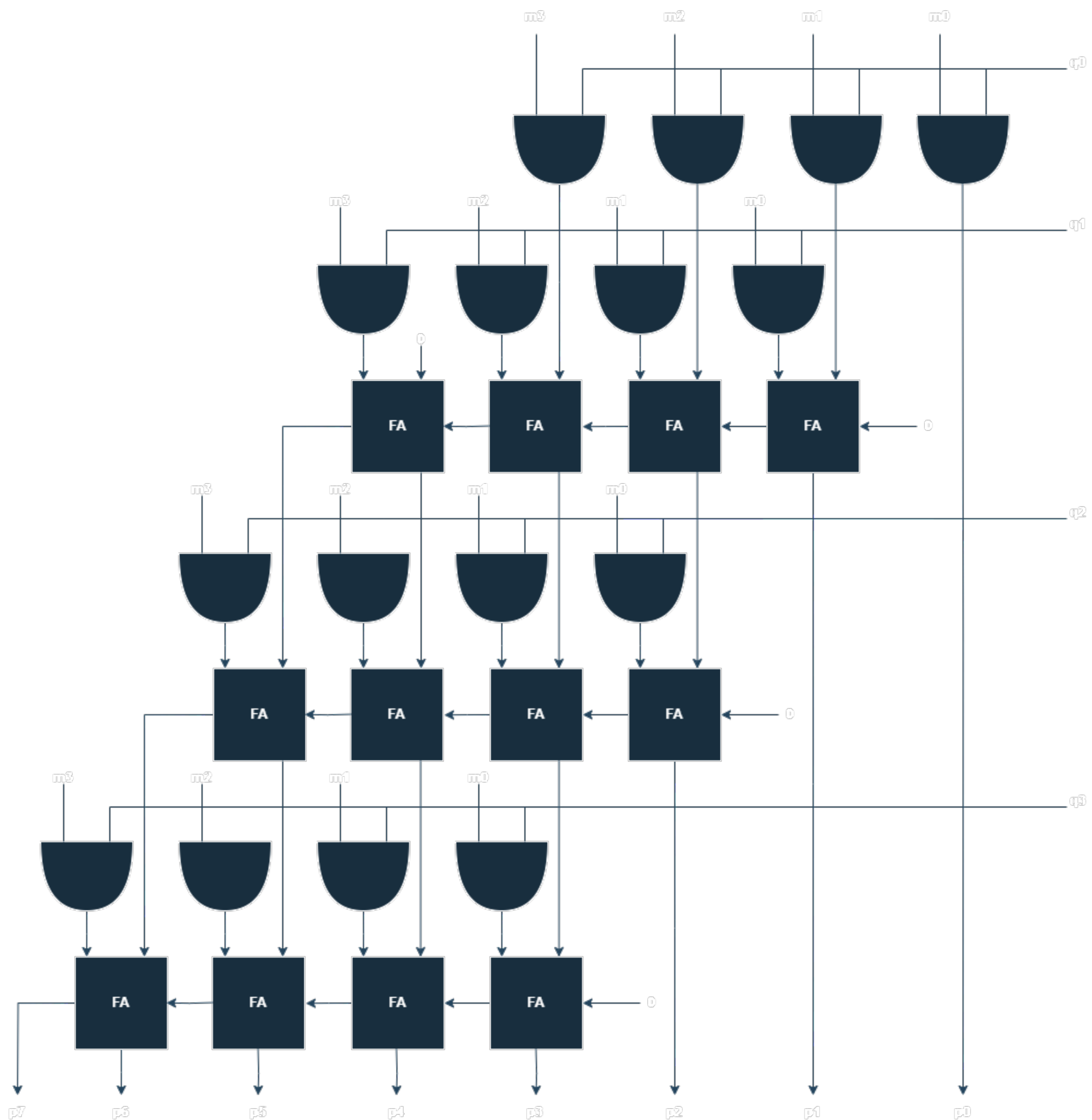
#	Input	Output	Bidirectional
0	IN0	OUT0	
1	IN1	OUT1	
2	IN2	OUT2	
3	IN3	OUT3	
4	IN4	OUT4	
5	IN5	OUT5	
6	IN6	OUT6	
7	IN7	OUT7	

ECE2204 4x4 Array Multiplier [203]

- Author: Jack Li Bill Li
- Description: 4x4 Structural Array Multiplier
- GitHub repository
- HDL project
- Mux address: 203
- Extra docs
- Clock: 0 Hz

How it works

This is a array multiplier to multiply two 4-bit binaries. It uses full adders to process bitwise multiplications, the circuit diagram is as shown below:



How to test

a testbench called test.py is given. where tests can be given to the project, to create a new test, change the value for dut.ui_in.value to 0x(any two integers), the product of the two integers will be calculated in the multiplier, and for assert dut.uo_out.value, change it to the expected output of the two integers you just inserted, if the program runs properly, no error should occur.

External hardware

N/A

Pinout

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

TinyTapeout1 [204]

- Author: Matthew H
- Description: Change Display Segments
- GitHub repository
- Wokwi project
- Mux address: 204
- Extra docs
- Clock: 0 Hz

How it works

LED display to show letter H

How to test

turn on 2,3,4, 6,7,8

External hardware

what is that

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	

comparator [205]

- Author: prtx
- Description: comparator
- GitHub repository
- Wokwi project
- Mux address: 205
- Extra docs
- Clock: 0 Hz

How it works

Will deal with this later

How to test

Will deal with this later

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	A_IN	A_OUT	
1	B_IN	B_OUT	
2			
3			
4		LT	
5		GT	
6		EQ	
7			

FB GDS [206]

- Author: Fahad Bastaki
- Description: Toggle the pins to turn on the LED display
- GitHub repository
- Wokwi project
- Mux address: 206
- Extra docs
- Clock: 0 Hz

How it works

Seven segment with switches

How to test

Turn on switches

External hardware

SSD, switches

Pinout

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

4x4 Array Multiplier [207]

- Author: Adrian Lopez and Jack Verdis
- Description: 4x4 Structural Array Multiplier
- GitHub repository
- HDL project
- Mux address: 207
- Extra docs
- Clock: 0 Hz

How it works

A 4-bit array multiplier is a combinational circuit that multiplies two 4-bit binary numbers using AND gates and full adders. Each bit of one number is multiplied by each bit of the other to create partial products. These products are aligned in a grid, with each row shifted one position to the left, like multiplication. These partial products are then added together using half-adders and full adders. Each takes three inputs and creates a sum resulting in an 8-bit binary number.

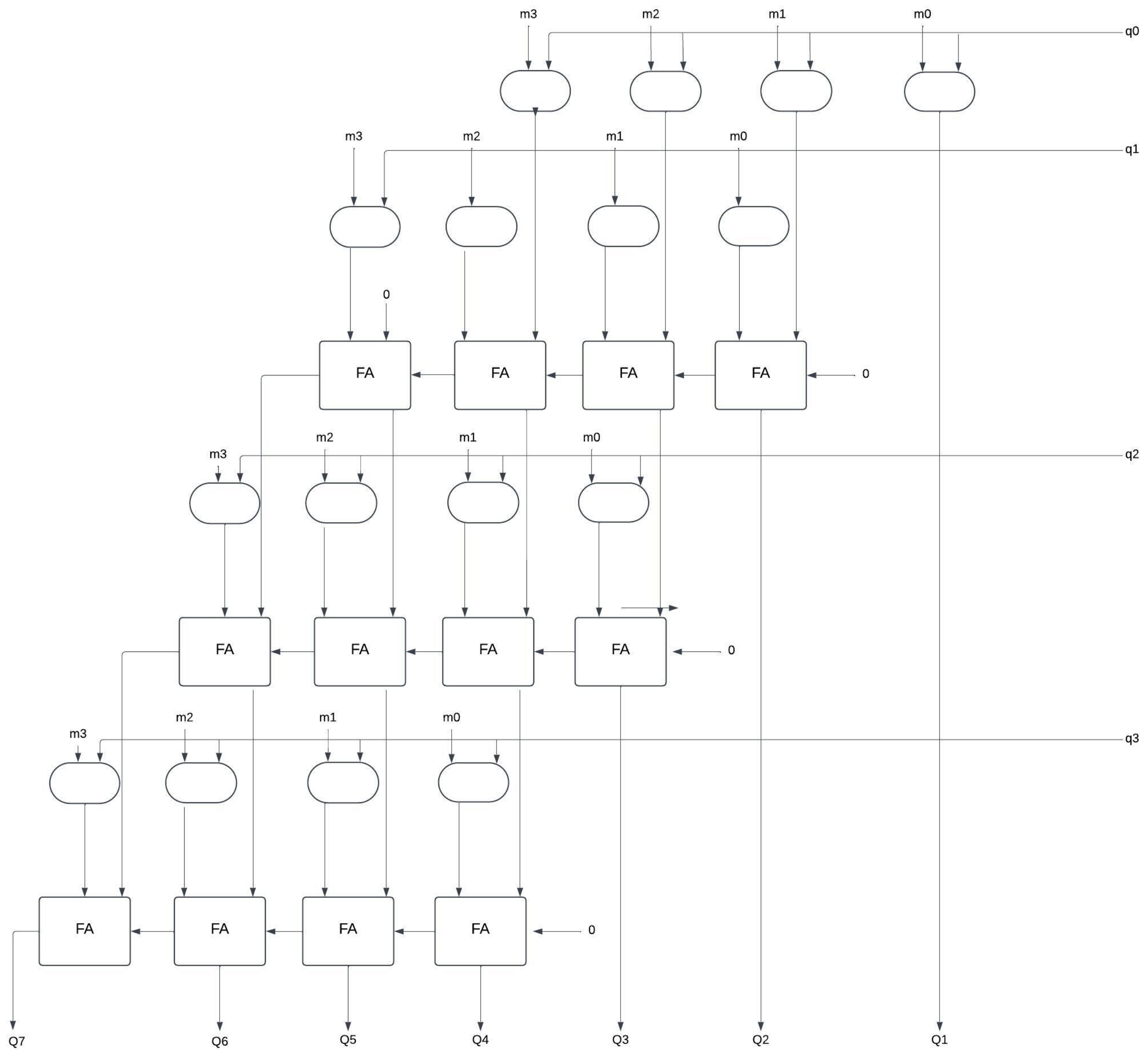


Figure #1 4x4 Mutipler Array

How to test

In order to test the product, two binary inputs need to be inputted into the code: the first four bits being the first number and the last four bits being the last number. These numbers are then run through the code, which outputs an 8-bit number that is the result of multiplying the two given numbers.

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	q[0]	p[0]	
1	q1	p1	
2	q2	p2	
3	q[3]	p[3]	
4	m[0]	p[4]	
5	m1	p[5]	
6	m2	p[6]	
7	m[3]	p[7]	

Semana UCU Verilog [224]

- Author: Universidad Católica del Uruguay
- Description: Union of proyects done in class
- GitHub repository
- HDL project
- Mux address: 224
- Extra docs
- Clock: 0 Hz

Summary

This project is a compilation of designs created by students with little to no knowledge in electronics, as a part of a hands-on learning course during * SEMANA UCU* , with their outputs multiplexed so we can test all. There were 15 total projects submitted, based on 3 different guidelines. Select the project using mux_in[0:3].

Guidelines

1- Basic Project

- Description: A shift register with ui_in[0] as input and ui_in1 as external clock. When the shift register contains a specific key chosen by the students, ui_out[0] is driven to 1.
- How to test: Connect ui_in1 with an external clock and insert the key via ui_in1 form MSB to LSB

2- Advanced Project N°1

- Description: Decoder from 3 bits to 7 segment display with ui_in[2:0] as inputs. Some groups upped it to 4 bits
- How to test: Input a 3 bit number through ui_in[2:0] and check if the output lights up the correct number (Watch out, most groups made ui_in[0] be the MSB and ui_in2 be the LSB of your input)

3- Advanced Project N°2

- Description: A 3 bit counter, driven by an external clock through ui_in[0], connected to the 3 bits to 7 segment display decoder from Advanced Project N°1. Once again some groups upped both the counter and the decoded to 4 bits.
- How to test: Connect ui_in[0] to an external clock and check if the 7 segment display lights up correctly.

Projects (Ordered by mux value)

Group 0

- Member(s): Locatelli, Roldós
- Wokwi: https://wokwi.com/projects/410732069226456065
- Guideline chosen for project: Advanced Project N°1
- Details: Uses 4 bits as input, and a common cathode display

Group 1

- Member(s): Giacometti, Salvo, Varela
- Wokwi: https://wokwi.com/projects/410463015062285313
- Guideline chosen for project: Advanced Project N°2
- Details: Uses a common cathode display, and counts up to 10 and overflows.

Group 2

- Member(s): Raposo
- Wokwi: https://wokwi.com/projects/410724169008053249
- Guideline chosen for project: Advanced Project N°1
- Details: Uses 3 bits as input, and a common cathode display

Group 3

- Member(s): Bava, Perez
- Wokwi: https://wokwi.com/projects/410732939207035905
- Guideline chosen for project: Advanced Project N°1
- Details: Uses 3 bits as input, and a common cathode display

Group 4

- Member(s): Firpo, Pursals
- Wokwi: https://wokwi.com/projects/410570046815176705
- Guideline chosen for project: Advanced Project N°1
- Details: Uses 3 bits as input, and a common cathode display

Group 5

- Member(s): Martinez
- Wokwi: https://wokwi.com/projects/410640428205329409
- Guideline chosen for project: Advanced Project N°2
- Details: Uses a common cathode display, and counts up to 8 and overflows.

Group 6

- Member(s): Nasso, Juarez
- Wokwi: https://wokwi.com/projects/410553650788005889
- Guideline chosen for project: Advanced Project N°2
- Details:Uses a common cathode display, and counts up to 16 and overflows. (Only displays correctly up to 9)

Group 7

- Member(s): Lenzuen, Gauthier
- Wokwi: https://wokwi.com/projects/410463710171875329
- Guideline chosen for project: Advanced Project N°2
- Details:Uses a common anode display, and counts up to 8 and overflows. In this case, clock is driven by ui_in1, and ui_in[0] sets the counter to 7

Group 8

- Member(s): Mendez, Vago
- Wokwi: https://wokwi.com/projects/410463176068023297
- Guideline chosen for project: Advanced Project N°2
- Details:Uses a common cathode display, and counts up to 16 and overflows. (Only displays correctly up to 9)

Group 9

- Member(s): Albín
- Wokwi: https://wokwi.com/projects/410462842465590273
- Guideline chosen for project: Basic Project
- Details: Key is 0x11

Group 10

- Member(s): Muniz
- Wokwi: https://wokwi.com/projects/410463191701250049
- Guideline chosen for project: Basic Project
- Details: Key is 0xB2

Group 11

- Member(s): Cerizola, Mesa
- Wokwi: https://wokwi.com/projects/410555856765101057
- Guideline chosen for project: Basic Project
- Details: Key is 0x80

Group 12

- Member(s): Romano, Ventós
- Wokwi: https://wokwi.com/projects/410463349567547393
- Guideline chosen for project: Basic Project
- Details: Both 0x7F and 0xFF work as key

Group 13

- Member(s): Locatelli, Roldós
- Wokwi: https://wokwi.com/projects/410639448686247937
- Guideline chosen for project: Basic Project
- Details: Key is 0x49

Group 14

- Member(s): Hernández, Pedron
- Wokwi: https://wokwi.com/projects/410643958389030913
- Guideline chosen for project: Basic Project
- Details: Key is 0x55

Schematic

External hardware

7 segment displays (common anode and common cathode) LEDs

Pinout

#	Input	Output	Bidirectional
0	ui_in[0]	uo_out[0]	
1	ui_in1	uo_out1	
2	ui_in2	uo_out2	
3	ui_in[3]	uo_out[3]	
4	mux_in[0]	uo_out[4]	
5	mux_in1	uo_out[5]	
6	mux_in2	uo_out[6]	
7	mux_in[3]		

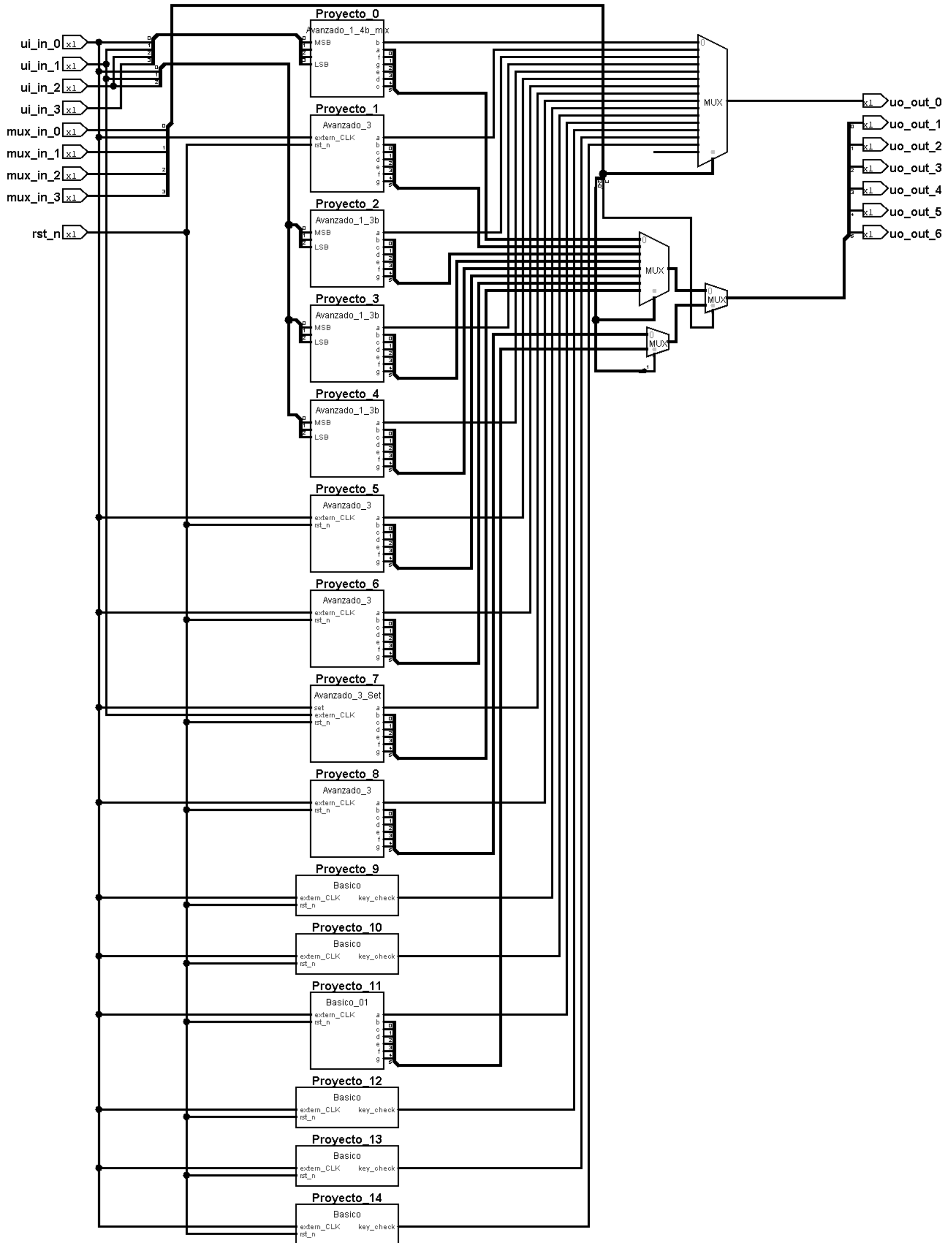


Figure 21: block diagram

4 by 4 Array Multiplier [226]

- Author: Hanyuan (Bob) Huang
- Description: 4x4 Structural Array Multiplier
- GitHub repository
- HDL project
- Mux address: 226
- Extra docs
- Clock: 0 Hz

How it works

The structural 4 by 4 binary array multiplier generates four partial products by ANDing each bit of one 4-bit input with each bit of the other. Each partial product is then shifted according to its significance (based on bit position). The shifted rows are summed using binary adders, yielding an 8-bit product. This structured approach is called an array multiplier.

How to test

To test a 4x4 binary multiplier, apply a set of 4-bit input pairs, covering typical, edge, and corner cases. For each pair, verify that the output matches the expected 8-bit product. Automate tests to check all possible inputs (total of 256 combinations) if feasible, or focus on key cases to ensure accuracy and catch potential design errors.

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	q[0]	p[0]	
1	q1	p1	
2	q2	p2	
3	q[3]	p[3]	
4	m[0]	p[4]	
5	m1	p[5]	

#	Input	Output	Bidirectional
6	m2	p[6]	
7	m[3]	p[7]	

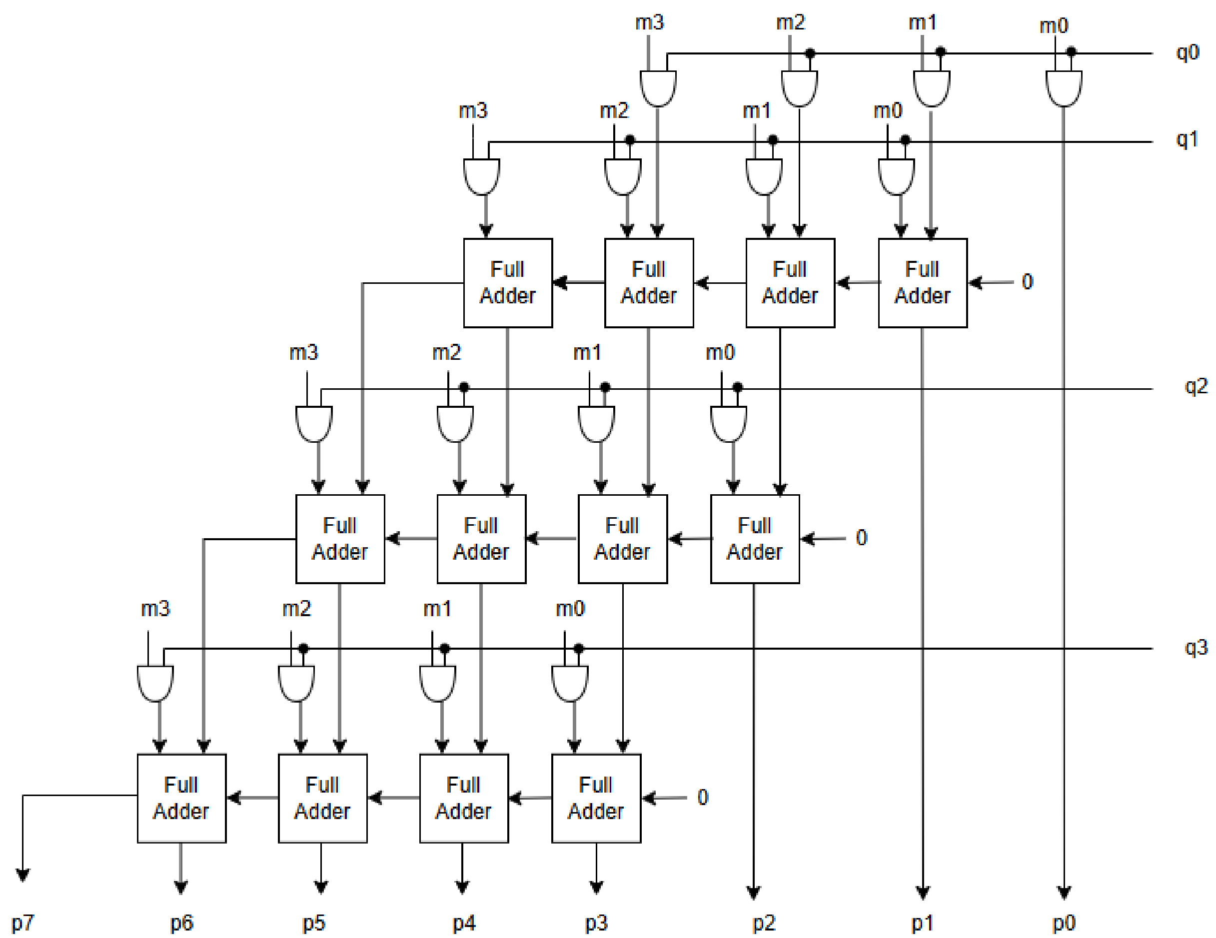


Figure 22: 4x4 Array Multiplier

4-bit multiplier [228]

- Author: Annie Huang and Sharon Chi
- Description: This project takes in two 4-bit numbers to produce an 8 bit product
- GitHub repository
- HDL project
- Mux address: 228
- Extra docs
- Clock: 0 Hz

How it works

The project takes in two 4-bit numbers and performs multiplication to produce a product p. For each bit in the 4-bit inputs, the partial product was calculated by performing bitwise AND between the two inputs, resulting in 4 different partial products. Then the partial product would be shifted according to their multiplication process by shifting them in bits. Lastly, the partial products would be summed up to find the 8-bit product.

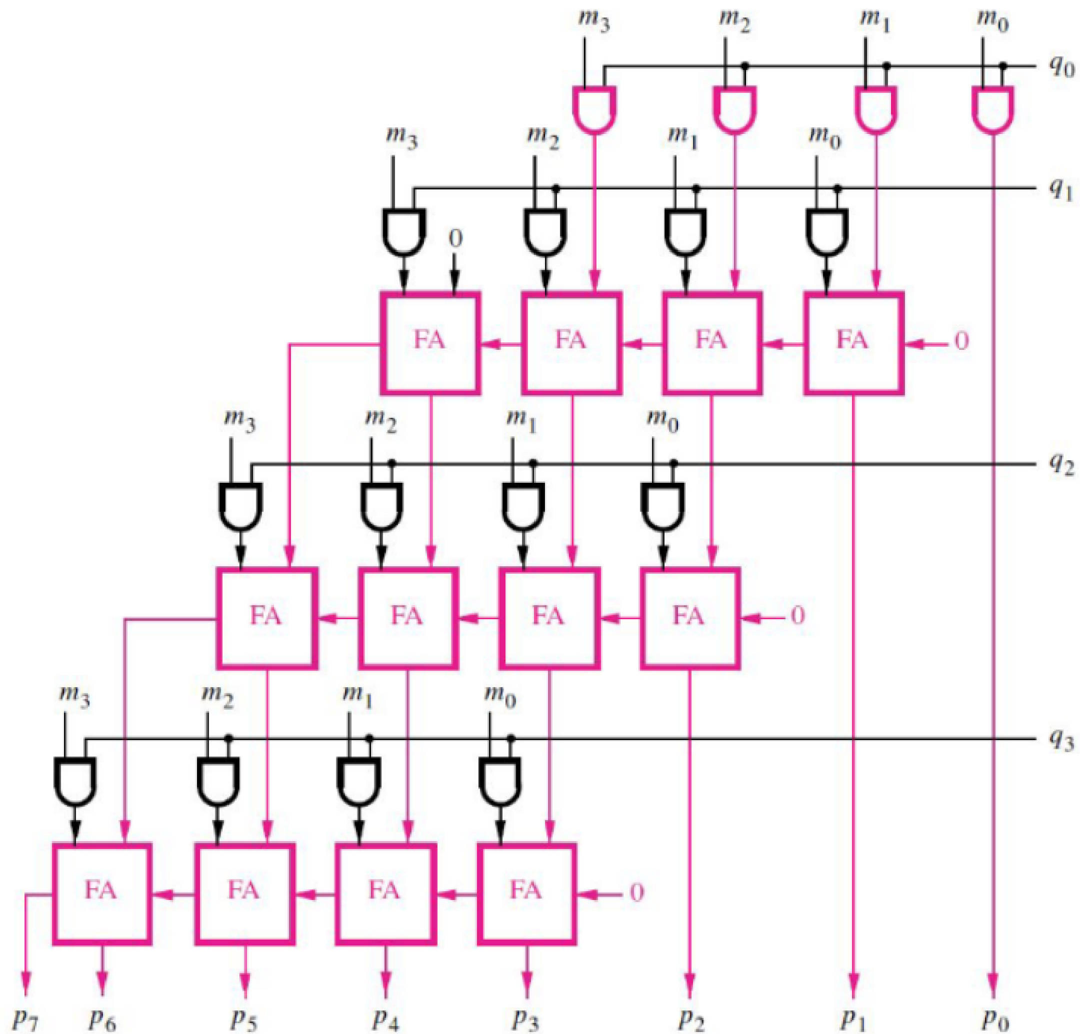


Figure 1: 4×4 Array Multiplier

How to test

We tested the code by creating a test bench giving it different inputs and the product it should get for each set of inputs after multiplication. If the result after running the code is the same as what was entered in the test bench, then the code would pass the test.

External hardware

None

Pinout

#	Input	Output	Bidirectional
0	a[0]	c[0]	
1	a1	c1	
2	a2	c2	
3	a[3]	c[3]	
4	b[0]		
5	b1		
6	b2		
7	b[3]		

OpenRAM SRAM macro [229]

- Author: K.Makise
- Description: test OpenRAM sram macro, 32x16
- GitHub repository
- HDL project
- Mux address: 229
- Extra docs
- Clock: 0 Hz

Introduction

This project is aim to test the OpenRAM macros(modified) in tt.

Structure

This project caontains 1 32x16 sram macrp, 1 sram controller, 1 UART port(RX & TX).

How to test

This project relies on the UART to communicate with. There are 2 phases to control the sram, one is the address phase, which tells the sram controller which address and which operation you want to do; Another is the data phase, depending on the operation, it could be the data being read out or the data you want to write into the sram. When transfering the address to the sram controller, in order to make " write " operation, the [5] Bit needs to be set to " 0 ", and vice versa, the [5] Bit needs to be set to" 1 " to do the " read " operation.

There's a dpu inside it, the [7] is used to activate the dpu, which will read the data in sram, do some operation, and then write back to the sram.

Just to make sure uart cmd timing. Suggest to transfer the first cmd only when uart_ready == 1

#	Input	Output	Bidirectional

Pinout

#	Input	Output	Bidirectional
0		uart_ready	
1			
2			
3	rx_in		
4			tx_out
5			
6		overrun	
7		error	

Array Multiplier [230]

- Author: Theodore Hua
- Description: Implementing a 4x4 multiplier using structural logic
- GitHub repository
- HDL project
- Mux address: 230
- Extra docs
- Clock: 0 Hz

How it works

Github Link: https://github.com/th3474/tt09-array-multiplier.git

Given two 4-bit inputs of m and q. We multiply m and q to produce the 8-bit output of p.

Figure 1: Diagram of a 4-bit multiplier: https://github.com/th3474/tt09-array-multiplier/blob/main/Multiplier%20Diagram%201.png

We implemetented a 4x4 array of row q and column m. Using logic AND gate, we fill each index of the arrays to contain the product of m[i]q[j] with i and j is the corresponding index of m and q.

We draw diagonal lines in the array as shown in the below diagram to see that every summation of all terms included in each diagonal lines is equal to each bit of the output p, starting from m[0]q[0] = p[0], m1q[0]+m1q1 = p[0],...,m[3]q[3] = p[6], with the carryout of p[6] = p[7].

To obtain each bit of the 8-bit output p, we use the 1-bit fulladder module to slowly all up the term included in each diagonal line. An 1-bit fulladder module requires 3 inputs of x, y, carry in and outputs the sum of x + y with its carryout.

Figure 2: Breakdown Diagram to implement a 4-bit multiplier: https://github.com/th3474/tt09-array-multiplier/blob/main/Multiplier%20Breakdown.png

How to test

Enter an 8-bit octaldecimal value input, with the first 4 bits represent the value of the first term m, and the last 4 bits represent the value of the second term q. The output is the 8-bit product of the first 4 bits and last 4 bits of the input.

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	q[0]	p[0]	
1	q1	p1	
2	q2	p2	
3	q[3]	p[3]	
4	m[0]	p[4]	
5	m1	p[5]	
6	m2	p[6]	
7	m[3]	p[7]	

VGA Pride [231]

- Author: Rebecca G. Bettencourt
- Description: A VGA demo for showing pride flags
- GitHub repository
- HDL project
- Mux address: 231
- Extra docs
- Clock: 0 Hz

How it works

Displays pride flags on the screen.

To add another flag, create a flag.v file and add it to `src/flag_index.v`, `test/Makefile`, and `info.yaml`, using the existing flags as examples.

How to test

Connect to a VGA monitor. Set the following inputs to change the displayed flag:

- `ui_in[7]` to display the first flag
- `ui_in[6]` to display the next flag
- `ui_in[5]` to display the previous flag
- `ui_in[4]` to display the flag whose index is on `uio_in`

Index	Flag
0	Rainbow flag, 6 stripes
1	Rainbow flag, 7 stripes
2	Rainbow flag, 8 stripes
3	Rainbow flag, 9 stripes
4	Philadelphia rainbow flag
5	Progress rainbow flag
6	Progress rainbow flag 2021 version
7	Trans pride flag
8	Abrosexual pride flag
9	Aceflux pride flag
10	Aegosexual pride flag
11	Agender pride flag
12	Androgyne pride flag
13	Androsexual pride flag

Index	Flag
14	Aporagender pride flag
15	Aroace pride flag
16	Aroflux pride flag
17	Aromantic pride flag
18	Asexual pride flag
19	Aspec pride flag
20	Bigender pride flag (pink purple white purple blue)
21	Bigender pride flag (blue white purple white pink)
22	Bigender pride flag (pink yellow white purple blue)
23	Bisexual pride flag
24	Ceterosexual pride flag
25	Demiandrogyne pride flag (pink purple blue)
26	Demiandrogyne pride flag (green white green)
27	Demiboy pride flag
28	Demifluid pride flag
29	Demiflux pride flag
30	Demigender pride flag
31	Demigirl pride flag
32	Demiromantic pride flag
33	Demisexual pride flag
34	Disability rights flag (gold silver bronze tricolor)
35	Disability rainbow flag
36	Gender-neutral pride flag
37	Genderfluid pride flag
38	Genderflux pride flag
39	Genderqueer pride flag
40	Greygender pride flag
41	Greysexual pride flag
42	Gynosexual pride flag
43	Intersex pride flag (purple circle)
44	Intersex pride flag (blue/pink gradient)
45	Thislesbianlife lesbian pride flag (pink and red)
46	Sadlesbeandisaster lesbian pride flag, 7 stripes (orange and pink)
47	Sadlesbeandisaster lesbian pride flag, 5 stripes (orange and pink)
48	Lydiandragon lesbian pride flag (violet crocus dill rose)
49	Maya Kern lesbian pride flag (violet rose crocus dill)
50	RebeccaRGB femme lesbian pride flag (violet lavender pink rose)
51	Littleender pride flag
52	Maverique pride flag
53	Leonis Ignis MLM pride flag (brown and blue)

Index	Flag
54	Vincian MLM pride flag, 7 stripes (green and blue)
55	Vincian MLM pride flag, 5 stripes (green and blue)
56	Vincian MLM pride flag (light blue and light green)
57	Multigender pride flag
58	Multisexual pride flag
59	Neptunic pride flag
60	Neutrois pride flag
61	Nonbinary pride flag
62	Objectum pride flag
63	Omnisexual pride flag
64	Pangender pride flag
65	Pansexual pride flag
66	Polyamory pride flag (blue, red, black with yellow pi)
67	Polyamory pride flag (blue, magenta, purple with yellow heart)
68	Polygender pride flag
69	Polysexual pride flag
70	Pomosexual pride flag
71	Proculsexual pride flag
72	IBM PS/2 pride flag
73	Queer pride flag
74	Trains pride flag (*Train Landscape*, Ellsworth Kelly, 1953)
75	Transfeminine pride flag
76	Transmasculine pride flag
77	Transneutral pride flag
78	Trigender pride flag
79	Unlabeled pride flag
80	Uranic pride flag
81	Voidpunk pride flag

External hardware

TinyVGA PMOD

Pinout

#	Input	Output	Bidirectional
0	address mode	R1	A0
1		G1	A1

#	Input	Output	Bidirectional
2		B1	A2
3		VSync	A3
4	set	R0	A4
5	prev	G0	A5
6	next	B0	A6
7	reset	HSync	A7

4-bit Array Multiplier [232]

- Author: Minjae Kim, Jiawei Ding
- Description: 4-bit array multiplier using structural Verilog
- GitHub repository
- HDL project
- Mux address: 232
- Extra docs
- Clock: 0 Hz

How it works

This project implements a **4-bit array multiplier** using structural Verilog. It takes two 4-bit binary inputs (`m` and `q`) and computes their product, outputting the result as an 8-bit binary number. The multiplication is performed using an array multiplier architecture, which generates partial products for each bit of the inputs and sums them using a series of adders.

Partial products are calculated based on the bits of each 4-bit input. The summation of partial products is arranged in stages, and the final result is accumulated through these stages, producing an 8-bit result (`uo_out[7:0]`).

How to test

1. **Input Setup**: Connect the first 4-bit input `m` through `ui_in[3:0]` and the second 4-bit input q through `uio_in[3:0]`.
2. **Observing Output**: The 8-bit product of `m` and `q` will be output on `uo_out[7:0]`.
3. **Clock and Reset**: Although there are `clk` and `rst_n` signals in the design, they are not utilized in this version of the array multiplier, which operates as a combinational circuit.

Example Test Case

- **Inputs**: Set `m` to `4'b0011` (decimal 3) and `q` to `4'b0010` (decimal 2).
- **Expected Output**: `uo_out` should be `8'b00000110` (decimal 6).

Testing can be performed on a simulation platform (such as Verilog testbenches in ModelSim or other simulation tools) by assigning values to `ui_in` and `uio_in` and verifying the `uo_out` output.

External hardware

This project does not require any external hardware. All inputs and outputs are managed internally within the module, which can be tested in simulation environments or FPGA-based hardware setups.

Circuit Diagram

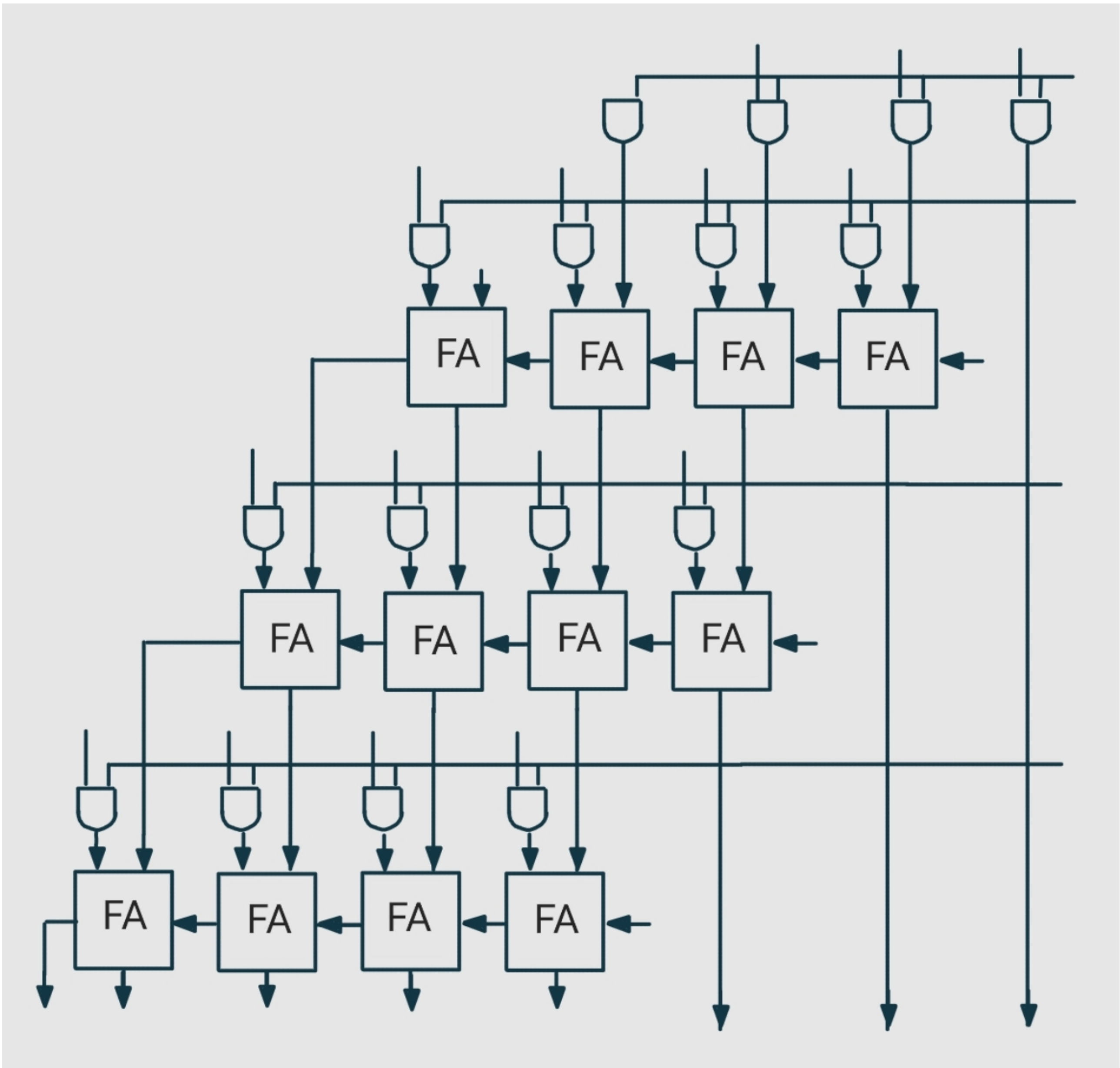


Figure 23: Circuit Diagram

Pinout

#	Input	Output	Bidirectional
0	m[0]		
1	m1		
2	m2		
3	m[3]		
4	q[0]		
5	q1		
6	q2		
7	q[3]		

Noise test for a CDAC capacitor chain [233]

- Author: Venkadesh Eswaranandam & Allan Huang
- Description: This project was made to answer the question of ow much does bad layout affect performance of a SAR ADC
- GitHub repository
- Analog project
- Mux address: 233
- Extra docs
- Clock: 0 Hz

How to test

This is a test of a C2C array of MIM capacitors to determine how suceptible they are to noise. Clock the digital pins to see what areas are most suceptible to noise. This layout was done intentionaly bad to determine as to what degree layout matters.

External hardware

No external hardware used.

Pinout

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

Analog pins

`ua#`	`analog#`	Description
0	5	Analog output

ECE-UY 2204 4x4 Array Multiplier [234]

- Author: Jane Manalu, Isabella Menshouse, KJ Moses
- Description: Performs a 4x4 structural array multiplier using fulladder
- GitHub repository
- HDL project
- Mux address: 234
- Extra docs
- Clock: 0 Hz

How it works

This project will receive two four-bit numbers, where both will be multiplied by the 4x4 array multiplier. The multiplier executes using combinations of full adders (FAs) and two-input AND gates, linked together as shown in the attached circuit diagram. Each row represents partial products generated by the AND gates, where each bit of one number is ANDed with each bit of the other number. The results are then added column by column using full adders, with carries propagated to the next stage. The final outputforms the product of the two four-bit input numbers

How to test

To test this 4x4 array multiplier, provide an 8-bit number, which then will be split into two 4-bit inputs by the multiplier's design source. The circuit will calculate the product, displayed on output. It will compare this output to the expected result to verify the answer. Testing with different 8-bit values will ensure reliable functionality across various inputs.

External hardware

N/A

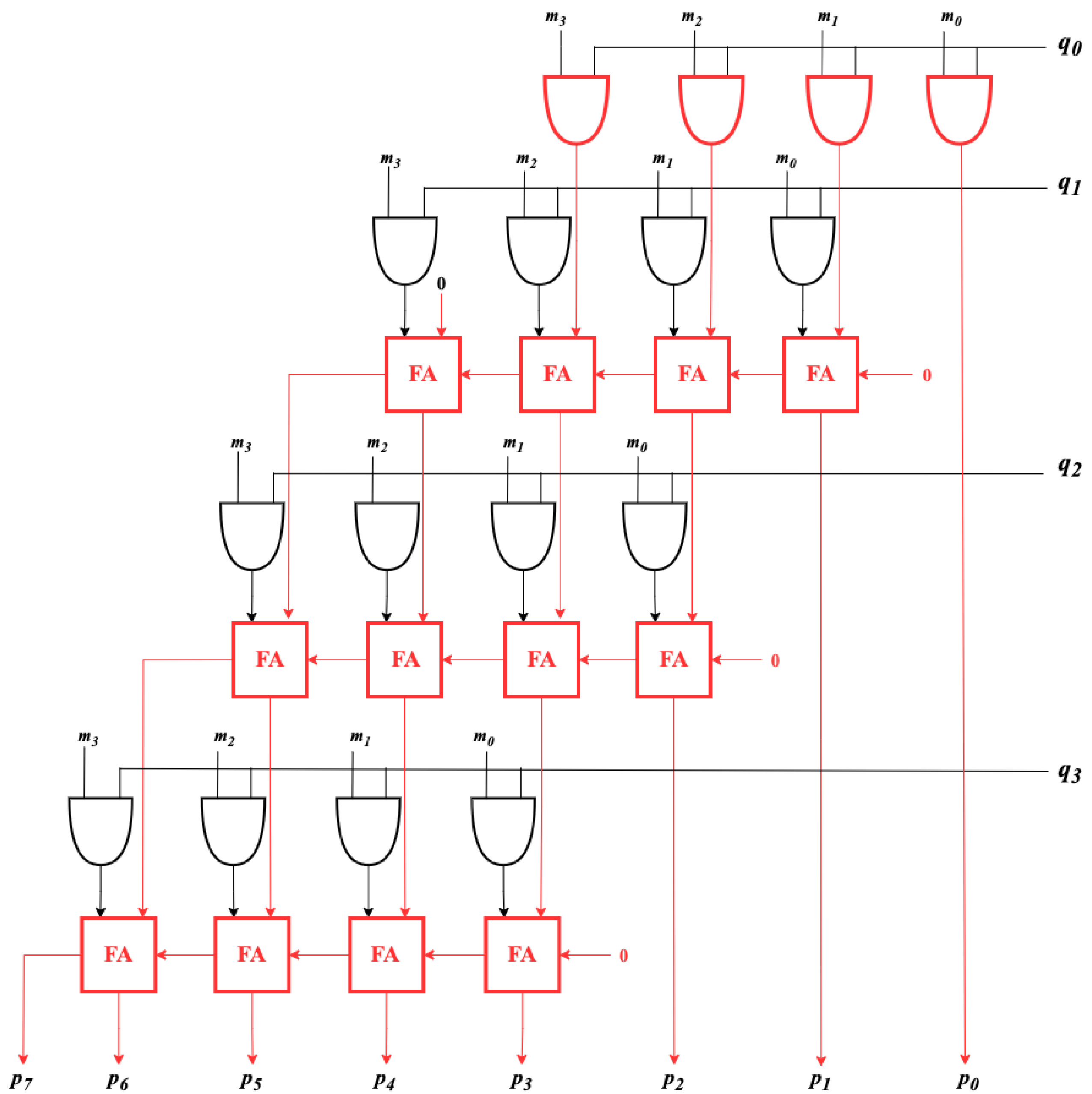


Figure 1: Logic Diagram for 4X4 Array Multiplier.

Pinout

#	Input	Output	Bidirectional
0	q[0]	p[0]	
1	q1	p1	
2	q2	p2	
3	q[3]	p[3]	
4	m[0]	p[4]	
5	m1	p[5]	
6	m2	p[6]	

#	Input	Output	Bidirectional
7	m[3]	p[7]	

Analog Switch [235]

- Author: Andrew Dona-Couch
- Description: A simple analog switch.
- GitHub repository
- Analog project
- Mux address: 235
- Extra docs
- Clock: 0 Hz

How it works

This is a basic analog switch. When the control input is high, the X and Y analog inputs are connected. When the control input is low, they are not connected.

How to test

Connect X to digital high. Connect a pull-down resistor from Y to ground.

Change the value of the control input. Verify that the inverse of the control input is correct. Verify that the voltage at Y matches.

Pinout

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

Analog pins

ua#	analog#	Description
0	4	X
1	1	Y

array_multiplier [236]

- Author: xg2523_cw4483
- Description: a 4-bit multiplier
- GitHub repository
- HDL project
- Mux address: 236
- Extra docs
- Clock: 0 Hz

Array Multiplier

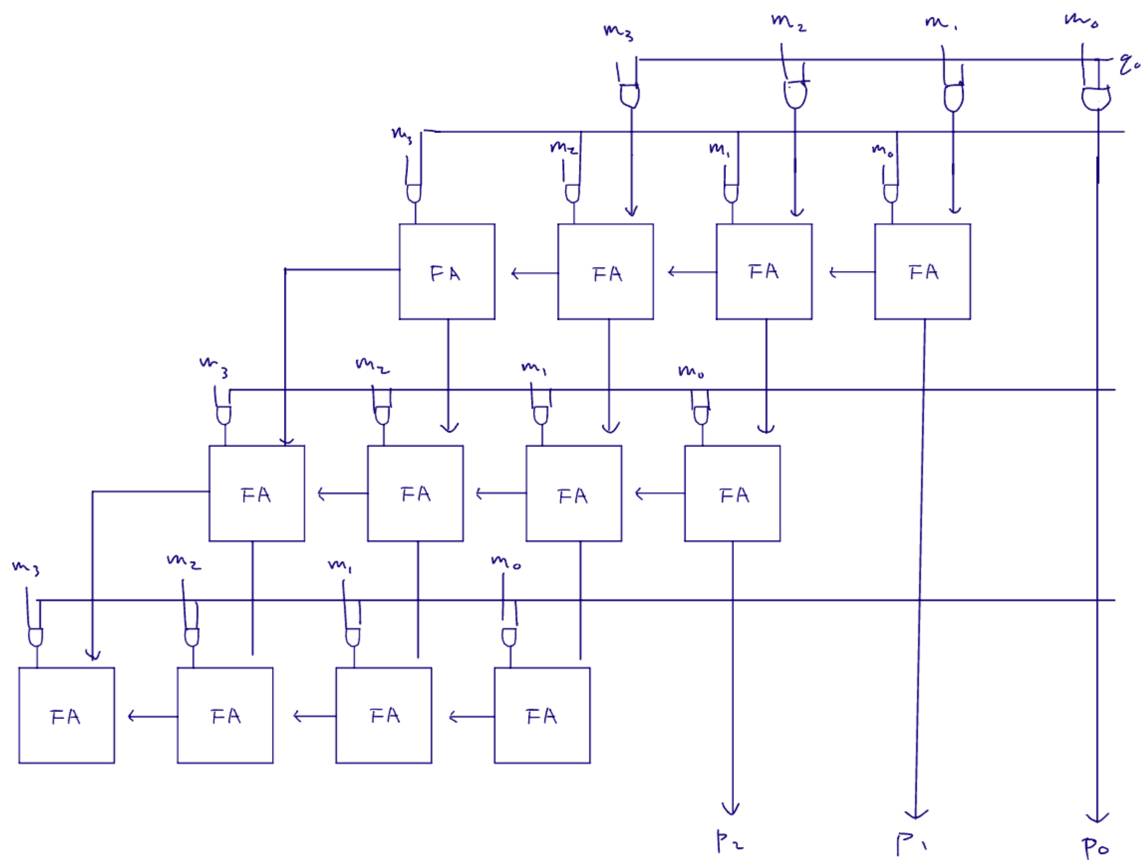
A 4-bit multiplier that outputs 8-bit result. Author: Xiaoyu Guan, Andy Wu xg2523 cw4483 Lab D 8

How it works

Partial Product Generation: The m and q inputs are both 4-bit binary numbers. To perform multiplication, partial products are generated by performing AND operations between each bit of m and each bit of q. These partial products are assigned to wires mp0, mp1, mp2, and mp3.

Summing Partial Products Using Full Adders: The first row of partial products (mp0) is directly assigned to s0, representing the first sum. There is no carry-in for this row. The partial products are added together using a series of full adders (full_adder module). A full adder takes three inputs: two data bits and a carry-in and outputs a sum and a carry-out. The subsequent rows of partial products (mp1, mp2, and mp3) are added together row by row using the full adders, and the resulting sums and carry bits are propagated forward to the next stage.

Final Product: After the partial products are added in the stages using the full adders, the final product is formed. The final result, p, is an 8-bit number that represents the product of the 4-bit multiplicand and the 4-bit multiplier.



How to test

The 8-bit input represents two 4_bit inputs, the 8-bit output should be the product(all unsigned binary numbers) For example: input:00010010 This means the input m=0001,q=0010. The operation is (in decimal) 12*, result p=12*=2. So output P should be 00000010.

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	m[0]	p[0]	

#	Input	Output	Bidirectional
1	m1	p1	
2	m2	p2	
3	m[3]	p[3]	
4	q[0]	p[4]	
5	q1	p[5]	
6	q2	p[6]	
7	q[3]	p[7]	

Digital OTA [237]

- Author: UABC
- Description: Low_Voltage operational transconductance amplifier
- GitHub repository
- Analog project
- Mux address: 237
- Extra docs
- Clock: 10000 Hz

How it works

The following circuit amplifies transconductance, high output impedance

How to test

Use a function generator

External hardware

No external hardware used

Pinout

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

Analog pins

ua#	analog#	Description
0	5	Vo
1	0	VA
2	4	VB

8-bit-CARRY_SKIP [238]

- Author: Aaquil Kasham, Temiloluwa Omomuwasan
- Description: 8 bit input adder
- GitHub repository
- HDL project
- Mux address: 238
- Extra docs
- Clock: 0 Hz

How it works

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

How to test

To test the carry-skip adder:

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

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

External hardware

No external hardware is required for this project. List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Pinout

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

Telephone hybrid [239]

- Author: htfab
- Description: Converts two unidirectional wires to a bidirectional one
- GitHub repository
- Analog project
- Mux address: 239
- Extra docs
- Clock: 0 Hz

How it works

To be added later. In the meanwhile, see the Wikipedia page for Telephone hybrid.

How to test

Add a wire between LINE1 and LINE2 (or LINE1 and the line pin of another hybrid). Voltage signals sent to IN1 should appear on OUT2 while those sent to IN2 should appear on OUT1.

External hardware

Analog test equipment (e.g. function generator and oscilloscope)

Pinout

#	Input	Output	Bidirectional
0	divider bit 0	debug out 0	debug out 8
1	divider bit 1	debug out 1	debug out 9
2	divider bit 2	debug out 2	debug out 10
3	divider bit 3	debug out 3	debug out 11
4	pass gate / debug in 0	debug out 4	debug out 12
5	debug in 1	debug out 5	debug out 13
6	debug in 2	debug out 6	debug out 14
7	debug in 3	debug out 7	debug out 15

Analog pins

`ua#`	`analog#`	Description
0	5	IN1
1	0	OUT1
2	4	LINE1
3	1	LINE2
4	3	OUT2
5	2	IN2

Array Multiplier [256]

- Author: Jeryl Ho & Justin Park
- Description: 4x4 Structural Array Multiplier
- GitHub repository
- HDL project
- Mux address: 256
- Extra docs
- Clock: 0 Hz

How it works

An array multiplier is a combinational circuit that performs binary multiplication by generating and summing partial products. Here's how a 4x4 array multiplier operates:

1. **Binary Multiplicand and Multiplier**: A 4x4 multiplier takes two 4-bit binary numbers (e.g., (A = A_3 A_2 A_1 A_0) and (B = B_3 B_2 B_1 B_0)) as inputs. Each bit in (A) is multiplied by each bit in (B), creating 16 partial products.

2. **Partial Product Generation**: Each bit in (A) is ANDed with each bit in (B), forming a matrix of partial products. For instance, if (A = 1011) and (B = 1101), then (A_3 $\times$B_3), (A_3 $\times$B_2), and so forth are calculated.

3. **Shifting and Summing**: Each row of partial products corresponds to a shifted version based on the position of the bits in (B). For example, the row generated by (A_3) will be shifted three places to the left.

4. **Adding Partial Products**: The shifted partial products are summed column by column, similar to traditional addition in binary, often using full adders or half adders.

5. **Final Product**: The result is an 8-bit product that represents the multiplication of the two 4-bit inputs.

Here's a visual representation of how an array multiplier works:

```
graph TD
    A[Multiplicand A - Bits A3 A2 A1 A0]
    B[Multiplier B - Bits B3 B2 B1 B0]

    subgraph Partial_Products
        A3B0[A3 * B0] --> A3B1[A3 * B1] --> A3B2[A3 * B2] --> A3B3[A3 * B
        A2B0[A2 * B0] --> A2B1[A2 * B1] --> A2B2[A2 * B2] --> A2B3[A2 * B
```

```
        A1B0[A1 * B0] --> A1B1[A1 * B1] --> A1B2[A1 * B2] --> A1B3[A1 * B
        A0B0[A0 * B0] --> A0B1[A0 * B1] --> A0B2[A0 * B2] --> A0B3[A0 * B
    end

    Sum[Sum of Partial Products]
    Output[Final Product]

    A --> Partial_Products
    B --> Partial_Products
    Partial_Products --> Sum
    Sum --> Output
```

How to test

To test the array multiplier:

1. Set up the multiplier by providing binary inputs for both the multiplicand (A) and the multiplier (B).
2. Run the simulation or test in hardware to verify that each partial product is calculated correctly.
3. Ensure that the partial products are properly shifted and summed to produce the final product.
4. Compare the final output with the expected result from standard binary multiplication to confirm accuracy.

External hardware

No external hardware is required for this project. The array multiplier can be tested within a simulation environment or with an FPGA setup if hardware verification is needed.

Pinout

#	Input	Output	Bidirectional
0	q[0]	p[0]	
1	q1	p1	
2	q2	p2	
3	q[3]	p[3]	
4	m[0]	p[4]	

#	Input	Output	Bidirectional
5	m1	p[5]	
6	m2	p[6]	
7	m[3]	p[7]	

Array multiplier [258]

- Author: Wyte wu ,Xintong Hu
- Description: 4x4 Structural Array multiplier
- GitHub repository
- HDL project
- Mux address: 258
- Extra docs
- Clock: 0 Hz

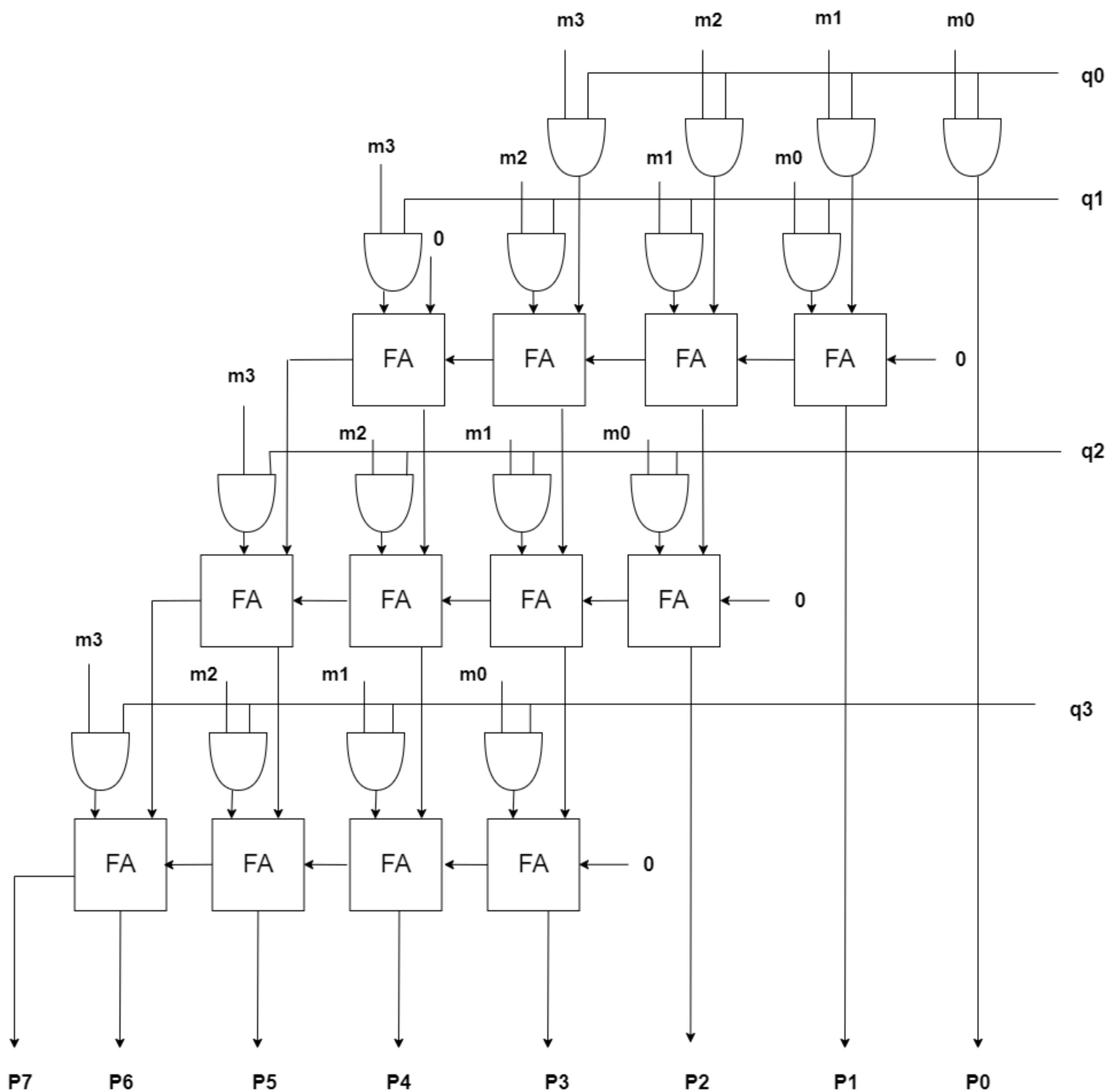


Figure 24: alt text

How it works

When multiplying two 4-bit numbers, m = 1011 (11 in decimal) and q = 1101 (13 in decimal), the multiplier first generates partial products (m0, m1, m2, and m3) by AND-ing each bit of q with all bits of m. For instance, m0 results from q[0] & m, producing 1101, and similarly, m2 and m3 are also 1101, while m1 is 0000 since q[1] is zero. These partial products are then summed column-wise using full adders, combining overlapping bits and propagating carries. For example, in the first column, p[0] is directly assigned from m0[0], which is 1. Moving to the second column, we add m0[1] and m1[0] with any carry (which is zero in this case), giving a sum of 1 and a carry of zero, resulting in p[1] = 1. In the third column, adding m0[2], m1[1], and the carry results in a sum of 0 and a carry of zero, making p[2] = 0. In the fourth column, adding m0[3] and m1[2] with zero carry gives a sum of 1 and no carry, so p[3] = 1. Continuing this process for all columns and partial products, the final 8-bit product p is formed as 10011111 (143 in decimal), representing the correct product of 11 and 13.

How to test

The Cocotb testbench for your project sets up a clock running at 100 KHz and initializes the design by asserting and de-asserting a reset signal (rst_n). It tests our Verilog module by setting various input values (ui_in, uio_in) and checking the resulting output (uo_out) using assertions to ensure correctness. For each test case, like **Test Case 3**, the test sets specific input values (e.g., 0x2_6), waits for one clock cycle to allow the inputs to propagate through the design, and verifies that the output matches an expected value (in this case, 12). This structured approach allows us to efficiently validate the behavior of our module for different input scenarios and edge cases.

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	q[0]	p[0]	
1	q1	p1	
2	q2	p2	

#	Input	Output	Bidirectional
3	q[3]	p[3]	
4	m[0]	p[4]	
5	m1	p[5]	
6	m2	p[6]	
7	m[3]	p[7]	

Array Multiplier [260]

- Author: Rebecca Boadu & Sarah Herrera
- Description: 4x4 Structural Array Multiplier
- GitHub repository
- HDL project
- Mux address: 260
- Extra docs
- Clock: 0 Hz

How it works

Our 4x4 array multiplier multiplies two 4-bit binary numbers to produce an 8-bit product by breaking down the process into partial products. Each bit of one input is ANDed with each bit of the other input, creating a 4x4 grid of partial products. These partial products are organized into rows and shifted leftward, simulating the alignment process in traditional multiplication. Full adders within each "box" of the grid add these partial products, managing both the sum and the carry bits. Each carry moves to the next box, allowing us to systematically add all rows and carry values. Once all additions are complete, the result is an accurate 8-bit binary product of the two inputs. A block diagram is also included to illustrate the structure of partial products, adders, and carry management within the multiplier.

How to test

Testing the 4x4 array multiplier is handled with a Verilog testbench that instantiates the module, wiring inputs and outputs, along with a Python script for validation. The Python script applies specific input values, then checks output correctness using assertions to verify expected multiplication results. For each test, values are set for the inputs, clock cycles are awaited, and the output is asserted to match the expected product, making it easy to identify any errors.

External hardware

N/A

Pinout

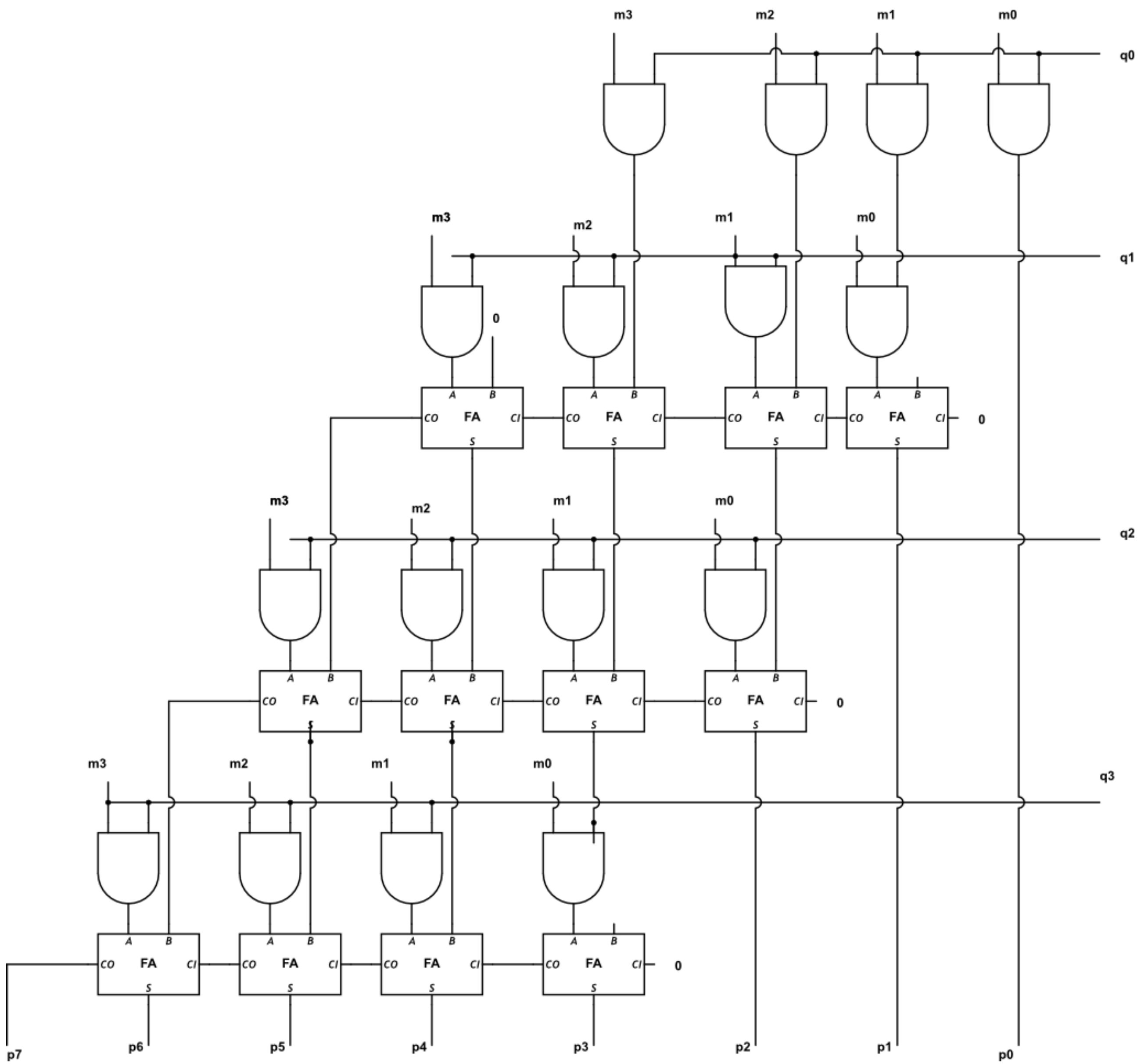


Figure 25: 4x4 Multiplier Diagram

#	Input	Output	Bidirectional
0	q[0]	p[0]	
1	q1	p1	
2	q2	p2	
3	q[3]	p[3]	
4	m[0]	p[4]	
5	m1	p[5]	
6	m2	p[6]	
7	m[3]	p[7]	

1bit_am_sdr [261]

- Author: James Sharp
- Description: 1bit AM software defined radio
- GitHub repository
- HDL project
- Mux address: 261
- Extra docs
- Clock: 50000000 Hz

How it works

This project is a Software Defined Radio pipeline for AM radio reception written in verilog. It works using an external comparator as a 1-bit ADC frontend which is oversampled and decimated 4096 times to give an extra 6 bits of precision. It is based on this Hackaday Project: https://hackaday.io/project/170916-fpga-3-r-1-c-mw-and-sw-sdr-receiver by Alberto Garlassi.

Although this is a fully digital core, but there are plans to make an analog frontend circuit as an analog design in future Tiny Tapeouts, so both designs would be hooked up together to create a radio with few external components.

Also, this core is very big - 3x2 Tiny Tapeout tiles (@ 64% utilisation). An area of future development could be to simplify the demodulation pipeline to reduce gate count.

How to test

You need to connect an external comparator and RC network. You will probably need a simple RF amplifier as well. See below for more information.

The core has a SPI interface for setting the demodulation frequency and gain. It consists of a single 32-bit shift register. It has the following format:-

Bits 31 - 30	Bits 29 - 26	Bits 25 - 0
Unused	Gain	NCO Phase incr.

The gain can take on the following values:

"Gain" value	Actual Gain
0	x1

“Gain” value	Actual Gain
1	x2
2	x4
3	x8
4	x16
5 - 7	x32

If the gain is set too high, the demodulated signal will wrap and sound distorted, so adjust the gain down to the minimum needed to hear the station clearly.

The “NCO Phase increment” is the value that is added to the NCO phase every clock cycle. Use the following python code to calculate the value to write, based on the desired carrier frequency:

```
hex(int((1<<26) * <carrier frequency> / <chip clock frequency>))
```

E.g., for 936kHz (ABC Radio national Hobart) at 50MHz clock frequency, it would be:

```
> hex(int((1<<26) * 936000 / 50000000))
'0x132b55'
```

External hardware

- External comparator
- Resistor bias network
- RC network
- External SPI microcontroller to set station
- RF amplifier

Pinout

#	Input	Output	Bidirectional
0	COMP_IN	COMP_OUT	
1	SPI_MOSI	PWM	
2	SPI_SCK		
3	SPI_CSb		
4			

#	Input	Output	Bidirectional
5			
6			
7			

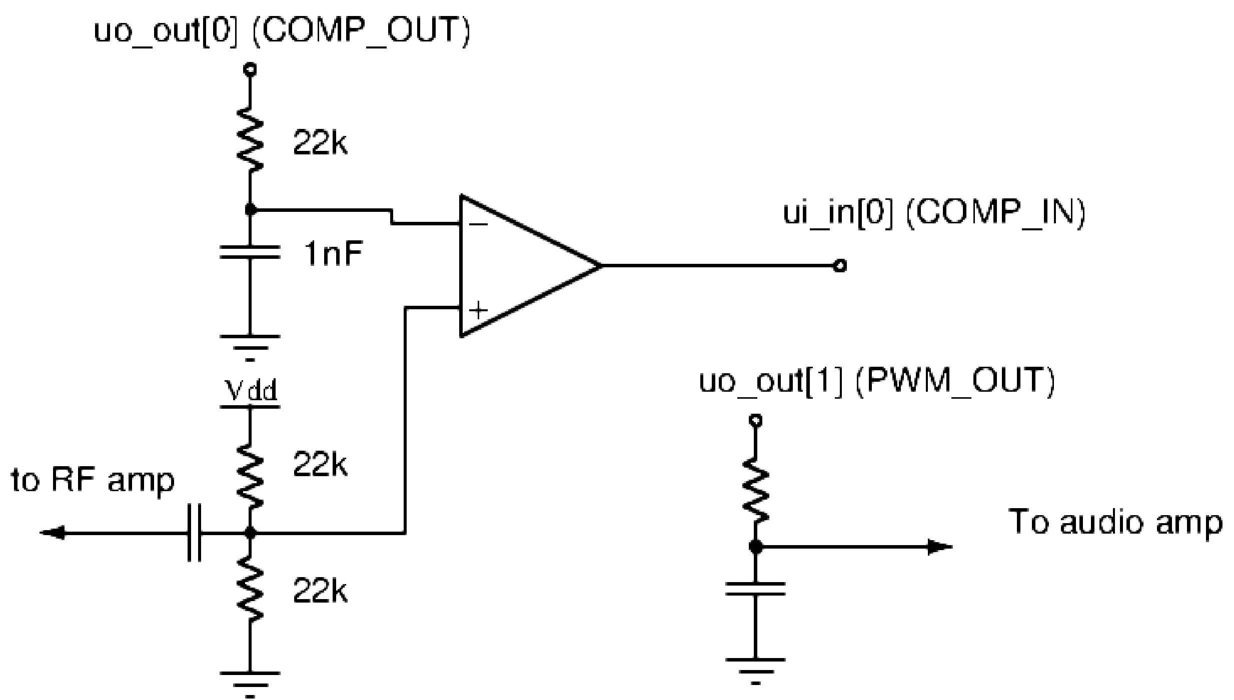


Figure 26: Schematic diagram of external circuitry

Array Multiplier [262]

- Author: Will Shang, Tyler Huynh
- Description: 4x4 Array Multiplier that multiplies two four-bit numbers
- GitHub repository
- HDL project
- Mux address: 262
- Extra docs
- Clock: 0 Hz

How it works

This project uses a 4x4 Array Multiplier to multiplies two four-bit numbers together, using a series of full adders to result in an 8 bit product (figure 1). The multiplier works by systematically multiplying each bit of the first number with each bit of the second number. These partial products are then combined using a series of full adders to form the final result.

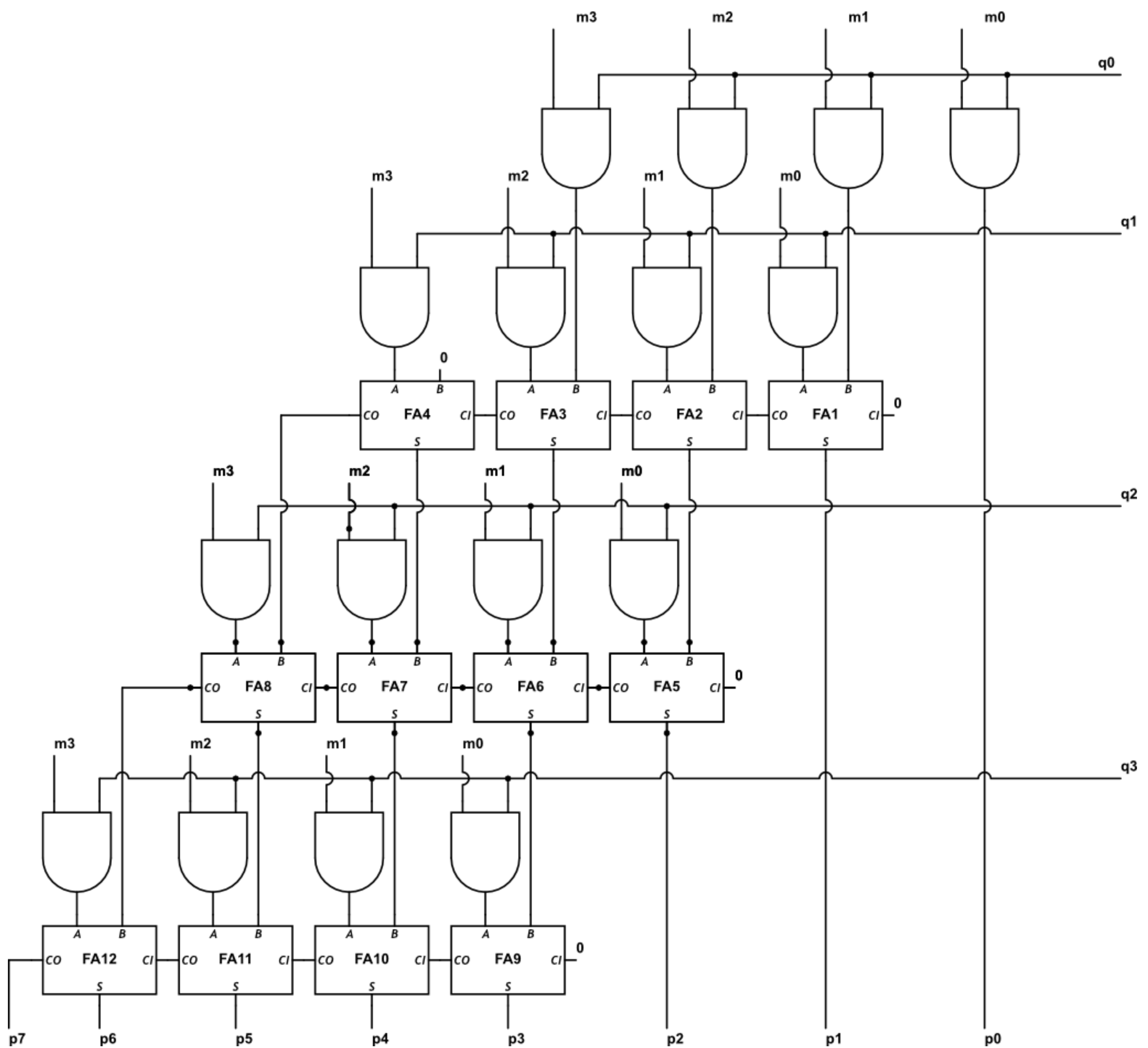


Figure 1: 4x4 Array Multiplier

How to test

Input two 4-bit binary numbers and manually verify the output. For example: 1st num: 1001 2nd num: 1011 Output: 1100011 (binary), or 0x63 (hexadecimal) The format of the output can be adjusted in test.py, but the value they represent should be accurate to the product of the two 4-bit binary numbers.

Pinout

#	Input	Output	Bidirectional
0	m[0]	p[0]	

#	Input	Output	Bidirectional
1	m1	p1	
2	m2	p2	
3	m[3]	p[3]	
4	q[0]	p[4]	
5	q1	p[5]	
6	q2	p[6]	
7	q[3]	p[7]	

Time to Digital Converter [263]

- Author: Jeremiasz Hauck
- Description: Phase difference measuring circuit with digital output
- GitHub repository
- Analog project
- Mux address: 263
- Extra docs
- Clock: 0 Hz

How it works

A Pseudo-Differential Time to Digital Converter (TDC)

How to test

The TDC has one analog input that is then split into start and stop signals. Because this TDC has a resolution of around 80 ps, it would be difficult to provide signals with such a small phase difference, that is why there is an extra variable delay circuit that delays the stop signal relative to the start signal. You can change the stop signal delay by configuring the digital input. To test the circuit drive the stop signal for a given configuration of delay.

External hardware

No external hardware needed.

Pinout

#	Input	Output	Bidirectional
0	stop fine delay 0	tdc bit 0	stop coarse delay 4
1	stop fine delay 1	tdc bit 1	start fine delay 0
2	stop fine delay 2	tdc bit 2	start fine delay 1
3	stop fine delay 3	tdc bit 3	start fine delay 2
4	stop coarse delay 0	tdc bit 4	start fine delay 3
5	stop coarse delay 1	tdc bit 5	start enable
6	stop coarse delay 2	tdc bit 6	
7	stop coarse delay 3	tdc bit 7	

Analog pins

ua#	analog#	Description
0	11	stop

Delta RNN and Leaky Integrate-and-Fire Nueron Circuit [264]

- Author: Katherine Rogacheva
- Description: A physical representation of a delta recurrent neural network (Delta RNN) and a leaky integrate-and-fire (LIF) neuron, that creates an artificial spike when the difference in the previous and current state is greater than a set delta threshold.
- GitHub repository
- HDL project
- Mux address: 264
- Extra docs
- Clock: 0 Hz

How it works

Takes inputput voltages and treats that as the input current injection into the LIF neuron

How to test

N/A

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	Input current bit [0]	State variable bit [0]	Difference in states bit 1
1	Input current bit 1	State variable bit 1	Difference in states bit 2
2	Input current bit 2	State variable bit 2	Difference in states bit [3]
3	Input current bit [3]	State variable bit [3]	Difference in states bit [4]

#	Input	Output	Bidirectional
4	Input current bit [4]	State variable bit [4]	Difference in states bit [5]
5	Input current bit [5]	State variable bit [5]	Difference in states bit [6]
6	Input current bit [6]	State variable bit [6]	Difference in states bit [7]
7	Input current bit [7]	State variable bit [7]	Difference in states bit [8]

tt_um_tim2305_adc_dac [265]

- Author: Timonas Juonys
- Description: 8bit dac and 4bit flash adc
- GitHub repository
- Analog project
- Mux address: 265
- Extra docs
- Clock: 50000000 Hz

How it works

8 bit r2r dac inputs are connected directlly to the digital input pins. its output can be connected to the analog pin by setting the dac_connect pin high. the connection is made by a transmission gate. 4bit flash adc has an input range of 0-1 volts, it refference voltages are set by a resistive voltage divider. the upper bound (the top voltage) used by the divider can be connected to the analog pin for calibration by setting adc_cal_connect pin high. The output of the adc is multiplexed on 3 4 bit busses, this way the frequency on the digital outputs pins is 3 times lower than the clock.

How to test

to test the dac: set adc_connect = 1 and adc_cal_connect = 0 put your number on the 8 input pins and read the analog voltage

to test the adc just clock the design and read the bus pins. you can use the internal dac or you can disconnect the dac and connect an external voltage source.

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	dac0	bus0[0]	bus2[0]
1	dac1	bus01	bus21
2	dac2	bus02	bus22

#	Input	Output	Bidirectional
3	dac3	bus0[3]	bus2[3]
4	dac4	bus1[0]	dac_conn
5	dac5	bus11	adc_cal_conn
6	dac6	bus12	sti_conn
7	dac7	bus1[3]	sti_dac_conn

Analog pins

ua#	analog#	Description
0	10	a

Verilog ring oscillator [266]

- Author: algofoogle (Anton Maurovic)
- Description: Simple ring oscillator by instantiating a sky130 inv_2 inverter ring
- GitHub repository
- HDL project
- Mux address: 266
- Extra docs
- Clock: 0 Hz

What is this?

Everyone has done a ring oscillator using inverter cells. Now it's my turn!

This simple example uses verilog to instantiate a ring of (an odd number of) `sky130_fd_sc_hd__inv_2` cells.

It produces its output on `uo_out[0]`.

Assuming each inverter introduces a delay of ~70ps, and there are 1001 of them, then hopefully this will oscillate at ~14MHz?

Pinout

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

2-bit Flash ADC [267]

- Author: Brandon S. Ramos
- Description: Flash ADC that outputs 2-bit encoded data
- GitHub repository
- Analog project
- Mux address: 267
- Extra docs
- Clock: 0 Hz

How it works

Converts a 1.8v to 0v analog signal into an encoded 2-bit digital signal

How to test

You can test by taking a voltage of 0 to 1.8v to the analog pin 0 and the encoded bits come out to the dedicated outputs 0 and 1

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0		encoded_out_0	
1		encoded_out_1	
2			
3			
4			
5			
6			
7			

Analog pins

ua#	analog#	Description
0	9	analog_pin_in
1	8	

Adaptive Leaky Integrate and Fire Neuron [268]

- Author: Sydnie Figuerres
- Description: Simulates an adaptive leaky integrate and fire neuron
- GitHub repository
- HDL project
- Mux address: 268
- Extra docs
- Clock: 0 Hz

How it works

This neuron model is an adaptive leaky integrate and fire neuron. It behaves similarly to a traditional leaky integrate and fire neuron(LIF), but takes into consideration the frequency of spikes occurring. In other words, if a simple LIF were to spike at a consistent rate, the adaptive LIF model will spike less often over time.

How to test

The file test.py shpuld be used to test this model. By applying varying input values with an expected output, you can measure the accuracy of this implementation of an adaptive leaky integrate and fire neuron.

External hardware

There is no external hardware for this model.

Pinout

#	Input	Output	Bidirectional
0	Input current bit[0]	State variable bit[0]	
1	Input current bit1	State variable bit1	
2	Input current bit2	State variable bit2	
3	Input current bit[3]	State variable bit[3]	

#	Input	Output	Bidirectional
4	Input current bit[4]	State variable bit[4]	
5	Input current bit[5]	State variable bit[5]	
6	Input current bit[6]	State variable bit[6]	
7	Input current bit[7]	State variable bit[7]	Spike bit

pll [269]

- Author: Mickey Cheah
- Description: 100 Mhz PLL
- GitHub repository
- Analog project
- Mux address: 269
- Extra docs
- Clock: 0 Hz

How it works

100MHz VCO, vary vctrl to change frequency from ~100MHz to 180Mhz

How to test

Apply vctrl=1.0V, observe vo or vo_dig pins

External hardware

none

Pinout

#	Input	Output	Bidirectional
0		up	
1		vosc_16	
2		down	
3			
4			
5			
6			
7	vref	vo_dig	

Analog pins

ua#	analog#	Description
0	10	vo
1	11	vstart
2	6	vctrl

Matmul System [270]

- Author: Abarajithan
- Description: Matmul System
- GitHub repository
- HDL project
- Mux address: 270
- Extra docs
- Clock: 0 Hz

How it works

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

How to test

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

External hardware

None

Pinout

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

Analog MUX module [271]

- Author: Pat Deegan
- Description: Pipe 4 or 8 analog signals around through switchable passgates, useful for adding debug and inspection to analog projects without wasting many pins
- GitHub repository
- Analog project
- Mux address: 271
- Extra docs
- Clock: 0 Hz

How it works

This is a package that gives two different mulitplexers a spin. The core is a set of 4 passgates (letting signals pass between A and Z sides) which are "one-hot", so one of them is enabled at a time. If you tie all the Zs together, you have a simple 4:1 mux. The 8 passgate version is an extension of this, and it may be used as 8 analog switches (only one on at a time) or an 8:1 mux.

In order to give it a good run, numerous test scenarios were implemented that allow charactirizing the passgates themselves, as well as chaining the muxes together.

This system has, in addition to the muxes, a ring oscillator and driver (created by Matt Venn), a manually laid out digital block to convert bits to the one-hot signals needed to control the passgates, and an openlane generate simple counter, that is clocked directly from the ring oscillator, such that we can drive it at hundreds of megahertz, divide down the clocking and shoot it through the mux over an analog pin.

How to test

Watch my video.

External hardware

Analog stuff.

Pinout

#	Input	Output	Bidirectional
0	RSEL0		
1	RSEL1		
2	RSEL2		
3	SEL0		
4	SEL1		
5	enable_counter		
6	enable_ringosc		
7			

Analog pins

ua#	analog#	Description
0	10	VRES
1	7	RINGOUT
2	9	MUXOUT
3	8	LADDEROUT

Steven's Wokwi Test [288]

- Author: Steven Abrego
- Description: Switch to 7-Segment Display
- GitHub repository
- Wokwi project
- Mux address: 288
- Extra docs
- Clock: 0 Hz

How it works

Switch number N (1-8) will display digit "N" on the 7-segment display

How to test

Flip switch N, where N is switch 1-8, and the corresponding digit will appear on the displau

External hardware

8-Switch array, 7-Segment Display (common cathode)

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	

2-Bit-Adder [289]

- Author: Jamin
- Description: 2 Bit Adder
- GitHub repository
- Wokwi project
- Mux address: 289
- Extra docs
- Clock: 0 Hz

How it works

Serves as a 2-bit adder with IN1-4 as inputs and OUT1-3 as outputs. IN0 is connected to OUT0 via an inverter, and this is not part of the 2-bit adder.

How to test

Toggle IN1-4 between HIGH and LOW and observe the OUT1-3, which should follow a 2-bit adder.

External hardware

LED display

Pinout

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

8-Bit CPU [290]

- Author: University of Waterloo - Fall 2024 ECE 298A
- Description: A basic 8-bit CPU design building off the SAP-1
- GitHub repository
- HDL project
- Mux address: 290
- Extra docs
- Clock: 50000000 Hz

How it works

This project is a basic 8-bit CPU design building off the SAP-1. It is a combination of various modules developed as a part of the ECE298A Course at the University of Waterloo.

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

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

The Instruction register stores the current instructions and breaks it up into the opcode and address, which are passed into corresponding locations

The 16 Byte memory module consists of 16 memory locations that store 1 byte each. The memory allows for both read and write operations, controlled by input signals, as well as data supplied by the MAR.

The MAR is a register which handles RAM interactions, namely specifying the address for store/load, as well as the data to be stored.

The 8-bit ripple carry adder assumes 2s complement inputs and thus supports addition and subtraction. It pushes the result to the bus via tri-state buffer. It also includes a zero flag and a carry flag to support conditional operation using an external microcontroller. These flags are synchronized to the rising edge of the clock and are updated when the adder outputs to the bus.

The Accumulator register functions to store the output of the adder. It is synchronized to the positive edge of the clock. The accumulator loads and outputs its value from the bus and is connected via tri-state buffer. The accumulator's current value is always available as an output (and usually connected to the Register A input of the ALU)

The B register stores the second operand for ALU operations which is loaded from RAM.

The Output register outputs the value from register A onto the uo_out pins.

The 8 Bit Bus is driven by various blocks. We allow multiple blocks that are able to write using tri-state buffers.

Supported Instructions

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

Instruction Notes

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

Control Signal Descriptions

Control Signal	Array	Component	Function
Cp	14	PC	Increments the PC by 1
Ep	13	PC	Enable signal for PC to drive the bus
Lp	12	PC	Tells PC to load value from the bus
nLma	11	MAR	Tells MAR when to load address from the bus
nLmd	10	MAR	Tells MAR when to load memory from the bus
nCE	9	RAM	Enable signal for RAM to drive the bus

Control Signal	Array	Component	Function
nLr	8	RAM	Tells RAM when to load memory from the MAR
nLi	7	IR	Tells IR when to load instruction from the bus
nEi	6	IR	Enable signal for IR to drive the bus
nLa	5	A Reg	Tells A register to load data from the bus
Ea	4	A Reg	Enable signal for A register to drive the bus
Su	3	ALU	Activate subtractor instead of adder
Eu	2	ALU	Enable signal for Adder/Subtractor to drive the bus
nLb	1	B Reg	Tells B register to load data from the bus
nLo	0	Output Reg	Tells Output register to load data from the bus

Sequencing Details

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

Instruction Micro-Operations

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

Stage	**LDA**	**ADD**	**SUB**	**OUT**
T0	Ep, nLma	Ep, nLma	Ep, nLma	Ep, nLma
T1	Cp	Cp	Cp	Cp

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

Instruction Micro-Operations Notes

- First three micro-operations are common to all instructions.
- NOP operation executes only the first three micro-operations.
- Cp signal is not asserted during the HLT instruction in T2.
- ** Halt internal register is set to 1. More on this later

Programmer

Stage	**Control Signals**	**Programmer specific signals**
T0	Ep, nLMA	ready = 1
T1	Cp	ready = 0
T2	-	-
T3	nLmd	read_ui_in = 1
T4	nLr	read_ui_in = 0, done_load = 1
T5	-	done_load = 0

Detailed Overview T0: Control Signals the same as the typical default microinstruction – load the MAR with the address of the next instruction. Assert ready signal to alert MCU programmer (off chip) that CPU is ready to accept next line of RAM data.

T1: Increment the PC, the same as the typical default microinstruction. De-assert ready signal since the MCU programmer is polling for the rising edge.

T2: Do nothing to allow an entire clock cycle for programmer to prepare the data.

T3: Load the MAR with the data from the bus. Also, assert the read_ui_in signal which controls a series of tri-state buffers, attaches the ui_in pins straight to the bus.

T4: Load the RAM from the MAR. De-assert the read_ui_in signal (disconnect the ui_in pins from driving the bus since the ui_in pin data might be now inaccurate). Assert the done_load signal to indicate to the MCU that the chip is done with the ui_in data.

T5: De-assert done_load signal.

Programmer Notes The MCU must be able to provide the data to the ui_in pins (steady) between receiving the ready signal (assume worst case end of T0), and the bus needing the values (assume worst case beginning of T3).

Therefore, the MCU must be able to provide the data at a maximum of 2 clock periods.

IO Table: CB (Control Block)

Name	Verilog	Description	I/O	Width	Trigger
clk	clk	Clock signal	I	1	Edge Transition
resetn	rst_n	Set stage to 0	I	1	Active Low
opcode	opcode	Opcode from IR	I	4	NA
out	control_signals	Control Signal Array	O	15	NA
programming	programming	Programming mode	I	1	Active High
done_load	done_load	Executed Load during prog	O	1	Active High
read_ui_in	read_ui_in	Push ui_in onto bus	O	1	Active High
ready	ready_for_ui	Ready to prog next byte	O	1	Active High
HF	HF	Halting flag	O	1	Active High

IO Table: PC (Program Counter)

Name	Verilog	Description	I/O	Width	Trigger
bus	bus[3:0]	Connection to bus	IO	4	NA
clk	clk	Clock signal	I	1	Falling Edge
clr_n	rst_n	Clear to 0	I	1	Active Low
cp	Ep	Allow counter increment	I	1	Active High
ep	Cp	Output to bus	I	1	Active High
lp	Lp	Load from bus	I	1	Active High

PC (Program Counter) Notes

- Counter increments only when Cp is asserted, otherwise it will stay at the current value.
- Ep controls whether the counter is being output to the bus. If this signal is low, our output is high impedance (Tri-State Buffers).
- When CLR is low, the counter is cleared back to 0, the program will restart.
- The program counter updates its value on the falling edge of the clock.
- Lp indicates that we want to load the value on the bus into the counter (used for jump instructions). When this is asserted, we will read from the bus and instead of incrementing the counter, we will update each flip-flop with the appropriate bit and prepare to output.
- The least significant 4 bits from the 8-bit bus will be used to store the value on the program counter (0-15). Will be read from (JMP asserted) and written to (Ep asserted).
- clr_n has precedence over all.
- Lp takes precedence over Cp.

IO Table: Instruction Register (IR)

Name	Verilog	Description	I/O	Width	Trigger
bus	bus	Connection to bus	IO	8	NA
clk	clk	Clock signal	I	1	Rising Edge
clear	~rst_n	Clear to 0	I	1	Active High
opcode	opcode	Opcode from IR	O	4	NA
n_load	nLi	Load from Bus	I	1	Active Low
n_enable	nEi	Output to bus	O	1	Active Low

Instruction Register (IR) Notes

- The A Register updates its value on the rising edge of the clock.
- nEi controls whether the instruction is being output to the bus[3:0]. If this signal is high, our output is high impedance (Tri-State Buffers).
- nLi indicates that we want to load the value on the bus into the IR. When this is low, we will read from the bus and write to the register.
- When clear is high, the opcode is cleared back to NOP.
- IR always outputs the current value of the register to CB.

IO Table: RAM

Name	Verilog	Description	I/O	Width	Trigger
addr	mar_to_ram_addr	Address for read/write	I	4	NA
data_in	mar_to_ram_data	Data for write	I	8	NA
data_out	bus	Connection to bus	O	8	NA
lr_n	nLr	Load data from MAR	I	1	Active Low
ce_n	nCE	Output to bus	I	1	Active Low
clk	clk	Clock Signal	I	1	Rising edge
rst_n	‘1’	Clear RAM	I	1	Active Low

RAM Notes

- Addressing: The memory is 4-bit addressable, where the address specifies which register (out of 16) is being accessed for reading or writing.
- Write operation: A byte of data is written to specific register in RAM, where the location is determined by the address. Requires write enable lr_n signal as active (low) and the clock edge to occur.
- Read operation: Data can be read from a specific register in RAM determined by the input address. Requires chip enable ce_n signal as active (low). The data is output on the bus, and it is updated on the clock edge.
- Output: Data is presented on the bus line when the chip is enabled for reading, and high-impedance (Z) otherwise.
- RAM is never reset, rather, we always flash it.

IO Table: MAR

Name	Verilog	Description	I/O	Width	Trigger
bus	bus	Connection to bus	IO	8	NA
clk	clk	Clock signal	I	1	Rising Edge
addr	mar_to_ram_addr	Address for read/write	O	4	NA
data	mar_to_ram_data	Data for write	O	8	NA
n_load_data	nLmd	Load data from Bus	I	1	Active Low
n_load_addr	nLma	Load address from Bus	I	1	Active Low

MAR Notes

- The MAR updates its value on the rising edge of the clock.

- nLmd indicates that we want to load the value on the bus into the data register. When this is low, we will read from the bus and write to the register.
- nLma indicates that we want to load the value on the bus[3:0] into the address register. When this is low, we will read from the bus and write to the register.
- MAR always outputs the current value of the data and address registers to the RAM module.

IO Table: ALU (Adder/Subtractor)

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

ALU (Adder/Subtractor) Notes

- Eu controls whether the counter is being output to the bus. If this signal is low, our output is high impedance (Tri-State Buffers).
- A Register and B Register always provide the ALU with their current values.
- When sub is not asserted, the ALU will perform addition: Result = A + B
- When sub is asserted, the ALU will perform subtraction by taking 2s complement of operand B: Result = A - B = A + !B + 1
- Carry Out and Result Zero flags are updated on rising clock edge.

IO Table: Accumulator (A) Register

Name	Verilog	Description	I/O	Width	Trigger
clk	clk	Clock Signal	I	1	Rising edge

Name	Verilog	Description	I/O	Width	Trigger
bus	bus	Connection to bus	IO	8	NA
load	nLa	Load from bus	I	1	Active Low
enable_out	Ea	Output to bus	I	1	Active High
Register A	reg_a	Accumulator Register	O	8	NA
clear	rst_n	Clear Signal	I	1	Active Low

Accumulator (A) Register Notes

- The A Register updates its value on the rising edge of the clock.
- Ea controls whether the counter is being output to the bus. If this signal is low, our output is high impedance (Tri-State Buffers).
- nLa indicates that we want to load the value on the bus into the A Register. When this is low, we will read from the bus and write to the register.
- When CLR is low, the register is cleared back to 0.
- (Register A) always outputs the current value of the register to the ALU.

IO Table: B Register

Name	Verilog	Description	I/O	Width	Trigger
bus	bus	Connection to bus	IO	8	NA
clk	clk	Clock Signal	I	1	Rising edge
n_load	nLb	Load from bus	I	1	Active Low
value	reg_b	B Register value	O	8	NA

B Register Notes

- The B Register updates its value on the rising edge of the clock.
- nLb indicates that we want to load the value on the bus into the B Register. When this is low, we will read from the bus and write to the register.
- B Register always outputs the current value of the register to the ALU.

IO Table: Output Register

Name	Verilog	Description	I/O	Width	Trigger
bus	bus	Connection to bus	IO	8	NA
clk	clk	Clock Signal	I	1	Rising edge
n_load	nLo	Load from bus	I	1	Active Low
value	uo_out	B Register value	O	8	NA

Output Register Notes

- The Output Register updates its value on the rising edge of the clock.
- nLo indicates that we want to load the value on the bus into the B Register. When this is low, we will read from the bus and write to the register.

How to test

Provide input of op-code. Check that the correct output bits are being asserted/de-asserted properly.

Setup

1. **Power Supply**: Connect the chip to a stable power supply as per the voltage specifications.
2. **Clock Signal**: Provide a stable clock signal to the `clk` pin.
3. **Reset**: Ensure the `rst_n` pin is properly connected to allow resetting the chip.

Testing Steps

1. **Initial Reset**:

    - Perform a sync reset by pulling the `rst_n` pin low, waiting for 1 clock signal, and then pulling pulling the `rst_n` high to initialize the chip.

2. **Load Program into RAM**:

    - Use the `ui_in` pins to load a test program into the RAM. Ensure the `programming` pin is high during this process.

    - Perform a sync reset by pulling the `rst_n` pin low, waiting for 1 clock signal, and then pulling pulling the `rst_n` high to initialize the chip.

- Wait for the ready_for_ui signal to go high, indicating that the CPU is ready to accept data.
- Provide the first byte of data on the ui_in pins.
- Wait for the done_load signal to go high, indicating that the data has been successfully loaded into the RAM.
- Repeat the process for each byte of data:
  - Wait for ready_for_ui to go high.
  - Provide the next byte of data on the ui_in pins.
  - Wait for done_load to go high.
- Example program data:

```
0x10,  # NOP
0x73,  # JMP 0x3
0x00,  # HLT
0x4F,  # LDA 0xF
0x2E,  # ADD 0xE
0x6D,  # STA 0xD
0x50,  # OUT
0x3F,  # SUB 0xF
0x50,  # OUT
0x4D,  # LDA 0xD
0x50,  # OUT
0x72,  # JMP 0x2
0x10,  # NOP
0x00,  # Padding/empty instruction
0x02,  # Constant 2 (data)
0x01   # Constant 1 (data)
```

3. **Run Test Program**:
   - Set the `programming` pin low to exit programming mode.
   - Perform a sync reset by pulling the `rst_n` pin low, waiting for 1 clock signal, and then pulling pulling the `rst_n` high to initialize the chip.
   - Monitor the `uo_out` and `uio_out` pins for expected outputs.
   - Verify the control signals and data outputs at each clock cycle.

4. **Functional Tests**:
   - Perform specific functional tests for each instruction (e.g., ADD, SUB, LDA, STA, JMP, HLT).

- Verify the correct execution of each instruction by checking the output and control signals.

Example Test Cases

- **HLT Instruction**: Example program data:

```
0x4E,  # LDA 0xE
0x50,  # OUT
0x00,  # HLT
0x4F,  # LDA 0xF
0x50,  # OUT
0x00,  # HLT
0x00,  # Padding/empty instruction
0x00,  # Padding/empty instruction
0x00,  # Padding/empty instruction
0x00,  # Padding/empty instruction
0x00,  # Padding/empty instruction
0x00,  # Padding/empty instruction
0x00,  # Padding/empty instruction
0x00,  # Padding/empty instruction
0x09,  # Constant 9 (data)
0xFF   # Constant 255 (data)
```

  This program should first output 9 and then NOT change that to 255. HF should be set to 1

- **NOP Instruction**: Example program data:

```
0x42,  # LDA 0x2
0x50,  # OUT
0x10,  # NOP / Constant 16 (data)
0x1F,  # NOP
0x1F,  # NOP
0x1F,  # NOP
0x1F,  # NOP
0x1F,  # NOP
0x1F,  # NOP
0x1F,  # NOP
0x1F,  # NOP
0x1F,  # NOP
0x4E,  # LDA 0xF
0x50,  # OUT
```

```
0x1F,  # NOP
0x1F,  # NOP / Constant 31 (data)
```

This program should flash the lower 4 bits of the output register on and off with different on/off times

- **NOP Instruction**: Example program data:

```
0x42,  # LDA 0x2
0x50,  # OUT
0x10,  # NOP / Constant 16 (data)
0x1F,  # NOP
0x1F,  # NOP
0x1F,  # NOP
0x1F,  # NOP
0x1F,  # NOP
0x1F,  # NOP
0x1F,  # NOP
0x1F,  # NOP
0x1F,  # NOP
0x4E,  # LDA 0xF
0x50,  # OUT
0x1F,  # NOP
0x1F,  # NOP / Constant 31 (data)
```

This program should flash the lower 4 bits of the output register on and off with different on/off times

- **ADD Instruction** Example program data:

```
0x50,  # OUT
0x2E,  # ADD 0xE
0x70,  # JMP 0x0
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
```

```
0x01,  # Constant 1 (data)
0xFF,  # Padding/empty instruction
```

This program should add 1 to the A register, display it and loop back to the start. The output should be a counter from 0 to 255, then repeat.

CF should be set to 1 when the A register overflows, and 0 when it doesn't. CF=1 happens when the A register is 255 and 1 is added to it.

ZF should be set to 1 when the A register is 0, and 0 otherwise.

- **SUB Instruction** Example program data:

```
0x50,  # OUT
0x3E,  # SUB 0xE
0x70,  # JMP 0x0
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0x01,  # Constant 1 (data)
0xFF,  # Padding/empty instruction
```

This program should subtract 1 to the A register, display it and loop back to the start. The output should be a counter from 255 to 0, then repeat.

CF should be set to 1 when the A register overflows, and 0 when it doesn't. CF=0 happens when the A register is 0 and 1 is subtracted from it.

ZF should be set to 1 when the A register is 0, and 0 otherwise.

- **LDA Instruction**

  See above for example program data.

- **OUT Instruction**

  See above for example program data.

- **STA Instruction**

  Example program data:

```
0x4E,  # LDA 0xE
0x2F,  # ADD 0xF
0x5F,  # OUT
0x6E,  # STA 0xF
0x2F,  # ADD 0xE
0x5F,  # OUT
0x00,  # HLT
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0x09,  # Constant 9 (data)
0xFF   # Constant 255 (data) -> Constant 8 (data)
```

This program should load 9 to the A register, add 255 to it, resulting in 8 (CF should set to 1) display it, store it in 0xF, add 9 to it, resulting in 17 (CF should set to 0) and display it. Then, it should halt, and set HF to 1.

- **JMP Instruction**

  Example program data:

```
0x44,  # LDA 0x4
0x5F   # OUT
0x7D,  # JMP 0xD
0x0F,  # HLT
0x00,  # Constant 0 (data)
0xFF,  # Constant 5 (data)
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0xFF,  # Padding/empty instruction
0x45,  # LDA 0x5
0x5F   # OUT
0x0F,  # HLT
```

This program should load 0x4 (0) to the A register, display it, NOT HALT, jump to 0xD, then load 0x5 (255) to the A register, display it, and halt. HF should be set to 1.

Acknowledgements

- Darius Rudaitis, Eshann Mehta: RAM
- Evan Armoogan, Catherine Ye: PC
- Damir Gazizullin, Owen Golden: ALU, Accumulator
- Roni Kant, Jeremy Kam: MAR, B Register, Output Register, Instruction Register
- Gerry Chen, Siddharth Nema: Control Block and Programmer
- ECE 298A Course Staff: Prof. John Long, Prof. Vincent Gaudet, Refik Yalcin

Pinout

#	Input	Output	Bidirectional
0	prog_in_0	output_register_0	in_programming
1	prog_in_1	output_register_1	out_ready_for_ui
2	prog_in_2	output_register_2	out_done_load
3	prog_in_3	output_register_3	out_CF
4	prog_in_4	output_register_4	out_ZF
5	prog_in_5	output_register_5	out_HF
6	prog_in_6	output_register_6	
7	prog_in_7	output_register_7	

fulladder [291]

- Author: Keoni Gandall
- Description: A full adder made with wokwi
- GitHub repository
- Wokwi project
- Mux address: 291
- Extra docs
- Clock: 0 Hz

How it works

This implements a full adder on IN0, IN1, and IN2 switches (for A, B, and Carry). It uses 2 XOR gates, 2 AND gates, and an OR gate.

How to test

Output will be LEDs on OUT0 and OUT1. The output will be the binary sum of IN0+IN1+IN2.

External hardware

LEDs are used on OUT0 and OUT1.

Pinout

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

RLE Video Player [292]

- Author: Mike Bell
- Description: Reads run length encoded data from QSPI flash, displays on VGA
- GitHub repository
- HDL project
- Mux address: 292
- Extra docs
- Clock: 24000000 Hz

How it works

A 6bpp run length encoded image or video is read from a W25Q128JV or similar QSPI flash, and output to 640x480 VGA.

This is perfect for displaying the Bad Apple music video.

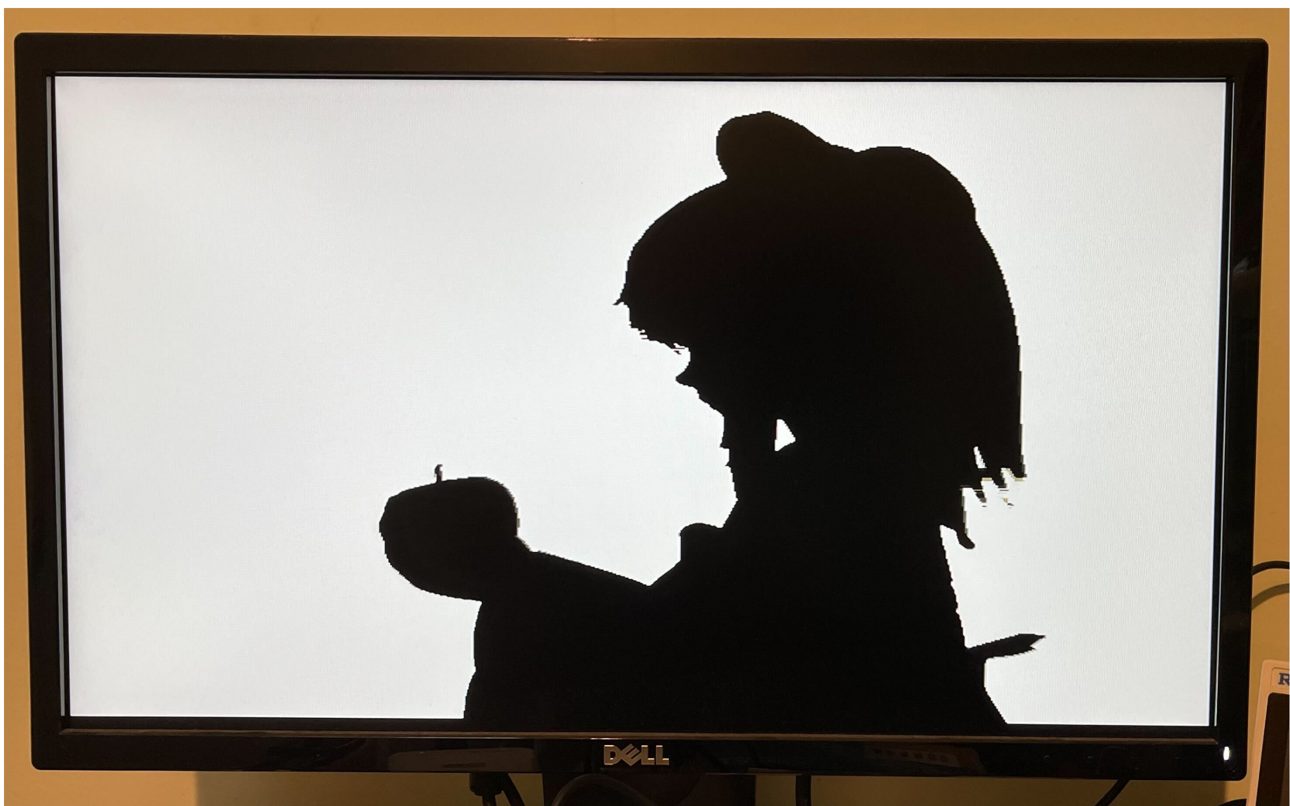


Figure 27: A frame from Bad Apple, rendered by the FPGA version of this design

Run Length Encoding The encoding uses 16-bit words. Most words are a run length in the top 10 bits, and a colour in the bottom 6 bits. A run must come to the end at the end of each row.

A run must be at least 2 pixels, and any group of 3 consecutive runs within a row must be at least 12 pixels, otherwise the data buffer will empty.

8-bit mono audio data can be interleaved into the video stream. The PWM output value is updated by the value `0xC000 + sample`, these must be at the end of a row, but do not have to be present on every row. With a 24MHz project clock the row clock is exactly 30kHz.

To compress the audio slightly, sample deltas can also be used, packing 2, 3 or 4 samples into one command. These add a signed offset to the current sample value at the end of the next 2, 3 or 4 rows:

- `0xD000 + (offset1 << 6) + offset2` with 2 6-bit signed offsets
- `0xE000 + (offset1 << 8) + (offset2 << 4) + offset3` with 3 4-bit signed offsets
- `0xF000 + (offset1 << 9) + (offset2 << 6) + (offset3 << 3) + offset4` with 4 3-bit signed offsets

This means that quieter audio takes less space!

Note that row and frame repeat, which were supported on the TT07 and TT IHP 0p2 versions are not supported here because audio data is interleaved into the video data.

The data is read starting at address 0. The special word `0xBFC0` causes the player to stop and restart from address 0 at the beginning of the next frame, restarting the video. This could also be used to display a still image.

How to test

Create a RLE binary file (docs/scripts to do this TBD) and load onto the flash. The pinout matches the QSPI Pmod. This should be plugged into the audio Pmod, and then the audio Pmod plugged into the bidir pins. Note the flash must support the h6B Fast Read Quad Output command, with 8 dummy cycles between address and data.

Connect the Tiny VGA PMOD to the output pins.

Inputs 2-0 set the read latency for the SPI in half clock cycles, it's likely that will need to be set to 2 (set input 1 high and inputs 0 and 2 low). This latency depends on the total round trip time through the mux and out to the flash and back. Valid values are 1 to 4.

Run with a 24MHz clock.

Maximum file size The 16MB flash is only enough for the first minute of Bad Apple. But because the flash read is just one very long read it would be straightforward to supply the data stream from the RP2040 or other external source. To make it easier to do this from the demo board RP2040, the QSPI pin configuration can be modified by setting `in3` high so that the 4 data pins are contiguous.

External hardware

- QSPI PMOD plugged into Audio PMOD
- Tiny VGA PMOD

Pinout

#	Input	Output	Bidirectional
0	SPI latency[0]	R1	CS
1	SPI latency1	G1	SD0 / SCK
2	SPI latency2	B1	SD1 / SD0
3	Select QSPI pinout	vsync	SCK / SD1
4		R[0]	SD2
5		G[0]	SD3
6		B[0]	Unused CS
7		hsync	PWM audio

Hopfield Network with Izhikevich-type RS and FS Neurons [293]

- Author: Daniel Solis
- Description: An on-chip implementation of a Hopfield neural network using Izhikevich-type regular spiking (RS) and fast spiking (FS) neurons with on-chip Hebbian learning for pattern storage and retrieval.
- GitHub repository
- HDL project
- Mux address: 293
- Extra docs
- Clock: 16000000 Hz

How it works

It is a leaky Integrated Neuron

How to test

Just Test

External hardware

No

Pinout

#	Input	Output	Bidirectional
0	learning_enable		spikes[0]
1	pattern_input[0]		spikes1
2	pattern_input1		spikes2
3	pattern_input2		spikes[3]
4	pattern_input[3]		spikes[4]
5			spikes[5]
6			spikes[6]
7			

4-bit Multiplier [294]

- Author: Sarp Sevil
- Description: A 4x4 array multiplier that multiplies two 4-bit numbers to produce an 8-bit product.
- GitHub repository
- HDL project
- Mux address: 294
- Extra docs
- Clock: 0 Hz

How it works

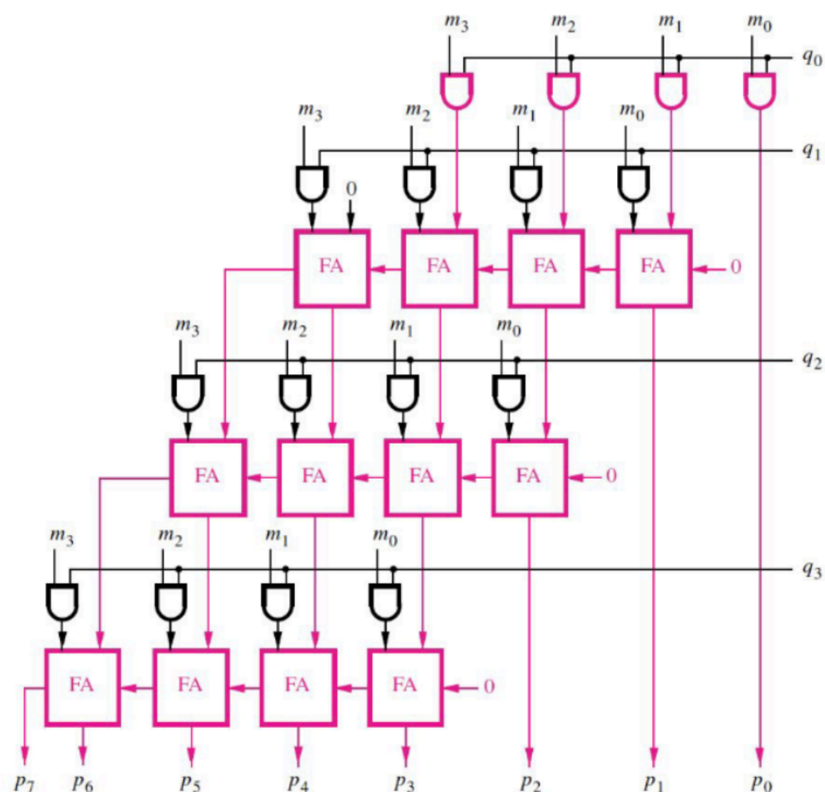


Figure 28: image

Above is a diagram that represents a 4-bit multiplier, which takes in two 4-bit integers and outputs a single 8-bit integer.

This was created using a manual structural design. We utilized a 1-bit full adder module in our implementation.

- AND-Gates are utilized to multiply each bit of input `m` with each bit of input `q`.

- We align partial products diagonally to mimic that of manual binary multiplication.
- We use 1-bit Full Adders to add products and handle carries.
- The outputs of the Full Adders eventually went to the bits of our output p which is an 8-bit integer.

How to test

Creating your own test cases:

- Go to the test folder and locate test.py.
- Edit test.py and add your own custom test cases.

```
# Example
# TEST CASE #0 -> 0 * 1
dut.ui_in.value = 0b00000001
await ClockCycles(dut.clk, 1)
assert dut.uo_out.value = 0b00000000
```

- Run the test with make and check the tests passed.

- If you've forked the repository
- Commit and push your changes to your forked repository
- Check Github Actions to check if your tests have passed

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	m[0]	p[0]	
1	m1	p1	
2	m2	p2	
3	m[3]	p[3]	
4	q[0]	p[4]	

#	Input	Output	Bidirectional
5	q1	p[5]	
6	q2	p[6]	
7	q[3]	p[7]	

Perceptron [295]

- Author: Mimi Rapoport
- Description: Simulates a perceptron
- GitHub repository
- HDL project
- Mux address: 295
- Extra docs
- Clock: 0 Hz

How it works

The perceptron takes in three inputs, multiplies them by weights and then sums the products. It then weighs the sum against a threshold to decide whether to outpur 1 or 0. The perceptron also takes in a desired output and performs a weight update when the desired output and actual output don't match. .

How to test

Make sure that the clock and reset are working.

External hardware

None

Pinout

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

Histogramming [296]

- Author: isil isiksalan
- Description: histogramming unit
- GitHub repository
- HDL project
- Mux address: 296
- Extra docs
- Clock: 0 Hz

Histogramming on Chip for Short Luminescence Signals

Isil Isiksalan

09 November 2024

Background To measure the lifetime of short luminescence signals effectively, Time-Correlated Single Photon Counting (TCSPC) is commonly used. TCSPC measures the time intervals between photon pulse detections and a synchronized reference signal, usually from a laser. This data is used to create a histogram of photon arrival times, which helps in calculating luminescence lifetimes.

```
This module is useful in TCSPC systems, particularly after a Time-to-Digi
or other time-tagging components. It processes 6-bit time-tagged data by
bins. Designed for systems capable of supporting up to 64 bins, our modul
numbered bins, mapping data into 32 bins to save space. This approach all
missing bins after the decay fitting process.

## Main Idea

The main idea is to integrate histogramming functionality within a digita
processing.


## Overview of thettumhistogrammingModule

Thettumhistogrammingmodule is designed for digital signal processing, par
that require data binning based on their values. Implemented in Verilog,
an 8-bit input stream, using the last 6 bits to classify values, and comm
```

results of its operations through a finite state machine with states IDLE
RESETBINS.

Description of the Module

Inputs and Outputs:

- Inputs:
    - uiin[7:0]: Main 8-bit input where binning is derived from the last
    - uioin[7:0]: Auxiliary input, not used in the current logic.
    - clk: Clock input for synchronization.
    - rstn: Active-low reset signal.
    - ena: Enable signal to activate histogramming.
- Outputs:
    - uoout[7:0]: Outputs the count of the current bin.
    - uioout[7:0]: Provides status flags including data validity, last bi
      ness for new data.
    - uiooe[7:0]: Output enable signal foruioout.

Working Principle:

1.Initialization and Resetting:Clears bins to zero and sets the module for new data intake.

2.Data Handling and Binning:Receives data, determines the bin index, and updates bin counts according to the input conditions.

3.State Management:Manages data output and resets based on binning outcomes.

Module Testing

The module underwent thorough testing using a testbench that simulated va
narios, including:

- Initial reset and setup.
- Ignoring even-numbered inputs.

- Filling multiple odd bins and managing overflow conditions.
- Checking reset functionality after data output.

- Testing operational robustness with manipulated enable signals.

All tests verified the module's functionality.

Pinout

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

test_friday2 [297]

- Author: Niles Peter
- Description: class
- GitHub repository
- HDL project
- Mux address: 297
- Extra docs
- Clock: 0 Hz

8-bit KoggeStone Adder

Author: Niles Villaverde, Joshua Cho Language: Verilog

How it works

The KoggeStone Adder computes in parallel, first the sum from the two different inputs and then computes the carry-out for each bit. Then uses the calculated carry-out and sum of each bit to compute the final result of the adder. *Note: No carry-out so values greater than 255 can not be outputted*

In the project.v file, there are 5 different modules: BigCircle, SmallCircle, Square, Triangle, and tt_um_koggestone_adder8.

Shown in figure 1 below is the block diagram for the flow for the KoggeStone Adder

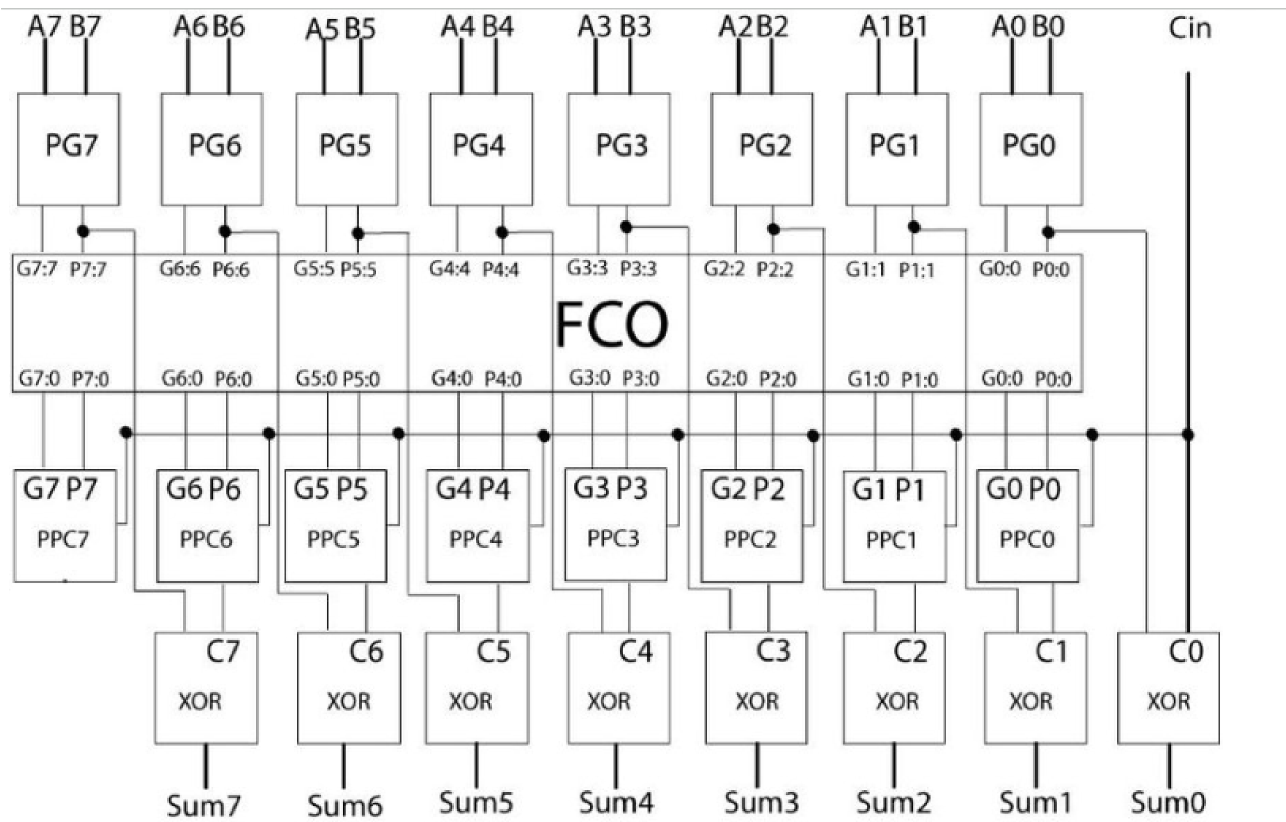


Figure 1: KoggeStone Adder Block Diagram

BigCirle Module The BigCircle module represents the carry generator for the KoggeStone Adder. It calculates the generated and propagated signal in each bit stage in the Adder. In comparison to carry-ripple adders, the KoggeStone adder allows for the carry information to propagate efficiently to multiple bit positions. This allows for the number of sequential steps in calculating the final carry-out to be reduced.

The BigCircle takes in the generate and propagate signals from the current position in the adder and the previous position in the adder. Using these signals, BigCircle updates the generate signal for the bit position to reflect if the carry is generated from this bit position or propagated from the previous. Then calculates the propagation signal to decide whether if a carry can be passed through this position.

SmallCircle Module The SmallCircle module passes the carry in signal and generated carry signal to the next position

Square Module The Square module calculates the current generate and propagate signal by ANDing the inputs A and B as well as XORing the inputs A and B respectively.

Triangle Module The Triangle module calcualtes the sum bit by XORing the propagate bit with the previous carry-in bit.

tt_um_koggestone_adder8 Module The tt_um_koggestone_adder8 module takes in two 8-bit inputs, ui_in and uio_in. The module also outputs an 8-bit output, uo_out. **Input Signals:** Two 8-bit, a and b which are mapped to ui_in and uio_in, respectively. Cin, carry-in for the addition which is set to zero. g and p, generate and propagation signal for each bit. c, carries for each bit position.

The first sequence is to use the Square Module to create the initial generate and progagate calculations. Then uses the BigCircle Module to calculate the intermediate generate and propagation signals of each bit. In the second stage of the BigCircle Module, by combining the signals over groups of 4 bits, it further propagates the carry. In the third stage of the BigCircle Module, it continues the carry propagation over an even wider spans of bits. Then using the SmallCircle Module, the final Carry-Out signals for each position are calculated. Then the final sum is calculated using the Triangle Modules.

How to test

The two different inputs, ui_in[7:0] and uio_in[7:0] are iterated through each possible combination of 8-bit numbers to test all corner cases. The outputs are set to the calculated values calculated by the KoggeStone Adder. If the sum between the two values are greater than 255, the test is skipped as limitations on the hardware prevent us from having a carry-out value.

External hardware

no external hardware

Pinout

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

#	Input	Output	Bidirectional
6	a[6]	sum[6]	b[6]
7	a[7]	sum[7]	b[7]

Perceptron Neuron [298]

- Author: Michael Chun
- Description: Makes a NAND gate with a perceptron neuron
- GitHub repository
- HDL project
- Mux address: 298
- Extra docs
- Clock: 0 Hz

How it works

Placeholder

How to test

Placeholder

External hardware

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

Pinout

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

#	Input	Output	Bidirectional
6	Input current bit [6]	State variable bit [6]	
7	Input current bit [7]	State variable bit [7]	Output bit

carry_select [299]

- Author: Juan, Leyang
- Description: This project designs a 8-bit carry select adder.
- GitHub repository
- HDL project
- Mux address: 299
- Extra docs
- Clock: 0 Hz

How it works

The 8-bit carry select adder works through the full adder and mux. The Carry Select Adder works by essentially using two ripple adders, with one having cin = 0 and the other cin = 1. Through this procedure, we are able to speed up the calculation of selecting which sum depending on our cin.

The ripple adder works by using a cascade of several full adders connected in series with each other. Each full adder is resposible for their adding their corresponding bits from both inputs and outputs their carryout to the carryin of the next full adder until both inputs have been fully added together. The ripple adder, and by extension the carry select adder is simple to implement and requires minimal logic gates to implement, making it inexpensive and space-efficient compared to other methods of addition. However, there is a delay due to the carry propagation which limits the ripple adder (and therefore the carry-select adder) in its effective speed with larger bitwidth inputs. However, for this application (8-bits), this adder is very efficient in both space and speed.

This project uses 'https://github.com/FCHXWH823/Verilog-Adders' as reference.

How to test

We tested all the combinations. This means two 8 bits input sum to a 8 bit output, and we ignore the carry out bit.

Therefore, we expect both the input and the output to be in the range of 0 to 255.

External hardware

We did not use any external hardware.

Pinout

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

I2C and SPI [300]

- Author: Vidyamol and Arun A V
- Description: Design of I2C and SPI communication protocols
- GitHub repository
- HDL project
- Mux address: 300
- Extra docs
- Clock: 400000 Hz

How it works

I2C and SPI protocol

How to test

send data enable clk

External hardware

No external hardware

Pinout

#	Input	Output	Bidirectional
0	i2c_data_in	sck_o	
1	i2c_clk_in	mosi_o	
2	miso_i	i2c_data_out	
3		i2c_clk_out	
4		i2c_data_oe	
5	i2c_wb_err_i	i2c_clk_oe	
6	i2c_wb_rty_i		
7			

Lab C 4x4 Mult-Array [301]

- Author: Justin Morris, Alexa
- Description: This focuses on designing a 4x4 multiplier array with simple parallel multiplier that calculates the product of two 4-bit binary numbers.The final design will be used as a Tiny Tapeout project for the purpose of a multiplier.
- GitHub repository
- HDL project
- Mux address: 301
- Extra docs
- Clock: 0 Hz

How it works

In this lab, the functionality of a 4x4 multiplier array utilizing full adders to perform binary multiplication. The process began with two 4-bit binary numbers, A and B, from which we generated four partial products by multiplying each bit of B with the entirety of A.These partial products binary are then aligned for addition.To sum the partial products, we use full adders, which combined the bits from each partial product while managing carries through each bit position. This systematic addition ultimately yielded an 8-bit result, representing the product of the two original 4-bit numbers. This experiment demonstrates the principles of binary multiplication and the role of full adders in digital circuit design.

How to test

To test a 4x4 multiplier, apply different combinations of 4 bit input signals while varying the select lines to ensure the correct input is routed to the output.The output for each combination should be recorded and compared against the expected output based on the select line values.Any discrepancies will indicate a fault in the multiplier design or implementation, allowing for troubleshooting.

External hardware

N/A

Pinout

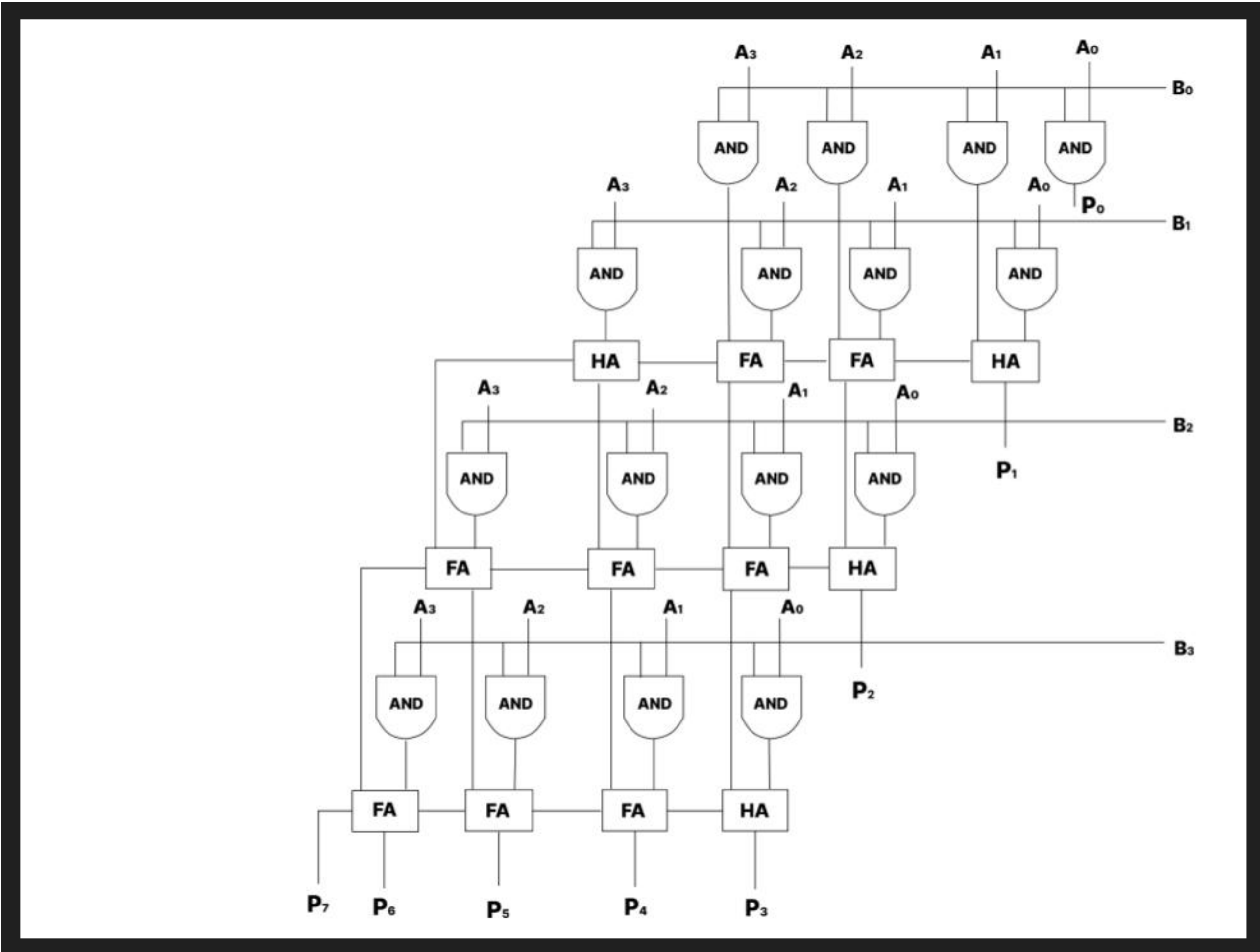


Figure 29: 4x4 Array Multiplier

#	Input	Output	Bidirectional
0	m[0]	p[0]	
1	m1	p1	
2	m2	p2	
3	m[3]	p[3]	
4	q[0]	p[4]	
5	q1	p[5]	
6	q2	p[6]	
7	q[3]	p[7]	

Configurable Logic Block [302]

- Author: Gary Mejia
- Description: A small CLB with a LUT3
- GitHub repository
- HDL project
- Mux address: 302
- Extra docs
- Clock: 0 Hz

How it works

The chip takes in two 8-bit inputs uin_in, this is the three arguments to the boolean function, write enable of the LUT, and clock enable of the CLB, and uio_in is the actual boolean function. The single output is the evaluation of the boolean function given the argument.

How to test

A simple hardware test would be to set the uio_in to 011111111 to get a NAND3. Use uin_in[3] to program the LUT with the seed and use uin_in[4] to make the output synchronous. Use uin_in[2:0] to input values into the NAND3.

External hardware

Switches on all inputs and leds on all outputs.

Common Boolean Functions and Seeds

Function	Seed
NAND3	01111111
NOR3	00000001
NOT	01010101
XOR2	01100110
Majority	11101000
Even Parity	01101001
One Hot	00010110

Pinout

#	Input	Output	Bidirectional
0	ui_in[0]	uo_out[0]	uio_in[0]
1	ui_in1	uo_out1	uio_in1
2	ui_in2	uo_out2	uio_in2
3	ui_in[3]	uo_out[3]	uio_in[3]
4	ui_in[4]	uo_out[4]	uio_in[4]
5	ui_in[5]	uo_out[5]	uio_in[5]
6	ui_in[6]	uo_out[6]	uio_in[6]
7	ui_in[7]	uo_out[7]	uio_in[7]

Tiny RAM DFF 2r1w [303]

- Author: Darryl Miles
- Description: RAM made from DFF 2r1w (32x8)
- GitHub repository
- HDL project
- Mux address: 303
- Extra docs
- Clock: 10000000 Hz

How it works

This is a really bad implementation of RAM that uses standard verilog to implement a dual-port-read single-port-write RAM using D-type flipflops.

- DO_A Data Out Port-A
- DI_A Data In Port-A
- DO_B Data Out Port-B
- AD_A Address Port-A
- AD_B Address Port-B
- LOHI_A Nibble (4bit) select Port-A
- LOHI_B Nibble (4bit) select Port-B
- W_EN Write Enable (Port-A implied)

2 pages of 16 bytes (8-bits) is the total storage. The high 1-bit of address are set via RST_N configuration, see below. the low 4-bits of address are supplied on the signal lines.

How to test

The external ports are as you would expect for a RAM module, similar to other ram modules based on the pin descriptions.

Memory reads occur all the time, there is no read-enable. Only port-A can be used to write into. The RST_N does not change the contents of the RAM storage area.

The RST_N release (posedge) is used to latch some additional configuration bits, so the following values are significant and can only be changed by clocking RST_N with a posedge which causes capture:

- uio_in[0] ADDRHI 1-bit to change the RAM page that can be accessed. This is a way to fill out the TT 1x1 tile space a little and allow the upper storage area to be accessible.
- uio_in[3:1] unused
- uio_in[4] READ_BUFFERED_A enable this will enable a synchronous output buffer register on the PORT-A to be enable, so the read value becomes available at the next cycle (pipelined) and held between cycles. If this works as expected this makes the port output asynchronous or synchronous.
- uio_in[5] READ_BUFFERED_B enable, same as above but for PORT-B.
- uio_in[6] WRITE_THROUGH this will activate a MUX bypass that has the effect of implementing a READ_AFTER_WRITE policy, so the currently written value is also the value found at the output port. When inactive (set logic 0) a READ_BEFORE_WRITE policy should be in effect. TODO check this works as expected when READ_BUFFERED_A is active.
- uio_in[7] unused, due to it also being the WRITE-ENABLE bit when in normal operations so allowing the CLK to run freely across reset and an unwanted write occurring.

External hardware

The standard PCB and RP2040 can be used to access. I expect a Micro Python interface to follow in an update.

Future areas to explore

Write a custom placement and wiring router to perform better placement and congestion architecture so that RAM size and WORD WIDTH. This would perform placement into the standard cell track layout so it can be run as a first pass to pack a solution into a design. Ideally leaving the external signals accessible at the edges of the area.

This might allow packing of any width, any depth, single/dual port (as options into the placement process) allowing for consistent size estimations to be made.

It seems when standard placement is left to solve this problem you don't get a result that scales with increased area usage. TODO some research into exactly what occurs in those scenarios, it is expected this maybe due to wiring congestion problem of cells just being in the wrong place / locality and requiring a lot of wiring to get a solution.

I pick dual-port-read support as that should provide a harder problem to solve as a single-port-read needs less wiring.

NOTES

PL_TARGET_DENSITY_PCT=95%
PL_RESIZER_HOLD_SLACK_MARGIN=0.08
GRT_RESIZER_HOLD_SLACK_MARGIN=0.03
CLOCK_PERIOD=100 (10MHz)

- 32x8 3 slew, 26 fanout vio, +106 buffers, resized 646, +16 tie, +238 hold buffers, No room for 156 instances.
- 28x8 1 slew, 36 fanout vio, +124 buffers, resized 763, +16 tie, +201 hold buffers, No room for 23 instances.
- 26x8 1 slew, 28 fanout vio, +105 buffers, resized 646, +16 tie, +191 hold buffers,
  10091 vio, 6289 vio after 6th, did not get much better, 6H to 4025 (incomplete pass)
- 24x8 0 slew, 26 fanout vio, +97 buffers, resized 632, +16 tie, +176 hold buffers,
  9084 vio, 5305 vio after 6th, best 2134 vio after 24th
- 22x8 0 slew, 20 fanout vio, +82 buffers, resized 555, +16 tie, +171 hold buffers,
  7079 vio, 3583 vio after 6th, best 428 vio after 29th, 6H to 427
- 20x8 4 slew, +101 buffers, +101 buffers, resized 649, +16 tie, +151 hold buffers,
  6622 vio, 3390 vio after 6th, best 991 vio after 24th
- 18x8 2 slew, 23 fanout vio, +72 buffers, resized 496, +16 tie, +138 hold buffers,
  4379 vio, 1299 vio after 6th, 0 vio after 43rd,
  SUCCESS
- 16x8 0 slew, 24 fanout vio, +76 buffers, resized 506, +16 tie, +120 hold buffers,
  4802 vio, 1631 vio after 6th, 1 vio after 56th, 6H to 64th

Pinout

#	Input	Output	Bidirectional
0	DI_A[0]	DO_A[0]	AD_B[0] (in)
1	DI_A1	DO_A1	AD_B1 (in)
2	DI_A2	DO_A2	AD_B2 (in)
3	DI_A[3]	DO_A[3]	AD_B[3] (in)
4	AD_A[0]	DO_B[0]	LOHI_A (in)
5	AD_A1	DO_B1	LOHI_B (in)

#	Input	Output	Bidirectional
6	AD_A2	DO_B2	
7	AD_A[3]	DO_B[3]	W_EN (in)

ECE-2204 4x4 Array Multiplier [320]

- Author: Evan Dworkin, Dante Minasyan
- Description: Multiplies two 4-bit numbers together
- GitHub repository
- HDL project
- Mux address: 320
- Extra docs
- Clock: 0 Hz

How it works

This project takes in two 4-bit inputs and multiplies them together into an 8-bit output. It uses an array of 12 full adders and 16 AND gates to do so (Figure 1). The operands are represented by the input pins with [0:3] representing operand 1 and [7:4] representing operand 2. The product is represented by the 8-bit output pin.

How to test

Input two 4-bit numbers via ui_in. The 4 most significant bits are taken to be the first term, the 4 least significant bits are taken to be the second term. The 8 uo_out bits are the output, in binary. For example, an input of 10011101 would be (1001) * (1101), or 9 * 13. The product would be (01110101), or 117.

Pinout

#	Input	Output	Bidirectional
0	m0	p0	
1	m1	p1	
2	m2	p2	
3	m3	p3	
4	q0	p4	
5	q1	p5	
6	q2	p6	
7	q3	p7	

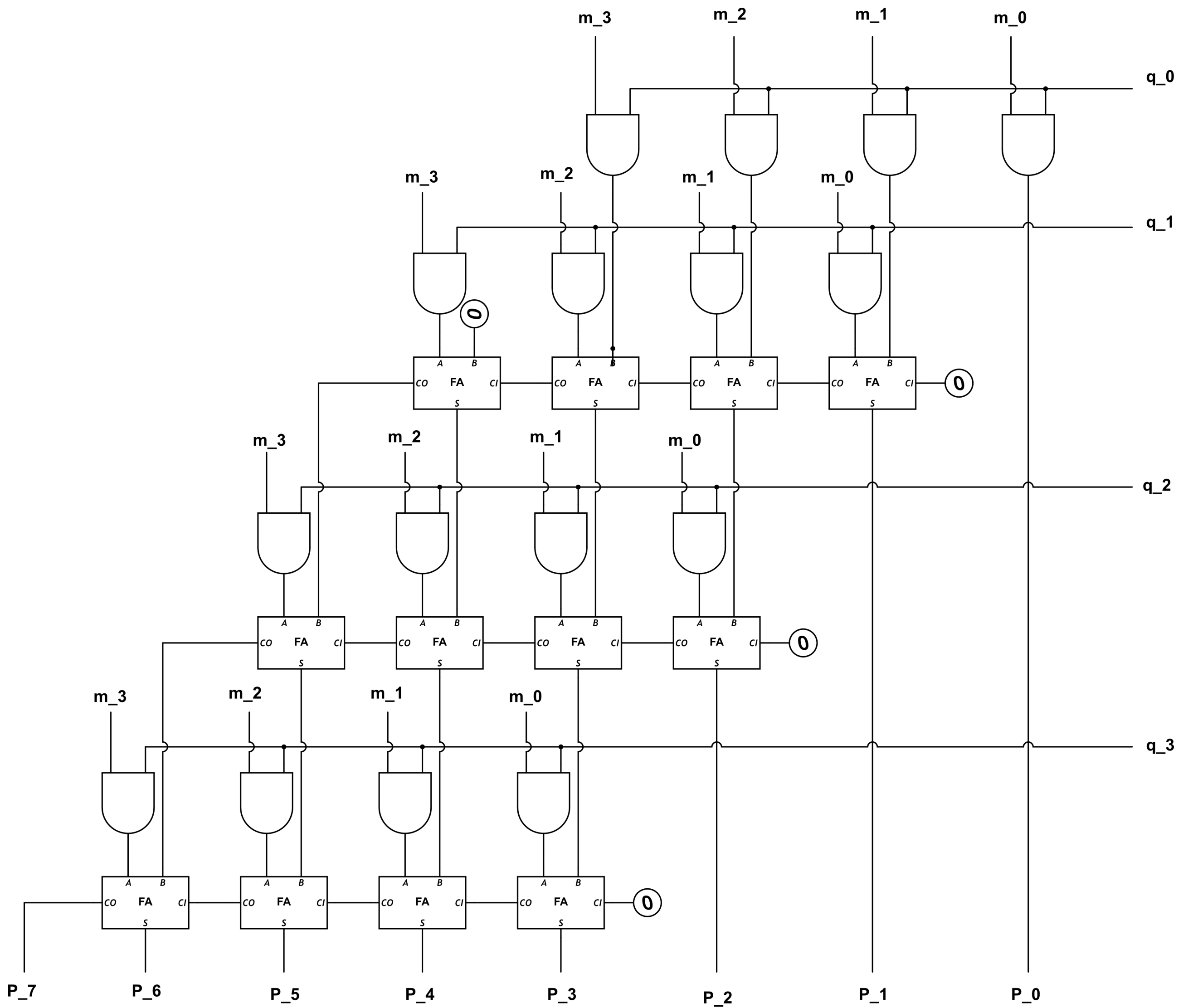


Figure 30: 4x4 Array Multiplier

Senol Gulgonul tt09 [321]

- Author: Senol Gulgonul
- Description: Display the letters of SEnOLGULGONUL on 7-Seg using internal oscillator
- GitHub repository
- HDL project
- Mux address: 321
- Extra docs
- Clock: 0 Hz

How it works

Displays letters of SEnOLGULGONUL on 7-Seg display by using internal three gate oscillator

How to test

Connect external R1, R2 and C for three gate oscillator and clk input and watch letters on 7-seg

External hardware

output pins a,b,c,d,e,f,g,dp are connected to a 7-Seg display, two inout and two bioutput for oscillator

Pinout

#	Input	Output	Bidirectional
0	inv3_in	a	inv3_out
1	inv1_in	b	inv2_out
2		c	
3		d	
4		e	
5		f	
6		g	
7		dp	

ECE2204 4x4 Array Multiplier [322]

- Author: Jason Brandon
- Description: 4x4 Structural Array Multiplier
- GitHub repository
- HDL project
- Mux address: 322
- Extra docs
- Clock: 0 Hz

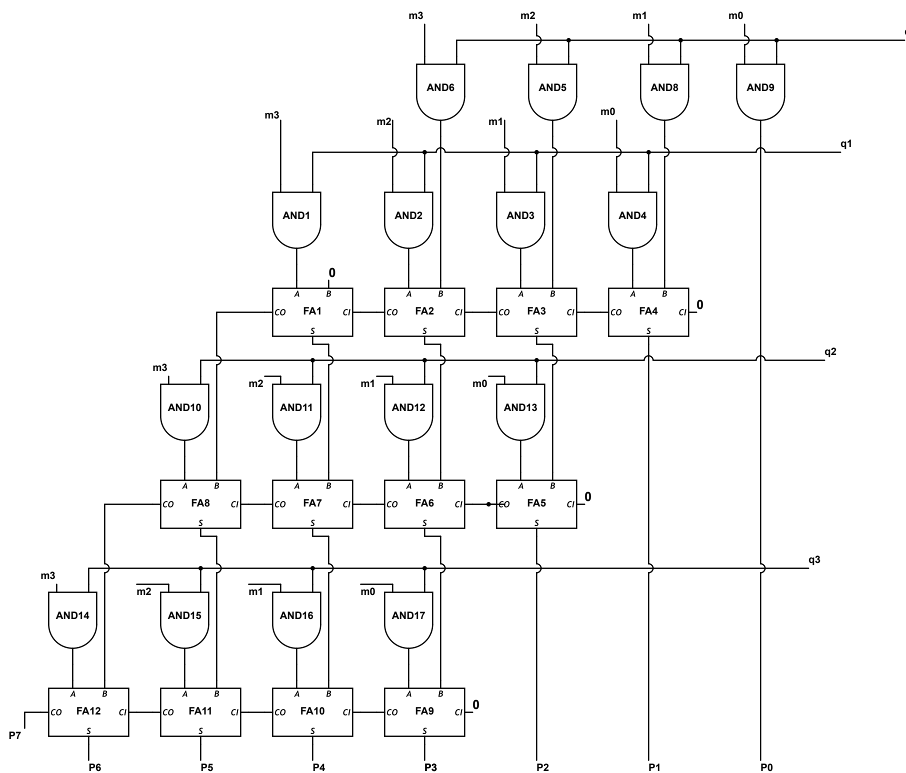


Figure 31: alt text

How it works

The Verilog code is for a 4x4 array multiplier that takes two 4-bit numbers, m and q, and produces an 8-bit result. It starts by splitting an 8-bit input into two parts: the upper 4 bits represent m and the lower 4 bits represent q. To calculate the product, the code computes partial products using bitwise AND operations. Each partial product corresponds to a bit of q, and there are four arrays created for these. After that, the code sums these partial products with a series of full adders. Each full adder takes

inputs from the previous stage and the next bits of the partial products, managing carries as needed. The final sum is stored in an 8-bit variable p, which is then sent to the output. The full adder used in this process helps calculate the sum and carry-out from two inputs and a carry-in. This method of breaking down the multiplication into partial products and adding them up demonstrates a clear and structured approach to implementing a basic multiplier in hardware.

How to test it

To test the Verilog program for the 4x4 array multiplier, the user would create a testbench, which is a separate module designed to evaluate how the multiplier functions. They would begin by setting up a variety of 4-bit input values, including simple cases such as all zeros and all ones, as well as some random combinations. This approach helps ensure that the multiplier can handle different scenarios correctly. Next, the user would run the testbench with these inputs and check the outputs against the expected results of the multiplication. Since the multiplication of two 4-bit numbers can yield an 8-bit result, they would calculate the expected outputs manually or use a calculator for verification. To facilitate this process, the user would add print statements to display the outputs on the console. If any outputs do not match the expected values, they would review the code to identify and correct any mistakes. By systematically testing various inputs and confirming the results, the user can ensure that the multiplier operates as intended.

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	q[0]	p[0]	
1	q1	p1	
2	q2	p2	
3	q[3]	p[3]	
4	m[0]	p[4]	
5	m1	p[5]	
6	m2	p[6]	
7	m[3]	p[7]	

Space Detective Maze Explorer [323]

- Author: Esteban Oman Mendoza
- Description: A maze explorer game, output uses qty5 7segment displays or LED equivalent
- GitHub repository
- HDL project
- Mux address: 323
- Extra docs
- Clock: 50000 Hz

How it works

This is a maze running on hardware. 3 user inputs are used. user_input[0] is used to walk forward on low, and user_input[2:1] is used as direction select where 2'b00 = N, 2'b01= East, 2'b10 = South, and 2'b11 = West" bit 2 is the most significant bit

How to test

You will need to wire up qty 5 7-segment displays or led equivalent. seg 0 is the right most or least significant segment, and seg4 being the left most of Most significant segment. Hook up all the common pins. for example pin 1 from seg0 connects to all other pin1 on the other 4 segments, they are then connected to the corresponding output pins uo[7:0]

Outputs

uo[0]: " a uo[0] = a " uo1: " — uo1 = b " uo2: " f | g | b uo2 = c " uo[3]: " | | uo[3] = d " uo[4]: " — uo[4] = e " uo[5]: " | | uo[5] = f " uo[6]: " e | d | c uo[6] = g " uo[7]: " | | uo[7] = dp" — dp uo([7:0] is the decoded segment signals to display the game output.

using 5 pnp transitors with Vcc (I used. 3.3V) at the emmiter, and the common anode (I used http://www.xlitx.com/datasheet/5161AS.pdf) connected to the collector, make a connection to uio[4:0] to represent seg4-seg0. example uio 5'b011111 would turn on seg 4 (low = on) each segment is mapped to uio[0]: "state LSB" uio1: "state MSB" uio2: "Direction LSB" uio[3]: "Direction MSB" uio[4]: "Top half of segment used for wall representation. 0-0, 1-1,...,5-5.

External hardware

qty 5 7-segment display or LED equivalent to visualize the game qty 5 current limitting resistors for the 7 segments qty 5 current limitting resistors to manage current through the output pins. these are connected to the base breadboard and enough wiring to make all the connections

Pinout

#	Input	Output	Bidirectional
0	user_input[0] Move forward (move one step in selected direction	a segments[0] = a	state LSB
1	user_input1 (considered least significant bit used in selection direction) Used to select facing direction where 2'b00 = N, 2'b01= East, 2'b10 = South, and 2'b11 = West	— segments1 = b	state MSB

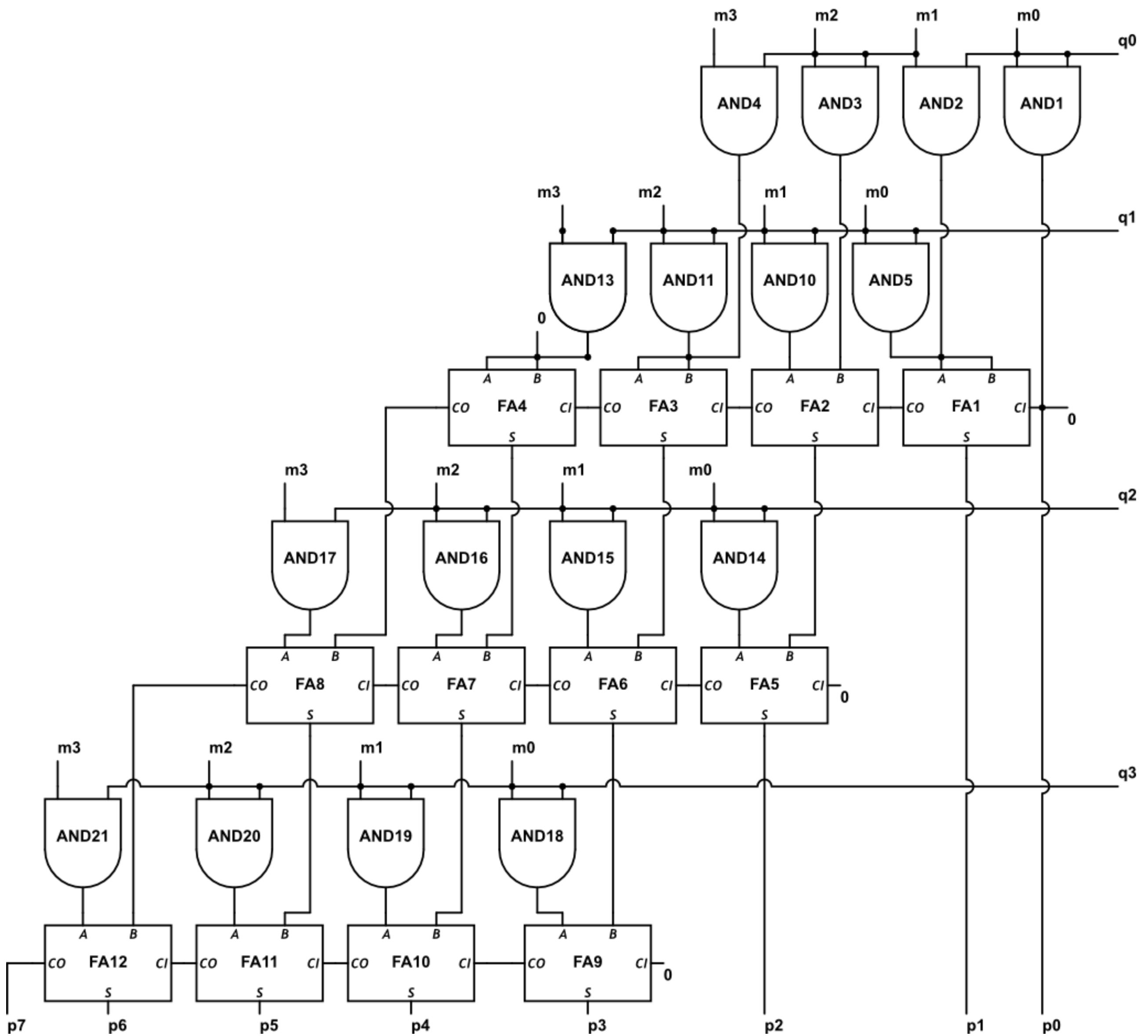
#	Input	Output	Bidirectional
2	user_input2 (considered least most significant bit used in selection direction) Used to select facing direction where 2'b00 = N, 2'b01= East, 2'b10 = South, and 2'b11 = West	f	g
3	not used		
4	not used	— segments[4] = e	Top half of segment used to display the walls of the room as seen from above (birds eye view). The top most segment(a) represents the wall directly in front of you in the chosen direction N,E,S, or West.
5	not used		
6	not used	e	d
7	not used		

Array Multiplier [324]

- Author: Leon Ha, Jegyeoung An
- Description: 4*4 Structural Array structural multiplier
- GitHub repository
- HDL project
- Mux address: 324
- Extra docs
- Clock: 0 Hz

How it works

The project is a 4x4 array multiplier. The inputs m and q are 4-bit factors that get multiplied to produce the output p which is a 8-bit product of m and q. This was created using manual structural design. The module array_mult_structural multiplies two 4 bit inputs (m and q) to produce an 8-bit product. Design components such as AND gates and full adders were applied to construct the design. The full adder module takes three 1-bit inputs, a, b, and cin, and produces a 1-bit sum and a 1-bit cout. The assign statement was used to assign the results. Four 4-bit partial products were generated. The LSB (Least Significant Bit) was assigned as the LSB of the first partial product. The sum of the 4-bit partial products are assigned to the output, and the carryout is stored.



How to test

In order to use this project, two 4-bit factors can be assigned to m and q. The output p should be the product of m and q.

0. 10 * 5 = 50
1. 4 * 2 = 8
2. 0 * 0 = 0
3. 2 * 1 = 2
4. 7 * 7 = 49

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	q[0]	p[0]	
1	q1	p1	
2	q2	p2	
3	q[3]	p[3]	
4	m[0]	p[4]	
5	m1	p[5]	
6	m2	p[6]	
7	m[3]	p[7]	

Hamming Code (7,4) [325]

- Author: Sebastien Paradis
- Description: (7,4) Hamming Encoder/Decoder
- GitHub repository
- HDL project
- Mux address: 325
- Extra docs
- Clock: 0 Hz

How it works

This implementation of the (7,4) Hamming Code allows for the same input to be used for encoding and decoding, with dynamic selection of the mode using the MSB of the input.

Hamming Encoder (7,4) Overview The Hamming (7,4) encoder is a linear error-correcting code that encodes 4 data bits into 7 bits by adding 3 parity bits, which can detect and correct a single-bit error.

Parity Format

{p1 p2 p3}

Data Format

{d1 d2 d3 d4}

Input An 8-bit input "ui" with the following format (note the form is {7 6 5 4 3 2 1 0})

Input Pins

- ui[0] - Bit 0 for 4-bit data input, d4
- ui1 - Bit 1 for 4-bit data input, d3
- ui2 - Bit 2 for 4-bit data input, d2
- ui[3] - Bit 3 for 4-bit data input, d1
- ui[4] - X
- ui[5] - X
- ui[6] - X
- ui[7] - Mode Selector (0 => Encode, uses ui[3:0]; 1 => Decode, uses ui[6:0])

Output An 8-bit output "uo" with the following format (note the form is {7 6 5 4 3 2 1 0})

Output Pins

- uo[0] - Bit 0 for 7-bit encoded output, d4
- uo1 - Bit 1 for 7-bit encoded output, d3
- uo2 - Bit 2 for 7-bit encoded output, d2
- uo[3] - Bit 3 for 7-bit encoded output, p3
- uo[4] - Bit 4 for 7-bit encoded output, d1
- uo[5] - Bit 5 for 7-bit encoded output, p2
- uo[6] - Bit 6 for 7-bit encoded output, p1
- uo[7] - X

Encode Mode

- Encode Mode is selected by setting the MSB of the input (bit 7) LOW (0).
- If encode mode is chosen, the encoder will use bits 3:0 as the four data bits to be encoded, and produce a 7-bit encoded output.
- Bit 6:4 are not involved in any encoding.

Encode Mode Input Format

{selector, X, X, X, d1, d2, d3, d4}

Encode Mode Output Format

{p1, p2, d1, p3, d2, d3, d4}

Parity Bit Calculations

1. p1 covers bits d1, d2, and d4.
   - p1 = d1 XOR d2 XOR d4
2. p2 covers bits d1, d3, and d4.
   - p2 = d1 XOR d3 XOR d4
3. p3 covers bits d2, d3, and d4.
   - p3 = d2 XOR d3 XOR d4

Expected Outputs of Encode Mode

- 0XXXd1d2d3d4 -> 0p1p2d1p3d2d3d4
- 00000000 -> 00000000
- 00000010 -> 00101010
- 00000001 -> 01101001
- 00000011 -> 01000011
- 00000100 -> 01001100
- 00000101 -> 00100101
- 00000110 -> 01100110
- 00000111 -> 00001111
- 00001000 -> 01110000
- 00001001 -> 00011001
- 00001010 -> 01011010
- 00001011 -> 00110011
- 00001100 -> 00111100
- 00001101 -> 01010101
- 00001110 -> 00010110
- 00001111 -> 01111111

Hamming Decoder (7,4) Overview The decoder checks the received 7-bit word for errors and corrects a single-bit error if detected. The process involves recalculating the parity bits and comparing them with the received parity.

Decode Mode

- Decode Mode is selected by setting the MSB of the input (bit 7) HIGH (1).
- If decode mode is chosen, the decoder will use bits 7:0, both the data and parity bits, and produce a 7-bit decoded output. The decoded output will be the originally encoded input as long as there were less than 2 flipped bits between encoder output and decoder input.

Decode Mode Input Format

{p1, p2, d1, p3, d2, d3, d4}

Decode Mode Output Format

{p1, p2, d1, p3, d2, d3, d4}

- a maximum of 1 bit could be flipped at position {S2, S1, S0}.

Syndrome Calculation The syndrome indicates the position of an error (if any):

1. S0 is recalculated using the same bits used to calculate p1 during encoding:
   - S0 = p1' XOR d1 XOR d2 XOR d4
2. S1 recalculates p2:
   - S1 = p2' XOR d1 XOR d3 XOR d4
3. S2 recalculates p3:
   - S2 = p3' XOR d2 XOR d3 XOR d4

Error Correction The syndrome {S2, S1, S0} gives the error location:

- If the syndrome is 000, no error is detected.
- If the syndrome is non-zero, the position of the error corresponds to the syndrome value (1 for the least significant bit, 7 for the most significant bit).
- E.g. if syndrome is 010, then. Our error bit is at bit 4
- If an error is detected, flip the bit at the position indicated by the syndrome.

How to test

Testing can be done by applying known data inputs with LOW as the value of the 7th bit (encode mode), and ensuring that the output is the expected encoding value (see table of expected outputs in encode mode).

Similarly, known encoded values can by used as input, with the 7th bit as HIGH (decode mode), and we can ensure that the output is the exact same as the original encoded value, even if we flip 1 bit. This should be done for each of the 7 bits for all encoded values

External hardware

TBD based on implementation.

Pinout

#	Input	Output	Bidirectional
0	LSB/Bit 0 for 4-bit Encoder Input OR LSB/Bit 0 for 7-bit Decoder Input	LSB/Bit 0 for 7-bit Encoder OR Decoder Output	
1	Bit 1 for 4-bit Encoder Input OR Bit 1 for 7-bit Decoder Input	Bit 1 for 7-bit Encoder OR Decoder Output	
2	Bit 2 for 4-bit Encoder Input OR Bit 2 for 7-bit Decoder Input	Bit 2 for 7-bit Encoder OR Decoder Output	
3	MSB/Bit 3 for 4-bit Encoder Input OR Bit 3 for 7-bit Decoder Input	Bit 3 for 7-bit Encoder OR Decoder Output	
4	Bit 4 for 7-bit Decoder Input	Bit 4 for 7-bit Encoder OR Decoder Output	
5	Bit 5 for 7-bit Decoder Input	Bit 5 for 7-bit Encoder OR Decoder Output	
6	MSB/Bit 6 for 7-bit Decoder Input	MSB/Bit 6 for 7-bit Encoder OR Decoder Output	

#	Input	Output	Bidirectional
7	Mode Selector (0 => Encode, uses ui[3:0]; 1 => Decode, uses ui[6:0])	Mode Selector (0 => Encode; 1 => Decode)	

ece2204 project for tapeout [326]

- Author: Yiqiao, Geno
- Description: class project 4x4 structal arary multiplier
- GitHub repository
- HDL project
- Mux address: 326
- Extra docs
- Clock: 0 Hz

How it works

The `tt_um_array_mult_structural` module performs 4x4 unsigned multiplication by using bitwise multiplication and a series of Carry-Save Adders (CSAs) followed by a final Carry-Propagate Adder (CPA). It takes an 8-bit input (`ui_in`) divided into two 4-bit operands: `m = ui_in[7:4]` and `q = ui_in[3:0]`. Each bit of `m` is multiplied with each bit of `q`, generating partial products. These partial products are accumulated row by row using full adders, with the carry propagated through successive rows. The final product, `p`, is composed from the sum and carry values of the last rows and is assigned to the 8-bit output (`uo_out`).

How to test

To test the module, you can create a Verilog testbench that applies different 8-bit input values for `ui_in` representing two 4-bit numbers and observes the resulting 8-bit output `uo_out`. For example, inputting `ui_in = 8'b0011_0011` (3 * 3) should yield `uo_out = 8'b0000_1001` (9). By applying various combinations of operands, such as `ui_in = 8'b1111_1111` (15 * 15), and using a simulation tool to verify the waveforms, you can ensure that the design correctly computes the products and functions as expected.

External hardware

N/A

Pinout

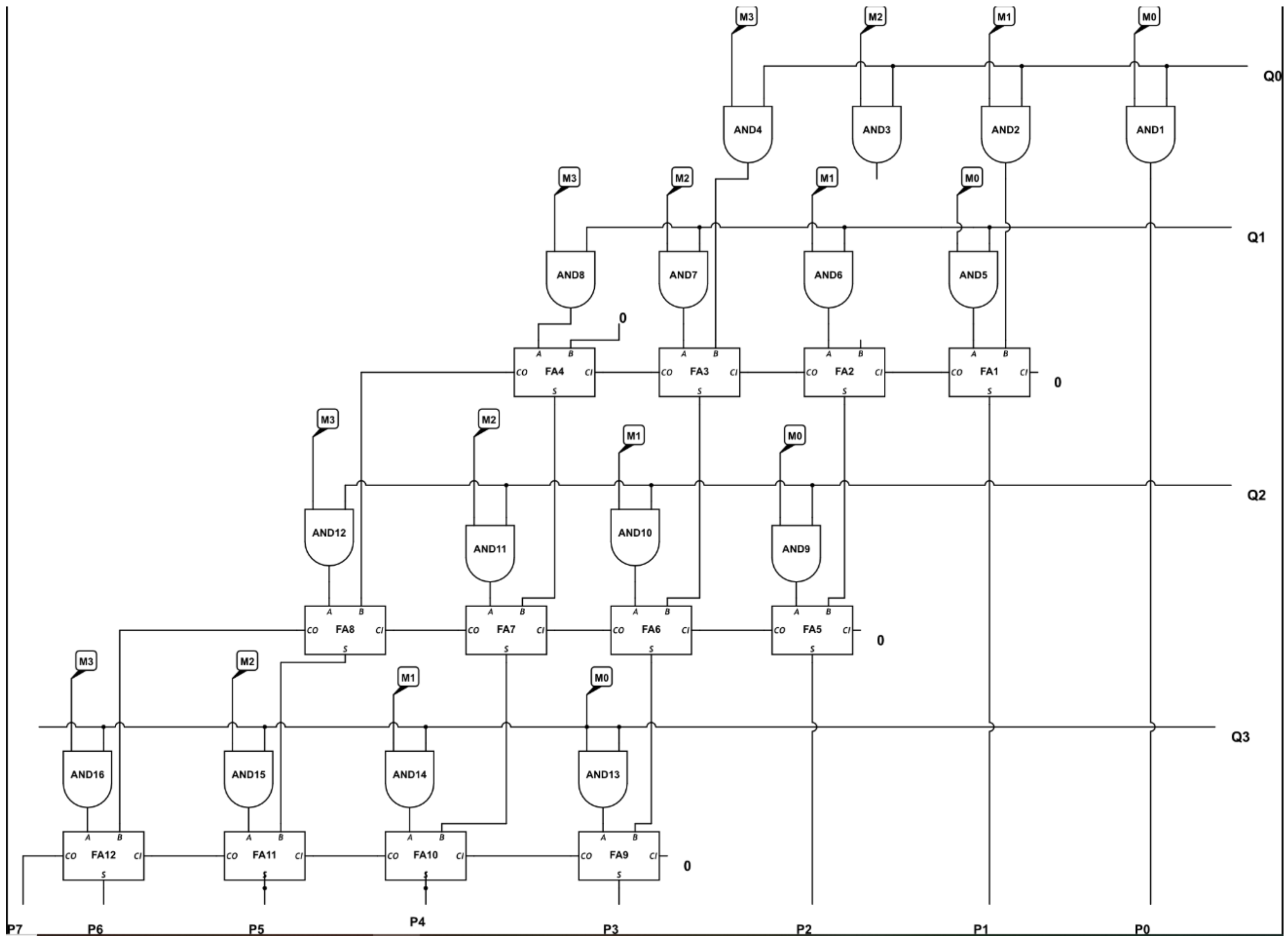


Figure 32: Alt text

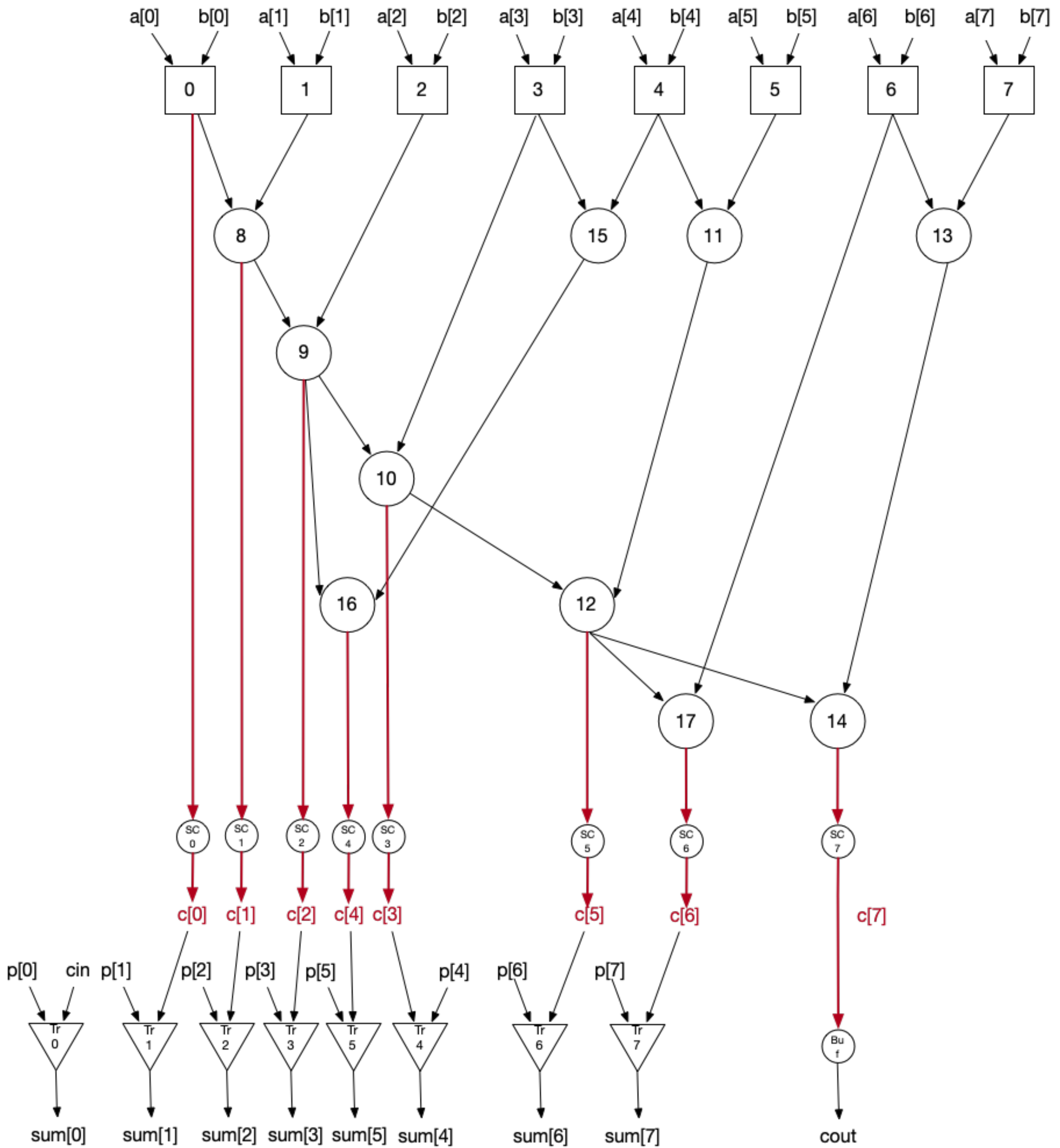
#	Input	Output	Bidirectional
0	q[0]	p[0]	
1	q1	p1	
2	q2	p2	
3	q[3]	p[3]	
4	m[0]	p[4]	
5	m1	p[5]	
6	m2	p[6]	
7	m[3]	p[7]	

tiny-tapeout-8bit-GPTPrefixCircuit [327]

- Author: Weihua Xiao
- Description: In this project, we use large language model to automatically create a totally-new prefix network-based high speed adder, for getting a good trade-off between PPA (power performance and area).
- GitHub repository
- HDL project
- Mux address: 327
- Extra docs
- Clock: 0 Hz

How it works

LLM-aided design of a totally-new 8-bit prefix network-based high speed adder:



In this figure, the squares represent the Square module in project.v, the circles represent the BigCircle module in project.v, the small circles represent the SmallCircle in project.v, and the triangles represent the Triangle module in project.v. Each carry signal (c[i]) is generated by circles and each sum signal (sum[i]) is generated by triangles.

How to test

This test systematically applies all combinations of 8-bit values to dut.a and dut.b, verifies the resulting sum dut.sum against the expected 8-bit result ((dut.a + dut.b) & 0xFF), and asserts that the dut behaves correctly.

External hardware

No external hardware

Pinout

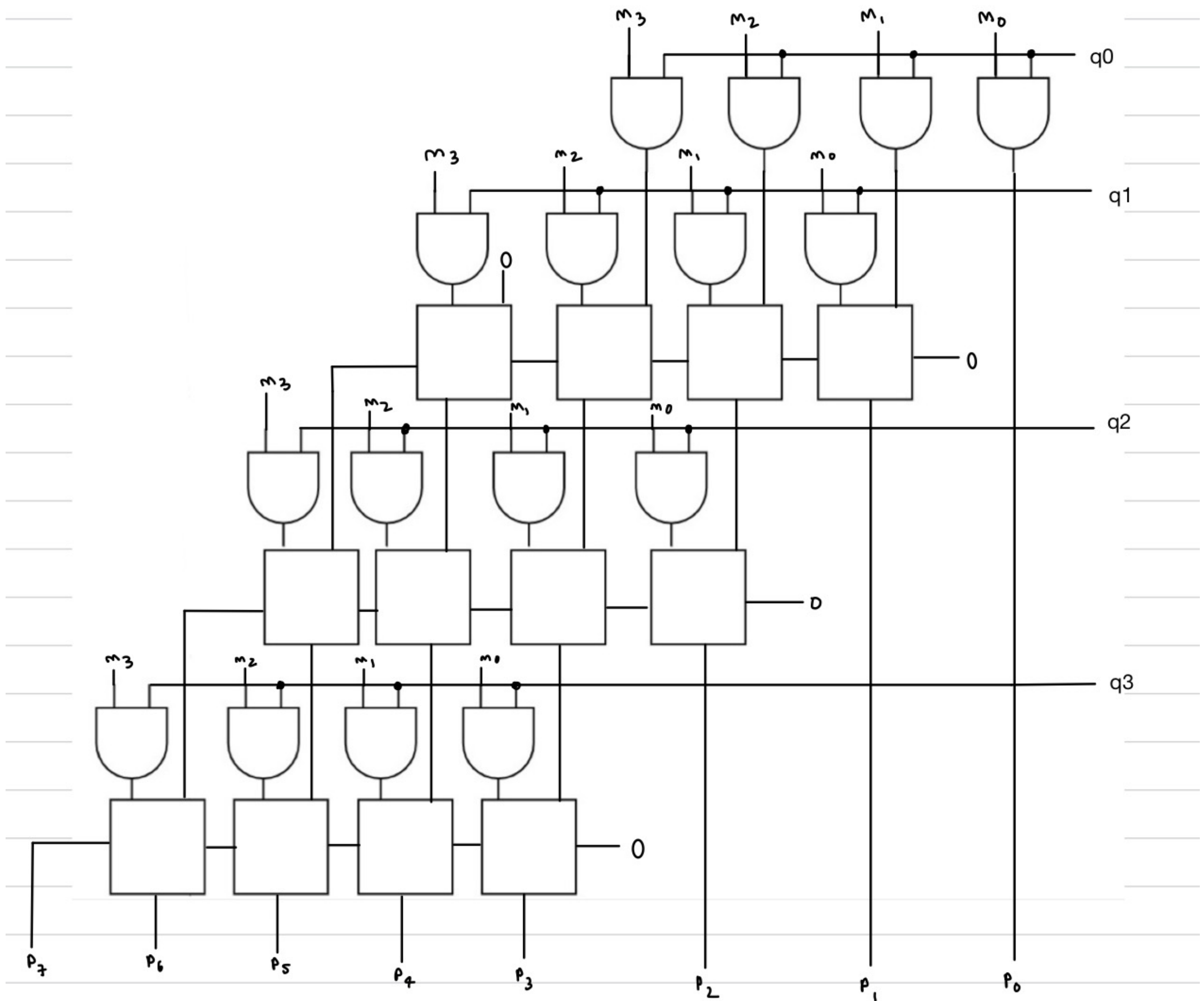
#	Input	Output	Bidirectional
0	a[0]	sum[0]	b[0]
1	a1	sum1	b1
2	a2	sum2	b2
3	a[3]	sum[3]	b[3]
4	a[4]	sum[4]	b[4]
5	a[5]	sum[5]	b[5]
6	a[6]	sum[6]	b[6]
7	a[7]	sum[7]	b[7]

4x4 array multiplier [328]

- Author: Gabriela Perez, Martha McQuillan
- Description: 4x4 structural array multiplier
- GitHub repository
- HDL project
- Mux address: 328
- Extra docs
- Clock: 0 Hz

How it works

A 4x4 multiplier is a digital circuit that multiplies two 4-bit binary numbers to produce an 8-bit result. It works by generating partial products: each bit of one 4-bit number is multiplied by each bit of the other, producing 16 partial results. These are then organized in rows, with each row shifted left according to the position of the bit being multiplied. Finally, the rows are summed using binary adders, yielding the final 8-bit product representing the multiplication result



This is a 4x4 array muliplier that takes in two 4-bit factors, m and q, and uses a full adder to output an 8-bit product of m and q, p.

How to test

Test the multiplier with a test bench of 10 varrying values of 4-bit factors with their multiplcation value. This is an unsigned decimal multiplier.

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	q[0]	p[0]	
1	q1	p1	
2	q2	p2	
3	q[3]	p[3]	
4	m[0]	p[4]	
5	m1	p[5]	
6	m2	p[6]	
7	m[3]	p[7]	

LIF on a Ring Topology [329]

- Author: Taylor Kergan
- Description: LIF neurons connected in a ring that displays different firing patterns.
- GitHub repository
- HDL project
- Mux address: 329
- Extra docs
- Clock: 0 Hz

How it works

This project implements eight leaky integrate-and-fire (LIF) neurons that are connected in a ring topology. Each neuron:

1. Integrates input current over time
2. Leaks voltage according to a decay constant
3. Fires when voltage reaches threshold
4. Influences its neighbors through coupling currents

The system supports multiple firing patterns:

- Independent: Neurons fire based only on their input current
- Wave: Activity propagates around the ring
- Synchronous: All neurons tend to fire together
- Clustered: Neurons form synchronized pairs
- Burst: Strong neighbor coupling creates burst patterns

How to test

The system can be tested through several inputs:

1. Base current (ui_in[7:3]): Controls the fundamental firing rate
2. Pattern select (ui_in[2:0]): Chooses the firing pattern
3. Coupling strength (uio_in[7:0]): Sets the strength of inter-neuron connections

To observe behavior:

1. Monitor spike outputs (uo_out[7:0]): Each bit represents one neuron's spikes
2. Watch voltage state (uio_out[7:0]): Shows membrane potential of first neuron
3. Run different patterns to see:

- Wave propagation
- Synchronization
- Burst patterns
- Clustering effects

Test sequence:

1. Apply reset (rst_n)
2. Enable system (ena)
3. Set desired pattern and current
4. Monitor outputs for expected behavior

External hardware

N/A.

Pinout

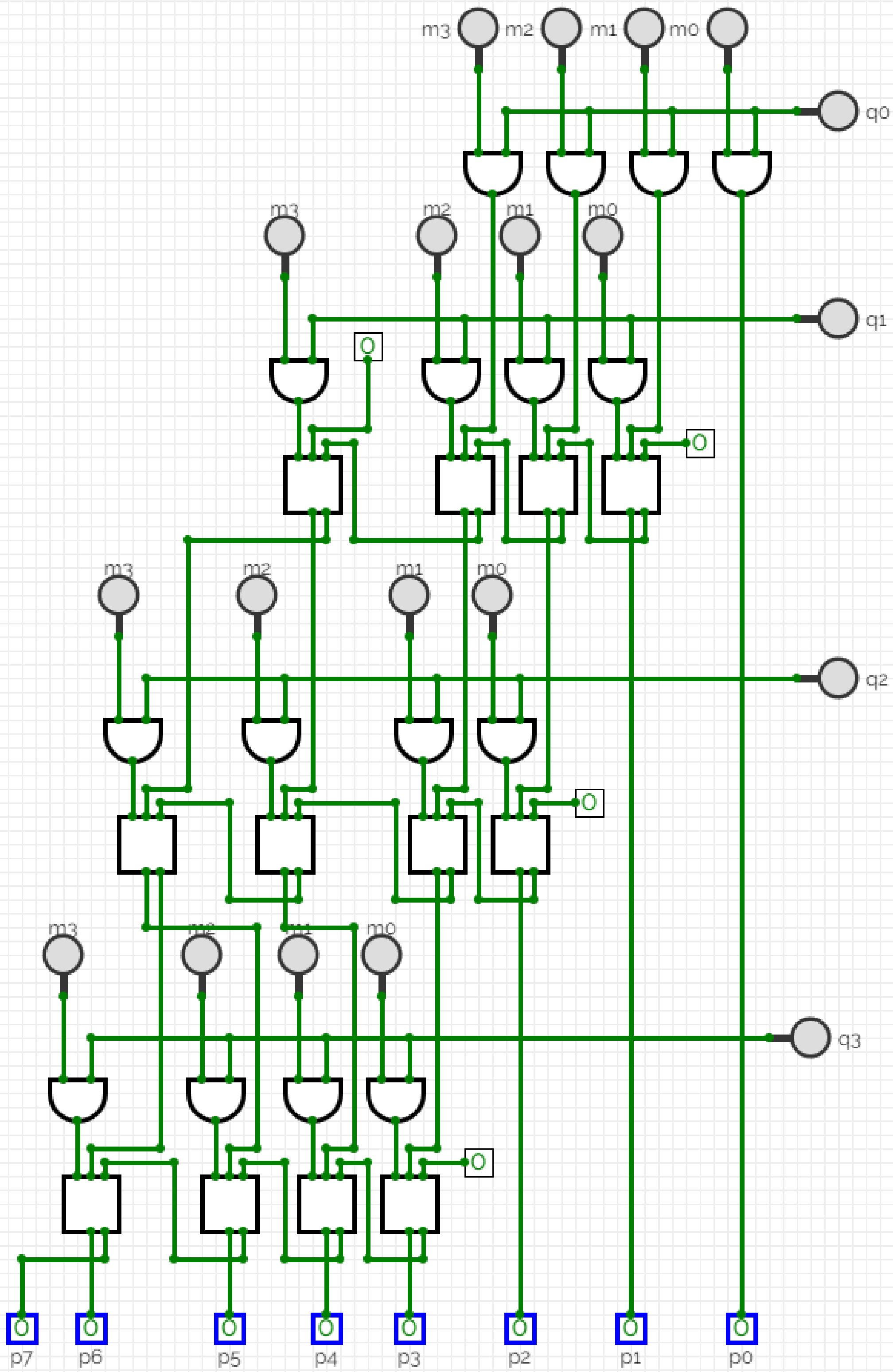
#	Input	Output	Bidirectional
0	Pattern select bit 0 (LSB)	Spike output from neuron 0	Coupling strength bit 0 (LSB)
1	Pattern select bit 1	Spike output from neuron 1	Coupling strength bit 1
2	Pattern select bit 2 (MSB)	Spike output from neuron 2	Coupling strength bit 2
3	Base current scaling bit 0 (LSB)	Spike output from neuron 3	Coupling strength bit 3
4	Base current scaling bit 1	Spike output from neuron 4	Coupling strength bit 4
5	Base current scaling bit 2	Spike output from neuron 5	Coupling strength bit 5
6	Base current scaling bit 3	Spike output from neuron 6	Coupling strength bit 6
7	Base current scaling bit 4 (MSB)	Spike output from neuron 7	Coupling strength bit 7 (MSB)

4-bit-array-multiplier [330]

- Author: HenryZ-ErickR
- Description: 4-bit array mutiplication between two arrays
- GitHub repository
- HDL project
- Mux address: 330
- Extra docs
- Clock: 0 Hz

How it works

This project is a 4x4 array multiplier which multiplies two 4-bit numbers to produce an 8-bit result. The miltiplier work by generating partial products through bit-wise AND operations between the individual bits of the two input numbers. These partial products are then summed using a series of full adders. Which handle both the sum and the carry bits. The structure of the code starts from the least significant bits (LSB) and progresses to the most significant bits (MSB), adding the partial products in stages. Each stage involves full adders that sum three inputs: two partial products and a carry from the previous stage. The final product is generated by combining the sums and carries, with the last carry assigned to the most significant bit of the result. This approach efficiently organizes binary multiplication using logical AND gates and full adders. An illustration of the structure of this multiplier can be seen in the figure below.



How to test

To test the functionality of this multiplier, a test bench file would be used to instantiate the multiplier module, provide different 4-bit values for the inputs m and q, and

observe the 8-bit output p. For each test case, the testbench compares the result of the multiplier's output with the expected result of multiplying m and q using simple binary arithmetic. By applying a variety of test inputs, including edge cases such as all zeros, all ones, and alternating bit patterns, we can verify that the multiplier handles all cases correctly. The testbench would also use initial and always blocks to display the results of each multiplication using $display statements, allowing us to validate the behavior in a simulation environment like Vivado.

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	q[0]	p[0]	
1	q1	p1	
2	q2	p2	
3	q[3]	p[3]	
4	m[0]	p[4]	
5	m1	p[5]	
6	m2	p[6]	
7	m[3]	p[7]	

Delta-Sigma ADC Decimation Filter [331]

- Author: Alexander Sheldon
- Description: Decimation filter for output of a delta-sigma ADC.
- GitHub repository
- HDL project
- Mux address: 331
- Extra docs
- Clock: 50000000 Hz

How it works

Digital low pass and decimation filter for use at the output of a delta-sigma ADC. Analog will hopefully be included on the next shuttle.

How to test

Input 1 bit data on ui_in[0] at 50MHz representing the output of a delta-sigma modulator Will generate 16 bit data on the GPIOs at 50MHz/64=781.25kHz

External hardware

TBD

Pinout

#	Input	Output	Bidirectional
0	dec_in	mux_out[0]	div_clk8x
1		mux_out1	
2		mux_out2	div_clk
3		mux_out[3]	
4		mux_out[4]	
5		mux_out[5]	
6		mux_out[6]	
7		mux_out[7]	

Array_Multiplier [332]

- Author: Taegahm Kang
- Description: Multiplies two 4-bit numbers
- GitHub repository
- HDL project
- Mux address: 332
- Extra docs
- Clock: 0 Hz

How it works

The project takes one 8-bit input. The 8-bit input is split to form two 4-bit inputs. The inputs are put into an array that uses a combination of AND gates and Full Adders to get the individual values of the product. The Full Adders are coded using a combination of AND, XOR, and OR gates. The 4-bit array multiplier used for this project is shown in Figure 1.

Figure 1: 4-bit array multiplier

How to test

Give an 8-bit input and check if the output is the correct product of the first and last 4-bits of the input.

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	q[0]	p[0]	
1	q1	p1	
2	q2	p2	
3	q[3]	p[3]	
4	m[0]	p[4]	
5	m1	p[5]	
6	m2	p[6]	

#	Input	Output	Bidirectional
7	m[3]	p[7]	

an lfsr with synaptic neurons (excitatory or inhibitatory) [333]

- Author: kai juarez-jimenez
- Description: each bit edge in the LFSR will mimic synaptic input that either excites / inhibits the next "neuron" , shwoing behaviors similar to how synapses manage signal in nns.
- GitHub repository
- HDL project
- Mux address: 333
- Extra docs
- Clock: 0 Hz

How it works

this project implements a neuromorphic-inspired Linear Feedback Shift Register (LFSR) with "synaptic neurons" that simulate excitatory/inhibitory responses. each bit in the LFSR behaves like a neuron, where transitions (rising/falling edges) from 0 to 1 or 1 to 0 generate excitatory or inhibitory signals, simulating synaptic inputs in neural networks. these signals modify the LFSR's feedback path, resulting in pseudo-random output sequences that mimic synaptic interactions by either enhancing (excitatory) or suppressing (inhibitory) activity.

additionally, this design allows for customizable seed inputs, set through external input pins, enabling users to initialize the LFSR with a specific seed to observe varying sequence outputs. this feature provides added flexibility and control over the pseudo-random behavior.

How to test

1. clock initialization: Run a clock signal to provide timing for the LFSR operation.
2. reset: hold the reset pin active (low) to initialize the LFSR state with the selected seed.
3. seed testing: configure the seed by setting the ui_in input pins, then observe the LFSR output sequence through uo_out.
4. cycle observation: monitor the output sequence over multiple clock cycles to verify pseudo-random behavior, and repeat for different seed values for varied sequences.

External hardware

N/A

Pinout

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

Generador PWM multiproposito con frecuencia y ciclo de trabajo modulable [334]

- Author: Marco Vázquez, Paúl González, Abimael Jimenez, UACJ
- Description: A PWM generator with a 6-bit input that allows the user to enter a denominator that divides the frequency. Using a pair of control inputs, we can increase or decrease the duty cycle of the modulated output by 10%.
- GitHub repository
- HDL project
- Mux address: 334
- Extra docs
- Clock: 5000 Hz

How it works

Overall, the module converts a high-speed clock signal into a PWM signal with adjustable frequency and duty cycle. The user receives a high-frequency clock signal and, through a frequency divider, generates a lower-frequency clock. Then, they control the high duration of the PWM signal using buttons that increase or decrease the duty cycle value.

A 5kHz signal is received; the 6-bit divider only accepts numbers from 2 to 63 (decimal). The possible output frequencies for the PWM range from 2500Hz (5kHz/2) to 79Hz (5kHz/63), which can be used in different electronic components such as RGB LEDs, servomotors, stepper motors, sensors, and other circuits.

How to test

- Connect the clock signal: Assign a high-frequency clock.
- Apply the reset signal: Initially set the reset to high to restart the module. This will reset all counters and the duty cycle to their initial values.
- Set the frequency divider: Define the frequency divider value to adjust the speed of the clock used. This value controls the PWM signal frequency. A higher divider value will result in a lower PWM frequency, and vice versa.
- Duty cycle adjustment buttons: When activating the increment button, the duty cycle will increase by 10%. When activating the decrement button, the duty cycle will decrease by 10%.

Recommendation: Use the PWM signal only as a control signal; the power supply for the devices it is applied to should come from an external power source.

External hardware

The PWM output should go to a PMOD to have that control signal available on a device.

Pinout

#	Input	Output	Bidirectional
0	increase_duty	pwm_out0	
1	decrease_duty	pwm_out1	
2	divisor[0]	pwm_out2	
3	divisor1	pwm_out3	
4	divisor2	pwm_out4	
5	divisor[3]	pwm_out5	
6	divisor[4]	pwm_out6	
7	divisor[5]	pwm_out7	

Perceptron [335]

- Author: Clarence Chan
- Description: Hardware implementation of a single layer perceptron
- GitHub repository
- HDL project
- Mux address: 335
- Extra docs
- Clock: 0 Hz

How it works

Given 8 bits of inputs and 8 bits of weights, the single layer perceptron will classify the inputs as class 0 or 1.

How to test

Initialize inputs and expected output in test.py, then run `make` in the test subdirectory.

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	Input bit 1	Perceptron class output	Weight bit 1
1	Input bit 2		Weight bit 2
2	Input bit 3		Weight bit 3
3	Input bit 4		Weight bit 4
4	Input bit 5		Weight bit 5
5	Input bit 6		Weight bit 6
6	Input bit 7		Weight bit 7
7	Input bit 8		Weight bit 8

2_bit_7seg [416]

- Author: Nathaniel_Laurente
- Description: displays 0-3
- GitHub repository
- Wokwi project
- Mux address: 416
- Extra docs
- Clock: 0 Hz

How it works

You use in1 and in0 as binary inputs to display a number in decimal on the 7 segment display. ex. {in0, ~in1}, will display "2".

How to test

Use the "1" switch to toggle the most significant bit, and the "2" switch to toggle the least significant bit.

External hardware

LED display, 8 input switch.

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	

Adbe_Project [417]

- Author: Aditya_Bedekar
- Description: basic project
- GitHub repository
- Wokwi project
- Mux address: 417
- Extra docs
- Clock: 0 Hz

How it works

Aditya's init

How to test

A basic project

External hardware

Need to fill in

Pinout

#	Input	Output	Bidirectional
0	IN0	OUT0	
1	IN1		
2			
3			
4			
5			
6			
7			

8 bit LFSR [418]

- Author: Aaron Nowack
- Description: 8 Bit LFSR, aka 8 Bit Pseudo Random Number Generator
- GitHub repository
- Wokwi project
- Mux address: 418
- Extra docs
- Clock: 0 Hz

How it works

A simple 8 bit LFSR I took from this paper that popped up in a google search of LFSR designs https://nandland.com/lfsr-linear-feedback-shift-register/

How to test

The 8 output pins should output a pseudo random 8 bit number, about once per second

External hardware

The 8 output pins can be connected to LEDs.

Pinout

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

Odd or even [419]

- Author: Eliana
- Description: odd or even input
- GitHub repository
- Wokwi project
- Mux address: 419
- Extra docs
- Clock: 0 Hz

How it works

Depending on weather the input is odd or even the green LED will light up for odd and the red for even.

How to test

Turn on different outputs to see if they are odd or even with the different LEDs.

External hardware

None

Pinout

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

Broken Two Bit Adder [420]

- Author: Mann
- Description: Performs binary addition on two 2-bit numbers
- GitHub repository
- Wokwi project
- Mux address: 420
- Extra docs
- Clock: 0 Hz

How it works

Adds two 2-bit numbers using logic gates.

How to test

IN0, IN1 represents the first 2-bit number, IN2, IN3 represents the second 2-bit number.

External hardware

Switches and LEDs.

Pinout

#	Input	Output	Bidirectional
0	IN0		
1	IN1		
2	IN2		
3	IN3		
4			
5			
6			
7			

Manchester Encoder [421]

- Author: Prajwal Shashidhar Chavadi
- Description: Manchester Encoder
- GitHub repository
- Wokwi project
- Mux address: 421
- Extra docs
- Clock: 0 Hz

How it works

It encodes incoming serial data to manchester encoded serial data via a FSM with a clock frequency twice the frequency of the input data frequency

How to test

Apply incoming serial data with a frequency f and the design with a frequency x2 of the original frequency. check output for encoded data.

External hardware

NOT Gate, AND gate, D FlipFlops, OR gate, Clocks

Pinout

#	Input	Output	Bidirectional
0	Encoder clock x2 frequency	Encoded Data	
1	Input Data x1 frequency		
2			
3			
4			
5			
6			
7			

4 bit adder [422]

- Author: Angel Lim Hui Yi
- Description: a 4-bit adder
- GitHub repository
- Wokwi project
- Mux address: 422
- Extra docs
- Clock: 0 Hz

How it works

The project is a 4-bit adder

How to test

input 0/1 and it will output the addition of it

External hardware

no external hardware

Pinout

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

Tiny_Tapeout_Adder! [423]

- Author: Abhinav Chaubey
- Description: This project synthesizes to an Adder!
- GitHub repository
- Wokwi project
- Mux address: 423
- Extra docs
- Clock: 0 Hz

How it works

This project works by putting two numbers (x and y), and a carry in. This module adds the two numbers & the carry_in, and returns carry_out and sum

How to test

Check is x and y is equal to sum.

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	X	Sum	
1	Y	Carry-out	
2	Carry-In		
3			
4			
5			
6			
7			

TinyTapeout workshop - Wokwi 8 Bit LFSR [424]

- Author: Nate Voorhies
- Description: An 8 bit Fibonacci lfsr
- GitHub repository
- Wokwi project
- Mux address: 424
- Extra docs
- Clock: 0 Hz

How it works

Just a 8-bit LFSR that zooms along. RST_N slaps a 1 in case it comes up all zeros. Period should be 255

How to test

Raise RST_N, then lower. We should see all 255 patterns of 8 bits values that != 8'h0 over the next 255 clocks.

External hardware

None.

Pinout

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

Morse Code for J and R [425]

- Author: Jainil Rao
- Description: A morse code generator circuit for alphabets J and R
- GitHub repository
- Wokwi project
- Mux address: 425
- Extra docs
- Clock: 0 Hz

How it works

I have used a 4bit synchronous counter to used And and Or gates to simulate "." and "-" of morsecode.

How to test

We can test it by connecting a 7 segment display and we can change inputs with a button(for J it's ".—" and for R it's ".-.").

External hardware

button, 7 segment display, clock

Pinout

#	Input	Output	Bidirectional
0		OUT0	
1		OUT1	
2			
3			
4			
5			
6			
7			

3bitFullAdder [426]

- Author: Isabella Phung
- Description: 3-bit full adder, where A[3:0] is inputs IN0-IN2, B[3:0] is inputs IN3-IN5, and the carry in is IN6. Output at OUT0 to OUT2, carryout at OUT3
- GitHub repository
- Wokwi project
- Mux address: 426
- Extra docs
- Clock: 0 Hz

How it works

3 bit adder, A[0:3] on inputs IN0-IN3, B[0:3] on inputs IN4-IN5, output on outputs OUT0-OUT2, carry out on OUT3.

How to test

flip switches, should connect to hex 7 seg leds to display output.

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	IN5		
6	IN6		
7			

XorTree [427]

- Author: Ammar Ratnani
- Description: Computes the XOR of all inputs
- GitHub repository
- Wokwi project
- Mux address: 427
- Extra docs
- Clock: 0 Hz

How it works

Computes the sum-mod-2 of all the inputs, and puts the output on a pin.

How to test

N/A

External hardware

N/A

Pinout

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

Sigma-Delta ADC [428]

- Author: Martin Schoeberl
- Description: Analog to digital converter - and back
- GitHub repository
- Wokwi project
- Mux address: 428
- Extra docs
- Clock: 50000000 Hz

How it works

Provide an analog signal at Vin (e.g., some music) and listen to the output at Vout (using an amplifier). The circuit will probably add some noise, as it is very crude. But it went from the analog domain to digital and then back.

Future work should be to use that sigma-delta coded signal to do some fun audio processing.

Anyone knowing how to do DSP in the sigma-delta domain?

How to test

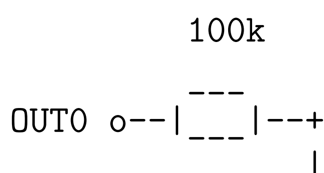
Connect an analog source to your design and listen to the music (output).

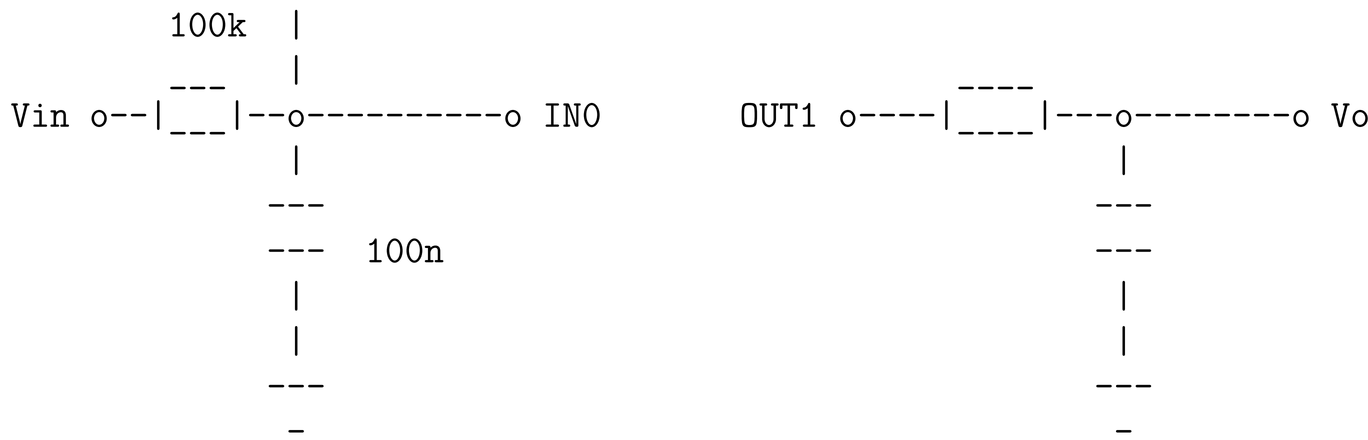
External hardware

This is a sigma-delta AD converter and a DA converter.

The input is mixed with the feedback delay/inversion and uses the threshold of the DFF input as a comparator, serving as a single bit ADC.

The R and C values depend on the input signal and can be discussed and should be explored.





Pinout

#	Input	Output	Bidirectional
0	IN0	OUT0	
1		OUT1	
2			
3			
4			
5			
6			
7			

tt09-4bit-adder-dhags [429]

- Author: Danny
- Description: This is a 4-bit adder that takes two 4-bit inputs and adds them, outputting them as a 5-bit number.
- GitHub repository
- Wokwi project
- Mux address: 429
- Extra docs
- Clock: 0 Hz

How it works

This is a 4-bit adder that takes two 4-bit inputs and adds them, outputting them as a 5-bit number.

How to test

Flip the DIP Switch with inputs 0-3 corresponding to the first number and 4-7 corresponding to the second number. See if the output leds light up in a way that makes sense if the two numbers were added.

External hardware

DIP Switch and a 7-segment display or leds.

Pinout

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

Mini-Adder and Clock Divider [430]

- Author: Marcus
- Description: 2-digit full adder and clock divider
- GitHub repository
- Wokwi project
- Mux address: 430
- Extra docs
- Clock: 0 Hz

How it works

The first 4 inputs are 2 two-digit binary inputs. The MSB of each input is at IN0 and IN2, respectively. These are added together into a three-digit binary output that is displayed as the first 3 vertical lines of the seven segment display - specifically, segments f, b, and e, where segment f is the MSB of the output.

Additionally, there is a clock divider that divides 10kHz by 2^16 that turns DP on and off.

How to test

Enabling the below inputs should result in the following ouput: | IN0 | IN1 | IN2 | IN3 | F | B | E | |—–|—–|—–|—–|—|—|—| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 1 | 0 | 0 | 0 | 0 | 1 | | 1 | 0 | 0 | 0 | 0 | 1 | 0 | | 0 | 0 | 1 | 1 | 0 | 1 | 1 | | 1 | 1 | 0 | 1 | 1 | 0 | 0 | | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

Pinout

#	Input	Output	Bidirectional
0	IN0		
1	IN1	OUT1	
2	IN2		
3	IN3		
4		OUT4	
5		OUT5	
6			
7			

7-seg display checker [431]

- Author: Ryan Taylor
- Description: Converts the inputs to seven seg display
- GitHub repository
- Wokwi project
- Mux address: 431
- Extra docs
- Clock: 0 Hz

How it works

Just converts inverts to 7segdisplay using combinational logic

How to test

Just test each input with all others zero, so in1 = 1 and in2 through in in8 = 0; should show 1. Same for all numbers. Also all inputs should show the dot.

External hardware

Nothing 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	IN7	OUT6	
7	IN0	OUT7	

Drew's First Wokwi Design [448]

- Author: ReanimationXP
- Description: Drew's First Wokwi Design
- GitHub repository
- Wokwi project
- Mux address: 448
- Extra docs
- Clock: 0 Hz

How it works

It is a thing, and it does stuff.

How to test

Hook it up to things, and see if it does stuff.

External hardware

'Tis currently a mystery.

Pinout

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

Shadoff Test [449]

- Author: David Shadoff
- Description: Gate Sample
- GitHub repository
- Wokwi project
- Mux address: 449
- Extra docs
- Clock: 0 Hz

How it works

This project demonstrates trivial AND gates and inverters. To be used for demonstrating logical inputs/outputs, tool flow, and 3D visualization.

How to test

OUT0 is driven by the logical AND value from IN0 and RST_N OUT1 is driven by the logical AND value from IN1 and IN0

OUT2 is driven by the opposite (inverted) logic value from IN2 OUT3 is driven by the opposite (inverted) logic value from IN3

OUT4 is driven by the logic value at IN4 OUT5 is driven by the logic value at IN5 OUT6 is driven by the logic value at IN6 OUT7 is driven by the logic value at IN7

External hardware

RESET pushbutton DIP Switch (for inputs) LED display

Pinout

#	Input	Output	Bidirectional
0	IN0	OUT0	
1	IN1	OUT1	
2	IN2	OUT2	
3	IN3	OUT3	
4	IN4	OUT4	
5	IN5	OUT5	

#	Input	Output	Bidirectional
6	IN6	OUT6	
7	IN7	OUT7	

Pseudo Random Generator Using 2 Ring Oscillators [450]

- Author: Michael Yim
- Description: Pseudo Random Generator Using 2 Ring Oscillators
- GitHub repository
- Wokwi project
- Mux address: 450
- Extra docs
- Clock: 10000 Hz

How it works

Just connect power. In theory, the two ring oscillators will start to oscillate. The random output will be sampled at the D flip flop at every clock (10K Hz). Out1 and Out2 will be the complementary outputs of the random generator.

How to test

Just connect power. In theory, the two ring oscillators will start to oscillate. The random output will be sampled at the D flip flop at every clock (10K Hz). Out1 and Out2 will be the complementary outputs of the random generator.

External hardware

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

Pinout

#	Input	Output	Bidirectional
0		OUT0	
1		OUT1	
2			
3			
4			
5			
6			
7			

Tiny Tapeout Take 2 [451]

- Author: Stephanie Rosales
- Description: We got nands
- GitHub repository
- Wokwi project
- Mux address: 451
- Extra docs
- Clock: 0 Hz

How it works

Pins 0, 1 are connected to a NAND Gate. Pines 2, 3 are connected to another nand gate. Those 2 nand gates are then tied to gether and the output is for pin 1.

How to test

Set pins 1-7 high

External hardware

N/A

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	

JonsFirstTapeout [452]

- Author: ghangas
- Description: Output J,O,N on a seven segment display
- GitHub repository
- Wokwi project
- Mux address: 452
- Extra docs
- Clock: 0 Hz

How it works

If pin 1 2 and 3 will spell Jon on a seven segment display.

How to test

Pin1 outputs J Pin 2 outputs O Pin 3 outs N

External hardware

Seven segment display

Pinout

#	Input	Output	Bidirectional
0	common1	out0	
1	In1	out1	
2	In2	out2	
3		out3	
4		out4	
5		out5	
6		out6	
7		out7	

Speller [453]

- Author: Aaron Eiche
- Description: Spells 'AAron' on a 7-segment display
- GitHub repository
- Wokwi project
- Mux address: 453
- Extra docs
- Clock: 0 Hz

How it works

Pressing the clock button should step through the 5 letters of my name, output on a 7-segment display.

How to test

Press the clock button, letters appear. Note that 0 and 1 are the same letter.

External hardware

None (Well, the 7-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	

And Gates that don't do much [454]

- Author: Chris Collins
- Description: My introduction to tiny tapeout
- GitHub repository
- Wokwi project
- Mux address: 454
- Extra docs
- Clock: 0 Hz

How it works

Still figuring out how it works.

How to test

Flip the dip switches

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any 7 segment display

Pinout

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

RAYS FIRST TAPEOUT rev 2 [455]

- Author: RAY STITS
- Description: rays first tapeout V3
- GitHub repository
- Wokwi project
- Mux address: 455
- Extra docs
- Clock: 3 Hz

How it works

clock input goes into string of d flip flops making the led segments illuminate in a circle. may want to hit reset to clear the d flip flops if more than 1 segment is illuminated.

How to test

turn on clock switch or press step button.

External hardware

clock button 7seg led

Pinout

#	Input	Output	Bidirectional
0		out0	
1		out1	
2		out2	
3		out3	
4		out4	
5		out5	
6			
7		out7	

SimplePattern [456]

- Author: Poorn
- Description: Display P on 7-seg display when 0x46 is inputted
- GitHub repository
- Wokwi project
- Mux address: 456
- Extra docs
- Clock: 0 Hz

How it works

This is a very simple project that displays ‘P’ on the 7-segment display, when the 8-bit input is 0x46. This is achieved through simple logic design & a few number of basic logic gates.

How to test

By default, when all inputs are 0 (0x00), the 7-segment output should have inversed segment states required for displying P (which basically means that if you turn OFF all segments required for P and turn ON the rest of the ‘digit’ making segments then that’s what it would be showing). It wasn’t intentional but I think it’s pretty cool that it turned out this way.

As different patterns are inputted, different segments of the display will turn on/off. When the right combination (0x46 –> IN1, IN2, IN6 are logic 1 and every other INs are 0) is provided in the input, the 7-segment display should light up segments to display P (OUT2, OUT3, OUT7 are OFF, every other is lit up or ON).

External hardware

A breadboard, 1.8V power supply, some connector wires, 8 position DIP switch, and a 7 segment display is required.

#	Input	Output	Bidirectional

Pinout

#	Input	Output	Bidirectional
0	IN0		
1		OUT1	
2	IN2		
3	IN3		
4		OUT4	
5	IN5	OUT5	
6	IN6		
7	IN7		

6 Bit shift register [457]

- Author: MOMO
- Description: 6 Bit shift register
- GitHub repository
- Wokwi project
- Mux address: 457
- Extra docs
- Clock: 0 Hz

How it works

6 Bit shift register

How to test

Apply Clk and Din to inputs 0/1 and observe data on output pins 0-5

Pinout

#	Input	Output	Bidirectional
0	Clk	Dout 0	NIC
1	Data	Dout 1	NIC
2	NIC	Dout 2	NIC
3	NIC	Dout 3	NIC
4	NIC	Dout 4	NIC
5	NIC	Dout 5	NIC
6	NIC	NIC	NIC
7	NIC	NIC	NIC

sphereinabox hello [458]

- Author: Nick Winters
- Description: Hello World
- GitHub repository
- Wokwi project
- Mux address: 458
- Extra docs
- Clock: 0 Hz

How it works

I've built 8-input logic gates, all using each of the 8 inputs.... or will eventually

How to test

Set the inputs 0..7 to your desired 8 inputs.

Observe the outputs the corresponding output pins.

External hardware

No specific external hardware is expected for this project.

Pinout

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

Duffy [459]

- Author: Jonathan Duffy
- Description: trying out an oscillator or delay line
- GitHub repository
- Wokwi project
- Mux address: 459
- Extra docs
- Clock: 0 Hz

How it works

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

How to test

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

External hardware

Nothing specific, switches or digital in to the input

Pinout

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

Input Counter [460]

- Author: Benjamin Meyer
- Description: Counts the number of switches turned on
- GitHub repository
- Wokwi project
- Mux address: 460
- Extra docs
- Clock: 0 Hz

How it works

This project counts the number of input switches turned on.

How to test

Just flip the switches!

External hardware

No external hardware is needed

Pinout

#	Input	Output	Bidirectional
0	IN0	OUT0	
1	IN1	OU1	
2	IN2	OUT2	
3	IN3	OUT3	
4	IN4	OUT4	
5	IN5	OUT5	
6	IN6	OUT6	
7	IN7	OUT7	

Will It NAND? [461]

- Author: Daniel Samarin
- Description: A bunch of nand gates to test the tool chain… for now.
- GitHub repository
- Wokwi project
- Mux address: 461
- Extra docs
- Clock: 0 Hz

How it works

Yo, it's just a bunch of NANDs.

How to test

Be a man, use your hand to connect up your NAND.

External hardware

Put it in the sand, like it's silicon, because you're a silly con.

Pinout

#	Input	Output	Bidirectional
0	NAND1a	NAND1out	
1	NAND1b	NAND2out	
2	NAND2a	NAND3out	
3	NAND2b	NAND4out	
4	NAND3a		
5	NAND3b		
6	NAND4a		
7	NAND4b		

4 bit ALU [462]

- Author: Gabriela Alfaro
- Description: A simple design of an Arithmetic Logic Unit capable of basic operations: addition, substraction , multiplication, division and some logic operations.
- GitHub repository
- HDL project
- Mux address: 462
- Extra docs
- Clock: 0 Hz

How it works?

The 4-bit ALU (Arithmetic Logic Unit) is designed to perform a range of arithmetic and logical operations on two 4-bit inputs, A and B. The operation is determined by a 3-bit control signal, Opcode, which specifies the function to execute, such as addition, subtraction, multiplication, division, and bitwise operations (AND, OR, NOT, XOR).

When an arithmetic operation like addition is selected, the ALU outputs an 8-bit result, ALU_Result, to accommodate larger sums or products, and it sets a Carry flag if there's an overflow. For logical operations like AND or OR, the ALU applies the operation bit-by-bit between A and B. The Zero flag is activated when the result is zero, providing a useful condition for further logic. This flexibility allows the ALU to handle various computational tasks, making it a crucial part of digital systems that require multi-functional data processing.

How to test?

To test the design, the operation codes are:

- Addition (000)
- Substraction (001)
- Multiplication (010)
- Division (011)
- Logic AND (100)
- Logic OR (101)
- Logic NOT (110)
- Logic XOR (111)

Pinout

#	Input	Output	Bidirectional
0	A[0]	ALU_Out[0]	ZeroFlag
1	A1	ALU_Out1	CarryOut
2	A2	ALU_Out2	
3	A[3]	ALU_Out[3]	
4	B[0]	ALU_Out[4]	
5	B1	ALU_Out[5]	
6	B2	ALU_Out[6]	
7	B[3]	ALU_Out[7]	

Bad Logic [463]

- Author: AaronV
- Description: Basic (broken) logic
- GitHub repository
- Wokwi project
- Mux address: 463
- Extra docs
- Clock: 0 Hz

How it works

Converts ASCII binary 'A' to 'V'

How to test

Input 0x37 on the input pins. The output pins should display 0x56.

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	

Full Adder [481]

- Author: Harish Prabhakaran
- Description: Simple Full Adder Schematic in Wokwi
- GitHub repository
- Wokwi project
- Mux address: 481
- Extra docs
- Clock: 0 Hz

How it works

My Wokwi design is a simple full adder schematic. There are three inputs that correspond to two 1-bit inputs and a carry bit. There are two outputs hooked up to two LEDs (one red and one blue) that correspond to the sum and carry bits respectively.

How to test

:/

External hardware

External hardware for this project includes two LEDs (colors are up to discretion)

Pinout

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

2048 sliding tile puzzle game (VGA) [482]

- 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: 482
- Extra docs
- Clock: 0 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. 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

After resetting the game, you will see a jumping "2048" animation on the screen. Press any of the `ui_in[3:0]` pins to start the game. The game will start with two tiles with the number 2 on the board. Use the `ui_in` pins 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 by setting `ui_in[6]`.

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

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		R0	debug_data
5		G0	debug_data
6	retro_colors	B0	debug_data
7	debug_mode	HSync	debug_data

TT-Farhad [483]

- Author: Farhad
- Description: simple design
- GitHub repository
- Wokwi project
- Mux address: 483
- Extra docs
- Clock: 0 Hz

How it works

It's 2 flip flops connected to the IO.

How to test

D pin of flip flops are IN0 and IN2, Q is OUT0 and OUT2.

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		OUT1	
2	IN2	OUT2	
3		OUT3	
4			
5			
6			
7			

Four Bit Adder [485]

- Author: Anahit
- Description: Adds two four-bit numbers together.
- GitHub repository
- Wokwi project
- Mux address: 485
- Extra docs
- Clock: 0 Hz

How it works

Two numbers in [0-15] are converted into 4 bit binary numbers. For example, A = 8 and B = 3: A = 1000 B = 0011 The two numbers are then added.

How to test

Enter the numbers using the switches (inputs 0-3 for A (LSB=0), inputs 4-7 for B (LSB=4)) and observe the ouput (LSB=0)

External hardware

Switches for input (8), LEDs for output (5).

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	

SPI Logic Analyzer with Charlieplexed Display [486]

- Author: ParallelLogic-
- Description: Displays contents of register map on charlieplexed display. Generates waveforms for PWM, UART, WS2812 in response to trigger.
- GitHub repository
- HDL project
- Mux address: 486
- Extra docs
- Clock: 10000000 Hz

How it works

The bi-directional pins are used to drive a charliplexed 8*7 LED display. A SPI serial connection is used to set the values in a register map. Auxilary functions are implemented, space/time permitting, ex: LFSR, PWM, freqency counting, ultrasonic distance sensing

How to test

Use SPI to read/write values to the register map, observe the output on the LEDs and/or in the serial response. CS active low SPI MODE 0 SPI_CLK <= SYS_CLK/2 Most signigicant bit is exchanged first

External hardware

Charlielexed 7*8 LED display

Pinout

#	Input	Output	Bidirectional
0	CS	ASIC_OUT_0	MAT0
1	SCLK	ASIC_OUT_1	MAT1
2	MOSI	ASIC_OUT_2	MAT2
3	TRIGGER	ASIC_OUT_3	MAT3
4	ASIC_IN_0	ASIC_OUT_4	MAT4
5	ASIC_IN_1	ASIC_OUT_5	MAT5
6	ASIC_IN_2	ASIC_OUT_6	MAT6

#	Input	Output	Bidirectional
7	ASIC_IN_3	MISO	MAT7

2 bit adder [487]

- Author: Aadarsha Kandel
- Description: this is a 2 bit adder
- GitHub repository
- Wokwi project
- Mux address: 487
- Extra docs
- Clock: 0 Hz

How it works

It works by adder two bit numbers

How to test

set all inputs

External hardware

Buttons and leds

Pinout

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

pio-ram-emulator example: Julia fractal [488]

- Author: Toivo Henningsson
- Description: Example of using pio-ram-emulator to draw a Julia fractal
- GitHub repository
- HDL project
- Mux address: 488
- Extra docs
- Clock: 50400000 Hz

How it works

This is an example of the using the https://github.com/toivoh/pio-ram-emulator RAM emulator for Tiny Tapeout. The RAM is used to store a frame buffer, 320x480 at 2 bits/pixel. The frame buffer is continuously read to output a 640x480 @60 Hz VGA signal. At the same time, the logic computes a Julia fractal, writing 16 bits to the frame buffer for every 8 pixels computed. After about a second, the whole frame buffer is filled in.

For more info about the RAM emulator, see https://github.com/toivoh/pio-ram-emulator/blob/main/docs/pio-ram-emulator.md.

The project contains some helper code for working with with the RAM emulator:

- `pio_ram_emulator.v` and `pio_ram_emulator.vh` (`sb_io.v` is also need) contain the modules `pio_ram_emu_transmitter` and `pio_ram_emu_receiver`
  - These are used to transmit and receive messages using the RAM emulator's message format
  - The design still has to follow the rules in https://github.com/toivoh/pio-ram-emulator/blob/main/docs/pio-ram-emulator.md about which messages can be sent when
  - See `julia_top.v` for an example of how to use these modules
- `test/pio_ram_emulator_model.v` contains a simulation model of the RAM emulator
  - See `test/tb.v` for an example of how to use the simulation model in a test
  - See `verilator/vtop.v` for an example of how to use the simulation model in a verilator setup

- The model will try to detect behavior that violates the rules in https://github.com/toivoh/pio-ram-emulator/blob/main/docs/pio-ram-emulator.md, in which case it will set an error flag and stop responding (see the `ERROR_RESPONSE` parameter)
- The simulation model is helpful, but might not capture the ful behavior of the RAM emulator. Please try to run your design on an FPGA against the actual RAM emulator as well.

How to test

Plug in a TinyVGA VGA Pmod to the output Pmod. The https://github.com/toivoh/pio-ram-emulator RAM emulator must be running on the RP2040. **TODO: Instructions for how to set up.** Start the project.

Controls The appearance of the Julia fractal is controlled by the `C` parameter, which can be seen as a complex value or 2d vector. The `C` paramter can be changed using the `ui_in` port:

- `button_up` / `button_down` / `button_left` / `button_right` move the `C` value.
- `button_incstep` doubles the step length.
- `button_decstep` halves the step length.

A new `ui_in[5:0]` value must be stable for 2^19 cycles, or approximately 10 ms (at a 50.4 MHz clock rate), before it is accepted. The `use_both_button_dirs` input changes how the input is interpreted:

- When `use_both_button_dirs = 0`, an input is triggered when one of the `button_` signals goes from high to low (and is stable for 10 ms). Recommended if the inputs are connected to buttons.
- When `use_both_button_dirs = 1`, an input is triggered when one of the `button_` signals goes from high to low or low to high (and is stable for 10 ms). Recommended if the inputs are connected to toggle switches.

External hardware

This project needs a TinyVGA VGA Pmod.

Pinout

#	Input	Output	Bidirectional
0	button_up	R1	
1	button_down	G1	
2	button_right	B1	
3	button_left	vsync	
4	button_incstep	R0	tx_out[0]
5	button_decstep	G0	tx_out1
6		B0	rx_in[0]
7	use_both_button_dirs	hsync	rx_in1

AND and NOT gate testing [489]

- Author: Aman Maldar
- Description: AND and NOT gate testing
- GitHub repository
- Wokwi project
- Mux address: 489
- Extra docs
- Clock: 0 Hz

How it works

Added 1 AND gate and 1 NOT gate for testing. AND gate is IN0 and IN1 as input. OUT1 is output. NOT gate has IN2 as input. OUT2 is output.

How to test

check AND gate truth table. check NOT gat truth table

External hardware

OUT1 and OUT2 drives LED

Pinout

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

Analog 8 bit 3.3v R2R DAC [490]

- Author: Matt Venn
- Description: A simple 8 bit DAC with a sine waveform driver and 3.3v output
- GitHub repository
- Analog project
- Mux address: 490
- Extra docs
- Clock: 0 Hz

How it works

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

3.3v output is achieved with level shifting drivers.

How to test

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

Then drive the 8 inputs and observe the analog output.

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

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

External hardware

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

#	Input	Output	Bidirectional

Pinout

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

Analog pins

ua#	analog#	Description
0	0	DAC output

Kanoa’s first Wokwi deseign Tinytapeout 2024 Nonsense [491]

- Author: Kanoa Mignard
- Description: Something random
- GitHub repository
- Wokwi project
- Mux address: 491
- Extra docs
- Clock: 0 Hz

How it works

It doesn’t yet

How to test

TBD

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	

Ring Oscillators [492]

- Author: Matt Venn
- Description: Ring Oscillators using analog output pins
- GitHub repository
- Analog project
- Mux address: 492
- Extra docs
- Clock: 0 Hz

How it works

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

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

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

- Ring oscillator 1
- Ring oscillator 2
- Driver

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

How to test

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

The 300 MHz ring oscillator had a problem with a missing contact in TT08, and is not expected to work. It has been fixed for TT09.

External hardware

Oscilloscope.

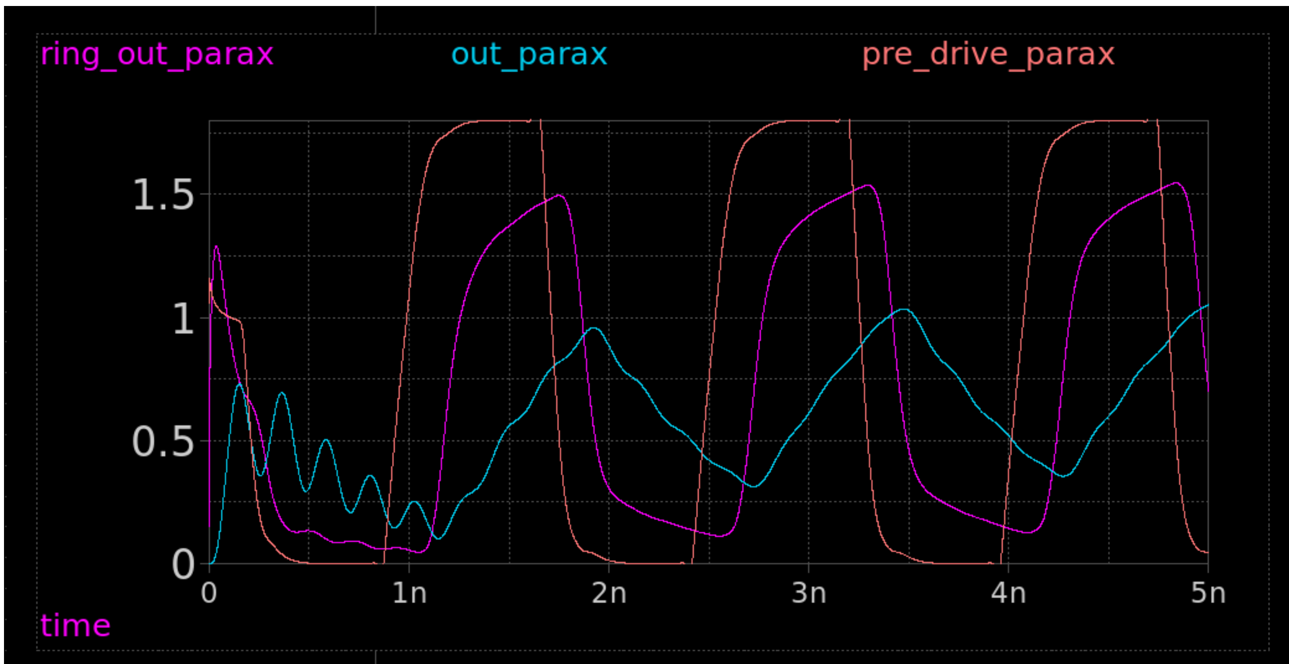


Figure 33: output waveform

Pinout

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

Analog pins

ua#	analog#	Description
0	3	
1	2	

add it [493]

- Author: alex b
- Description: half adder
- GitHub repository
- Wokwi project
- Mux address: 493
- Extra docs
- Clock: 0 Hz

How it works

Half adder

How to test

Click some inputs, see some xor and carry

External hardware

LEDS

Pinout

#	Input	Output	Bidirectional
0	IN0	OUT0	
1	IN1	OUT1	
2			
3			
4			
5			
6			
7			

AMS Chip ITS [494]

- Author: Astria Nur Irfansyah
- Description: ITS tinytapeout 3
- GitHub repository
- Analog project
- Mux address: 494
- Extra docs
- Clock: 0 Hz

How it works

This is planned to contain a simple ADC.

How to test

TBC

External hardware

TBC

Pinout

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

Analog pins

ua#	analog#	Description
0	1	out
1	3	
2	2	

one [495]

- Author: Neil
- Description: Shows 1 on the 7 segment display
- GitHub repository
- Wokwi project
- Mux address: 495
- Extra docs
- Clock: 0 Hz

How it works

If you turn on the first input, a 1 will be shown on the 7 segment display.

How to test

Wire up the outputs to a 7 segment display, and send a ON signal on IN0.

External hardware

7 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	

SIC-1 8-bit SUBLEQ Single Instruction Computer [518]

- Author: Uri Shaked
- Description: Hardware implementation of the 8-bit Single Instruction Computer
- GitHub repository
- HDL project
- Mux address: 518
- 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 (uo pins)

Programming the SIC-1

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

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

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

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

Pinout

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

4-bit R2R DAC [520]

- Author: David Parent
- Description: Converts a 4 bit wide signal to an analog signal
- GitHub repository
- Analog project
- Mux address: 520
- Extra docs
- Clock: 0 Hz

How it works

This is a simple 4-bit R2R DAC similar to the example.

How to test

Punt in a 4 bit strait case signal from 0000 to 1111 in steps of 1 on A0 A1 A2 and A3. Note that the LSB is controlled by pin UIN_0, not UIB_3.

External hardware

ADALM2000 in digital mode.

Pinout

#	Input	Output	Bidirectional
0	A3	Out	
1	A2		
2	A1		
3	A0		
4			
5			
6			
7			

Analog pins

ua#	analog#	Description
0	6	

Dickson Charge Pump [522]

- Author: Uri Shaked
- Description: Pumps the input voltage up to ~8V
- GitHub repository
- Analog project
- Mux address: 522
- Extra docs
- Clock: 2000000 Hz

How it works

A 3-stage dickson charge pump. The output voltage is `Vout = 4*(VAPWR - Vd) = ~9.6 V` where `VAPWR` is the analog input voltage (nominally 3.3 V), and Vd is the diode drop (~0.9 V). The output voltage is divided by two and available at the `ua[0]` pin.

How to test

Apply a clock signal of 2 MHz to the `clk` input. In TT09, the analog pin voltage is limited to VDDIO/VDDA (usually 3.3 V), so the output voltage will be divided by three. You can measure the divided output voltage at the `ua[0]` (vout_div) pin.

Simulation results

Post layout simulation showing the output voltage `x1.vout` and the divided output voltage on ta `ua[0]` pin. The output voltage stabilizes at ~8.7 V, and the divided output voltage at ~2.88 V. The current draw is about 1.2 uA (measured by adding a 1k resistor between `ua[0]` and VGND in simulation).

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

Project layout

Pinout

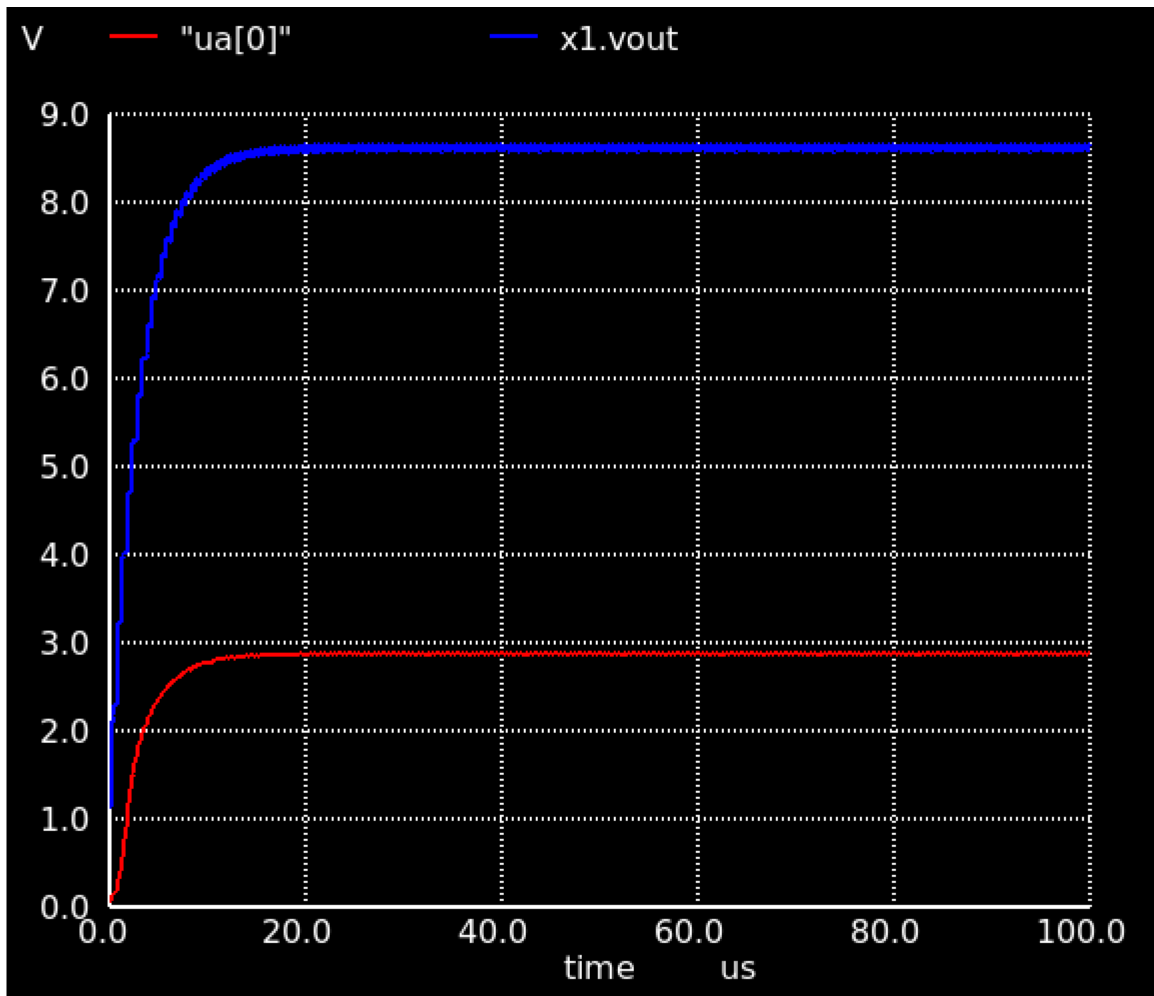


Figure 34: output voltage and divided voltage

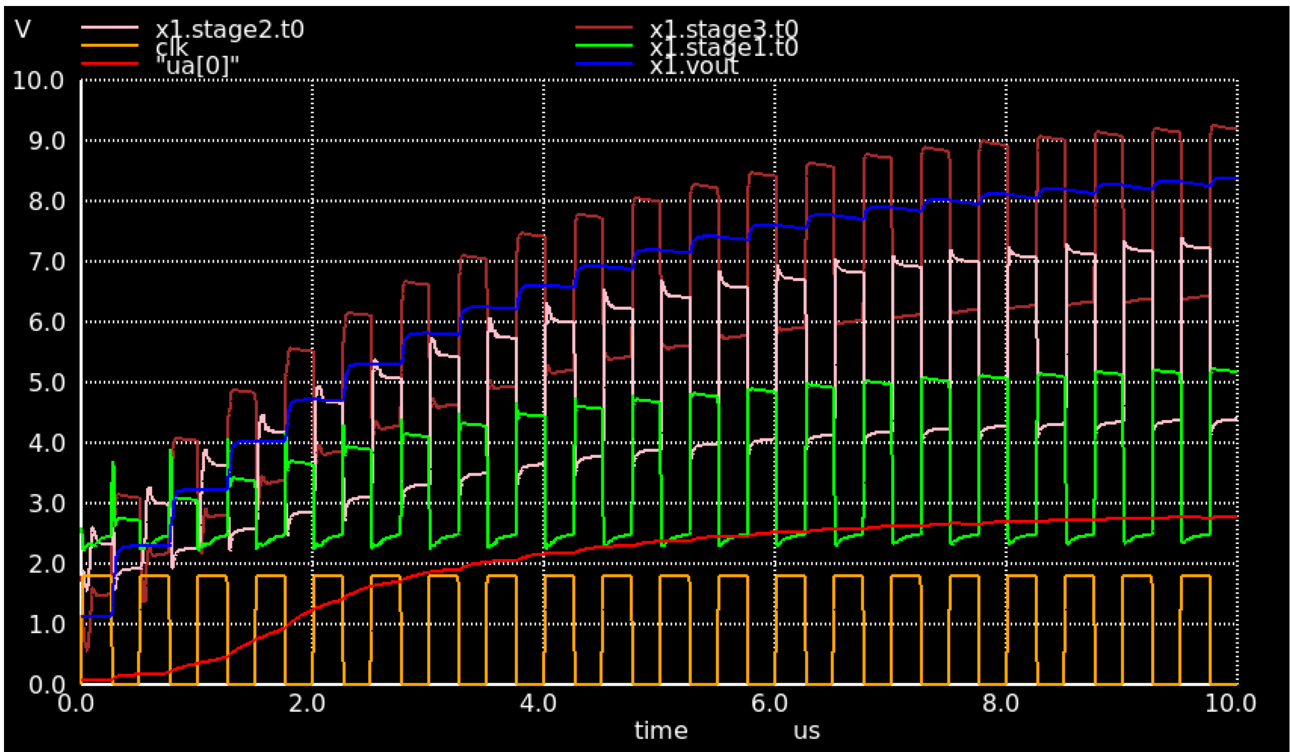


Figure 35: output voltage and intermediate voltages

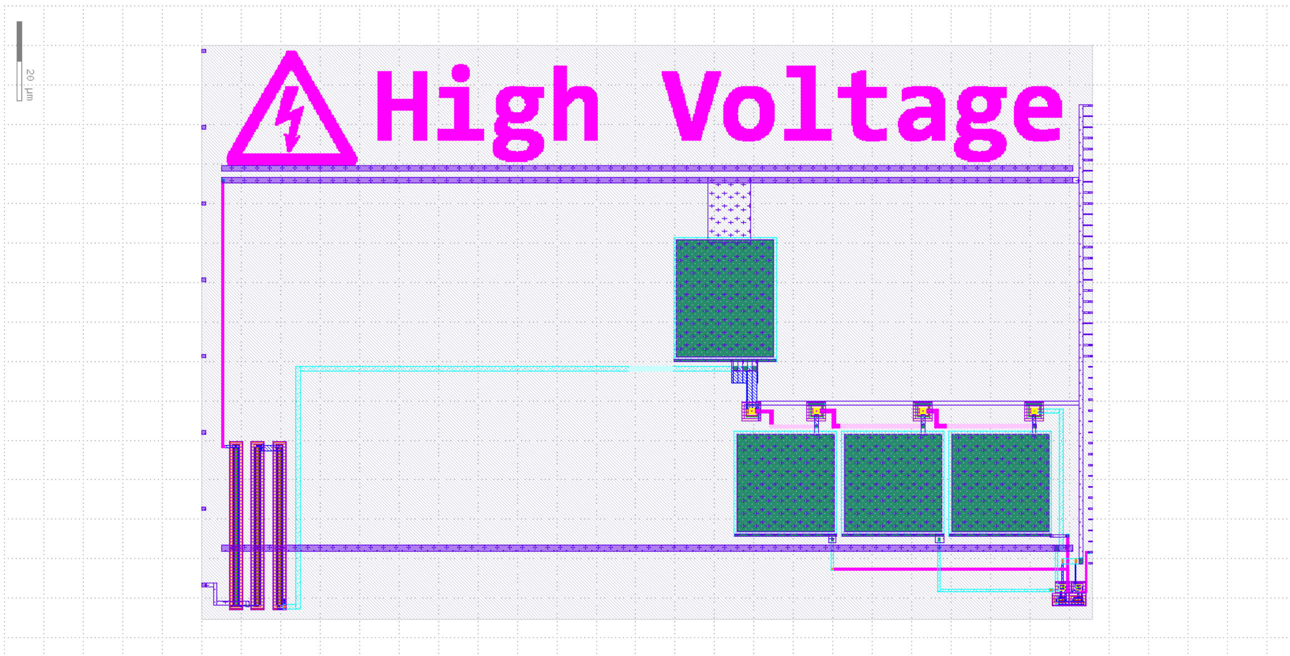


Figure 36: Project layout

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

Analog pins

ua#	analog#	Description
0	7	vout_div

Analog double inverter [524]

- Author: Matt Venn
- Description: A pair of inverters wired between 2 analog pins
- GitHub repository
- Analog project
- Mux address: 524
- Extra docs
- Clock: 0 Hz

How it works

A pair of inverters, a large one after a small one.

How to test

Put a signal into the input pin, and observe the output. It should match polarity.

The rise time of the output was simulated at less than 4ns.

External hardware

Signal generator, oscilloscope.

Pinout

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

Analog pins

ua#	analog#	Description
0	11	output
1	6	input

OpAmp 3stage [526]

- Author: Rod Burt
- Description: 3stage PMOS OpAmp
- GitHub repository
- Analog project
- Mux address: 526
- Extra docs
- Clock: 0 Hz

How it works

A test chip for a 3stage PMOS OpAmp.

How to test

Pinout: out -> ua[0], in- -> ua1, in+ -> ua2

External hardware

Typical analog bench setup.

Pinout

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

Analog pins

ua#	analog#	Description
0	7	out
1	9	in-
2	8	in+

Counter [544]

- Author: Alex Solomatnikov
- Description: Reverse 8-bit counter with reset
- GitHub repository
- Wokwi project
- Mux address: 544
- Extra docs
- Clock: 10 Hz

How it works

Dividing counter using a series of D-flops

How to test

Reset with reset button and press step button

External hardware

LEDs connected to every output pin + reset and step button, no input data buttons

Pinout

#	Input	Output	Bidirectional
0		out0	
1		out1	
2		out2	
3		out3	
4		out4	
5		out5	
6		out6	
7		out7	

Shifter [545]

- Author: Ethan Sifferman
- Description: Input » Inout
- GitHub repository
- HDL project
- Mux address: 545
- Extra docs
- Clock: 0 Hz

How it works

Output = Input[7:0] » Inout[7:0]

How to test

The LEDs will output the shifted value of Output = Input[7:0] » Inout[7:0].

External hardware

Switches on the inputs, LEDs on the outputs

Pinout

#	Input	Output	Bidirectional
0	ui_in[0]	uo_out[0]	uio_in[0]
1	ui_in1	uo_out1	uio_in1
2	ui_in2	uo_out2	uio_in2
3	ui_in[3]	uo_out[3]	uio_in[3]
4	ui_in[4]	uo_out[4]	uio_in[4]
5	ui_in[5]	uo_out[5]	uio_in[5]
6	ui_in[6]	uo_out[6]	uio_in[6]
7	ui_in[7]	uo_out[7]	uio_in[7]

7-bit arbiter [546]

- Author: Kira Tran
- Description: Simple combinational logic to arbit input (prioritize lowest of inputs 0 - 6)
- GitHub repository
- Wokwi project
- Mux address: 546
- Extra docs
- Clock: 0 Hz

How it works

OUT7 stays high as long as reset_n XOR clk is active (effectively, on when clk low). Other outputs (OUT0-6) have at most one high, based on inputs IN0-6, arbiting between multiple inputs. Arbiter prioritizes lowest active input.

How to test

Connect outputs to 7-segment LED display, or any other form of testable output (ex. LEDs). Change inputs and verify correct arbitration. OUT7, with a freerunning clock and stable reset, should oscillate periodically.

External hardware

Input switches, output 7-segment LED, possibly a button to step clk.

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		OUT7	

NAND Flip-Flop [547]

- Author: Luigi C.
- Description: Micro Design: Signle NAND Flip-Flop
- GitHub repository
- Wokwi project
- Mux address: 547
- Extra docs
- Clock: 0 Hz

How it works

It's a 4-NAND Flip-Flop, it works!

How to test

Try the switches

External hardware

2 LED's

Pinout

#	Input	Output	Bidirectional
0	CLK	OUT0	
1			
2	IN0		
3			
4			
5			
6			
7		OUT7	

LCA's first Wokwi design [548]

- Author: leahcorbett18
- Description: Number output from switches
- GitHub repository
- Wokwi project
- Mux address: 548
- Extra docs
- Clock: 0 Hz

How it works

The design outputs the number 1 on the 7-segment if switch[0] is high. This will repeat, so for switch 2 it will output number 2, and so on.

How to test

Turn on the switches and make sure that it outputs it's corresponding number.

External hardware

LED Display and switches

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 [549]

- Author: Olivia
- Description: lights up numbers
- GitHub repository
- Wokwi project
- Mux address: 549
- Extra docs
- Clock: 0 Hz

How it works

Produces numbers 1-3.

How to test

Turn on switch correlated with number. See LED display and output.

External hardware

none

Pinout

#	Input	Output	Bidirectional
0	in0	led0	
1	in1	led1	
2	in3	led2	
3		led3	
4		led4	
5		led5	
6		led6	
7		led7	

Tinysynth [550]

- Author: Erling Rennemo Jellum
- Description: A tiny square wave oscillator accepting MIDI commands.
- GitHub repository
- HDL project
- Mux address: 550
- Extra docs
- Clock: 50000000 Hz

How it works

Accepts MIDI commands over UART, generates a corresponding square wave signal using PWM.

How to test

External hardware

A Pmod AMP2 connected to the PMOD connector.

Pinout

#	Input	Output	Bidirectional
0	a	x0	b0
1	b	x1	b1
2	c	x2	b2
3	d	x3	b3
4	e	x4	b4
5	f	x5	b5
6	g	x6	b6
7	h	x7	b7

rhTinyTapeout [551]

- Author: Raphael Huang
- Description: Flashes my initials
- GitHub repository
- Wokwi project
- Mux address: 551
- Extra docs
- Clock: 10000 Hz

How it works

Using a clock divider and 2-input muxs, I created a circuit that flips between my initials.

How to test

Who needs tests?

External hardware

Switch at IN0, seven segment display at OUT0-OUT6

Pinout

#	Input	Output	Bidirectional
0	IN0	OUT0	
1		OUT1	
2		OUT2	
3		OUT3	
4		OUT4	
5		OUT5	
6		OUT6	
7		OUT7	

half adder [552]

- Author: Adam Wu
- Description: half adder with sum and carry out
- GitHub repository
- Wokwi project
- Mux address: 552
- Extra docs
- Clock: 0 Hz

half adder

takes two inputs and outputs a carry and a sum

Pinout

#	Input	Output	Bidirectional
0	IN0	OUT0	
1		OUT1	
2			
3			
4			
5			
6			
7			

rand [553]

- Author: mahi
- Description: just_random
- GitHub repository
- Wokwi project
- Mux address: 553
- Extra docs
- Clock: 0 Hz

How it works

Explain how your

How to test

Explain how to u

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	ino	out0	
1	in1	out1	
2	in2	out2	
3	in3	out3	
4	in4		
5	in5		
6	in6		
7	in7		

Tiny Tapeout 9 Template [554]

- Author: Jason
- Description: Flip da switches
- GitHub repository
- Wokwi project
- Mux address: 554
- Extra docs
- Clock: 0 Hz

How it works

Testing the LED digit

How to test

Flip switches as pleased

External hardware

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

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	

Ripple counter [555]

- Author: Marc Mignard
- Description: Initial test of Wokwi design
- GitHub repository
- Wokwi project
- Mux address: 555
- Extra docs
- Clock: 0 Hz

How it works

A 4-bit ripple counter, a NAND gate, and an AN gate

How to test

Try your best

External hardware

DIP switch, hex LED

Pinout

#	Input	Output	Bidirectional
0	CLK	OUT0	
1	IN0	OUT1	
2	IN1	OUT2	
3	IN2	OUT3	
4	IN3	OUT4	
5		OUT5	
6			
7			

four flip flops [556]

- Author: Arjun Vedantham
- Description: four flip flops tied together, drawing from input 0
- GitHub repository
- Wokwi project
- Mux address: 556
- Extra docs
- Clock: 1 Hz

How it works

Pulls the input from in0, and shifts the bit state along four flip flops -> outputs to a seven segment display

How to test

Flip the in0 switch :)

External hardware

Seven segment display/LEDs, switch

Pinout

#	Input	Output	Bidirectional
0	in0		
1			
2			
3			
4			
5			
6			
7			

adder-tt09 [557]

- Author: Philip Solomatnikov
- Description: 4bit adder with a carry output
- GitHub repository
- Wokwi project
- Mux address: 557
- Extra docs
- Clock: 0 Hz

How it works

The entire thing is based off of this image, so all credit to them. Uses XOR, AND, & OR gates for the entire thing.

Inputs

- **0.** - A0
- **1.** - B0
- **2.** - A1
- **3.** - B1
- **4.** - A2
- **5.** - B2
- **6.** - A3
- **7.** - B3

Outputs

- **0.** - Sum 0
- **1.** - Sum 1
- **2.** - Sum 2
- **3.** - Sum 3
- **4.** - Carry

How to test

Take 2 4-bit arrays and then feed them in in the following order:

- For array 1, write to A
- For array 2, write to B

External hardware

LEDs from Sum 0-3 for display. Switches from A0-3 and B0-3.

Pinout

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

Full Adder [558]

- Author: Amogha Srinivas
- Description: takes in two inputs a,b and Cin and outputs the sum and the carryout
- GitHub repository
- Wokwi project
- Mux address: 558
- Extra docs
- Clock: 0 Hz

How it works

takes in two inputs a and b, along with carry_in and gives output across 2 pins, sum and carry_out

How to test

to ensure correct working of the full adder , toggle with the inputs , if any two inputs are high , the sum should be LOW and the carry_out should be HIGH , if any one input is HIGH the sum should be HIGH and the carry_out should be LOW and if all three inputs are HIGH both sum and carry_out should be HIGH.

External hardware

LED display.

Pinout

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

NAND-Equ [559]

- Author: DanT
- Description: NAND Equivilant
- GitHub repository
- Wokwi project
- Mux address: 559
- Extra docs
- Clock: 0 Hz

How it works

Creates NAND equivilant to common gates

How to test

plug in and verify odd output matches even outputs

External hardware

dip switch and leds

Pinout

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

Elevator Design [576]

- Author: Jocelyn Zhu
- Description: Simulation of an elevator design on a digital clock display.
- GitHub repository
- HDL project
- Mux address: 576
- Extra docs
- Clock: 0 Hz

How it works

The project implements an elevator interface on a digital clock based on user floor selection. The user selects a floor using the board switches, and the display increments/decrements floor numbers according to the elevator's state (moving up, moving down, or idle). Once the elevator reaches the selected floor, the display shows the user-selected floor number until a different floor is chosen or the switches are reset. When the switches are reset, the display decrements back to the default floor.

How to test

Use board switches 0-7 to select the desired floor.

External hardware

A LED display is used to show elevator operation and the selected floor number.

Pinout

#	Input	Output	Bidirectional
0	Switch 0	Segment A	
1	Switch 1	Segment B	
2	Switch 2	Segment C	
3	Switch 3	Segment D	
4	Switch 4	Segment E	
5	Switch 5	Segment F	
6	Switch 6	Segment G	
7	Switch 7	A dot that appears during the IDLE state	

L display [578]

- Author: Matt Lamparter
- Description: Displays L character on a 7 seg display when 00101111 entered on the input (and pressing the Step button to enable display)
- GitHub repository
- Wokwi project
- Mux address: 578
- Extra docs
- Clock: 0 Hz

How it works

Enter the right combination of bits to display an “L” on the seven segment display. The combination is 0b00101111. Once that combination is entered you'll need to press the “Step” button in order to display the L on the display. Releasing the Step button will clear the display.

How to test

Try different combinations of inputs. The only time the output should be displayed is when the right bit combination is entered and the Step button is pressed.

External hardware

Requires a 7 segment display, a push button connected to power, and a 8 bit wide DIP switch.

Pinout

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

#	Input	Output	Bidirectional
7	IN7		

S-R latch [580]

- Author: Albert
- Description: S-R latch
- GitHub repository
- Wokwi project
- Mux address: 580
- Extra docs
- Clock: 0 Hz

How it works

This is a simple S-R latch

How to test

IN1 is S, so if IN1 is high, OUT0 is high and OUT1 is low. IN0 is R, so if IN0 is high, OUT0 is low and OUT1 is high.

External hardware

n/a

Pinout

#	Input	Output	Bidirectional
0	IN0	OUT0	
1	IN1	OUT1	
2			
3			
4			
5			
6			
7			

Gabe's Big AND [582]

- Author: GabeMake
- Description: Displays 1 if all inputs are high, 0 if not
- GitHub repository
- Wokwi project
- Mux address: 582
- Extra docs
- Clock: 0 Hz

How it works

I connected a bunch of AND gates together to make a really big AND gate.

How to test

Set all the inputs to 1 and the output should show 1 on the LED 7-segment display. Otherwise it should show a 0.

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	IN0	LCD_A	
1	IN1	LCD_B	
2	IN2	LCD_C	
3	IN3	LCD_D	
4	IN4	LCD_E	
5	IN5	LCD_F	
6	IN6	LCD_G	
7	IN7	LCD_P	

Secret Code [584]

- Author: Rex
- Description: Simple project for TinyTapeout
- GitHub repository
- Wokwi project
- Mux address: 584
- Extra docs
- Clock: 0 Hz

How it works

This project follows the "secret code" next step in the TinyTapeout workshop: https://tinytapeout.com/guides/workshop/simulate-a-gate/

If you input the correct code, you get the first letter of my name output on the 7-segment display ('r').

How to test

Set bits 10101010 in order on IN0 through IN7 ("1" is high, "0" low), then you should receive an output of 10001100 on OUT0 through OUT7. Any other input should output 00000000.

External hardware

- Dip switch for inputs
- 7-segment display for outputs

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	

#	Input	Output	Bidirectional
7	IN7	OUT7	

joes-first-tiny-tapeout [586]

- Author: securelyfitz
- Description: does nothing… yet
- GitHub repository
- Wokwi project
- Mux address: 586
- Extra docs
- Clock: 0 Hz

How it works

Great, in theory. (Note: it doesn't actually work yet)

How to test

Toggle inputs. See outputs toggle

External hardware

Buttons (in0 and in1) and leds (out0)

Pinout

#	Input	Output	Bidirectional
0	IN0	OUT0	
1	IN1		
2			
3			
4			
5			
6			
7			

Abey's 1st Chip Design [588]

- Author: Abey Varghese
- Description: 1st Project with 3 AND Gates
- GitHub repository
- Wokwi project
- Mux address: 588
- Extra docs
- Clock: 0 Hz

How it works

2 AND gates outputting to OUT 1 and OUT 2 along with anded AND gates on OUT3

How to test

Switch the dipswitch

External hardware

7-segment display, dipswitch, pushbutton

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	

patrick's project [590]

- Author: patrick marcus
- Description: it does nothing
- GitHub repository
- Wokwi project
- Mux address: 590
- Extra docs
- Clock: 0 Hz

How it works

it doesn't work

How to test

untestable!

External hardware

there is none!

Pinout

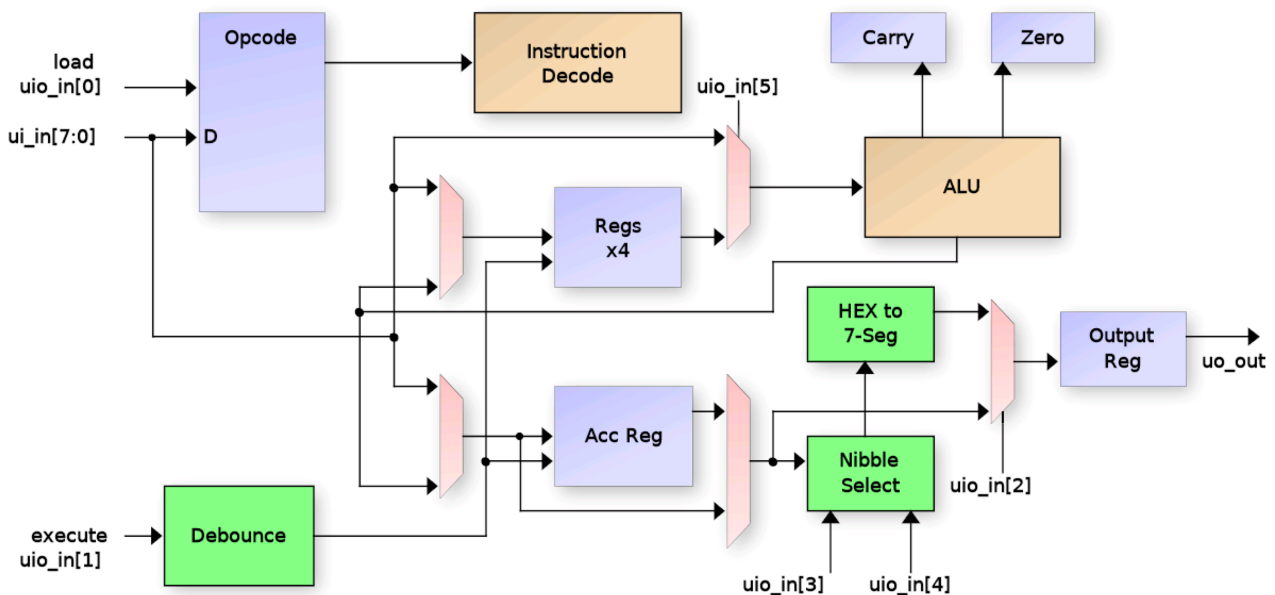
#	Input	Output	Bidirectional
0	1	o1	
1	2	o2	
2	3	03	
3	4	04	
4	5	05	
5	6	06	
6	7	07	
7	8	08	

tt09-pettit-wokproc-trainer [591]

- Author: Ken Pettit
- Description: An 8-bit CPU trainer
- GitHub repository
- Wokwi project
- Mux address: 591
- Extra docs
- Clock: 10000 Hz

What is WocProc Trainer?

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



How it works

It works by "feeding" it opcodes and data via the ui[7:0] and ui[0] input pins and then executing them by toggling the uio1 input pin. Some instructions require additional "Immediate Data" to be supplied via the ui[7:0] input pins prior to toggling the uio1 "execute" input.

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

How to test

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

   uio_in2: Selects 7-Segment (LOW) or binary (HIGH) output format uio_in[4]: Selects auto nibble / digit display (LOW) or manual (HIGH) uio_in[3]: Manual digit select when uio_in[4] is HIGH. uio_in[5]: Selects value to output (LOW = ACC reg, HIGH = new ACC load value)
3. Monitor the results using the 7-Seg display and uio_out[7:6] bits:

```
uio_out[6]: Indicates if CARRY bit is set
uio_out[7]: Indicates if ZERO bit is set
```

4. Perform an addition. After reset, the opcode register contains opcode 0x00, which is A = A + IMM. So supply a binry input value on ui_in[7:0] and toggle the EXECUTE input (uio_in1) HIGH then LOW. The 7-Seg display should display the HEX value of the sum.
5. The first addition just looked like a 'load' since Acc was zero from the reset. Add the value a second time (or supply a different value on ui_in[7:0]) and toggle the EXECUTE input again. The 7-Segment display should show the result of the addiiton.
6. Load register r0 from the A register. First enter the opcode (7'b1100_0000 from the opcode table) and then toggle the LOAD (ui_in[0]) input HIGH then LOW to load ui_in[7:] to the opcode register. Now toggle the EXECUTE (ui_in1) input HIGH then LOW. Register r0 should now contain the value from A.
7. Test if register r0 was loaded. First clear the Acc register (load opcode 7'b0111_0000 and EXECUTE it). The 7-Seg should show "00.". Now load and execute the opcode to load register r0 to Acc (opcode 7'b0110_0000). The 7-Seg should show the result of the summation that was stored in r0.
8. Perform a NOT operation on the A register by loading and executing opcode 7'0111_0001. The 7-Seg should show the compliment value of what was in A.

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

Opcodes supported:

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

Selecting the Output

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

uio_in2	uo_out Format
LOW	7-Segment
HIGH	Binary

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

uio_in[5]	uo_out Data
LOW	Acc Register
HIGH	ALU result (value loaded upon EXECUTE)

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

```
F1.
```

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

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

Hardware needed:

Dip switches and 7-Segment LED.

Pinout

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

Full adder Design [608]

- Author: Mithun
- Description: one bit full adder
- GitHub repository
- Wokwi project
- Mux address: 608
- Extra docs
- Clock: 0 Hz

How it works

The design has three inputs and two outputs. when a 1 bit 3 inputs are given, design gives the sum and carry.

How to test

Give 1 bit 3 inputs and check the 1 bit sum and 1 bit carry.

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		
3			
4			
5			
6			
7			

seven [609]

- Author: nikmign
- Description: seven
- GitHub repository
- Wokwi project
- Mux address: 609
- Extra docs
- Clock: 0 Hz

How it works

it works

How to test

click the buttons

External hardware

none

Pinout

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

Vincent's First Design [610]

- Author: Vincent Harkins
- Description: Not entirely sure
- GitHub repository
- Wokwi project
- Mux address: 610
- Extra docs
- Clock: 0 Hz

How it works

It takes a really long time but it should make a V.

How to test

turn on input 1

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		OUT1	
2		OUT2	
3		OUT3	
4		OUT4	
5		OUT5	
6			
7		OUT7	

gatesoup [611]

- Author: Elio Bourcart
- Description: a tasteful selection of logic gates between input and output
- GitHub repository
- Wokwi project
- Mux address: 611
- Extra docs
- Clock: 0 Hz

How it works

I connected a tasteful selection of logic gates between the inputs and outputs.

How to test

Get a snack, flip some switches, look at the pretty lights. Best enjoyed with a friend.

External hardware

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

Pinout

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

A Tale of Two NCOs [612]

- Author: Mike Ng
- Description: Two NCOs enter, one signal leaves
- GitHub repository
- HDL project
- Mux address: 612
- Extra docs
- Clock: 50000000 Hz

How it works

This design contains two NCOs, implemented with phase accumulators and sine lookup tables. The outputs of the NCOs are multiplied together by default. NCO B can be bypassed to a constant "one" or "half". There is also a boxcar filter for funsies.

When operating at 50 MHz, it should be possible to tune NCO A from 24.8 MHz to 0.195 MHz. NCO B has one less bit in its increment control, so it can only go up to 12.3 MHz.

How to test

The output is intended for something simple like an R-2R DAC. Don't expect it to be pretty at high frequency.

External hardware

- DAC
- Oscilloscope

Pinout

#	Input	Output	Bidirectional
0	phase_incr_A[0]	OUT0	phase_incr_B[0]
1	phase_incr_A1	OUT1	phase_incr_B1
2	phase_incr_A2	OUT2	phase_incr_B2
3	phase_incr_A[3]	OUT3	phase_incr_B[3]
4	phase_incr_A[4]	OUT4	phase_incr_B[4]
5	phase_incr_A[5]	OUT5	phase_incr_B[5]

#	Input	Output	Bidirectional
6	phase_incr_A[6]	OUT6	low_amplitude_B
7	filter_on	OUT7	bypass_B

Tiny Tapeout 9 Template Version 1 Tata Luka [613]

- Author: lukab
- Description: broken tatanator2000
- GitHub repository
- Wokwi project
- Mux address: 613
- Extra docs
- Clock: 1 Hz

How it works

testing 1 2 3 Explain how your project works

How to test

. Explain how to use your project

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	OUT2	
2			
3			
4			
5			
6			
7			

Workshop demo [614]

- Author: Tommy Thorn
- Description: Just a demo
- GitHub repository
- HDL project
- Mux address: 614
- Extra docs
- Clock: 50000000 Hz

How it works

It's magic

How to test

Connect the TX pin to your favorite terminal (more to be written)

External hardware

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

Pinout

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

UART TX [615]

- Author: Shaokai Lin
- Description: A UART transmitter modified from the one from the TinyTapeout website
- GitHub repository
- Wokwi project
- Mux address: 615
- Extra docs
- Clock: 50000000 Hz

How it works

This design is a UART TX that sends "SHAOKAI".

How to test

Connect it to your serial receiver and see if "SHAOKAI" gets printed.

External hardware

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

Pinout

#	Input	Output	Bidirectional
0		OUT0	
1		OUT1	
2			
3			
4			
5			
6	IN6		
7	IN7	OUT7	

LRC - Longitudinal Redundancy Check generator [616]

- Author: Steve Jenson <stevej@gmail.com>
- Description: LRC implementation for Tiny Tapeout 09
- GitHub repository
- HDL project
- Mux address: 616
- Extra docs
- Clock: 0 Hz

How it works

Calculates a running error correcting code. For each new byte applied to the input pins, calculates a running longitudinal redundancy code.

How to test

Supply a byte to `ui_in`, read the LRC on `uo_out`. Keep feeding it bytes and you'll keep getting new LRC codes. Code resets when the chip resets.

External hardware

No external hardware needed.

Pinout

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

my First WokWi Design [617]

- Author: Mani Rayabarapu
- Description: This is just a basic circuit that I designed in the WokWi environment with some basic logic gates.
- GitHub repository
- Wokwi project
- Mux address: 617
- Extra docs
- Clock: 0 Hz

How it works

This project is a simple counter circuit that increments and displays numbers on a 7-segment display. A "Step" button advances the count, and a "Reset" button resets it to zero. Logic gates control the segments based on the counter's output.

How to test

Press the "Step" button to increment the display and "Reset" to reset the count. Optionally, connect a clock for automatic counting.

External hardware

7-Segment Display Step and Reset buttons Clock source

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	

print [618]

- Author: Syeva
- Description: prints seven
- GitHub repository
- Wokwi project
- Mux address: 618
- Extra docs
- Clock: 0 Hz

How it works

Uses an and gate to make a seven on the LED display.

How to test

Toggle the 3 and 4 on in the switch, and it will print out a seven on the LED display.

External hardware

LED display

Pinout

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

Tiny Tapeout 9 [619]

- Author: Maya Choudhury
- Description: Testing logic
- GitHub repository
- Wokwi project
- Mux address: 619
- Extra docs
- Clock: 0 Hz

How it works

My project uses several flops and muxes.

How to test

Test carefully!

External hardware

Hook up to LEDs

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	

hello [620]

- Author: vishwajeet
- Description: Hello Bcd
- GitHub repository
- Wokwi project
- Mux address: 620
- Extra docs
- Clock: 0 Hz

How it works

Explain how your project

How to test

Explain how to use your

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	

tinydsp-lol [621]

- Author: Tassilo Tanneberger
- Description: testing digital dsp
- GitHub repository
- HDL project
- Mux address: 621
- Extra docs
- Clock: 2000000 Hz

How it works

This is just a test project to explore Chisel.

How to test

It should have a ChiselTest to run with `sbt test`.

External hardware

Nothing at the moment.

Pinout

#	Input	Output	Bidirectional
0	a	x0	b0
1	b	x1	b1
2	c	x2	b2
3	d	x3	b3
4	e	x4	b4
5	f	x5	b5
6	g	x6	b6
7	h	x7	b7

Full Adder [622]

- Author: David De La Luz
- Description: full adder from XOR, OR, AND gates
- GitHub repository
- Wokwi project
- Mux address: 622
- Extra docs
- Clock: 0 Hz

How it works

there are total of 3 inputs, the first two inputs are 1 bit inputs and they are added. the third input represents the carry value.

How to test

input your 1 bit values to be added to the first two inputs

External hardware

Two LEDs are being used, along with two XOR, two AND, and one OR gate

Pinout

#	Input	Output	Bidirectional
0	IN0	OUT0	BI1
1	IN1	OUT1	BI2
2	IN2	OUT2	BI3
3	IN3	OUT3	BI4
4	IN4	OUT4	BI5
5	IN5	OUT5	BI6
6	IN6	OUT6	BI7
7	IN7	OUT7	BI8

Leaky integrate and fire spiking neural network [623]

- Author: Aliyaa Islam
- Description: simulates a lif neuron
- GitHub repository
- HDL project
- Mux address: 623
- Extra docs
- Clock: 0 Hz

How it works

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

How to test

Do something

External hardware

NA

Pinout

#	Input	Output	Bidirectional
0	Input current bit [0]	State varibale bit [0]	
1	Input current bit 1	State varibale bit 1	
2	Input current bit 2	State varibale bit 2	
3	Input current bit [3]	State varibale bit [3]	
4	Input current bit [4]	State varibale bit [4]	
5	Input current bit [5]	State varibale bit [5]	

#	Input	Output	Bidirectional
6	Input current bit [6]	State varibale bit [6]	
7	Input current bit [7]	State varibale bit [7]	Spike bit

Stochastic Integrator [640]

- Author: Ciecen Lestari, Chih-Kuan Ho, David Parent
- Description: Use stochastic computing to implement integration
- GitHub repository
- HDL project
- Mux address: 640
- Extra docs
- Clock: 50000000 Hz

How it works

The stochastic integrator uses Euler's definition of integration to make it happen in the stochastic domain. This integrator follows unipolar probability.

REFERENCES USED

General Stochastic Integrator Design:

1 S. Liu and J. Han, "Hardware ODE solvers using stochastic circuits," 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, USA, 2017, pp. 1-6, doi: 10.1145/3061639.3062258. keywords: {Radiation detectors;Stochastic processes;Hardware;Generators;Clocks;Energy consumption;Throughput;stochastic integrator;ordinary differential equation;stochastic computing},

LFSR Design in Stochastic Computing:

2 Jason H. Anderson, Yuko Hara-Azumi, and Shigeru Yamashita. 2016. Effect of LFSR seeding, scrambling and feedback polynomial on stochastic computing accuracy. In Proceedings of the 2016 Conference on Design, Automation & Test in Europe (DATE '16). EDA Consortium, San Jose, CA, USA, 1550–1555. https://dl.acm.org/doi/abs/10.5555/2971808.2972171

How to test

Set ui_in[0] with a constant high and ui_in1 with constant low to see the equations described.

External hardware

ADALM2000

Pinout

#	Input	Output	Bidirectional
0	serial_input_1	serial_output_seq_integrator_a	
1	serial_input_2	serial_output_seq_integrator_b	
2		serial_output_seq_integrator_c	
3		serial_output_system_integrator_a	
4		serial_output_system_integrator_b	
5		serial_output_test_integrator_a	
6		serial_output_test_integrator_b	
7		output_sn_bit_seq_integrator_c	

E2M0 x INT8 Systolic Array [642]

- Author: ReJ aka Renaldas Zioma
- Description: Systolic array for testing (Septenary and Quinary) 2.6 bits/param packed weights
- GitHub repository
- HDL project
- Mux address: 642
- Extra docs
- Clock: 48000000 Hz

How it works

Reduced precision matrix multiplication base on systolic array architecture. Left side matrix is compressed to 2.6 bits per element.

How to test

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

External hardware

External hardware

External processor (RP2040 for example) is necessary to feed weights and input data into the accelerator and fetch the results.

Pinout

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

#	Input	Output	Bidirectional
6	packed weights	result	(in) activations
7	packed weights MSB	result MSB	(in) activations MSB

VGA Nyan Cat [644]

- Author: Andy Sloane
- Description: Displays the classic nyan.cat animation
- GitHub repository
- HDL project
- Mux address: 644
- Extra docs
- Clock: 25175000 Hz

VGA nyan cat

Figure 37: nyancat preview

How it works Outputs nyancat on VGA with music!

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

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

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

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

External hardware TinyVGA Pmod for video on o[7:0].

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

Pinout

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

15 channels emission counter [646]

- Author: Coline Chehense, Dinko Oletic
- Description: Counts the number of pulses received on each of 15 input channel and returns periodically a serial output of these values.
- GitHub repository
- HDL project
- Mux address: 646
- Extra docs
- Clock: 12000000 Hz

How it works

This is an early work-in progress test implementation of a digital readout, part of a low-power mixed-signal multichannel sensor interface for acoustic emission detection. The sensor interface is developed to support a passive, micromechanically-implemented ultrasonic signal frequency decomposition MEMS device, based on an array of piezo-electric micro-resonators: https://ieeexplore.ieee.org/document/9139151.

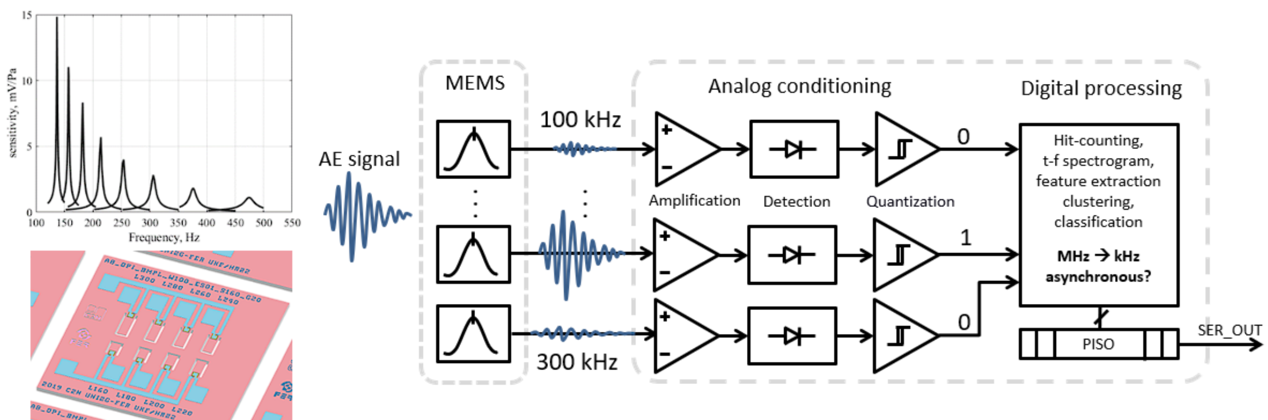


Figure 38: MEMS-based mixed-signal multichannel sensor interface for acoustic emission detection.

The digital readout part implemented here, tracks cumulative number of ultrasonic acoustic events/emissions occurences at each channel i.e. at the specific ultrasonic frequency, over a longer time interval. It is assumed that each acoustic emission event is represented by a short single digital input pulse. An analog conditioning circuit for pulse-shaping of input signals is not implemented here. The digital design consists of fifteen 12-bit channel-counters with overflow detection, a mm:ss real-time clock (RTC), a parallel-input-serial output (PISO) readout register, controlled by a readout state-machine. The counters store number of intermittently-occuring short digital input pulses, accumulated within the RTC's time-measurement interval 00:00 - 59:59,

at each of the four input channels. Periodically, after every RTC overflow (1 h with assumed 1 Hz RTC input clock signal), the state-machine performs sequential serial readout of the RTC time and all channels, and resets all channel counters. Additionally, readout and individual channel reset is initiated by overflow at any of individual input channel counter.

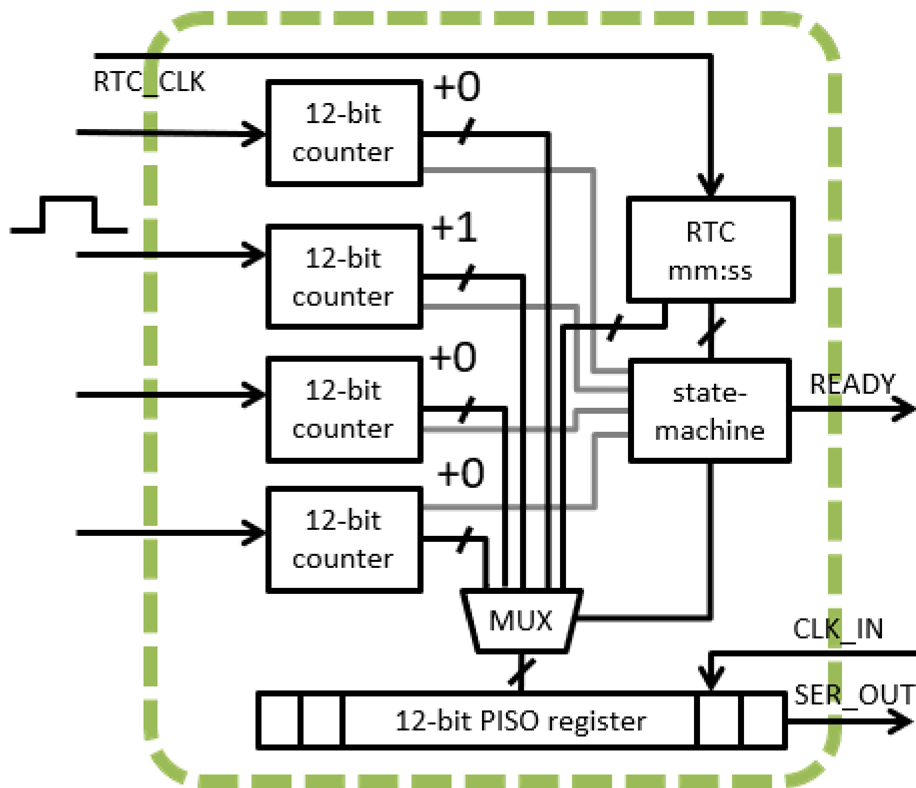


Figure 39: Digital multi-channel counter of ultrasonic acoustic events.

This design is part of research activities https://www.fer.unizg.hr/liss/aemems. The design is generally applicable for low-power wake-up sensor interfaces, acoustic event detection, non-destructive testing, particle-counters, or as a generic pulse-counting digital building block. This is the second TinyTapeout submission of the design. The first version was submitted to TT04, it featured 4 channels, and had timing issues during serial readout.

How to test

Input signals are short rising-edge digital pulses, connected to input pins "channel 1" to "channel 15" . Output data becomes ready for serial readout at the output pin "serial_out" when overflow is signalled via the output "ready" pin ovf_global. Output bits are serially clocked-out using the input pin "clk". Specifically, RTC overflow is signalled via output pin "ovf_RTC_out", and overflow at an individual channel via

the pin "ovf_ch_out". The rest of output pins are used for debugging of the state-machine's internal states.

External hardware

Logics analyzer will come handy.

Pinout

#	Input	Output	Bidirectional
0	RTC	serial_out	Channel 8
1	Channel 1	ovf_global	Channel 9
2	Channel 2	ovf_RTC	Channel 10
3	Channel 3	a0	Channel 11
4	Channel 4	a1	Channel 12
5	Channel 5	a2	Channel 13
6	Channel 6	a3	Channel 14
7	Channel 7	SL_out	Channel 15

Basic Oszilloscope and Signal Generator [648]

- Author: Pascal Gesell
- Description: Basic oscilloscope & signal generator on an ASIC
- GitHub repository
- HDL project
- Mux address: 648
- Extra docs
- Clock: 25000000 Hz

Authors: Pascal Gesell, Dr. Torsten Maehne, Dr. Theo Kluter

How it works

This is a basic oscilloscope design using the experimental VHDL template. It samples the input signal from channel 1 of an ADC Pmod (Digilent PmodAD1) and buffers the samples on an external FRAM. The captured signal is output on screen via a BlackMesa HDMI Pmod. Test signals are generated using Direct Digital Synthesis and are output on channel 1 of the DAC Pmod (Digilent PmodDA2). Four buttons and two switches allow to control the oscilloscope and choose the test signal to generate.

When the trigger button is pressed, a single-shot measurement is taken when the trigger criteria is met. The trigger criteria can be the vertical and horizontal position as well as the trigger level (positive edge or negative edge). The data is buffered onto the external FRAM, with the goal to contain 32k samples before the trigger event and 32k samples after the trigger event. After the data is collected, the data is displayed on the HDMI screen.

Since an external FRAM memory is used with no buffers on the chip, the displayed oscilloscope screen is actually rotated by 90° to the right. Thus only one sample needs to be read from the FRAM per output video line. A Python script is provided for convenience to read the video frames captured by an USB HDMI video grabber, rotate them by 90° to the left and display them on the screen.

The signal generator supports a few basic waveforms: sine, square, triangle and sawtooth. The frequency and amplitude can be adjusted using the buttons and switches. The signal generator is also used to test the trigger functionality and the display of the oscilloscope.

The scope settings are continuously output via UART at 9600 baud (8N1) on `uo_out(3)`.

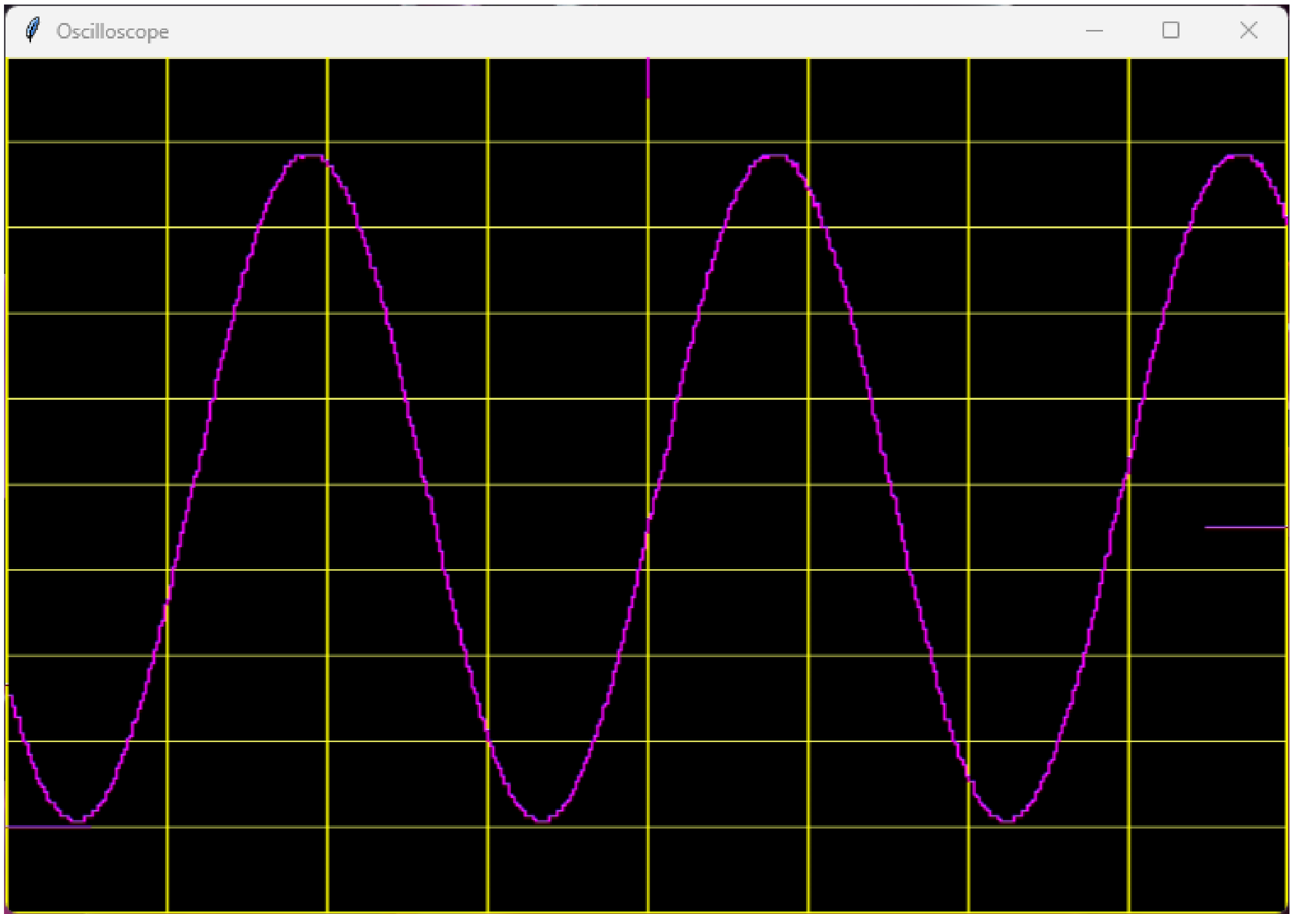


Figure 40: Image of Scope

How to test

Connect the various Pmods to the TinyTapeout 4+ demo board or FPGA board according to the pinout description in the info.yaml file. Connect the output of the DAC to the input of the ADC and connect the HDMI Pmod to a screen or HDMI capture card. Run the trigger to capture a single-shot measurement and display the data on the screen.

External hardware

To test and use this project, you will need the following hardware:

- 1 BlackMesaLabs 3-bit HDMI Pmod : A 3-bit HDMI Pmod
- 1 Digilent PmodAD1 : A 12-bit ADC Pmod
- 1 Digilent PmodDA2 : A 12-bit DAC Pmod
- 1 FM25W256G : 32k x 8 FRAM Pmod
- 1 Digilent PmodBTN : A 4 Buttons Pmod
- 1 Digilent PmodSWT : A 4 Switches Pmod, of which only 2 are used
- Optionally, an HDMI capture card to display the HDMI output on a computer screen

Attention: The above Pmods cannot be directly connected to the TinyTapeout 4+ demo board! The Pmods' pins need to be individually connected to the right Pmod pins of the TinyTapeout 4+ demo board, as documented in the IO section.

FPGA Implementation

The design has been implemented and tested on a Sipeed Tang Nano 9k FPGA board using my own base PCB to have enough available Pmods to test the design.

Acknowledgements

This work was realized under the supervision of Dr. Torsten Maehne and Dr. Theo Kluter as part of my project work in the 5th term of my Bachelor studies of electrical engineering and information technology at Berner Fachhochschule (BFH), Biel/Bienne, Switzerland.

Pinout

#	Input	Output	Bidirectional
0	FRAM MISO	ADC CS	HDMI Pmod Green
1	Button 1	DAC MOSI	HDMI Pmod Clock
2	Button 3	ADC SCLK	HDMI Pmod HSYNC
3	Switch 1	FRAM SCLK	UART_TX Settings Info (9600bps, 8N1)
4	ADC MISO	DAC CS	HDMI Pmod Red
5	Button 2	DAC SCLK	HDMI Pmod Blue
6	Button 4	FRAM CS	HDMI Pmod DE
7	Switch 2	FRAM MOSI	HDMI Pmod VSYNC

T3 (Tiny Ternary Tapeout) CSA [650]

- Author: Arnav Sacheti & Jack Adiletta
- Description: Ternary Matmul Processor using CSA
- GitHub repository
- HDL project
- Mux address: 650
- Extra docs
- Clock: 50000000 Hz

Tiny Ternary Tapeout Project Documentation

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

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

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

Loading the Weights (`tt_um_load.v`):

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

Matrix Multiplication (`tt_um_mult.v`):

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

Pipelined Architecture:

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

Outputting Results:

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

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

Step 1: Define a Ternary Array

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

Let's create a sample 4x2 ternary array:

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

Step 2: Define the Input Vector

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

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

Step 3: Compute the Output without Matrix Multiplication

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

For each column in the ternary array:

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

Step 4: Calculate Each Column's Output

Let's compute each column separately:

- **Column 1 Calculation:**

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

  Sum of Column 1: ($2 + 1 + 0 + 0 = 3$)

- **Column 2 Calculation:**

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

  Sum of Column 2: (0 - 1 - 3 + 0 = -4)

Final Output Vector

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

$$\text{Output} = \begin{bmatrix} 3 & -4 \end{bmatrix}$$

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

1. **Setup**:

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

Pinout

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

CORA-16 [652]

- Author: Andrew Dona-Couch
- Description: Simply 16-bit CPU
- GitHub repository
- HDL project
- Mux address: 652
- Extra docs
- Clock: 0 Hz

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

How it works

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

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

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

How to test

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

External hardware

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

Instruction set

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

Impact on the status flags is documented as:

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

One-byte instructions

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

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

Two-byte instructions

Name	Bit Pattern	Description	Status
Load	1000 0sss vvvv vvvv	Load a value into the accumulator	---- ----
Store	1001 0sss vvvv vvvv	Store a value to memory	---- ----
Add	1000 1sss vvvv vvvv	Add a value to the accumulator	---- -###
Sub	1001 1sss vvvv vvvv	Subtract a value from the accumulator	---- -###
And	1010 0sss vvvv vvvv	Bitwise and a value with the accumulator	---- --##
Or	1010 1sss vvvv vvvv	Bitwise or a value with the accumulator	---- --##
Xor	1011 0sss vvvv vvvv	Bitwise exclusive or a value with the accumulator	---- --##

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

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

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

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

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

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

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

Three-byte instructions

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

Pinout

#	Input	Output	Bidirectional
0	Data In 0	Data Out 0	SPI MOSI
1	Data In 1	Data Out 1	SPI CS
2	Data In 2	Data Out 2	SPI CLK
3	Data In 3	Data Out 3	SPI MISO
4	Data In 4	Data Out 4	Step
5	Data In 5	Data Out 5	Busy
6	Data In 6	Data Out 6	Halt
7	Data In 7	Data Out 7	Trap

ITS-RISCV [654]

- Author: Bambang T. Wibowo, Chazim Fikri A., Hernanda A. P., M. Hafidzh, Figo A. M., and Faiz S. K.
- Description: ITS RISC V based on the underserved TinyTapeout 07.
- GitHub repository
- HDL project
- Mux address: 654
- Extra docs
- Clock: 20000000 Hz

How it works

When the system boots up, it will start accessing the SPI bus to set up a connected SPI Flash memory in XIP mode and start executing instructions from there. The GPIO can be used to output data, e.g. as a bitbanged UART.

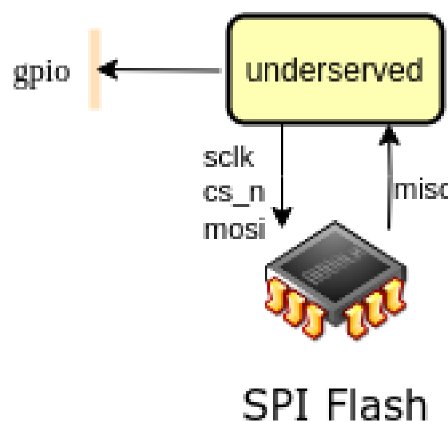


Figure 41: Environment

How to test

The testbench contains a model of an SPI Flash. A program in Verilog Hex format can be preloaded into the Flash model.

Underserved can easiest be run locally using FuseSoC.

Install FuseSoC

```
pip install fusesoc
```

Create and enter a new workspace

```
mkdir workspace && cd workspace
```

Register underserved as a library in the workspace

```
fusesoc library add underserved /path/to/prince
```

...if repo is available locally or... ...to get the upstream repo

```
fusesoc library add underserved https://github.com/olofk/underserved
```

Show available cores in workspace (probally just underserved for now if you haven't added other libraries)

```
fusesoc core list
```

Show info about underserved

```
fusesoc core show underserved
```

Run linting (static code checks) using Verilator

```
fusesoc run --target=lint underserved
```

Run underserved testbench

```
fusesoc run --target=sim underserved
```

Run with modelsim instead of default tool (icarus)

```
fusesoc run --target=sim underserved --tool=modelsim
```

External hardware

Expects a compatible SPI Flash. The XIP controller was stolen from PicoSoC which also contains some info about compatible SPI Flash components.

Pinout

#	Input	Output	Bidirectional
0		gpio0	
1		gpio1	
2		gpio2	
3		gpio3	
4		gpio4	
5		sclk	
6		cs_n	
7		mosi	miso

16 Bit Izhikevich Neuron [672]

- Author: Noah Williams
- Description: Izhikevich neuron model with 16 bit arithmetic.
- GitHub repository
- HDL project
- Mux address: 672
- Extra docs
- Clock: 0 Hz

How it works

```
Izhikevich model

The Izhikevich model is a simple spiking neuron model that builds on the
    dynamics of the simplistic leaky integrate-and-fire model, adding
    complexity of the Hodgkin-Huxley model with minimal computational
    cost.

    The model is described by the following system:

        v' = 0.04*v^2 + 5*v + 140 - u + I
        u' = a*(b*v - u)
        if v >= 30 then {v = c; u = u + d}

        where:
            a, b, c, d = dimensionless constants

                Regular Spiking (RS) Excitatory Neuron:
                    a = 0.02, b = 0.2, c = -65, d = 8


            v = membrane potential
            u = membrane recovery (Na and K, neg feedback to v)
            a = time scale of the recovery variable u (small = slow recov
            b = sensitivity of the recovery variable u to v
                    Larger values increase sensitivity and lead to more
                    spiking behavior. b<a(b>a) is saddle-node
            c = after spike reset value of v
                    caused by fast K+ channels
            d = after spike reset value of u
```

```
                   caused by slow Na+ & K+ channels
           I = input current

    The constants for the model differential equation v' are experimental
        determined by fitting the model to the desired neuron behavior. I
        the original paper (from which the equations are taken), the mode
        was fit to experimental data from Regular Spiking of a rat cortic
        neuron.


References:
https://www.izhikevich.org/publications/spikes.pdf
```

How to test

```
To test the model, use the supplied test-bench. The test-bench will run t
    three different scenarios. The first case is the reset test case, whi
    ensure that the model resets properly given a reset condition (res_n
    The next test case checks to make sure that the model doesn't spike w
    the input current is below threshold. The spike value for each of the
    included non-spike test cases should be 0. The final test case is the
    test that ensures the model spikes when the input is above the thresh
    includes a test for the maximum current to test overflow conditions.
    is checked with an assert statement.
```

External hardware

N/A at the moment :)

Pinout

#	Input	Output	Bidirectional
0	Input current bit [0]	State variable bit [0]	
1	Input current bit 1	State variable bit 1	
2	Input current bit 2	State variable bit 2	

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

Giant Ring Oscillator (3853 inverters) [673]

- Author: Uri Shaked
- Description: Configurable ring oscillator with up to 3853 inverters
- GitHub repository
- HDL project
- Mux address: 673
- Extra docs
- Clock: 0 Hz

How it works

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

Pin	Chain length
uo[0]	1
uo1	3
uo2	5
uo[3]	7
uo[4]	11
uo[5]	21
uo[6]	51
uo[7]	101
uio[0]	201
uio1	501
uio2	1001
uio[3]	2001
uio[4]	3001
uio[5]	3853

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

How to test

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

External hardware

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

Pinout

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

dff_mem [674]

- Author: dmrudait
- Description: 16 byte RAM built out of DFFs
- GitHub repository
- HDL project
- Mux address: 674
- Extra docs
- Clock: 0 Hz

How it works

This project implements a 16 Byte memory module (it consists of 16 memory locations that store 1 byte each). The memory allows for both read and write operations, controlled by input signals. The module requires a 4-bit address input, control signals lr_n and ce_n, a clock, 8-bit data input (for writes), and a reset signal.

Signals

- ui_in[7:0]: Dedicated input line for all control signals
- ce_n (Active Low): Chip enable signal for reading data.
- lr_n (Active Load): Load/write signal, enabling writing to memory.
- uio_in[7:0]: Bidirectional IO line for input (used an the input line for data)
- uio_out[7:0]: Bidirectional IO line for output.
- uio_oe (Active High): Used to set the bidirectional IO line to an input to be able to input data
- ena (Active High): Tiny Tapeout signal for enabling the module
- clk: global clock. Operations happen on the positive edge
- rst_n (Active low): Resets all contents in RAM to NULL.
- uo_out[7:0]: Dedicated output line (outputs ram contents when ce_n is low (active).

Addressing: The memory is 4-bit addressable, where the address specifies which register (out of 16) is being accessed for reading or writing.

Write operation: A byte of data is written to specific register in RAM, where the location is determined by the address. Requires write enable `lr_n` signal as active (low) and the clock edge to occur.

Read operation: Data can be read from a specific register in RAM determined by the input address. Requires chip enable `ce_n` signal as active (low). The data is output on the uo_out ports, and it is updated asynchronously (independant of the clock edge).

Output: Data is presented on the uo_out line when the chip is enabled for reading, and high-impedance (Z) otherwise.

How to test

To test, set the address and corresponding inputs to desired values. Clear `lr_n` for a write operation and `ce_n` for a read operation. Then pulse the clock to run signals.

The CocoTB testbenches located in the *test.py* file, test various scenarios for the module. First, it tests a write operation to each address in the module followed by a read operation at each address, to ensure correct behaviour. The script then sets `ui_in`, `lr_n` high and clears `ce_n` to setup for a Read with RAM output enabled. It then iterates over and reads from each address, comparing the recevied value (`uo_out`), to the expected byte from that address. If there are any mismatches, an assertion error is raised, specifying the faulty address and value.

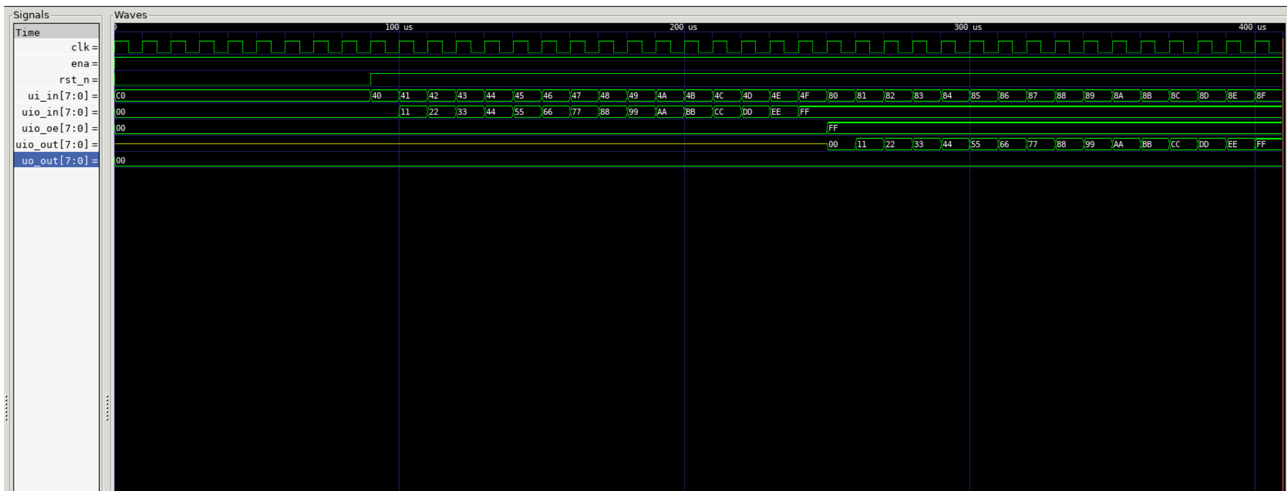


Figure 42: image

Figure 1: Gate level Test

Figure 2: Ideal Test

External hardware

This RAM module is intended to be integrated into an 8-bit processor. However, it is being submitted to TT as an indiviudal tile for testing. An external MAR would thus be required to program RAM and subseqently read memory. The MAR would act as a programmer according to the avove described specifications.

Figure 43: image

Pinout

#	Input	Output	Bidirectional
0	addr[0]	out[0]	in[0]
1	addr1	out1	in1
2	addr2	out2	in2
3	addr[3]	out[3]	in[3]
4	addr[4]	out[4]	in[4]
5		out[5]	in[5]
6	lr_n	out[6]	in[6]
7	ce_n	out[7]	in[7]

Lab B Group 10 Array Multiplier [675]

- Author: Abhinav and Annay
- Description: 4x4 Array Multiplier
- GitHub repository
- HDL project
- Mux address: 675
- Extra docs
- Clock: 0 Hz

How it works

An array multiplier is a combinational digital circuit used to multiply two binary numbers. It is structured similarly to the multiplication process, where partial products are generated and added to produce the final product. The array multiplier makes use of full adders, arranged in a systematic array to handle the binary addition of partial products. This process is shown in the image 4x4_array_multiplier.png located in the docs folder as well as shown in this preview.

How to test

To ensure new hardware is working correctly, start with a visual inspection for any physical damage and verify that all components are properly connected. Power up the hardware to confirm it initializes without errors, and check that your system detects it, typically through device manager or similar tools. Run any manufacturer-provided diagnostics or benchmarking tools to verify core functions and performance meet expectations. Test specific functionalities (such as I/O, communication, or display output), and update firmware or drivers if necessary. Conduct stress tests to check for stability under load, and perform any necessary compatibility and environmental checks to ensure the hardware is fully operational in its intended use case.

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	m[0]	p[0]	
1	m1	p1	
2	m2	p2	
3	m[3]	p[3]	
4	q[0]	p[4]	
5	q1	p[5]	
6	q2	p[6]	
7	q[3]	p[7]	

Verilog ring oscillator V2 [676]

- Author: algofoogle (Anton Maurovic)
- Description: Multiple simple ring oscillators by instantiating sky130 inv_2 inverter rings
- GitHub repository
- HDL project
- Mux address: 676
- Extra docs
- Clock: 0 Hz

What is this?

Everyone has done a ring oscillator using inverter cells. Now it's my turn!

I already submitted tt09-ring-osc on TT09 and rather than muck that up with extra stuff I decided to submit this alternate version which features:

- 4 simple independent rings instead of 1, hoping to run at different speeds:
  - `ring_125`: 125 inverters, *maybe* 112MHz out? Could be too fast for IO.
  - `ring_251`: 251 inverters, hopefully good for ~56MHz.
  - `ring_501`: 501 inverters, ~28MHz.
  - `ring_1001`: 1001 inverters, ~14MHz.
- Some other PWM experiments on faster ring oscillators.

Approximate frequences are estimated on the assumption that each inverter introduces a delay of ~70ps.

These use verilog to instantiate the rings of (an odd number of) `sky130_fd_sc_hd__inv_2` cells.

Pinout

#	Input	Output	Bidirectional
0	pwm2_in[0]	ring_125	dummy
1	pwm2_in1	ring_251	pwm3a_out
2	pwm3_in[0]	ring_501	
3	pwm3_in1	ring_1001	
4		c0_3	
5	pwm3a_in[0]	c1_3	

#	Input	Output	Bidirectional
6	pwm3a_in1	c2_5	pwm2_out
7	pwm3a_in2	c3_5	pwm3_out

TwoChannelSquareWaveGenerator [677]

- Author: Sam Kho
- Description: Like having two apple2-style speakers
- GitHub repository
- HDL project
- Mux address: 677
- Extra docs
- Clock: 256000 Hz

How it works

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

How to test

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

External hardware

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

Pinout

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

#	Input	Output	Bidirectional
6	TA6		TB6
7	TA7		TB7

Basic model for Systollic array implementation of LIF [678]

- Author: Sulaiman Islam
- Description: A model for systolic array implementation of LIF neurons. Hazard cases have been taken into account such as overstimulation of LIF nuerons with bypass cases.
- GitHub repository
- HDL project
- Mux address: 678
- Extra docs
- Clock: 0 Hz

How it works

The model represents how a generative implementation of SNN training can be implemented in verilog. Hazards that were predicted were that of LIF overstimulation due to excessive accumilation of the multiply accumilate MAC operations. In order to prevent this I implemented bypass conditions that made regulated the LIF inputs to choose between 0 and the output of the MAC operatios. The LIF would only take in the value of the MAC operation when the threeshold for firing was reached by the output of the MAC. One clock cylce later the MAC Accumilation would be reset. This was a weight stationary implemenation that had fixed constant weights on each of the MACS. Several Blocks of these weight stationary MACS could be implemented with their respective LIFS in theory, however due to size restrictions there is only one small block in the Top module. The user input ui_in is used to drive the MAC inputs. The uo_out individual bits were used to drive the spike to inidcate that the bypass had occured and that the LIF had spiked.

How to test

Using ui_in to vary X and checking uo_out for expected behavior based on the MAC operations.

External hardware

N/A

Pinout

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

RGB Mixer demo [679]

- Author: Matt Venn
- Description: Zero to ASIC demo project
- GitHub repository
- HDL project
- Mux address: 679
- Extra docs
- Clock: 10000000 Hz

How it works

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

How to test

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

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

External hardware

Use 3 digital encoders attached to the first 6 inputs.

Pinout

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

mips.sv [680]

- Author: Eric Smith
- Description: The MIPs core example from Weste & Harris
- GitHub repository
- HDL project
- Mux address: 680
- Extra docs
- Clock: 1000000 Hz

How it works

This is the example processor core from Weste & Harris written by Max Yi (byyi@hmc.edu) and David_Harris@hmc.edu 12/9/03 https://pages.hmc.edu/harris/cmosvlsi/4e/

How to test

Carefully

External hardware

Need a program store.

More to come later.

Pinout

#	Input	Output	Bidirectional
0	memdata[0]	writedata[0]	adr[0]
1	memdata1	writedata1	adr1
2	memdata2	writedata2	adr2
3	memdata[3]	writedata[3]	adr[3]
4	memdata[4]	writedata[4]	adr[4]
5	memdata[5]	writedata[5]	adr[5]
6	memdata[6]	writedata[6]	memread
7	memdata[7]	writedata[7]	memwrite

VGA clock [681]

- Author: Matt Venn
- Description: Shows the time on a VGA screen
- GitHub repository
- HDL project
- Mux address: 681
- Extra docs
- Clock: 31500000 Hz

How it works

Races the beam! Font is pre generated and loaded into registers. 6 bit colour keeps register count low.

Every minute the colours cycle.

How to test

Hook up a VGA monitor to the outputs and provide a clock at 31.5 MHz.

Adjust time with the inputs[2:0], and choose the type of VGA PMOD with the input[3].

External hardware

VGA PMOD - you can use one of these VGA PMODs:

- https://github.com/mole99/tiny-vga
- https://github.com/TinyTapeout/tt-vga-clock-pmod

Set input[3] low to use tiny-vga and high to use vga-clock

#	Input	Output	Bidirectional

Pinout

#	Input	Output	Bidirectional
0	adjust hours	hsync / R1	
1	adjust minutes	vsync / G1	
2	adjust seconds	B0 / B1	
3	PMOD type select	B1 / VS	
4		G0 / R0	
5		G1 / G0	
6		R0 / B0	
7		R1 / HS	

gta6 [682]

- Author: henry
- Description: gta6
- GitHub repository
- Wokwi project
- Mux address: 682
- Extra docs
- Clock: 0 Hz

How it works

Guess a pin that lights the LED

Explain how your project works

How to test

Pick a pin and see if it lights up.

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any
None

Pinout

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

8-bit CBILBO [683]

- Author: Devesh Bhaskaran, Om Shivshankar Shigarkanti, Garima Bajpayi
- Description: Concurrent Built–In Logic BlockIn Logic Block Observer for Memory Test
- GitHub repository
- HDL project
- Mux address: 683
- Extra docs
- Clock: 40000000 Hz

How it works

In this Verilog code, we implement a BILBO (Built-In Logic Block Observer) shift register with multiple stages, using a combination of logic gates (AND, XOR), D flip-flops (DFF), and multiplexers (MUX) for feedback and shifting operations. We include input and output paths for Tiny Tapeout and support asynchronous reset and clocked logic. The modules interact to store and shift data, providing internal feedback and driving outputs for observation.

How to test

To test this project, we would create a testbench that provides stimulus for the inputs (`ui_in`, `uio_in`, `clk`, `rst_n`) and checks the outputs (`uo_out`, `uio_out`, `uio_oe`). We would simulate the shifting and feedback behavior of the BILBO shift register, verifying that the data is properly shifted and the feedback logic functions correctly across all stages of the register.

External hardware

No external hardware required for this project.

Pinout

#	Input	Output	Bidirectional
0	ui_in[0]	uo_out[0]	uio[0]
1	ui_in1	uo_out1	uio1
2	ui_in2	uo_out2	uio2

#	Input	Output	Bidirectional
3	ui_in[3]	uo_out[3]	uio[3]
4	ui_in[4]	uo_out[4]	uio[4]
5	ui_in[5]	uo_out[5]	uio[5]
6	ui_in[6]	uo_out[6]	uio[6]
7	ui_in[7]	uo_out[7]	uio[7]

Name Speller [684]

- Author: Conor Van Bibber
- Description: it spells a name using a 3 bit clock divider
- GitHub repository
- Wokwi project
- Mux address: 684
- Extra docs
- Clock: 2 Hz

How it works

It has a 3 bit counter that counts up to 7 and then resets. The counter is displayed on a 7-segment display.

How to test

To test, connect the 7-segment display to the ASIC and run the program. The counter should be displayed on the display. The counter should reset to 0 after pressing the reset button.

External hardware

7-segment display, reset button for nreset

Pinout

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

Michaels Tiny Tapeout ALU [685]

- Author: Michael McCulloch
- Description: Should work as a 2 6 bit input ALU, which then can choose from the RISCV ALU opcodes to select the operation which will be outputed in 8bit
- GitHub repository
- HDL project
- Mux address: 685
- Extra docs
- Clock: 0 Hz

How it works

In short the first 6 bits of the bidirection IO is A, then bits 7 and 7 of the bidirectional and bits 0 to 3 of the single way input are B and the last 4 bits are the ALU opcode (based on RISCV) Values get outputted in 8bit from the single way output bus

How to test

Setting the inputs and testing the outputs for certain opcodes

External hardware

None at the moment… Could attach LEDs for testing

Pinout

#	Input	Output	Bidirectional
0	Bit 2 of ALU Input B	Bit 0 of ALU Output	Bit 0 of ALU Input A
1	Bit 3 of ALU Input B	Bit 1 of ALU Output	Bit 1 of ALU Input A
2	Bit 4 of ALU Input B	Bit 2 of ALU Output	Bit 2 of ALU Input A
3	Bit 5 of ALU Input B	Bit 3 of ALU Output	Bit 3 of ALU Input A
4	Bit 0 of ALU OpCode	Bit 4 of ALU Output	Bit 4 of ALU Input A

#	Input	Output	Bidirectional
5	Bit 1 of ALU Opcode	Bit 5 of ALU Output	Bit 5 of ALU Input A
6	Bit 2 of ALU OpCode	Bit 6 of ALU Output	Bit 0 of ALU Input B
7	Bit 3 of ALU OpCode	Bit 7 of ALU Output	Bit 1 of ALU Input B

2-bit Full Adder [686]

- Author: Shreya
- Description: Adds two numbers
- GitHub repository
- Wokwi project
- Mux address: 686
- Extra docs
- Clock: 0 Hz

How it works

Adds two numbers

How to test

Apply two input signals.

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			
5			
6			
7			

ovl abc chip [687]

- Author: oliver lazaras
- Description: displays a with in0 and . with in1
- GitHub repository
- Wokwi project
- Mux address: 687
- Extra docs
- Clock: 0 Hz

How it works

basically the idea is to have each input display corresponding letter output ex: in0 on –> A , in1 on –> B, in2 on or in0 & in1 –> C etc…

How to test

switch IN 0 and 1 and see what happens

External hardware

7 seg display

Pinout

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

Simon's Caterpillar [704]

- Author: htfab
- Description: Port of Caterpillar Logic to Simon Says PMOD
- GitHub repository
- HDL project
- Mux address: 704
- Extra docs
- Clock: 50000 Hz

How it works

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

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

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

How to test

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

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

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

(The empty sequence is neither valid nor invalid.)

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

- long-press red: clear buffer
- long-press yellow: erase last color from buffer (“backspace”)
- long-press blue: show buffer contents (as a series of led flashes)
- long-press green: activate challenge mode
- short-press green & yellow: show a random valid sequence (and load into buffer)
- short-press red & blue: show a random invalid sequence (and load into buffer)
- short-press blue & yellow: switch to next level
- short-press red & green: switch to previous level
- short-press green & blue: toggle sound

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

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

Other keys and combinations:

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

External hardware

Simon Says PMOD

Pinout

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

tt6502 [706]

- Author: Anders
- Description: tt6502
- GitHub repository
- HDL project
- Mux address: 706
- Extra docs
- Clock: 0 Hz

How it works

This thing is to test how big a 6502 is.

How to test

This thing is to test how big a 6502 is.

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	gpio input 0	gpio output 0	cs_n
1	gpio input 1	gpio output 1	mosi
2	gpio input 2	gpio output 2	miso
3	gpio input 3	gpio output 3	sclk
4	gpio input 4	gpio output 4	gpio bidir 4
5	gpio input 5	gpio output 5	gpio bidir 5
6	irq interrupt	gpio output 6	gpio bidir 6
7	nmi interrupt	sync	gpio bidir 7

Oscillating Bones [708]

- Author: Uri Shaked
- Description: A stylish ring oscillator built from SkullFET transistors
- GitHub repository
- HDL project
- Mux address: 708
- Extra docs
- Clock: 0 Hz

How it works

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

How to test

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

Simulation results

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

The outputs are shifted by 2 volts to make them easier to see in the graph. "uo_out[0]" is the main output of the oscillator, and "uo_out1", "uo_out2", and "uo_out[3]" are the divided outputs.

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

#	Input	Output	Bidirectional

Pinout

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

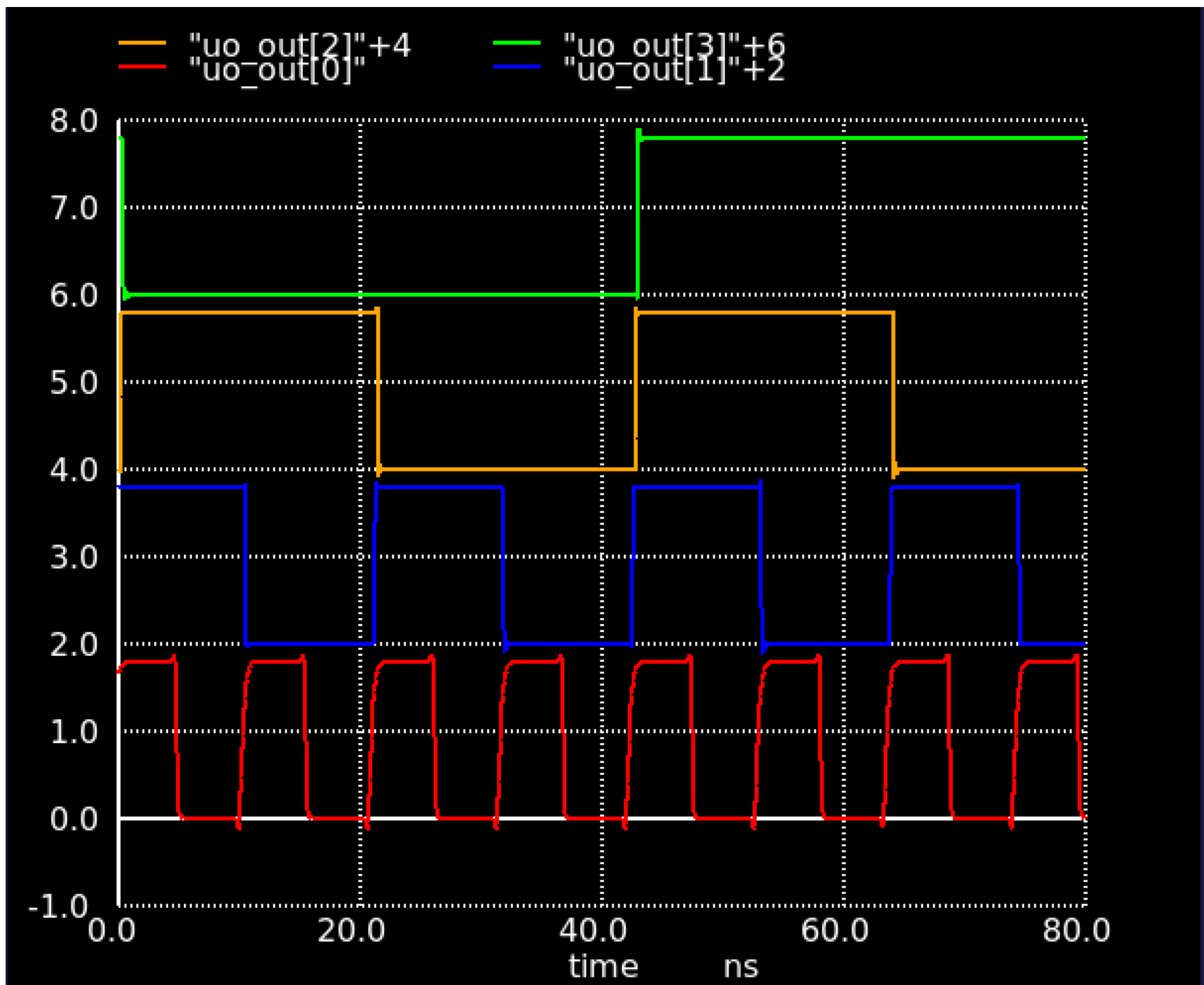


Figure 44: Simulation results

SoCET UART with FIFO buffers [710]

- Author: Miguel Isrrael Teran, Yashashwini Singh, Michael Li, Rafael Monteiro Martins Pinheiro, Vito Gamberini
- Description: General-purpose UART with hardware control flow and FIFO buffer capacity developed by Purdue's SoCET team
- GitHub repository
- HDL project
- Mux address: 710
- Extra docs
- Clock: 50000000 Hz

How it works

This project is a UART module that includes FIFO buffers to store bytes of data. The module has standard UART input and output pins, such as `rx`, `tx`, `cts`, and `rts`. Additional inputs allow the configuration of operation mode(s): **Idle**, **RX**, **TX** or **Buffer Clear**, and desired baud rate is user-configurable through the `Control` pins. Bidirectional data pins are used to send and receive test data. Additional outputs include an `error` flag, as well as the TX FIFO's `full` flag, and the RX FIFO's `empty` flag.

How to test

Steps for testing are the following:

1) Supply a 50 MHz clock signal to the UART
2) Configure the control settings: `Control[1:0]` (`ui[3:2]`) are used to choose between preloaded baud rates. Here are the following baud rate configurations based on the values of `Control[1:0]`:

Value of `ui[3:2]`	Baud rate (bits/s)
0	9600
1	19200
2	38400
3	115200

`Control[3:2]` (`ui[5:4]`) set the UART's mode of operation for the current byte of data being processed. Each non-idle control signal must be preceded with an idle

signal to perform a valid transaction/manage the FIFO buffers. Here are the following UART mode configurations determined by the values of `Control[3:2]`:

Value of `ui[5:4]`	Mode Configuration
0	IDLE
1	TX
2	RX
3	BUFFER CLEAR

3) If you have 2 PCBs with the TT09 ASIC, you can load the same UART design in both and cross-connect their `rx`, `tx`, `cts`, and `rts` pins as shown in the image below. Then, you can use one of them as a Transmitter and the other as a Receiver. If you only have 1 PCB, you can test the UART with the **FT232RL Mini USB to TTL Serial Adapter Module** (see next section).

External hardware

We suggest using **switches** for the `Control` pins (this way you can keep the mode of operation stable). Image below shows the **FT232RL** module that can be used for testing and connecting serially to a computer's USB port. More information on the product can be found here.

Pinout

#	Input	Output	Bidirectional
0	rx	tx	data[0]
1	cts	rts	data1
2	Control[0]	err	data2
3	Control1	tx_buffer_full	data[3]
4	Control2	rx_buffer_empty	data[4]
5	Control[3]		data[5]
6			data[6]
7			data[7]

VGA Drop (audio/visual demo) [712]

- Author: ReJ aka Renaldas Zioma, eriQue aka Erik Hemming, Matthias Kampa
- Description: Tiny 8 part Megademo! TBL^Nesnausk^SonikClique
- GitHub repository
- HDL project
- Mux address: 712
- Extra docs
- Clock: 25200000 Hz

How it works

VGA signal generator

How to test

We are learning how VGA and Sky130 works here

External hardware

VGA PMOD

Pinout

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

Warp [714]

- Author: sylefeb
- Description: Demo on TinyTapeout? Let's do something!
- GitHub repository
- HDL project
- Mux address: 714
- Extra docs
- Clock: 25000000 Hz

Warp

> Please make sure to watch the demo for a few minutes as various effects play out before it loops. At start it waits for a few seconds to ensure VGA sync is achieved.

How it works

> *But does it work?*

Preface This demo is written in Silice, my HDL. Here is the actual source. Silice now fully support TinyTapeout as a build target.

Graphics The core effect is a classical tunnel effect ; however this is normally done with a "huge" pre-computed table having one entry per-pixel. So I thought it'd be challenging and fun to do it while racing the beam! Plus, I really like this effect.

There are several tricks at play: a shallow CORDIC pipeline to compute an *atan* and *length*, and a few precomputed *1/x* distances to interpolate between – these form keypoint rings along the tunnel. All the effects are then obtained by combining multiple layers in various ways (like a *tunnel effect processor* which registers can be configured for various effects).

The demo uses a lot of dithering (ordered Bayer dithering) given the output is RGB 2-2-2. All computations are grayscale and the RGB lense effect is obtained by delaying the grayscale values using the tunnel distance in R and B.

I also tried to make the logo interesting by deviating from a classical pixelated look. It is composed of tiles, either full or triangular, with a comparator and a bit of logic to do all four possible triangles.

The tunnel viewpoint change is obtained simply by shifting the tunnel center. I was surprised that a simple translation gives such a convincing effect (almost as if the viewpoint was rotating).

The 'blue-orange' tunnel effect is obtained through temporal dithering, one frame being the standard tunnel, the other the rotated tunnel. This gets combined with the RGB lense distortion, achieving the final look.

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

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

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

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

External hardware

- VGA PMOD
- Audio PMOD

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

Pinout

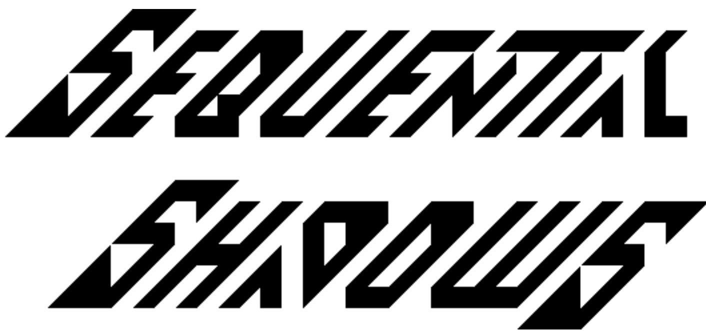
#	Input	Output	Bidirectional
0		R1	
1		G1	
2		B1	
3		VS	
4		R0	
5		G0	
6		B0	
7		HS	Audio

Sequential Shadows [TT08 demo competition] [716]

- Author: Toivo Henningsson
- Description: My contribution to the TT08 demo competition
- GitHub repository
- HDL project
- Mux address: 716
- Extra docs
- Clock: 50400000 Hz

Intro

Curly / Medieval presents



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

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

How it works

The demo code contains a few different parts:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Wave forms used:

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

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

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

The chords use different multipliers on the chord phase:

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

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

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

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

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

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

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

How to test

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

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

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

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

External hardware

This project needs

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

Pinout

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

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

achasen workshop validation [718]

- Author: adam chasen
- Description: validation description
- GitHub repository
- Wokwi project
- Mux address: 718
- Extra docs
- Clock: 0 Hz

How it works

single gate

How to test

take a look at the wokwi

External hardware

LED display and input switches

Pinout

#	Input	Output	Bidirectional
0	IN0	OUT0	
1	IN1		
2			
3			
4			
5			
6			
7			

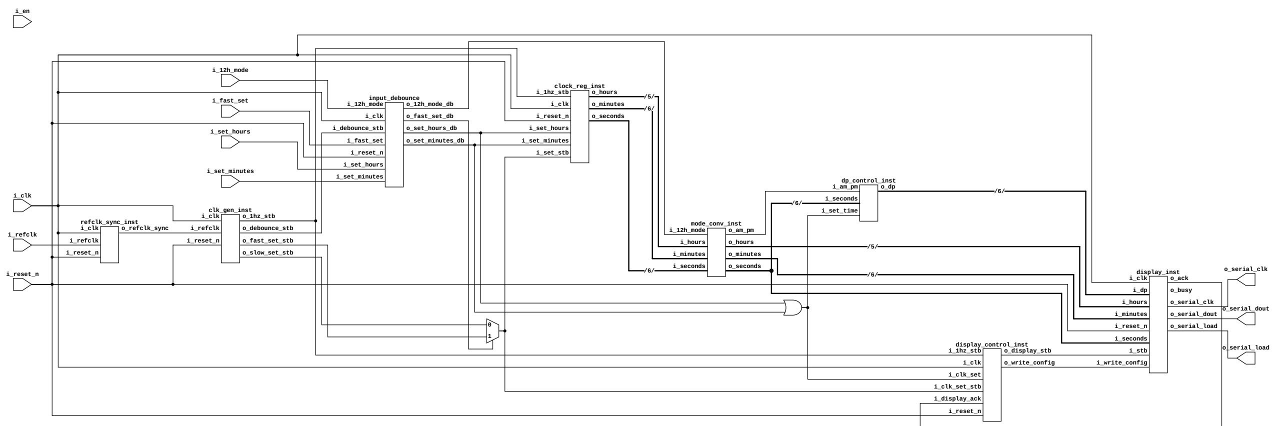
7-Segment Digital Desk Clock [736]

- Author: Samuel Ellicott
- Description: 7-Segment Desk Clock
- GitHub repository
- HDL project
- Mux address: 736
- Extra docs
- Clock: 50000000 Hz

How it works

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

A block diagram of the system is shown below.



How to test

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

External hardware

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

Pinout

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

TinySnake [737]

- Author: Ken Pettit
- Description: A snake slithers around the 7-Seg display
- GitHub repository
- Wokwi project
- Mux address: 737
- Extra docs
- Clock: 10000 Hz

How it works

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

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

How to test

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

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

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	speed[0]	seg_a	
1	speed1	seg_b	
2	speed2	seg_c	
3	speed[3]	seg_d	
4	speed[4]	seg_e	
5	speed[5]	seg_f	
6	speed[6]	seg_g	
7	speed[7]		

Basic Perceptron + ReLU [738]

- Author: UDXS
- Description: Basic Perceptron + ReLU Layer
- GitHub repository
- HDL project
- Mux address: 738
- Extra docs
- Clock: 0 Hz

How it works

It connects a small single-cycle multiply-accumulation unit to a ReLU output.

How to test

Reset and then, for every following cycle, provide pairs of signed 4-bit numbers representing the weight-input pair for a given model layer invocation. The output will change cycle-to-cycle. Sample it while providing your last inputs and then reset to attempt another invocation.

Pinout

#	Input	Output	Bidirectional
0	Weight[0]	ReLU[0]	ReLU[8]
1	Weight1	ReLU1	ReLU[9]
2	Weight2	ReLU2	ReLU[10]
3	Weight[3]	ReLU[3]	ReLU[11]
4	Input[0]	ReLU[4]	ReLU[12]
5	Input1	ReLU[5]	ReLU[13]
6	Input2	ReLU[6]	ReLU[14]
7	Input[3]	ReLU[7]	ReLU[15]

Classic 8-bit era Programmable Sound Generator SN76489 [739]

- Author: ReJ aka Renaldas Zioma
- Description: The SN76489 Digital Complex Sound Generator (DCSG) is a programmable sound generator chip from Texas Instruments.
- GitHub repository
- HDL project
- Mux address: 739
- Extra docs
- Clock: 4000000 Hz

How it works

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

The goals of this project

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

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

The future work

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

Chip technical capabilities

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

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

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

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

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

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

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

Historical use of the SN76489

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

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

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

The original pinout of the SN76489AN

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

Difference from the original hardware

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

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

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

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

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

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

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

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

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

A configurable clock divider was introduced in this implementation.

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

The reverse engineered SN76489

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

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

How to test

Summary of commands to communicate with the chip

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

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

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

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

Note frequency

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

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

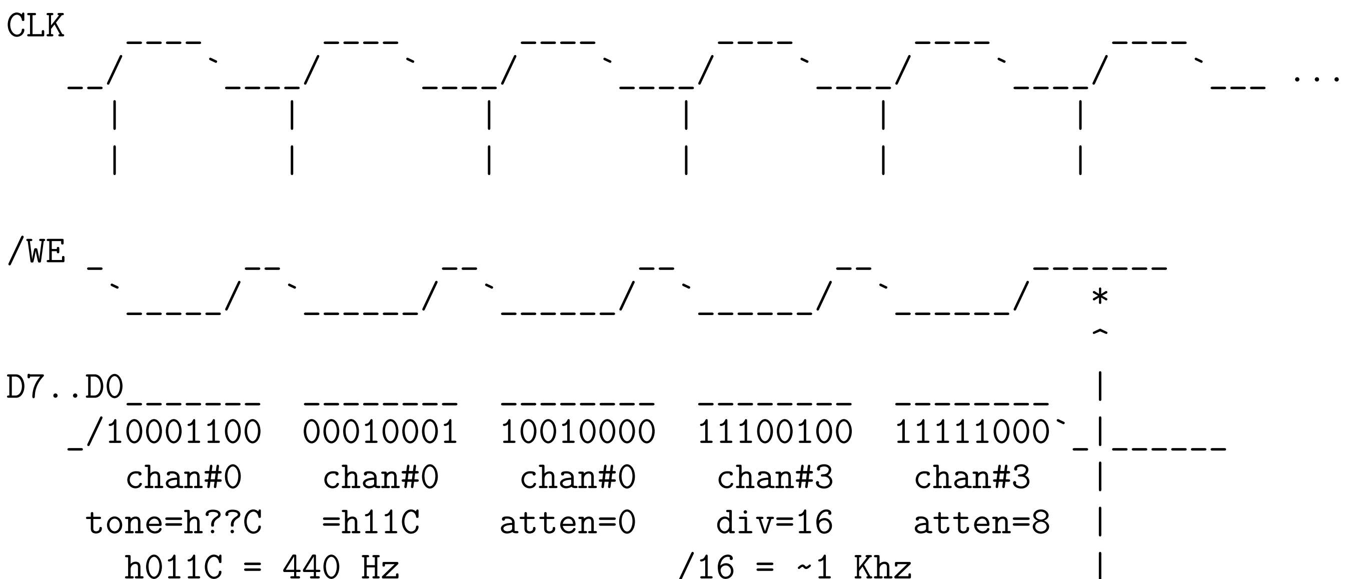
For example 10-bit value that plays 440 Hz note on a chip clocked at 4 MHz would be:

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

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

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

Timing diagram



```
                              white noise          |
                                                   |
                                             noise restarts
                                         after /WE goes high and
                                  there was a write to noise register
```

Configurable clock divider

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

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

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

Some examples of music recorded from the chip simulation

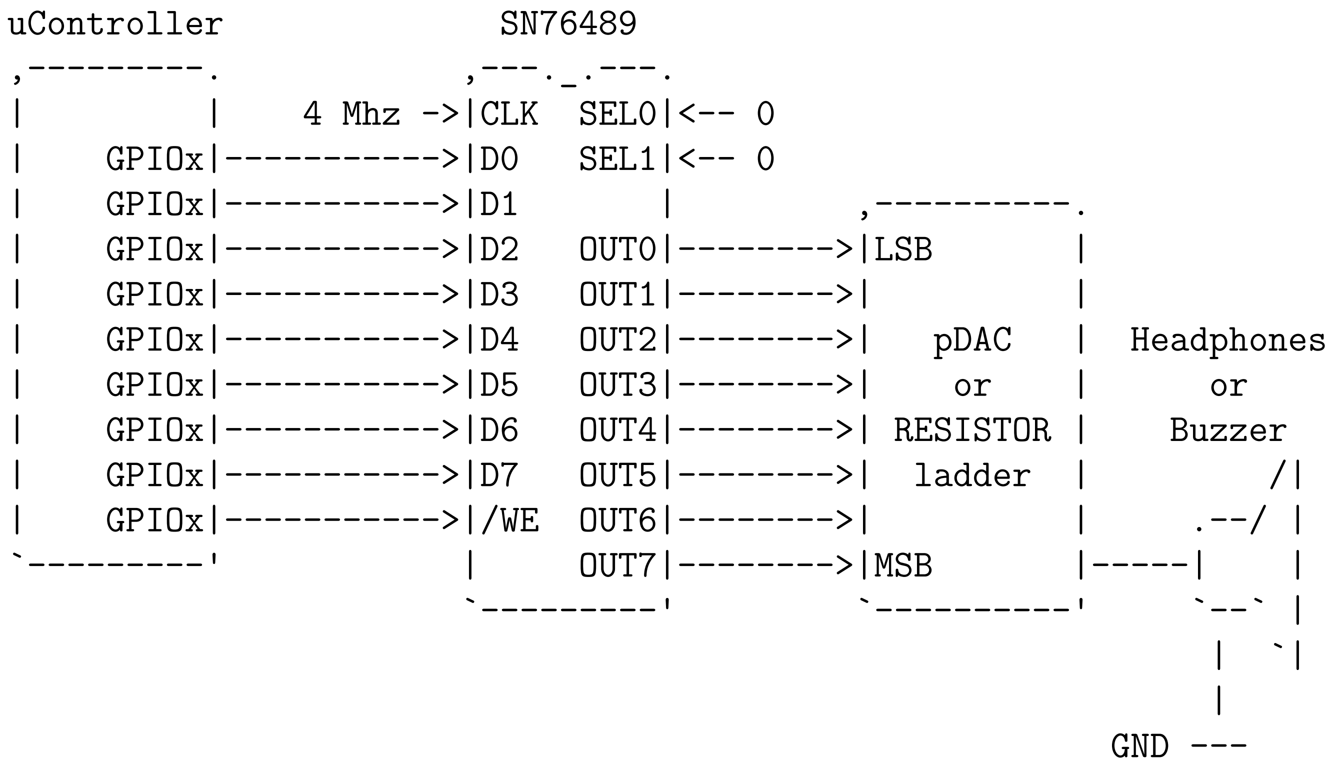
- https://www.youtube.com/watch?v=ghBGasckpSY
- https://www.youtube.com/watch?v=HXLAdA02I-w

External hardware

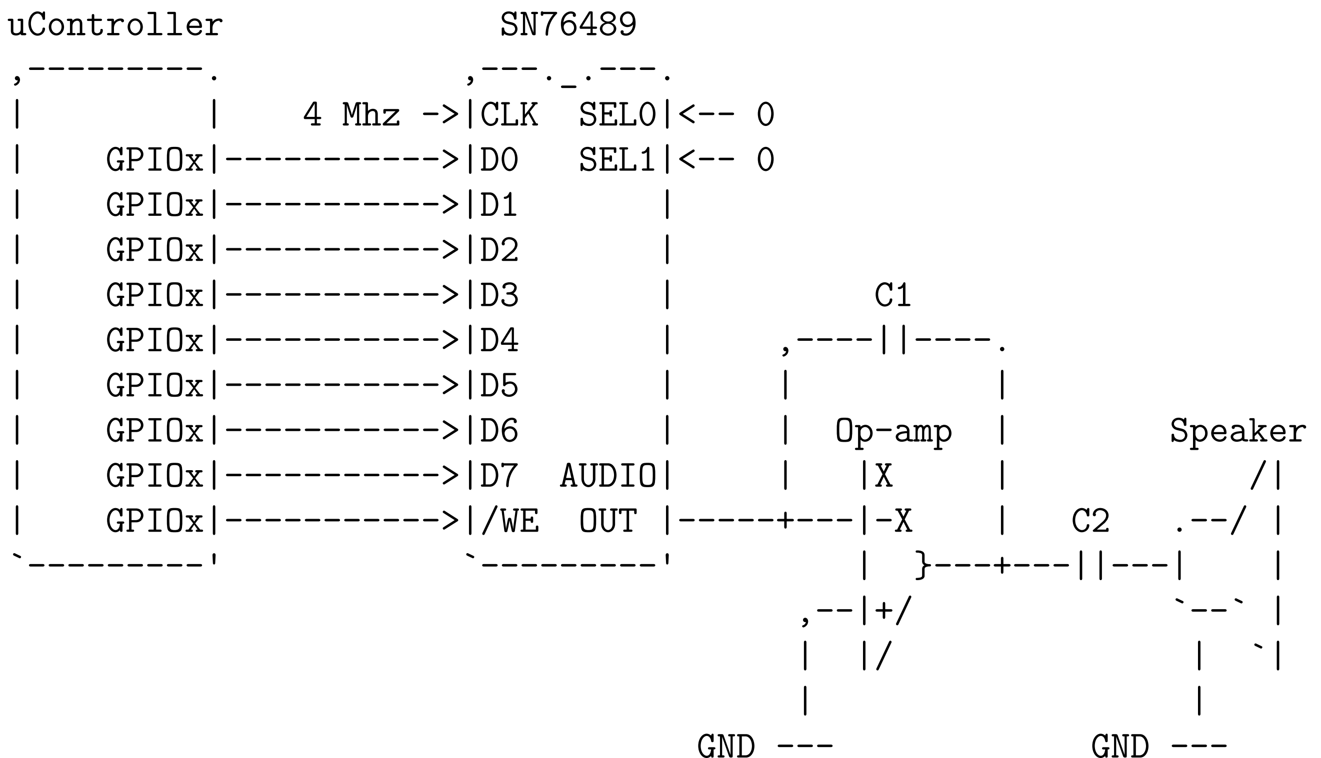
DAC (for ex. Digilent R2R PMOD), RC filter, amplifier, speaker.

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

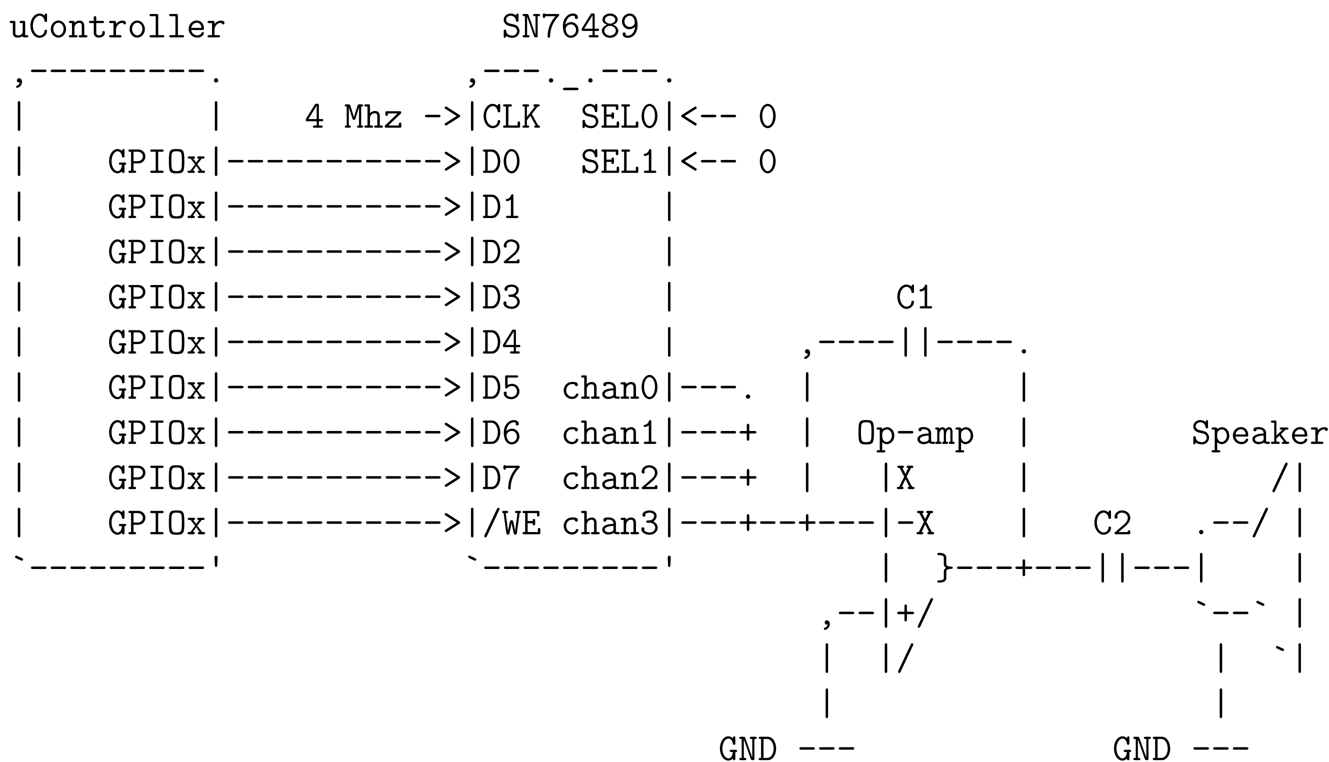
8-bit parallel output via DAC One option is to connect off the shelf data parallel Digital to Analog Converter (DAC) for example Digilent R2R Pmod to the output pins and route the resulting analog audio to piezo speaker or amplifier.



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



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



Pinout

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

Basic Matrix-Vector Multiplication [740]

- Author: Andy Ly
- Description: Basic matrix and vector multiplier that multiplies a 2x2 matrix with a 2x1 vector. Inputs are limited to 2 bit elements
- GitHub repository
- HDL project
- Mux address: 740
- Extra docs
- Clock: 0 Hz

How it works

Take input voltages and treats them as input current injection to lif neuron

How to test

Test it

External hardware

Possibly

Pinout

#	Input	Output	Bidirectional
0	Input bit [0] for matrix element 11	Output bit [0] for output vector element 1	Output bit [3] for output vector element 2
1	Input bit 1 for matrix element 11	Output bit 1 for output vector element 1	Output bit [4] for output vector element 2
2	Input bit [0] for matrix element 12	Output bit 2 for output vector element 1	

#	Input	Output	Bidirectional
3	Input bit 1 for matrix element 12	Output bit [3] for output vector element 1	
4	Input bit [0] for matrix element 21	Output bit [4] for output vector element 1	Input bit [0] for input vector element 1
5	Input bit 1 for matrix element 21	Output bit [0] for output vector element 2	Input bit 1 for input vector element 1
6	Input bit [0] for matrix element 22	Output bit 1 for output vector element 2	Input bit [0] for input vector element 2
7	Input bit 1 for matrix element 22	Output bit 2 for output vector element 2	Input bit 1 for input vector element 2

Classic 8-bit era Programmable Sound Generator AY-3-8913 [741]

- Author: ReJ aka Renaldas Zioma
- Description: The AY-3-8913 is a 3-voice programmable sound generator (PSG) chip from General Instruments. The AY-3-8913 is a smaller variant of AY-3-8910 or its analog YM2149.
- GitHub repository
- HDL project
- Mux address: 741
- Extra docs
- Clock: 2000000 Hz

How it works

This Verilog implementation is a replica of the classical **AY-3-8913** programmable sound generator. With roughly a 1500 logic gates this design fits on a **single tile** of the TinyTapeout.

The goals of this project

1. closely replicate the behavior and eventually the complete **design of the original** AY-3-891x with builtin DACs
2. provide a readable and well documented code for educational and hardware **preservation** purposes
3. leverage the **modern fabrication** process

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

Chip technical capabilities

- **3 square wave** tone generators
- A single **white noise** generator
- A single **envelope** generator able to produce 10 different shapes
- Chip is capable to produce a range of waves from a **30 Hz** to **125 kHz**, defined by **12-bit** registers.
- **16** different volume levels

Registers The behavior of the AY-3-891x is defined by 14 registers.

Register	Bits used	Function	Description
0	`xxxxxxxx`	Channel A Tone	8-bit fine frequency
1	`....xxxx`	—//—	4-bit coarse frequency
2	`xxxxxxxx`	Channel B Tone	8-bit fine frequency
3	`....xxxx`	—//—	4-bit coarse frequency
4	`xxxxxxxx`	Channel C Tone	8-bit fine frequency
5	`....xxxx`	—//—	4-bit coarse frequency
6	`...xxxxx`	Noise	5-bit noise frequency
7	`..CBACBA`	Mixer	Tone and/or Noise per channel
8	`...xxxxx`	Channel A Volume	Envelope enable or 4-bit amplitude
9	`...xxxxx`	Channel B Volume	Envelope enable or 4-bit amplitude
10	`...xxxxx`	Channel C Volume	Envelope enable or 4-bit amplitude
11	`xxxxxxxx`	Envelope	8-bit fine frequency
12	`xxxxxxxx`	—//—	8-bit coarse frequency
13	`....xxxx`	Envelope Shape	4-bit shape control

Square wave tone generators Square waves are produced by counting down the 12-bit counters. Counter counts up from 0. Once the corresponsding register value is reached, counter is reset and the output bit of the channel is flipped producing square waves.

Noise generator Noise is produced with 17-bit Linear-feedback Shift Register (LFSR) that flips the output bit pseudo randomly. The shift rate of the LFSR register is controller by the 5-bit counter.

Envelope The envelope shape is controlled with 4-bit register, but can take only 10 distinct patterns. The speed of the envelope is controlled with 16-bit counter. Only a single envelope is produced that can be shared by any combination of the channels.

Volume Each of the three AY-3-891x channels have dedicated DAC that converts 16 levels of volume to analog output. Volume levels are 3 dB apart in AY-3-891x.

Historical use of the AY-3-891x

The AY-3-891x family of programmable sound generators was introduced by General Instrument in 1978. Soon Yamaha Corporation licensed and released a very similar chip under YM2149 name.

Both variants of the AY-3-891x and YM2149 were broadly used in home computers, game consoles and arcade machines in the early 80ies.

- home computers: Apple II Mockingboard sound card, Amstrad CPC, Atari ST, Oric-1, Sharp X1, MSX, ZX Spectrum 128/+2/+3
- game consoles: Intellivision, Vectrex, Amstrad GX4000
- arcade machines: Frogger, 1942, Spy Hunter and etc.

The AY-3-891x chip family competed with the similar Texas Instruments SN76489.

The original pinout of the AY-3-8913

The **AY-3-8913** was a 24-pin package release of the AY-3-8910 with a number of internal pins left simply unconnected. The goal of AY-3-8913 was to reduce complexity for the designer and reduce the foot print on the PCB. Otherwise the functionality of the chip is identical to AY-3-8910 and AY-3-8912.

```
            ,--._.--.
    GND ---|1      24|<-- /cs*
   BDIR -->|2      23|<--  a8*
    BC1 -->|3      22|<-- /a9*
    DA7 <->|4      21|<-- /RESET
    DA6 <->|5      20|<-- CLOCK
    DA5 <->|6      19|--- GND
    DA4 <->|7      18|--> CHANNEL C OUT
    DA3 <->|8      17|--> CHANNEL A OUT
    DA2 <->|9      16|    not connected
    DA1 <->|10     15|--> CHANNEL B OUT
    DA0 <->|11     14|<-- test*
  test* <--|12     13|<-- VCC
            `-------'
    * -- omitted from this Verilog implementation
```

Difference from the original hardware

This Verilog implementation is a completely digital and synchronous design that differs from the original AY-3-8913 design which incorporated internal DACs and analog outputs.

Audio signal output While the original chip had no summation The module provides two alternative outputs for the generated audio signal:

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

Master output channel In contrast to the original chip which had only separate channel outputs, this implementation also provides an optional summation of the channels into a single master output.

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

No /A8, A9 and /CS pins The combination of **/A8**, **A9** and **/CS** pins orginially were intended to select a specific sound chip out the larger array of devices connected to the same bus. In this implementation this mechanism is omitted for simplicity, **/A8**, **A9** and **/CS** are considered to be tied **low** and chip behaves as always enabled.

Synchronous reset and single phase clock The original design employed 2 phases of the clock and asynchronous reset mechanism for operation of the registers.

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

The reverse engineered AY-3-891x

This implementation would not be possible without the reverse engineered schematics and analysis based on decapped AY-3-8910 and AY-3-8914 chips.

Explain how your project works

How to test

Summary of commands to communicate with the chip

The AY-3-8913 is programmed by updating its internal registers via the data bus. Below is a short summary of the communication protocol of AY-3-891x. Please consult AY-3-891x Technical Manual for more information.

BDIR	BC1	Bus state description
0	0	Bus is inactive
0	1	(Not implemented)
1	0	Write bus value to the previously latched register #
1	1	Latch bus value as the destination register #

Latch register address First, put the destination register adress on the bus of the chip and latch it by pulling both **BDIR** and **BC1** pins **high**.

Write data to register Put the desired value on the bus of the chip. Pull **BC1** pin **low** while keeping **BDIR** pin **high** to write the value of the bus to the latched register address.

Inactivate bus by pulling both **BDIR** and **BC1** pins **low**.

Register	Format	Description	Parameters
0,2,4	`ffffffff`	A/B/C tone period	`f` - low bits
1,3,5	`0000FFFF`	—//—	`F` - high bits
6	`000fffff`	Noise period	`f` - noise period
7	`00CBAcba`	Noise / tone per channel	`CBA` - noise off, `cba` - tone off
8,9,10	`000Evvvv`	A/B/C volume	`E` - envelope on, `v` - volume level
11	`ffffffff`	Envelope period	`f` - low bits
12	`FFFFFFFF`	—//—	`F` - high bits
13	`0000caAh`	Envelope Shape	`c` - continue, `a` - attack, `A` - alternate, `h` - hold

Note frequency

Use the following formula to calculate the 12-bit period value for a particular note:

$$toneperiod_{cycles} = clock_{frequency}/(16_{cycles} * note_{frequency})$$

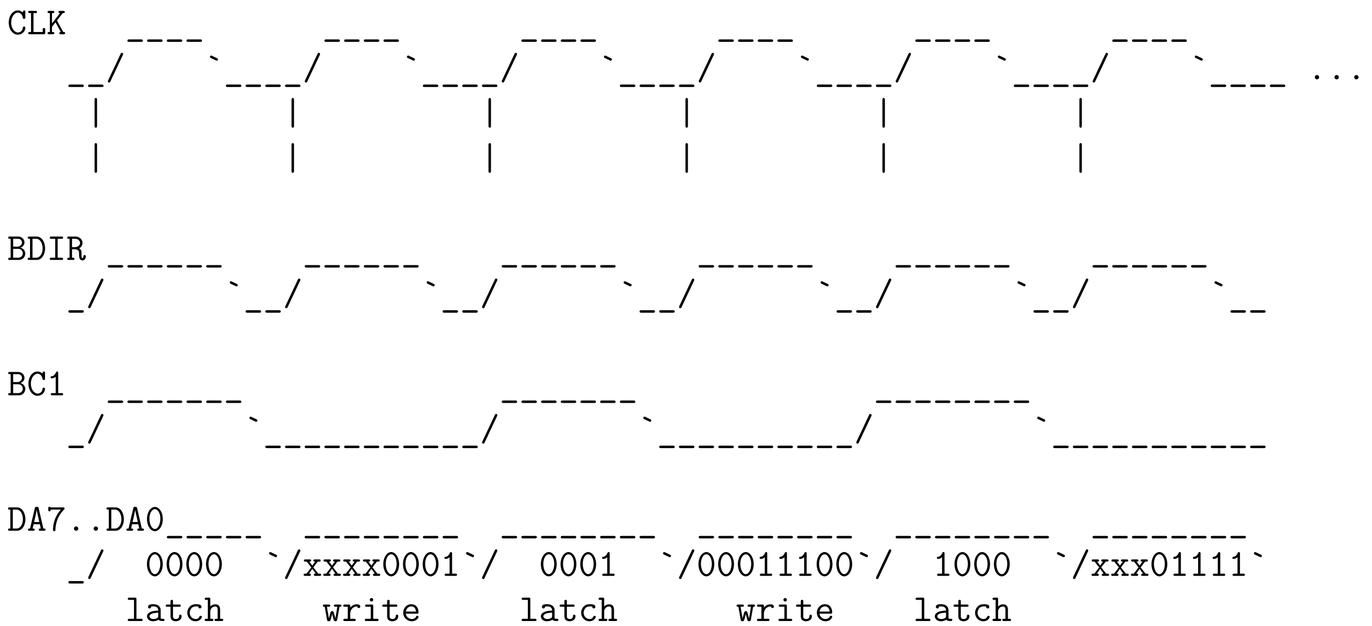
For example 12-bit period that plays 440 Hz note on a chip clocked at 2 MHz would be:

$$toneperiod_{cycles} = 2000000Hz/(16_{cycles} * 440Hz) = 284 = 11C_{hex}$$

An example to play a note at a maximum volume

BDIR	BC1	DA7..DA0	Explanation
1	1	`xxxx0000`	**Latch** tone A coarse register address $0 = 0000_{bin}$
1	0	`xxxx0001`	**Write** high 4-bits of the 440 Hz note $1 = 0001_{bin}$
1	1	`xxxx0001`	**Latch** tone A fine register address $1_{dec} = 0001_{bin}$
1	0	`00011100`	**Write** low 8-bits of the note $1C_{hex} = 00011100_{bin}$
1	1	`xxxx1000`	**Latch** channel A volume register address $8 = 1000_{bin}$
1	0	`xxx01111`	**Write** maximum volume level $15_{dec} = 1111_{bin}$ with the envelope disabled

Timing diagram



Externally configurable clock divider

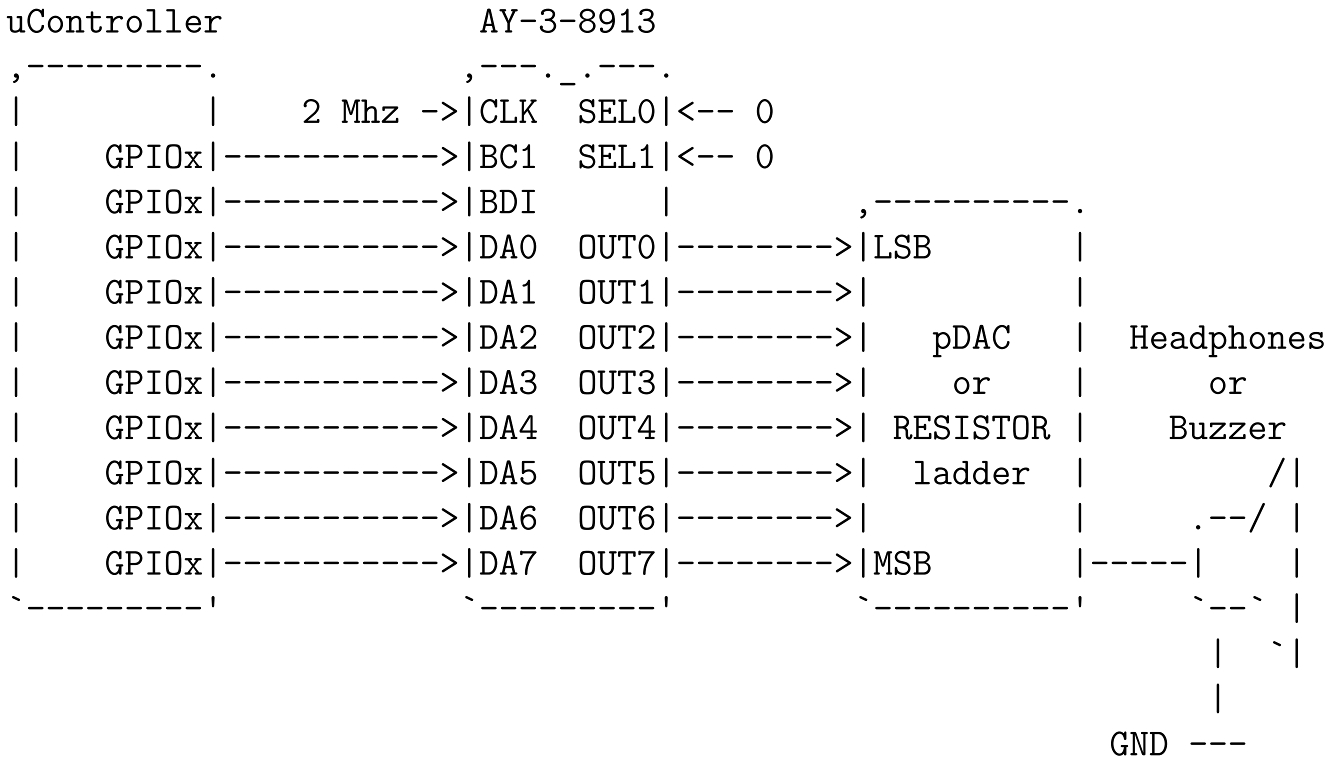
SEL1	SEL0	Description	Clock frequency
0	0	Standard mode, clock divided by 8	1.7 .. 2.0 MHz
1	1	—//—	1.7 .. 2.0 MHz
0	1	New mode for TT05, no clock divider	250 .. 500 kHZ
1	0	New mode for TT05, clock div. 128	25 .. 50 MHz

SEL1	SEL0	Formula to calculate the 12-bit tone period value for a note
0	0	$clock_{frequency}/(16_{cycles} * note_{frequency})$
1	1	—//—
0	1	$clock_{frequency}/(2_{cycles} * note_{frequency})$
1	0	$clock_{frequency}/(128_{cycles} * note_{frequency})$

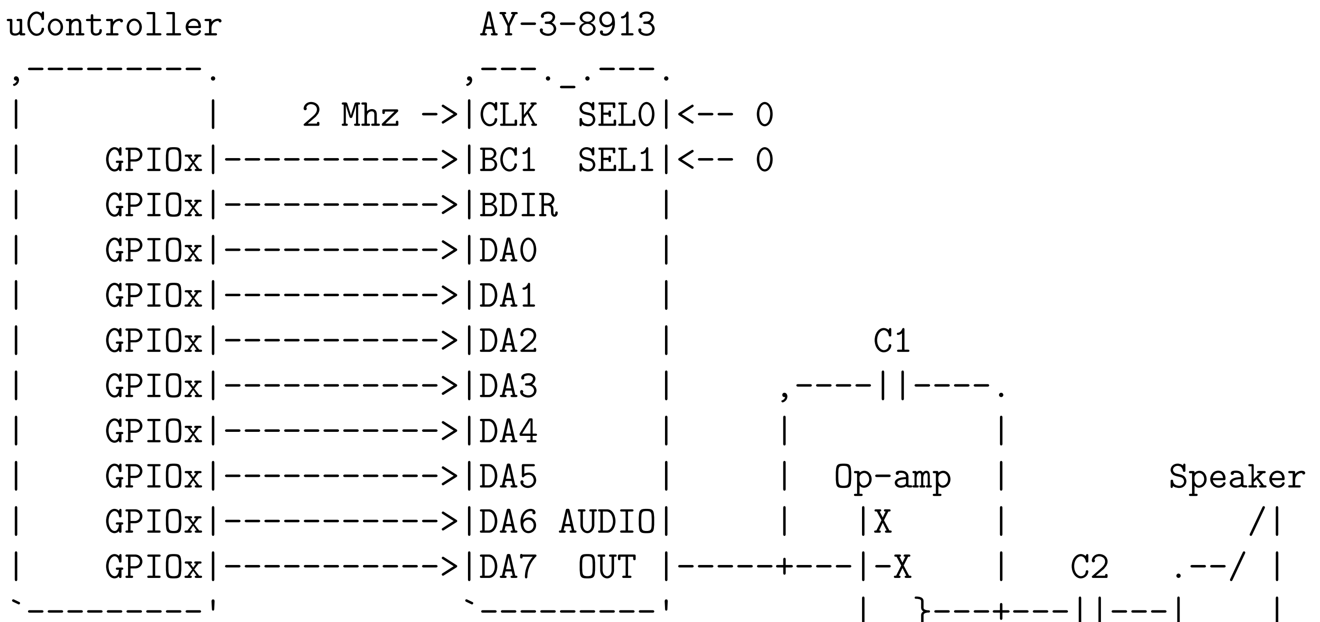
External hardware

The data bus of the AY-3-8913 chip has to be connected to microcontroller and receive a regular stream of commands. The AY-3-8913 produces audio output and has to be connected to a speaker. There are several ways how the overall schematics can be established.

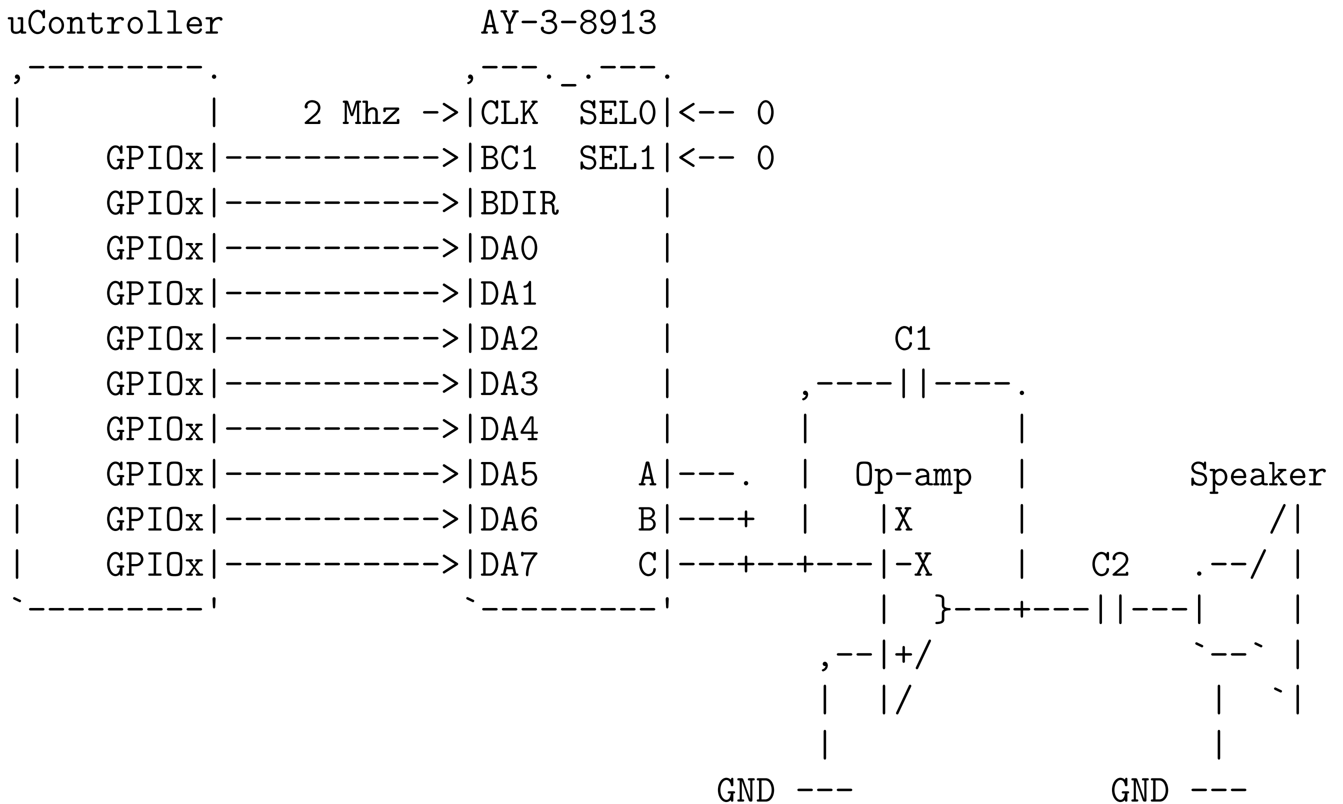
8-bit parallel output via DAC One option is to connect off the shelf data parallel Digital to Analog Converter (DAC) for example Digilent R2R Pmod to the output pins and route the resulting analog audio to piezo speaker or amplifier.



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



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



Pinout

#	Input	Output	Bidirectional
0	DA0 - multiplexed data/address bus LSB	audio out (PWM)	(in) **BC1** bus control
1	DA1 - multiplexed data/address bus	digita audio LSB	(in) **BDIR** bus direction

#	Input	Output	Bidirectional
2	DA2 - multiplexed data/address bus	digita audio	(in) **SEL0** clock divider
3	DA3 - multiplexed data/address bus	digita audio	(in) **SEL1** clock divider
4	DA4 - multiplexed data/address bus	digita audio	(out) channel A (PWM)
5	DA5 - multiplexed data/address bus	digita audio	(out) channel B (PWM)
6	DA6 - multiplexed data/address bus	digita audio	(out) channel C (PWM)
7	DA7 - multiplexed data/address bus MSB	digita audio MSB	(out) AUDIO OUT master (PWM)

8 bit MAC Unit [742]

- Author: Devesh Bhaskaran
- Description: Implementation Of 8-bit MAC Using Vedic Multipliers And Reversible Gates
- GitHub repository
- HDL project
- Mux address: 742
- Extra docs
- Clock: 40000000 Hz

How it works

The project aims to implement a 8-bit MAC unit for unsigned integer data type using Vedic Multipliers and Reversible gates. The two inputs are to be taken in through input pins and bi-directional pins using half a clock cycle and stored in registers. The MAC operation is performed on the values stored in these registers. The multiplier and adder takes half clock cycle each. The result of the operation is then sent through the output and bidirectional pins.

How to test

The project will be used to perform mac operations on 8-bit unsigned integers. This is mainly used in systems with fast computation and also primarly explores the concepts of reversible gates for energy efficiency.

External hardware

No external hardware is used for this project.

Pinout

#	Input	Output	Bidirectional
0	ui_in[0]	uo_out[0]	uio[0]
1	ui_in1	uo_out1	uio1
2	ui_in2	uo_out2	uio2
3	ui_in[3]	uo_out[3]	uio[3]
4	ui_in[4]	uo_out[4]	uio[4]

#	Input	Output	Bidirectional
5	ui_in[5]	uo_out[5]	uio[5]
6	ui_in[6]	uo_out[6]	uio[6]
7	ui_in[7]	uo_out[7]	uio[7]

Cgates [743]

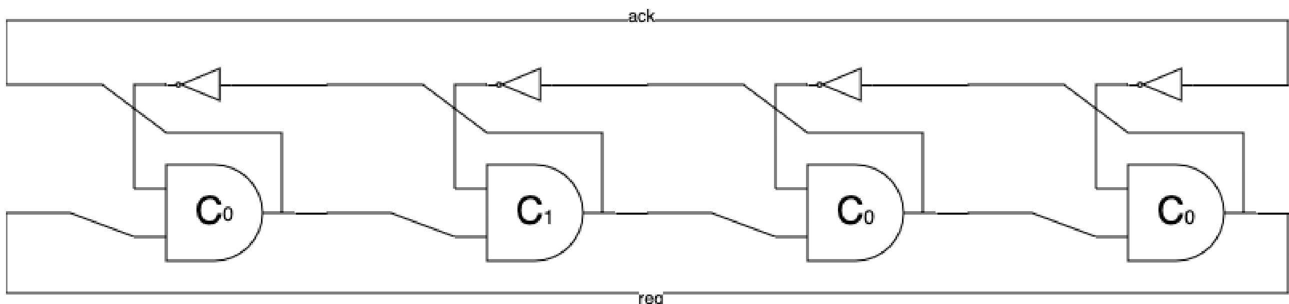
- Author: Tommy Thorn
- Description: Testing two different Cgate implementations and rings
- GitHub repository
- HDL project
- Mux address: 743
- Extra docs
- Clock: 0 Hz

How it works

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

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

We also build four rings from this, with uo2 and uo[3] being the output of a four stage build from Cl and Cc gates respectively. Similar for uo[4]/uo[5] except using 16 stage rings and uo[6]/uo[7] for 64 stage rings.



Since the pulse from each C-gate rings last only a few gate delay times, we use it to feed a toggle flip-flop, thus the corresponding output ping will toggle every time the pulse makes it round the ring. In other words, the cycle time of 4, 16, and 64 stage ring corresponding 8, 32, and 128 times the average stage delay of the corresponding ring.

Why is this interesting? Most asynchronous circuits disciplines rely heavily on the Cgate and this stage delay represents the absolute best-case for an asynchronous pipestage. Of course, for most interesting circuits the stage delay will be dominated by the computation performed.

How to test

Set ui[0] and ui1 different values and verify that uo[0]/uo1 only changes when both agree. The remaining six uo outputs corresponding to six rings, two 4-stage, two 16-stage, and two 64-stage. The first of each pair are built from latches, the latter from combinatorial logic. It will be interesting to see which is faster. The outputs are limited to 33 MHz / 30.3 ns, thus if the stage delay is less 3.8 ns we likely will not observe anything. For the other two the limits, are 947 ps and 237 ps, respectively. In hindsight, I should have made the rings more than an order of magnitude longer.

External hardware

For the basic test the rp2040 on the bringup board should be enough for the ring test, an oscilloscope is definely required to see anything from the rings.

Pinout

#	Input	Output	Bidirectional
0	A	Ql	
1	B	Qc	
2		R4l	
3		R4c	
4		R16l	
5		R16c	
6		RTBDl	
7		RTBDc	

Programmable PWM Generator [744]

- Author: Anas Alam
- Description: Programmabel PWM Generator
- GitHub repository
- HDL project
- Mux address: 744
- Extra docs
- Clock: 0 Hz

How it works

A programmable PWM generator. The desired frequency and duty cycle is programmed by setting pwm_top and pwm_threshold. A counter counts from 0 to pwm_top (over and over), the pwm signal is high as when the counter is $<=$ pwm_threshold.

pwm_top is wired to uio (all of them are used as inputs) pwm_threshold is wired to ui

They are encoded as follows

```
pwm_top &amp;lt;= uio(7 downto 5) &amp;lt;&amp;lt; uio(4 downto 0)
```

```
pwm_threshold &amp;lt;= ui(7 downto 5) &amp;lt;&amp;lt; ui(4 downto 0)
```

Resulting frequency of PWM signal is: $f_{out} = \frac{f_{in}}{pwm_{top}+1}$

Resulting duty cycle is: $f = \frac{pwm_{threshold}+1}{pwm_{top}+1}$

The goal is to have wide as possible frequency range while still being able to go from 0% to 100% in duty cycle.

How to test

Use above formulas to determine value of pwm_threshold and pwm_top, hard ware them to this value or connect through switches. Probe output on oscilloscope

External hardware

Switches and oscilloscope

Pinout

#	Input	Output	Bidirectional
0	pwm_threshold shift_amount[0]	pwm output	input: pwm_top shift_amount[0]
1	pwm_threshold shift_amount1	design is enabled (active high)	input: pwm_top shift_amount1
2	pwm_threshold shift_amount2	wired 0	input: pwm_top shift_amount2
3	pwm_threshold shift_amount[3]	wired 0	input: pwm_top shift_amount[3]
4	pwm_threshold shift_amount[4]	wired 0	input: pwm_top shift_amount[4]
5	pwm_threshold base[0]	wired 0	input: pwm_top base[0]
6	pwm_threshold base1	wired 0	input: pwm_top base1
7	pwm_threshold base[3]	wired 0	input: pwm_top base2

eksdee [745]

- Author: lucy revi
- Description: That's for none of us to know and all of us to find out.
- GitHub repository
- HDL project
- Mux address: 745
- Extra docs
- Clock: 0 Hz

How it works

I honestly don't know yet.

How to test

I honestly don't know yet.

External hardware

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

I honestly don't know yet.

Pinout

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

Verilog test project [746]

- Author: Alexander Symons
- Description: It adds the input and the IO pins
- GitHub repository
- HDL project
- Mux address: 746
- Extra docs
- Clock: 0 Hz

How it works

It adds the dedicated input to an internal register every clock cycle Least significant bits: dedicated output Most significant bits: bidirectional output

How to test

Put numbers on the input and see the accumulated value on all the leds

External hardware

Switches on inputs, leds on outputs and bidirectionals

Pinout

#	Input	Output	Bidirectional
0	ui_in[0]	uo_out[0]	uio_out[0]
1	ui_in1	uo_out1	uio_out1
2	ui_in2	uo_out2	uio_out2
3	ui_in[3]	uo_out[3]	uio_out[3]
4	ui_in[4]	uo_out[4]	uio_out[4]
5	ui_in[5]	uo_out[5]	uio_out[5]
6	ui_in[6]	uo_out[6]	uio_out[6]
7	ui_in[7]	uo_out[7]	uio_out[7]

ternary, E1M0, E2M0 decoders [747]

- Author: ReJ aka Renaldas Zioma
- Description: Ternary, Quinary and Septenary 1.6 .. 2.6 bits/param packed weights
- GitHub repository
- HDL project
- Mux address: 747
- Extra docs
- Clock: 0 Hz

How it works

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

How to test

Provide packed weights on INPUT pmods

External hardware

Use Pico

Pinout

#	Input	Output	Bidirectional
0	packed weights LSB	unpacked	(out) unpacked
1	packed weights	unpacked	(out) unpacked
2	packed weights	unpacked	(out) unpacked
3	packed weights	unpacked	(out) unpacked
4	packed weights	unpacked	(out) unpacked
5	packed weights	unpacked	(out) unpacked

#	Input	Output	Bidirectional
6	packed weights	unpacked	(out) dummy
7	packed weights MSB	unpacked	(in) Ternary / Septenary

Basic LIF Neuron [748]

- Author: stewedbeef
- Description: This is a basic LIF neuron
- GitHub repository
- HDL project
- Mux address: 748
- Extra docs
- Clock: 0 Hz

How it works

This is a simple leaky integrate-and-fire neuron which performs the integration by addition and leaks by dividing by two every time step. The neuron has an enable pin which causes the neuron to enable and move forward in time roughly once every second when fed a clock of approximately 50 MHz.

How to test

The LED wired up to output seven should turn on and off approximately once every second, with a period of approximately two seconds, to allow synchronisation by the user. Each time the LED switches on or off a time step has occurred. The user should stimulate the neuron by "providing" an input current, which is achieved by switching the inputs manually to indicate to the neuron, in binary, how much current should flow in. With enough stimulus, the neuron will fire a spike, visible on LEDs zero to six, for one time period. The neuron has a timeout which prevents it from having a constant output from overstimulation.

External hardware

Wire switches to all input ports and LEDs to all output ports. Bidirectional ports are unused.

Pinout

#	Input	Output	Bidirectional
0	ui_in[0]	uo_out[0]	
1	ui_in1	uo_out1	

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

Dynamic Threshold Leaky Integrate-and-Fire [749]

- Author: Kai Linsley
- Description: Leaky Integrate-and-Fire model simulating a spiking biological neuron
- GitHub repository
- HDL project
- Mux address: 749
- Extra docs
- Clock: 1000000 Hz

How it works

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

How to test

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

External hardware

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

Pinout

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

Integrate-and-Fire Neuron Circuit [750]

- Author: FNU Ashwine
- Description: A simple integrate-and-fire neuron model implemented in Verilog.
- GitHub repository
- HDL project
- Mux address: 750
- Extra docs
- Clock: 0 Hz

How it works

The Leaky Integrate-and-Fire (LIF) Neuron is a simple model of neuronal behavior. In this design, the neuron receives an input signal (spike) and integrates this input over time by increasing its internal membrane potential. If there is no input, the membrane potential "leaks" or decays gradually over time, simulating the natural loss of charge in biological neurons.

When the membrane potential reaches a defined threshold, the neuron fires a spike output, after which the membrane potential resets to zero. This process emulates the firing and reset cycle of biological neurons, providing a digital approximation of spiking behavior.

LIF Neuron Diagram - https://drive.google.com/uc?export=view&id=19_hF5C_uv8FfWdlOOItl

How to test

Do something

External hardware

NA

Pinout

#	Input	Output	Bidirectional
0	Input current bit [0]	State variable bit [0]	

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

tt09-C6-array-multiplier [751]

- Author: Jonathan Farah and Josef Anurov
- Description: 4x4 multiplier
- GitHub repository
- HDL project
- Mux address: 751
- Extra docs
- Clock: 0 Hz

How it works

4 x 4 Array Multiplier Circuit Diagram The circuit implements a 4 x 4 array multiplier using manual structural design. The array multiplier was created by using partial products and 4-bit adders. There are 2 inputs: m and q, which represent 4-bit input numbers to be multiplied. It outputs p, an 8-bit product of m and q. It has Wires w1,w2,w3 and w4. These represent the partial products generated from each bit of q multiplied by each bit of m. Wires: partial1, partial2, and partial3 store intermediate sums of partial products as they are added.

How to test

We test using some test numbers and checking the output. Wires C[2:0] are carry bits between the adders. Each w vector (from w1 to w4) represents the result of ANDing each bit of m with a specific bit of q. The add_4bit modules add these partial products together, simulating a ripple-carry addition for each shifted partial product. In terms of assigning the final product is constructed from the individual bits. The MSB comes from C2. Bits of partial3, partial2, partial1, and w1 make up the remaining bits, in that order. The implementation of the circuit was then tested using the provided Verilog testbench. The testbench was given a combination of inputs that effectively tested each case to ensure that the multiplier ran correctly.

External hardware

N/A

Pinout

#	Input	Output	Bidirectional
0	q[0]	p[0]	
1	q1	p1	
2	q2	p2	
3	q[3]	p[3]	
4	m[0]	p[4]	
5	m1	p[5]	
6	m2	p[6]	
7	m[3]	p[7]	

Zilog Z80 [770]

- Author: ReJ aka Renaldas Zioma
- Description: Z80 open-source silicon. Goal is to become a silicon proven, pin compatible, open-source replacement for classic Z80.
- GitHub repository
- HDL project
- Mux address: 770
- Extra docs
- Clock: 16000000 Hz

How it works

On April 15 of 2024 Zilog has announced End-of-Life for Z80, one of the most famous 8-bit CPUs of all time. It is a time for open-source and hardware preservation community to step in with a Free and Open Source Silicon (FOSS) replacement for Zilog Z80.

The implementation is based around Guy Hutchison's TV80 Verilog core.

The future work

- Add thorough instruction (including 'illegal') execution tests ZEXALL to test-bench
- Compare different implementations: Verilog core A-Z80, Netlist based Z80Explorer
- Create gate-level layouts that would resemble the original Z80 layout. Zilog designed Z80 by manually placing each transistor by hand.
- Tapeout QFN44 package
- Tapeout DIP40 package

Z80 technical capabilities

- nMOS original frequency 4MHz. CMOS frequency up to 20 MHz. This tapeout on 130 nm is expected to support frequency up to 50 MHz.
- 158 instructions including support for Intel 8080A instruction set as a subset.
- Two sets of 6 general-purpose reigsters which may be used as either 8-bit or 16-bit register pairs.
- One maskable and one non-maskable interrupt.
- Instruction set derived from Datapoint 2200, Intel 8008 and Intel 8080A.

Z80 registers

- `AF`: 8-bit accumulator (A) and flag bits (F)
- `BC`: 16-bit data/address register or two 8-bit registers

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

Z80 Pinout

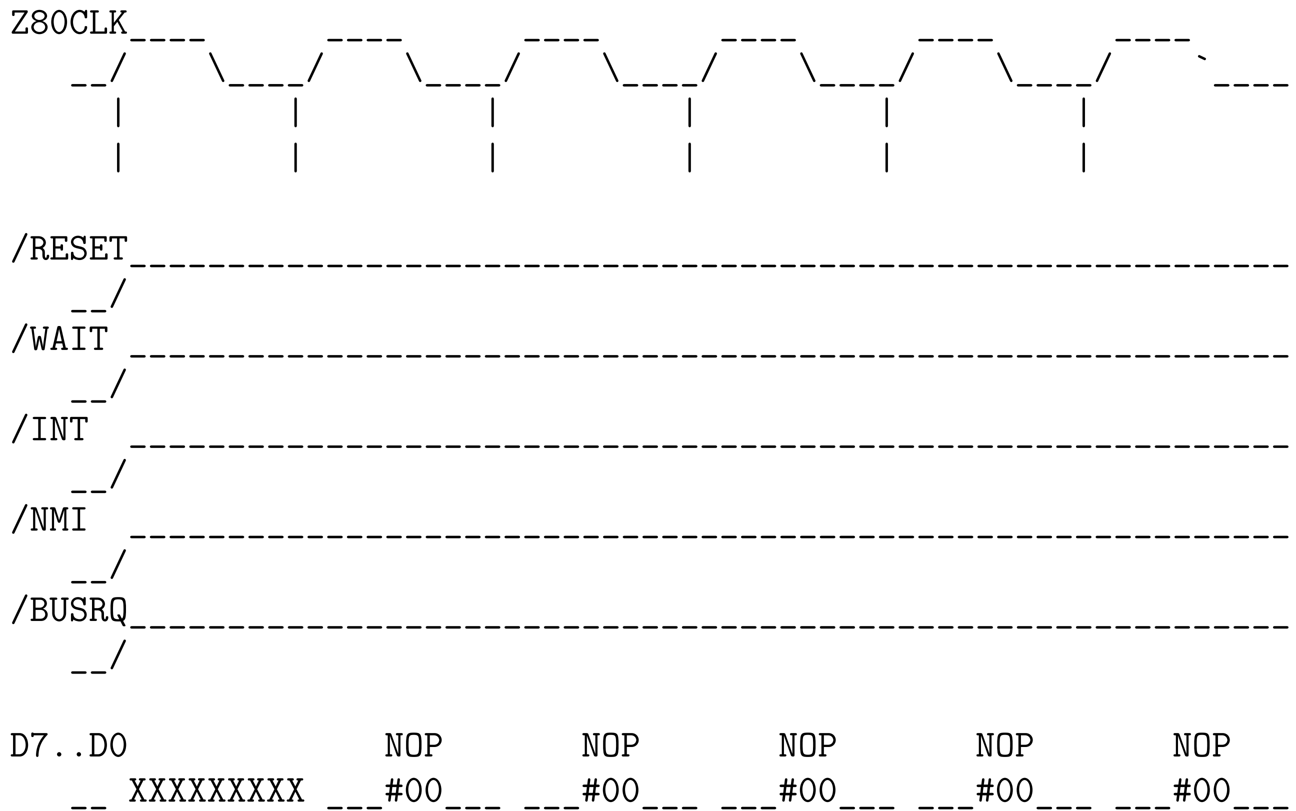
```
                 ,---------.__.---------.
<--     A11 |1                   40| A10     -->
<--     A12 |2                   39| A9      -->
<--     A13 |3        Z80 CPU    38| A8      -->
<--     A14 |4                   37| A7      -->
<--     A15 |5                   36| A6      -->
-->     CLK |6                   35| A5      -->
<->      D4 |7                   34| A4      -->
<->      D3 |8                   33| A3      -->
<->      D5 |9                   32| A2      -->
<->      D6 |10                  31| A1      -->
        VCC |11                  30| A0      -->
<->      D2 |12                  29| GND
<->      D7 |13                  28| /RFSH   -->
<->      D0 |14                  27| /M1     -->
<->      D1 |15                  26| /RESET  <--
-->    /INT |16                  25| /BUSRQ  <--
-->    /NMI |17                  24| /WAIT   <--
<--   /HALT |18                  23| /BUSAK  -->
<--   /MREQ |19                  22| /WR     -->
<--   /IORQ |20                  21| /RD     -->
            `----------------------'
```

How to test

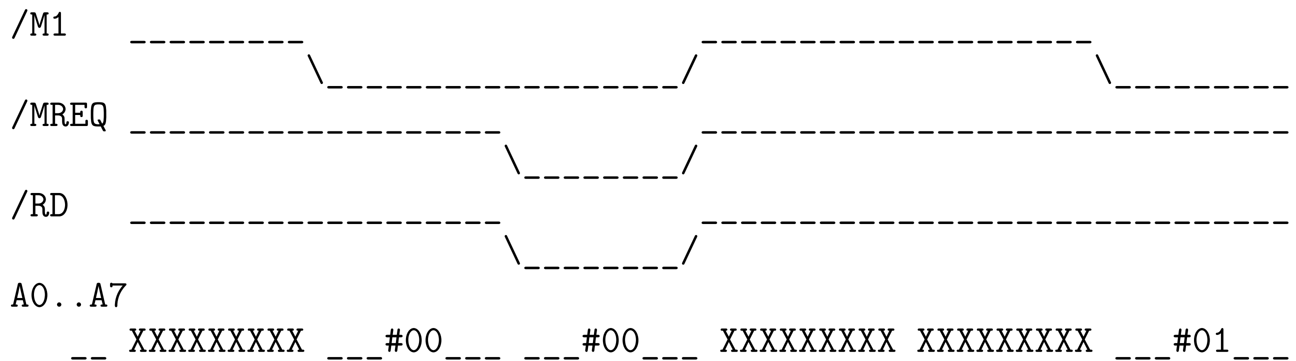
Hold all bidirectional pins (**Data bus**) low to make CPU execute **NOP** instruction. **NOP** instruction opcode is 0. Hold all input pins high to disable interrupts and signal that data bus is ready.

Every 4th cycle 8-bit value on output pins (**Address bus low 8-bit**) should monotonously increase.

Timing diagram, input pins



Expected signals on output pins



External hardware

Bus de-multiplexor, external memory, 8-bit computer such as ZX Spectrum.

Alternatively the RP2040 on the TinyTapeout test PCB can be used to simulate RAM and I/O.

Pinout

#	Input	Output	Bidirectional
0	/WAIT	/M1, A0, A8	D0
1	/INT	/MREQ, A1, A9	D1
2	/NMI	/IORQ, A2, A10	D2
3	/BUSRQ	/RD, A3, A11	D3
4		/WR, A4, A12	D4
5		/RFSH, A5, A13	D5
6	MUX – address lo/hi bits on the output pins	/HALT, A6, A14	D6
7	MUX – control signals on the output pins	/BUSAK, A7, A15	D7

Spectrogram extractor, 2 channels [782]

- Author: Coline Chehense, Dinko Oletic
- Description: Digital part of a time-frequency feature extraction sensor interface, two-channel real-time signal amplitude tracker. 7 input lines per channel represent thermometer code output of a flash ADC. Two-channel serial output.
- GitHub repository
- HDL project
- Mux address: 782
- Extra docs
- Clock: 1000000 Hz

How it works

This is an early work-in progress test implementation of a digital readout, part of a low-power mixed-signal multichannel sensor interface for acoustic emission detection. The sensor interface is developed to support a passive, micromechanically-implemented ultrasonic signal frequency decomposition MEMS device, based on an array of piezo-electric micro-resonators: https://ieeexplore.ieee.org/document/9139151.

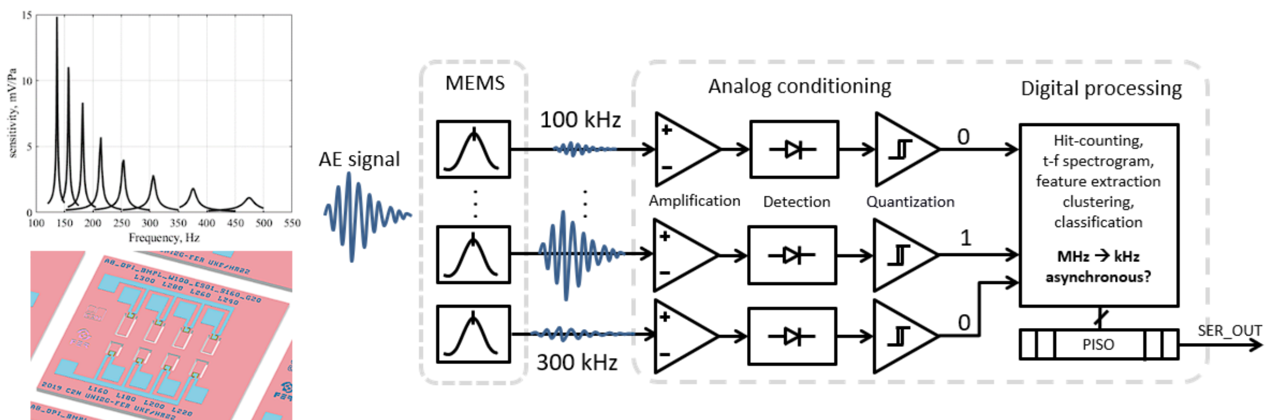


Figure 45: MEMS-based mixed-signal multichannel sensor interface for acoustic emission detection.

The digital part implemented here, performs real-time tracking of time-frequency spectrograms of individual acoustic emissions. It is assumed that each input-channel represents a signal amplitude envelope of the associated frequency-component over time, digitized using a flash ADC. The analog part of the ADC is not implemented here. Each input channel is represented by 7 input lines per channel representing the thermometer code output of a flash ADC. This test design implements only two-channels. The amplitude of an signal envelope at each channel is decoded into 3-bit BCD code.

Presence of an input signal at any channel (detector of acoustic emission start) initiates event-based 1 MHz sampling of the time-frequency amplitude spectrogram. The sampling lasts for 200 us. Once finished, the state machine controls reads-out the data stored. A double buffer composed of D-bistables is used to manage the storage and readout simultaneously. The stored data is sent serially for each channel. An RTC module is used to retrieve the time of the acoustic emission start.

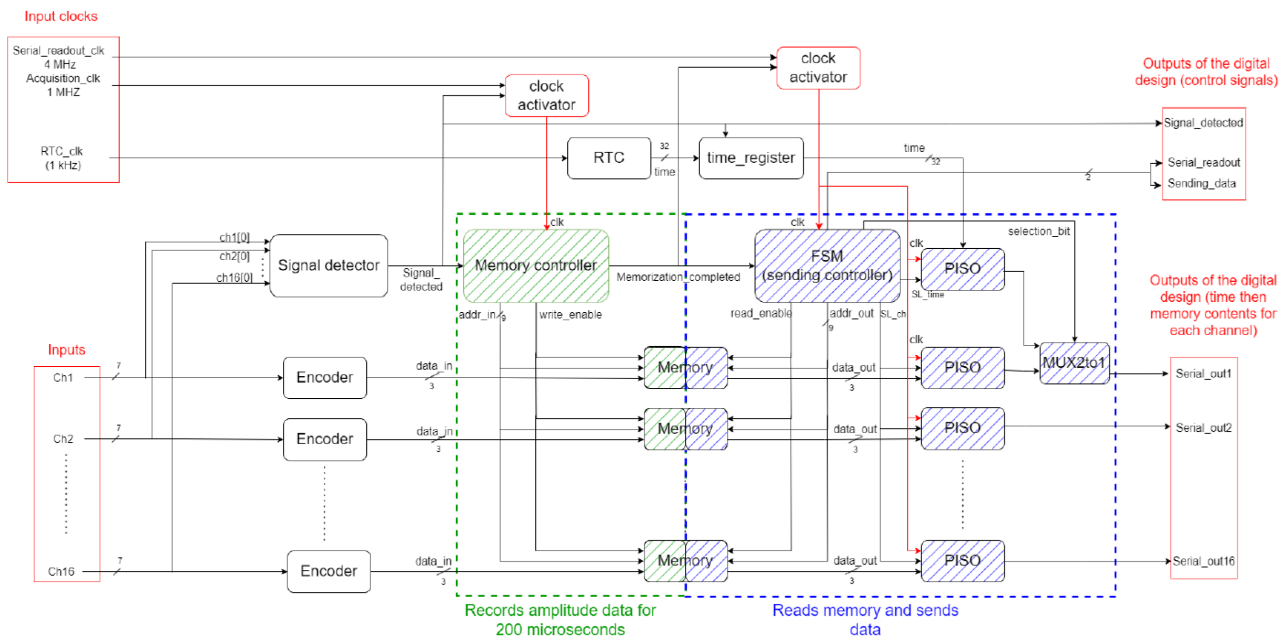


Figure 46: Digital multi-channel time-frequency amplitude spectrogram tracker.

This design is part of research activities https://www.fer.unizg.hr/liss/aemems. The design is generally applicable as a generic multi-channel time-series feature extraction block, and serve for subsequent clustering or classification, as part of an low-power MEMS-based sensor system-on-chip for acoustic event detection, or non-destructive testing. This is the first TinyTapeout submission of the design.

How to test

Please contact authors for detailed instructions on how to set-up the design.

External hardware

Logics analyzer will be useful for debugging.

Pinout

#	Input	Output	Bidirectional
0	ch1(0)	serial_out(0)	ch2(0)
1	ch1(1)	serial_out(1)	ch2(1)
2	ch1(2)	SL_time	ch2(2)
3	ch1(3)	SL_ch	ch2(3)
4	ch1(4)	signal_detected	ch2(4)
5	ch1(5)	memorization_completed	ch2(5)
6	ch1(6)	serial_readout	ch2(6)
7	RTC_clk(1kHz)	sending_data	serial_readout_clk(4Mhz)

Encoder [800]

- Author: Peilin
- Description: 8x3 encoder
- GitHub repository
- Wokwi project
- Mux address: 800
- Extra docs
- Clock: 0 Hz

How it works

Use the logic gates to make the 7 segment display the first letter of your name.

How to test

If the IN1-IN5 are on, the 7 segment will display an "F".

External hardware

No external hardware.

Pinout

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

chip_fab [801]

- Author: Aleksi
- Description: simple logic gate
- GitHub repository
- Wokwi project
- Mux address: 801
- Extra docs
- Clock: 0 Hz

How it works

Testing

How to test

Testing

External hardware

Here is my hardware

Pinout

#	Input	Output	Bidirectional
0	IN0	OUT0	
1	IN1	OUT1	
2			
3			
4			
5			
6			
7			

Clocked Display [802]

- Author: Dooseok
- Description: 8-bit Clocked Display
- GitHub repository
- Wokwi project
- Mux address: 802
- Extra docs
- Clock: 1 Hz

How it works

It allows output displays when the switch is clicked, or clock signal toggles. This was enabled by using flip-flops. And it also can be reset using the reset button.

How to test

Set input toggle switches as you want. Start test, and check its output changes by clicking the switch button.

External hardware

Flip-flops are added to enable display at each switch.

Pinout

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

YoshiTP [803]

- Author: Yeshua M (Yoshi)
- Description: Tiny Tapeout Microchip creates the letter M if flipped horizontally
- GitHub repository
- Wokwi project
- Mux address: 803
- Extra docs
- Clock: 0 Hz

How it works

This project is a tiny TDC constructed entirely of standard cells. During the Tapeout, we are creating a MOSFET. A MOSFET is a type of Transistor that can be used as a digital watch. The name stands for Metal Oxide Semiconductor Field Effect Transistor. When a voltage is applied across the gate and body, an electric field forms in the channel, attracting charge carriers to enable conduction. In an N-type MOSFET on a P-type substrate, minority carriers (electrons) form the conductive channel between the drain and the source. In a P-type MOSFET, the process is reversed: the substrate is N-type, and minority carriers (holes) create the channel. Since holes have lower mobility than electrons, P-type MOSFETs typically require a wider channel (2-3 times larger) than N-type MOSFETs for matching performance. To avoid increasing cell size, standard cells often don't match P and N strengths. Doping the silicon controls the ratio of charge carriers.

During the tape out when turning on or turning of the switches IN1-7 is causes a reaction on the lights that turn it on (OUT1-7).

How to test

To test the MOSFET you can switch on and off switches to cause a reaction on the "digital watch light"

External hardware

LED lights were used such as the Digital watch to view the numbers. (UNSURE) but the Logics "NAND Gate" and the "Not Gate" been used to create my initial of my last name

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	

A simple leaky integrate and fire neuron [804]

- Author: Heather Knight
- Description: Simulation of lif neuron
- GitHub repository
- HDL project
- Mux address: 804
- Extra docs
- Clock: 0 Hz

How it works

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

How to test

Do something

External hardware

NA

Pinout

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

#	Input	Output	Bidirectional
6	Input current bit [6]	State variable bit [6]	
7	Input current bit [7]	State variable bit [7]	Spike bit

Who knows what's happening Tiny Tapeout [805]

- Author: Ryan Kuo
- Description: Couldn't tell you
- GitHub repository
- Wokwi project
- Mux address: 805
- Extra docs
- Clock: 1 Hz

How it works

YOLO

How to test

I am still working on how to make it work. Test by runnning random input

External hardware

Im going to implement external led for funsies

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	

VGA Tiny Logo (1 tile) [806]

- Author: Renaldas Zioma
- Description: Large 480x480 pixels Tiny Tapeout logo with bling and dithered colors crammed into 1 tile!
- GitHub repository
- HDL project
- Mux address: 806
- Extra docs
- Clock: 25175000 Hz

How it works

Compressed VGA Logo

How to test

Connect to VGA monitor

External hardware

TinyVGA PMOD, VGA monitor

Pinout

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

Tiniest of tapeouts [807]

- Author: J Money
- Description: I'm still learning, trying my best
- GitHub repository
- Wokwi project
- Mux address: 807
- Extra docs
- Clock: 0 Hz

How it works

Guess a number! If the light lights up you win

How to test

Guess!

External hardware

Connect LED to output 0

Pinout

#	Input	Output	Bidirectional
0	IN3	OUT0	
1			
2			
3			
4			
5			
6			
7			

SK Test Workshop [808]

- Author: sreela
- Description: 8 bit buffer
- GitHub repository
- Wokwi project
- Mux address: 808
- Extra docs
- Clock: 0 Hz

How it works

takes 4 bit inputs, loads and displays

switches 7/8 to load upper 4 bits and then lower

How to test

play with switches

External hardware

none

Pinout

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

Tian TT9 [809]

- Author: Tianxin Wu
- Description: Flashy Lights
- GitHub repository
- Wokwi project
- Mux address: 809
- Extra docs
- Clock: 0 Hz

How it works

It is supposed to light up and hopefully flashes.

How to test

Run stimulation and redo things.

External hardware

LED Display, I believe.

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	

2-bit 2x2 Matrix Multiplier [810]

- Author: Kevin Ma
- Description: multiples two 2-bit 2x2 matrices
- GitHub repository
- HDL project
- Mux address: 810
- Extra docs
- Clock: 1000000 Hz

How it works

Computes matrix multiplication AB = C.

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

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

"A" top left: (0,0) -> IN7 | IN6
"A" top right: (0,1) -> IN5 | IN4
"A" bot left: (1,0) -> IN3 | IN2
"A" bot right: (1,1) -> IN1 | IN0

"B" top left: (0,0) -> IO7 | IO6
"B" top right: (0,1) -> IO5 | IO4
"B" bot left: (1,0) -> IO3 | IO2
"B" bot right: (1,1) -> IO1 | IO0

"C" top left: (0,0) -> OUT7 | OUT6
"C" top right: (0,1) -> OUT5 | OUT4
"C" bot left: (1,0) -> OUT3 | OUT2
"C" bot right: (1,1) -> OUT1 | OUT0

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

Each pin corresponds to one bit.

How to test

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

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

The matrix mulitplication of AB will be output.

External hardware

No external hardware needed.

Pinout

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

RISCV Processor Design [811]

- Author: KOUSHIK CSN
- Description: RISCV Processor Design
- GitHub repository
- HDL project
- Mux address: 811
- Extra docs
- Clock: 64000000 Hz

Project Datasheet: RISCV Processor Design

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

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

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

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

Pinout

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

Verilog ring oscillator V3 [812]

- Author: algofoogle (Anton Maurovic)
- Description: sky130 inv_2 ring oscillator with externally-selectable length
- GitHub repository
- HDL project
- Mux address: 812
- Extra docs
- Clock: 0 Hz

What is this?

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

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

This use verilog to instantiate the rings of (an odd number of) `sky130_fd_sc_hd__inv_2` cells.

Pinout

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

Test_project [813]

- Author: Ash
- Description: Full Adder
- GitHub repository
- Wokwi project
- Mux address: 813
- Extra docs
- Clock: 0 Hz

How it works

Simple full adder imnplementation. Three inputs and two outputs

How to test

Run all eight test cases duh

External hardware

Its very simple, not necessary.

Pinout

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

4-Bit Toy CPU [814]

- Author: Stefan Wallentowitz
- Description: This is a simple 4-bit CPU from a popular German textbook
- GitHub repository
- HDL project
- Mux address: 814
- Extra docs
- Clock: 0 Hz

How it works

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

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

How to test

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

Each cycle is driven externally usually as:

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

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

External hardware

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

Pinout

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

RISCV Processor Design [815]

- Author: Nishanth Kotla
- Description: RISCV Processor Design
- GitHub repository
- HDL project
- Mux address: 815
- Extra docs
- Clock: 64000000 Hz

Project Datasheet: RISCV Processor

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

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

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

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

Pinout

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

APA102 to WS2812 Translator [832]

- Author: Squidgeefish
- Description: Convert a 7-LED APA102 stream to a WS2812-compatible one
- GitHub repository
- HDL project
- Mux address: 832
- Extra docs
- Clock: 25000000 Hz

How it works

This is a converter from the SPI-style APA102 LED protocol to the single-line WS2812 protocol.

It's hard-coded to seven LEDs because I needed to set a limit, and this is clearly the simplest possible way to replace the Arduino Micro performing the same task on my 5n4ck3y-7r clone.

It clocks the SPI data on input bit 0 (clock) and bit 1 (data) and waits until it sees a string of 32 low bits to signal a valid start condition. At this point, it starts saving data into an internal shift register, handing that register's contents over to the WS2812 output data feed once all seven LEDs' values have been received. It continues clocking along until recognizing a stop condition (unconditionally 32 bits after the last LED value), at which point it goes back to waiting for a valid start condition. In order to address area concerns, I wound up cutting this down a bit - the internal mirror register was removed entirely, and the SPI reader now also handles discarding the first 8 bits of each 32-bit pixel value. Further tweaks that traded wiring complexity for combinatorics did not make it any better, unfortunately.

I wrote the SPI-parsing and bit-shuffling code from scratch, but the WS2812 output module is lifted from this TT05 submission. I did modify it to read the data stream MSB-first rather than LSB-first since that made my life a lot easier in bit-twiddler land.

Note that the first byte of each APA102 packet encodes an intensity, which I am ignoring since WS2812s do not support such a feature.

How to test

The way I will be testing this is by attaching `ui_in[0]` to SCK and `ui_in[1]` to SDO on a DEFCON badge that used APA102 LEDs. Attach `uo_out[0]` to drive a string of

at least seven WS2812s. I suspect that level shifters will be needed since TinyTapeout ICs run at around 1.8V?

Alternatively, you could probably stream something over in MicroPython.

If you're hand-crafting your packets, a few notes:

- A packet stream must start with a 32-bit start packet (0x00000000)
- APA102s reserve the first byte for intensity: `0b11100000 | <5-bit intensity>`. We're ignoring this completely.
- APA102 color order for the remaining three bytes is Blue, Green, Red.

There is also a random feature added in to fill space - there should be a continuous UART output of "Arglius Barglius" on `uo_out[1]` at approximately 115200 baud; this can be read out with a serial bridge or sufficiently advanced logic analyzer.

External hardware

Some sort of SPI driver is necessary, as is a string of at least seven WS2812 LEDs (or I suppose a logic analyzer can verify it if you're allergic to blinkies).

Pinout

#	Input	Output	Bidirectional
0	APA102_CK	WS2812_OUT	
1	APA102_SD	UART_OUT	
2			
3			
4			
5			
6			
7			

Collatz conjecture brute-forcer [834]

- Author: Vytautas Šaltenis
- Description: Runs a Collatz sequence calculation for a given number
- GitHub repository
- HDL project
- Mux address: 834
- Extra docs
- Clock: 0 Hz

How it works

The module takes a (large) integer number `N` as an input and computes the Collatz sequence until it reaches 1. When it does, it allows reading back two numbers:

1) The orbit length (i.e. the number of steps it took to reach 1)
2) The highest recorded value of the upper 16 bits of the 144-bit internal iterator

The latter number is an indicator for good candidates for computing path records. The non-zero upper bits indicate that the highest iterator value `Mx(N)` is in the range of the previous path records and should be recomputed in the full offline. (Holding on to the entire 144 bits of `Mx(N)` number would be more obvious, but this almost doubles the footprint of the design, hence, this optimisation).

How to test

The module can be in 2 states: IO and COMPUTE. After reset, the chip will be in IO mode. Since the input is intended to be much larger than the available pins, the input number is uploaded one byte at a time, increasing the address of where in the internal 144-bit-wide register that byte should be stored.

Same for reading the output, except that the output numbers are limited to 16-bits each, so it takes much fewer operations to read them.

The full loop of computations works like this:

1) Set input (see below)
2) Pull `start compute` pin to high. The chip will start computations and will pull `compute busy indicator` pin to high
3) Keep reading `compute busy indicator` pin until it gets low again
4) Read the output (see below)

Writing input:

1) Set `write enable` pin to low
2) Wait at least one cycle
3) Expose your input byte to `input0-7`
4) Expose the target address for that byte to `address0-4`
5) Wait at least one cycle
6) Set `write enable` pin to high

Reading output:

1) Set `orbit/max select` pin to low
2) Set `address0-4` to 0
3) Read low byte of orbit length from `output0-7`
4) Set `address0-4` to 1
5) Read high byte of orbit length from `output0-7`
6) Set `orbit/max select` pin to high
7) Repeat steps 2-5 to read the upper `Mx(N)` bits

Pinout

#	Input	Output	Bidirectional
0	input0	output0	address0
1	input1	output1	address1
2	input2	output2	address2
3	input3	output3	address3
4	input4	output4	address4
5	input5	output5	orbit/max select
6	input6	output6	start compute
7	input7	output7	write enable or compute busy indicator

TT09 SKY130 ROM Test [836]

- Author: Sylvain Munaut
- Description: Test of some prototype ROM macros
- GitHub repository
- HDL project
- Mux address: 836
- Extra docs
- Clock: 0 Hz

How it works

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

How to test

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

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

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

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

External hardware

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

The exact testing platform is still TBD.

#	Input	Output	Bidirectional

Pinout

#	Input	Output	Bidirectional
0	addr_in[0]	data[0]	data[5]
1	addr_in1	data1	data[6]
2	addr_in2	data2	data[7]
3	addr_in[3]	data[3]	data[8]
4	addr_in[4]	data[4]	data[9]
5	addr_in[5]		clk of data register
6	addr_in[6]		clk of addr register
7	clk of addr_ld[13:7] register		clk of addr_ld[6:0] register

TT09 SKY130 ROM Test (no LVT variant) [838]

- Author: Sylvain Munaut
- Description: Test of some prototype ROM macros modified to remove LVT implant on bitcell
- GitHub repository
- HDL project
- Mux address: 838
- Extra docs
- Clock: 0 Hz

How it works

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

How to test

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

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

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

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

External hardware

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

The exact testing platform is still TBD.

Pinout

#	Input	Output	Bidirectional
0	addr_in[0]	data[0]	data[5]
1	addr_in1	data1	data[6]
2	addr_in2	data2	data[7]
3	addr_in[3]	data[3]	data[8]
4	addr_in[4]	data[4]	data[9]
5	addr_in[5]		clk of data register
6	addr_in[6]		clk of addr register
7	clk of addr_ld[13:7] register		clk of addr_ld[6:0] register

PID Controller [840]

- Author: Kilian Bender
- Description: Hardware implementation of a naive PID Controller
- GitHub repository
- HDL project
- Mux address: 840
- Extra docs
- Clock: 1000000 Hz

How it works

The PID controller module works by continuously adjusting its output based on the difference between the desired value (setpoint) and the measured value (feedback). It does this using three components:

Proportional Term (P): This term corrects the error in proportion to the current difference between the setpoint and the feedback. It applies an immediate response to reduce the error.

Integral Term (I): This term accumulates the past error over time, helping to eliminate any steady-state error that may persist after the proportional correction.

Derivative Term (D): This term predicts future error by observing the rate of change of the current error, thus providing a damping effect to reduce overshooting.

The controller outputs a signal only in the positive direction. That means that we expect a system that naturally tends towards one point. Regarding a application in heating that means that we are not aiming to cool down the system when overshooting or if the setpoint is higher then our feedback but we just output 0 for control.

How to test

Set different values for setpoint and feedback and observe the output in response to it. Change the setpoint to play around.

External hardware

No specific external hardware is required to test the module in a simulation environment. However, for practical deployment, you may need:

Sensor: A sensor to provide the feedback signal, representing the process variable you wish to control.

Actuator: An actuator driven by the control_out signal to affect the process, such as a motor or a heating element.

Pinout

#	Input	Output	Bidirectional
0	setpoint 0	control_signal 0"	feedback 0
1	setpoint 1	control_signal 1	feedback 1
2	setpoint 2	control_signal 2	feedback 2
3	setpoint 3	control_signal 3	feedback 3
4	setpoint 4	control_signal 4	feedback 4
5	setpoint 5	control_signal 5	feedback 5
6	setpoint 6	control_signal 6	feedback 6
7	setpoint 7	control_signal 7	feedback 7

Frequency Counter SSD1306 OLED [842]

- Author: Pawel Sitarz (embelon)
- Description: Simple Frequency Counter displaying result on SSD1306 SPI OLED
- GitHub repository
- HDL project
- Mux address: 842
- Extra docs
- Clock: 1000000 Hz

How it works

Project measures frequency on ui[0] input by counting pulses during 100ms periods. Measured frequency is then displayed on graphical 128x32 pixels OLED display in form of emulated 7-segment display.

How to test

Internal logic needs 1MHz clock (to be generated by RPi Pico)

- Connect PMOD OLED display to see measurement
- Connect unknown frequency signal to be measured to ui[0]

External hardware

Freqquency is displayed on 128x32 OLED display with SSD1306 controller: PMOD OLED

Pinout

#	Input	Output	Bidirectional
0	clk_x - measured frequency input	OLED nRST	
1		OLED nVBAT	
2		OLED nVDC	
3		OLED nCS	
4		OLED Data/Command	
5		OLED CLK	
6		OLED Data Out	

#	Input	Output	Bidirectional
7			

Basys 3 Over UART Link [844]

- Author: Devin Atkin
- Description: Run the main Basys 3 Peripherals over a 115200 Uart Link
- GitHub repository
- HDL project
- Mux address: 844
- Extra docs
- Clock: 50000000 Hz

How it works

The Basys 3 is a normal board for learning FPGA design or prototyping certain designs. This project runs the main peripherals over a 115200 UART link. This code includes the main block that takes 16 "Led" inputs, 16 "Switch" Outputs, 12 "7 Segment Display" inputs, and 5 "Button" outputs; the block then gives a UART RX and UART TX which are routed to the bi-directional PMOD bus.

How to test

Use the associated PMOD board or interact with the UART. The following are the expected elements on the UART.

- "LD: 0xFFFF" Coming from this design going to the peripheral
- "SW: 0xFFFF" Coming from the peripheral going to the design
- "7S: 0xFFFF" Coming from this design going to the peripheral
- "BT: 0xFFFF" Coming from the peripheral going to the design

External hardware

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

Pinout

#	Input	Output	Bidirectional
0			uart_tx
1			uart_rx
2			uart_tx_ready

#	Input	Output	Bidirectional
3			uart_tx_valid
4			uart_rx_valid
5			uart_rx_ready
6			
7			

Tiny 1-bit AM Radio [846]

- Author: James Ross Sharp
- Description: Syntehsizable 1-bit AM radio core
- GitHub repository
- HDL project
- Mux address: 846
- Extra docs
- Clock: 50000000 Hz

How it works

This project is a Software Defined Radio pipeline for AM radio reception written in verilog. It works using an external comparator as a 1-bit ADC frontend. Demodulation is by the digital equivalent of a how a crystal radio works, i.e. bandpass filter followed by envelope detector. It is based on this Hackaday Project: https://hackaday.io/project/170916-fpga-3-r-1-c-mw-and-sw-sdr-receiver by Alberto Garlassi.

Although this is a fully digital core, but there are plans to make an analog frontend circuit as an analog design in future Tiny Tapeouts, so both designs would be hooked up together to create a radio with few external components.

This project is different from a previously submitted 3x2 tile tiny tapeout core, which used more conventional SDR techniques. This layout reduces area to 1x2 tiles, with a tradeoff in selectivity.

How to test

You need to connect an external comparator and RC network. You will probably need a simple RF amplifier as well. See below for more information.

The core has a SPI interface for setting the demodulation frequency and gain. It consists of a single 24-bit shift register. It has the following format:-

Bit 23	Bits 22 - 20	Bits 19 - 0
Unused	Gain	NCO Phase incr.

The gain can take on the following values:

"Gain" value	Actual Gain
0	x1
1	x2
2	x4
3	x8
4	x16
5 - 7	x32

If the gain is set too high, the demodulated signal will wrap and sound distorted, so adjust the gain down to the minimum needed to hear the station clearly.

The "NCO Phase increment" is the value that is added to the NCO phase every clock cycle. Use the following python code to calculate the value to write, based on the desired carrier frequency:

```
hex(int((1<<20) * <carrier frequency - (455000/<clock_frequency>*50e6)> /
```

E.g., for 936kHz (ABC Radio national Hobart) at 50MHz clock frequency, it would be:

```
> hex(int((1<<20) * 936000 / 50000000))
'0x2767'
```

External hardware

- External comparator
- Resistor bias network
- RC network
- External SPI microcontroller to set station
- RF amplifier

Pinout

#	Input	Output	Bidirectional
0	COMP_IN	COMP_OUT	
1	SPI_MOSI	PWM	
2	SPI_SCK		
3	SPI_CSb		

#	Input	Output	Bidirectional
4			
5			
6			
7			

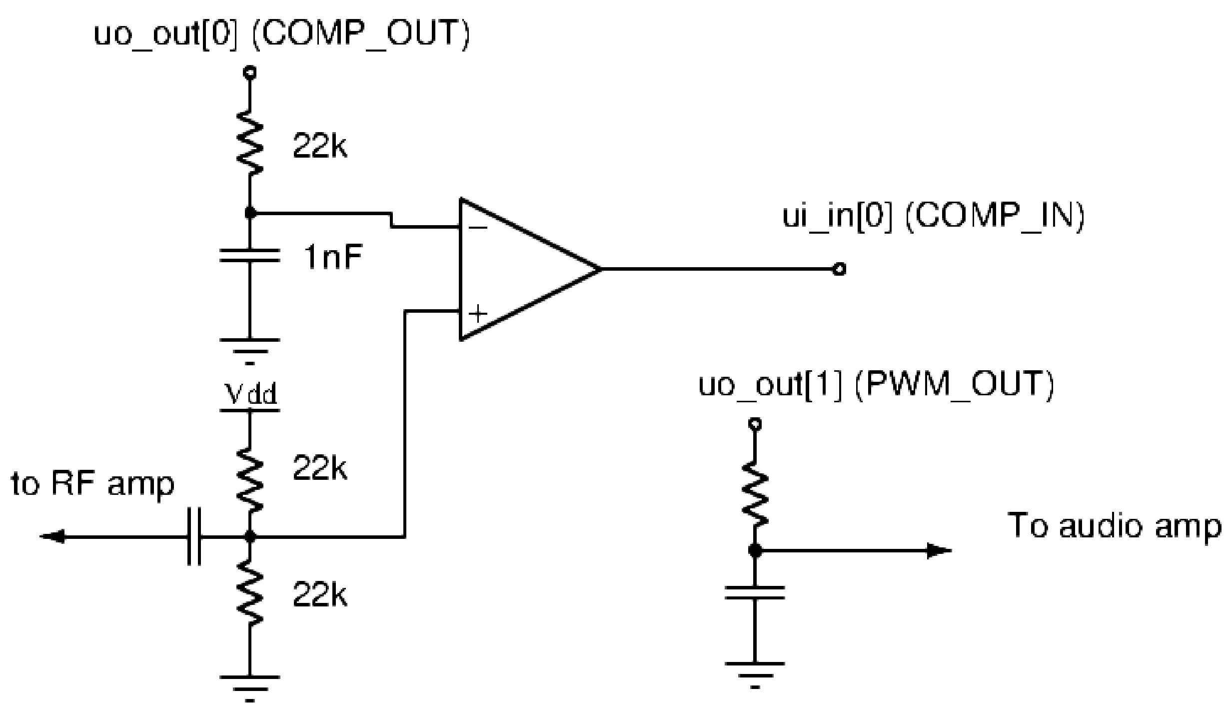


Figure 47: Schematic diagram of external circuitry

Encoder [864]

- Author: Ryan Schrader
- Description: 8x3 Encoder
- GitHub repository
- Wokwi project
- Mux address: 864
- Extra docs
- Clock: 0 Hz

How it works

This Wokwi project is an 8x3 Encoder

How to test

How to test is a WiP

External hardware

External hardware is a WiP

Pinout

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

dummy [865]

- Author: Naveen
- Description: does ntg
- GitHub repository
- Wokwi project
- Mux address: 865
- Extra docs
- Clock: 0 Hz

How it works

Explain how your project works by passing inputs leds will e on

How to test

just pass the input to it Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any 7 segment display,push buttons

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	

First Tapeout Chip - OCR [866]

- Author: Owen Robertson
- Description: Gate Demonstration Chip
- GitHub repository
- Wokwi project
- Mux address: 866
- Extra docs
- Clock: 0 Hz

How it works

This project functions as a demostration device for different gates. IN0 goes through a NOT gate to OUT0. IN1 and IN5 go through an OR gate to OUT1 and OUT5 and they go through a NOR gate to OUT7. IN2, IN3, and IN4 got through a 3-input XOR gate to OUT2 and OUT4. IN6 and IN7 goes through an AND gate to OUT6 and a NAND gate to OUT3.

How to test

Use the 8 different inputs to test the different gate operations on the board. Connecting the outputs to a seven-segment display or to 8 different LEDs can be used to verify the successful gate operations.

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	

#	Input	Output	Bidirectional
7	IN7	OUT7	

sarah's first chip [867]

- Author: sarah
- Description: testing, learning, chipping
- GitHub repository
- Wokwi project
- Mux address: 867
- Extra docs
- Clock: 60 Hz

How it works

Press play

How to test

Test well

External hardware

LED 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	

Half Adder [868]

- Author: Brendon
- Description: Half Adder circuit
- GitHub repository
- Wokwi project
- Mux address: 868
- Extra docs
- Clock: 0 Hz

How it works

Half adder adds 2 one bit integers, and I have 4 of them which produces 4 outputs onto the LED display.

How to test

External hardware

N/A

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 cipher 4 bit key [869]

- Author: sriram nimmala
- Description: a tiny encryption core that encryptes based on input key
- GitHub repository
- HDL project
- Mux address: 869
- Extra docs
- Clock: 5000000 Hz

How it works

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

How to test

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

External hardware

no external harware but memory to send test data

Pinout

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

Kai's Death Adder [870]

- Author: Kai Linsley
- Description: A simple full adder with a sum and carry out output
- GitHub repository
- Wokwi project
- Mux address: 870
- Extra docs
- Clock: 0 Hz

How it works

Takes a 1'bA + 1'bB, outputs Sum and Cout.

How to test

Test by checking design against A + B.

External hardware

LEDS were used for testing.

Pinout

#	Input	Output	Bidirectional
0	A	Sum	
1	B	Cout	
2	Cin		
3			
4			
5			
6			
7			

2 input multiplexor [871]

- Author: chad
- Description: 2 input multiplexor
- GitHub repository
- Wokwi project
- Mux address: 871
- Extra docs
- Clock: 0 Hz

How it works

I made a 2 input multiplexor

How to test

External hardware

nothing

Pinout

#	Input	Output	Bidirectional
0	IN0	OUT0	
1	IN1		
2	IN2		
3			
4			
5			
6			
7			

Kevin Project [872]

- Author: Kevin
- Description: half adder
- GitHub repository
- Wokwi project
- Mux address: 872
- Extra docs
- Clock: 0 Hz

How it works

this is a half adder Explain how your project works

How to test

this is a half adder Explain how to use your project

External hardware

this is a half adder 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	

Tutorial: Simple LIF Neuron [873]

- Author: Zack Bethel
- Description: It simulates a LIF neuron
- GitHub repository
- HDL project
- Mux address: 873
- Extra docs
- Clock: 0 Hz

How it works

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

How to test

do something

External hardware

NA

Pinout

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

#	Input	Output	Bidirectional
6	Input current bit[6]	State variable bit[6]	
7	Input current bit[7]	State variable bit[7]	Spike Bit

Leaky Neuron Network [874]

- Author: Matthew Randall
- Description: makes a leaky neuron network
- GitHub repository
- HDL project
- Mux address: 874
- Extra docs
- Clock: 0 Hz

How it works

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

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

How to test

1. Simulation: Use a Verilog simulator (e.g., ModelSim or Verilator) to test the neuron network. Apply various 5-bit input values to each of the three input neurons and observe when each neuron spikes in response. Check that the output neuron responds as expected to the combined weighted inputs by spiking when the sum of weighted spikes exceeds its threshold.

2. Hardware Testing (if implemented on FPGA): Synthesize the design and program it onto an FPGA. Connect switches or buttons to provide input signals for each neuron. Observe the final spike output on an LED to visualize when the output neuron spikes, or use an oscilloscope to verify spike timings and patterns for more detailed analysis.

External hardware

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

Pinout

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

Neuromorphic Hardware for SNN LSTM [876]

- Author: Hunter Schweiger
- Description: efficient neuromorphic hardware for running a SNN LSTM unit
- GitHub repository
- HDL project
- Mux address: 876
- Extra docs
- Clock: 50000000 Hz

Neuromorphic Hardware for SNN LSTM

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

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

The design operates through several key mechanisms:

1. Membrane Dynamics:
   - Integrates 8-bit input current
   - Applies leaky decay of 1/4 per cycle
   - Resets to 0 after spike
2. Adaptation Mechanism:
   - Learning-enabled threshold adjustment (controlled by uio_in[0])
   - Adaptation increases with each spike
   - Gradual decay when not spiking
3. Output Monitoring:
   - uo_out[7]: Refractory state indicator
   - uo_out[6:0]: Current membrane potential
   - uio_out[7]: Spike output
   - uio_out[6:0]: Spike counter

How to test Testing procedure:

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

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

Pinout

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

#	Input	Output	Bidirectional
5	Input current bit [5]	State variable bit [5]	
6	Input current bit [6]	State variable bit [6]	
7	Input current bit [7]	State variable bit [7]	Spike bit [7]

Project [878]

- Author: calculus
- Description: Exploring Wokwi/GDS
- GitHub repository
- Wokwi project
- Mux address: 878
- Extra docs
- Clock: 0 Hz

How it works

The various inputs will be used to determine if there is success on the output.

How to test

Change inputs to change the output

External hardware

No extra hardware needed.

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	

Hardware UTF Encoder/Decoder [897]

- Author: Rebecca G. Bettencourt
- Description: Converts Unicode code points between UTF-8, UTF-16, and UTF-32.
- GitHub repository
- HDL project
- Mux address: 897
- Extra docs
- Clock: 0 Hz

How it works

This project contains hardware logic to convert between the UTF-8, UTF-16, and UTF-32 encodings for Unicode text.

It will detect and raise an error signal on overlong encodings, out of range code point values, and invalid byte sequences.

(You can optionally disable range checking if you wish to use the original UTF-8 spec that supports values up to 0x7FFFFFFF.)

Basic operation

- In the initial state, all dedicated inputs should be set HIGH.
- At any time, set /RESET (rst_n) LOW and pulse CLK to reset all inputs and outputs to initial state.
- At any time, set /ROUT (input 0) LOW and pulse CLK to seek to the beginning of the output.
- You can set ERRS or /PROPS (input 1) HIGH to get an error status on the dedicated outputs.
- You can set ERRS or /PROPS (input 1) LOW to get character properties on the dedicated outputs.
- You can set CHK (input 2) HIGH to raise an error signal when the code point value is out of range (0x110000).
- You can set CHK (input 2) LOW to ignore out of range code point values and encode/decode values up to 0x7FFFFFFF.
- You can set CBE (input 3) HIGH to specify big endian order for UTF-32 and UTF-16 input and output.
- You can set CBE (input 3) LOW to specify little endian order for UTF-32 and UTF-16 input and output.

Inputting UTF-32

1. Set READ or /WRITE (input 4) LOW.
2. Set /CIO (input 5, character I/O) LOW.
3. Set bidirectional I/O to the first byte of the UTF-32 word and pulse CLK.
4. Set bidirectional I/O to the second byte of the UTF-32 word and pulse CLK.
5. Set bidirectional I/O to the third byte of the UTF-32 word and pulse CLK.
6. Set bidirectional I/O to the fourth byte of the UTF-32 word and pulse CLK.
7. Set /CIO (input 5, character I/O) HIGH.
8. Set READ or /WRITE (input 4) HIGH.
9. If READY (output 0) is HIGH and ERROR (output 5) is LOW, the input and output are both valid.
10. If READY (output 0) is LOW or ERROR (output 5) is HIGH, the input was out of range (0x110000 or, if CHK is LOW, 0x80000000).

Inputting UTF-16

1. Set ERRS or /PROPS (input 1) LOW.
2. Set READ or /WRITE (input 4) LOW.
3. Set /UIO (input 6, UTF-16 I/O) LOW.
4. Set bidirectional I/O to the first byte of the first UTF-16 word and pulse CLK.
5. Set bidirectional I/O to the second byte of the first UTF-16 word and pulse CLK.
6. If HIGHCHAR (output 3) is LOW, skip to step 9.
7. Set bidirectional I/O to the first byte of the second UTF-16 word and pulse CLK.
8. Set bidirectional I/O to the second byte of the second UTF-16 word and pulse CLK.
9. Set /UIO (input 6, UTF-16 I/O) HIGH.
10. Set READ or /WRITE (input 4) HIGH.
11. Set ERRS or /PROPS (input 1) HIGH.
12. If READY (output 0) is HIGH and ERROR (output 5) is LOW, the input and output are both valid.
13. If RETRY (output 1) is HIGH, the first word was a high surrogate but the second word was not a low surrogate. The output will be the high surrogate only; the last word will need to be processed again.

Inputting UTF-8

1. Set READ or /WRITE (input 4) LOW.
2. Set /BIO (input 7, byte I/O) LOW.
3. Set bidirectional I/O to the current byte of the UTF-8 sequence and pulse CLK.

4. Repeat step 3 until READY (output 0) or ERROR (output 5) is HIGH.
5. If READY (output 0) is HIGH and ERROR (output 5) is LOW, the input and output are both valid.
6. If RETRY (output 1) is HIGH, the UTF-8 sequence was truncated (not enough continuation bytes). The output will be the truncated sequence only; the last byte will need to be processed again.
7. If INVALID (output 2) is HIGH, the UTF-8 sequence was a single continuation byte or invalid byte (0xFE or 0xFF).
8. If OVERLONG (output 3) is HIGH, the UTF-8 sequence was an overlong encoding.
9. If NONUNI (output 4) is HIGH, the UTF-8 sequence was out of range (0x110000).

Outputting UTF-32

1. Set READ or /WRITE (input 4) HIGH.
2. Set /CIO (input 5, character I/O) LOW.
3. Pulse CLK and read the first byte of the UTF-32 word from the bidirectional I/O.
4. Pulse CLK and read the second byte of the UTF-32 word from the bidirectional I/O.
5. Pulse CLK and read the third byte of the UTF-32 word from the bidirectional I/O.
6. Pulse CLK and read the fourth byte of the UTF-32 word from the bidirectional I/O.
7. Set /CIO (input 5, character I/O) HIGH.
8. If the UTF-32 word is within range, the input and output are both valid.
9. If the UTF-32 word is not within range, then the input was either incomplete or invalid.

Outputting UTF-16

1. Set READ or /WRITE (input 4) HIGH.
2. If UEOF (output 6) is HIGH, then the input was either incomplete or invalid.
3. Set /UIO (input 6, UTF-16 I/O) LOW.
4. Pulse CLK and read the next byte of the UTF-16 sequence from the bidirectional I/O.
5. Repeat step 4 until UEOF (output 6) is HIGH.
6. Set /UIO (input 6, UTF-16 I/O) HIGH.

Outputting UTF-8

1. Set READ or /WRITE (input 4) HIGH.
2. If BEOF (output 7) is HIGH, then the input was either incomplete or invalid.
3. Set /BIO (input 7, byte I/O) LOW.
4. Pulse CLK and read the next byte of the UTF-8 sequence from the bidirectional I/O.
5. Repeat step 4 until BEOF (output 7) is HIGH.
6. Set /BIO (input 7, byte I/O) HIGH.

Error status

When ERRS or /PROPS (input 1) is HIGH, the dedicated outputs will be:

#	Name	Meaning
0	READY	The input and output are complete sequences.
1	RETRY	The previous input was invalid or the start of another sequence and was ignored. Process the output, reset, and try the previous input again.
2	INVALID	The input and output are invalid.
3	OVERLONG	The UTF-8 input was an overlong sequence.
4	NONUNI	The code point value is out of range (0x110000). (This is set independently of the CHK input; the CHK input only changes whether this counts as an error.)
5	ERROR	Equivalent to (RETRY or INVALID or OVERLONG or (NONUNI and CHK)).

If all of these outputs are LOW, the accumulated input is incomplete and more input is required (underflow).

Character properties

When ERRS or /PROPS (input 1) is LOW, the dedicated outputs will be:

#	Name	Meaning
0	NORMAL	The code point value is valid and not a C0 or C1 control character, surrogate, private use character, or noncharacter.
1	CONTROL	The code point value is valid and a C0 or C1 control character (0x00-0x1F or 0x7F-0x9F).
2	SURROGATE	The code point value is valid and a UTF-16 surrogate (0xD800-0xDFFF).
3	HIGHCHAR	The code point value is valid and either a high surrogate (0xD800-0xDBFF) or a non-BMP character (0x10000).
4	PRIVATE	The code point value is valid and either a private use character (0xE000-0xF8FF, 0xF0000) or the high surrogate of a private use character (0xDB80-0xDBFF).
5	NONCHAR	The code point value is valid and a noncharacter (0xFDD0-0xFDEF or the last two code points of any plane).

If all of these outputs are LOW, there is no valid code point in the output.

How to test

The `test.py` file covers a comprehensive set of test cases which are listed in a separate file to avoid bloating the TT09 manual.

External hardware

Any device that needs to process Unicode text.

Pinout

#	Input	Output	Bidirectional
0	/ROUT	READY; NORMAL	I/O LSB
1	ERRS, /PROPS	RETRY; CONTROL	I/O
2	CHK	INVALID; SURROGATE	I/O
3	CBE, /CLE	OVERLONG; HIGHCHAR	I/O
4	READ, /WRITE	NONUNI; PRIVATE	I/O
5	/CIO	ERROR; NONCHAR	I/O
6	/UIO	UEOF	I/O
7	/BIO	BEOF	I/O MSB

BINCounterAndGates [899]

- Author: conrad franke
- Description: Binary counter with io gates
- GitHub repository
- Wokwi project
- Mux address: 899
- Extra docs
- Clock: 1 Hz

How it works

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

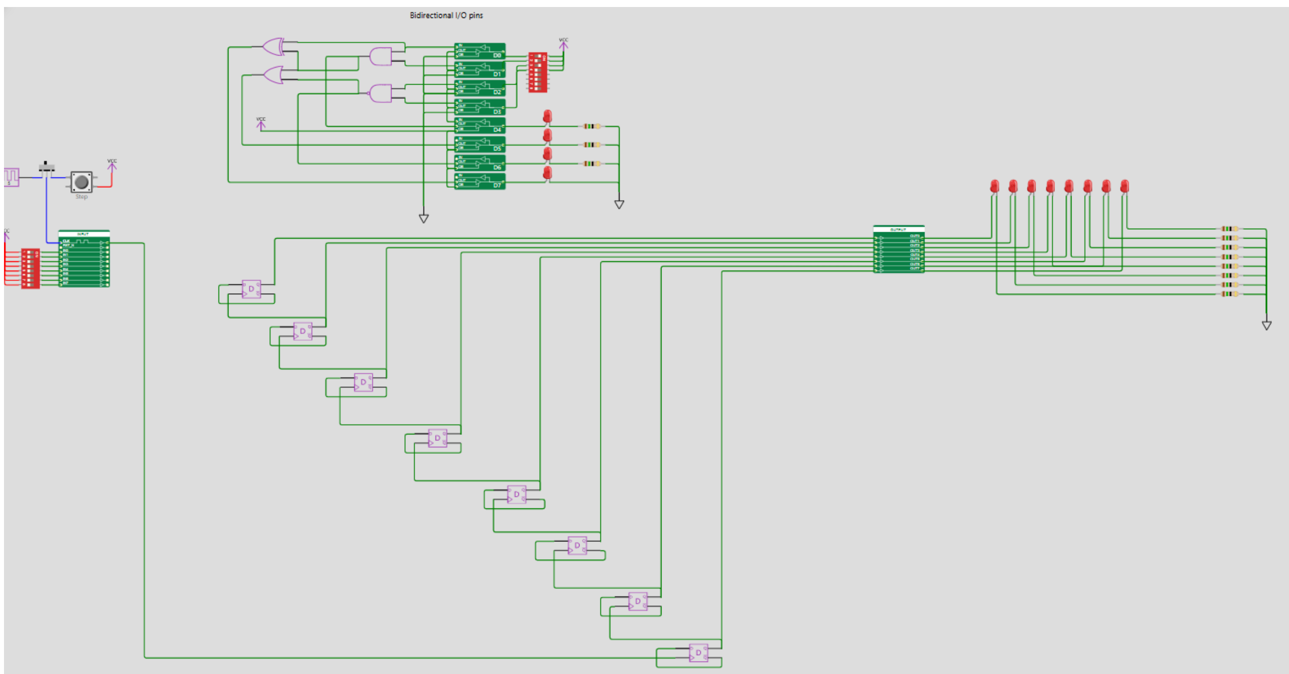


Figure 48: image

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

How to test

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

External hardware

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

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

Update

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

Pinout

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

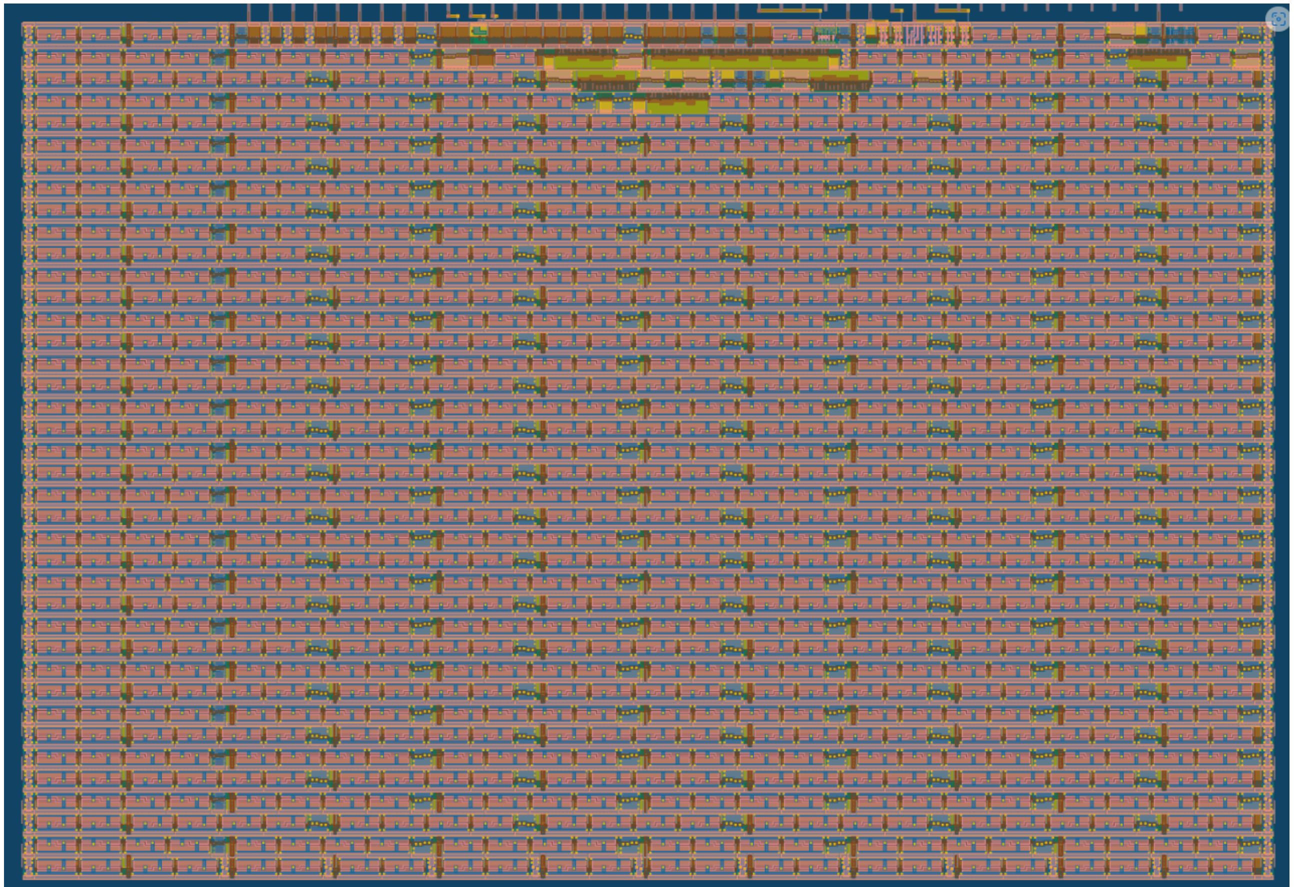


Figure 49: image

Color Bars [901]

- Author: Rebecca G. Bettencourt
- Description: VGA demo resembling NTSC color bars
- GitHub repository
- HDL project
- Mux address: 901
- Extra docs
- Clock: 0 Hz

How it works

Displays a test pattern on the screen resembling NTSC color bars. Optionally, you can add a station ID, make the ID scroll, and make the color bars scroll.

The colors displayed are NOT accurate to actual NTSC color bars. This cannot be used to adjust NTSC video equipment; it's just for fun.

Figure 50: Color bars with station ID

How to test

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

- `show_id` (`ui_in[0]`) to add a station ID,
- `custom_id` (`ui_in[1]`) to use a custom ID (address on `uio_out`, data on `ui_in[7:4]`),
- `scroll_id` (`ui_in[2]`) to make the ID scroll,
- `scroll_bars` (`ui_in[3]`) to make the color bars scroll.

External hardware

TinyVGA PMOD

Pinout

#	Input	Output	Bidirectional
0	show_id	R1	A0 (custom id)
1	custom_id	G1	A1 (custom id)
2	scroll_id	B1	A2 (custom id)
3	scroll_bars	VSync	A3 (custom id)
4	D3 (custom id)	R0	A4 (custom id)
5	D2 (custom id)	G0	A5 (custom id)
6	D1 (custom id)	B0	A6 (custom id)
7	D0 (custom id)	HSync	A7 (custom id)

Fuzzy Search Engine [903]

- Author: Peter Nørlund
- Description: A levenshtein based fuzzy search engine
- GitHub repository
- HDL project
- Mux address: 903
- Extra docs
- Clock: 50000000 Hz

How it works

tt09-levenshtein is a fuzzy search engine which can find the best matching word in a dictionary based on levenshtein distance.

Fundamentally its an implementation of the bit-vector levenshtein algorithm from Heikki Hyyrö's 2003 paper with the title *A Bit-Vector Algorithm for Computing Levenshtein and Damerau Edit Distances*.

Architecture The overall architecture is a Wishbone Classic system with two masters (The levenshtein engine and an SPI controlled master) and two slaves (The levenshtein engine and a QSPI SRAM controller).

Using the SPI interface, you store a dictionary and some bitvectors representing a search word in SRAM and then configures and activates the engine. The engine will then read the dictionary and bitvectors from the SRAM and, ultimately store the index and distance of the word in the dictionary with the lowest levenshtein distance in registers which can be read by the user.

SPI The device is organized as a wishbone bus which is accessed through commands on an SPI bus.

The maximum SPI frequency is 25% of the master clock (12.5MHz when the chip is running at 50MHz).

The bus uses SPI mode 3 (CPOL=1, CPHA=1)

Input bytes:

Byte	Bit	Description
0	7	READ=0 WRITE=1
0	6-0	Address bit 22-16

Byte	Bit	Description
1	7-0	Address bit 15-8
2	7-0	Address bit 7-0
3	7-0	Byte to write if WRITE, otherwise ignored

Output bytes:

Byte	Bit	Description
0	7-0	Byte read if READ, otherwise just 0x00

Since the SPI bridges to a wishbone bus which is shared by another master and because register and SRAM have different latencies, the response time is variable.

While the bus is working, the output bits will be zero. The final output byte will be preceeded by a one-bit.

Note that this means that the value 0x5A can appear 8 different ways on the SPI bus:

```
01 5A    0000000 1 01011010
02 B4    000000 1 01011010 0
05 68    00000 1 01011010 00
0A D0    0000 1 01011010 000
15 A0    000 1 01011010 0000
2B 40    00 1 01011010 00000
56 80    0 1 01011010 000000
AD 00    1 01011010 00000000
```

Memory Layout As indicated by the SPI protocol, the address space is 23 bits.

The address space is basically as follows:

Address	Size	Access	Identifier
0x000000	1	R/W	CTRL
0x000001	1	R/W	SRAM_CTRL
0x000002	1	R/W	LENGTH
0x000003	1	R/O	MAX_LENGTH
0x000004	2	R/O	INDEX
0x000006	1	R/O	DISTANCE
0x000200	512	R/W	VECTORMAP

Address	Size	Access	Identifier
0x000400	8M	R/W	`DICT`

CTRL

The control register is used to start the engine and see when it has completed.

The layout is as follows:

Bits	Size	Access	Description
0	1	R/W	Enable flag
1-7	7	R/O	Not used

Set the enable flag to start the engine. When the engine is finished, the enable flag is changed to `0`

SRAM_CTRL

Controls the SRAM

Bits	Size	Access	Description
0-1	2	R/W	Chip select
2-7	6	R/O	Not used

The chip select flag controls which chip select is used on the PMOD when accessing SRAM

Value	Pin	Notes
0	*None*	Default value
1	CS	Uses the default CS on the PMOD (Pin 1). Compatible with Machdyne's QQSPI PSRAM
2	CS2	Uses CS2 on the PMOD (pin 6). Compatible with mole99's QSPI Flash/(P)SRAM
3	CS3	Uses CS3 on the PMOD (pin 7). Compatible with mole99's QSPI Flash/(P)SRAM

LENGTH

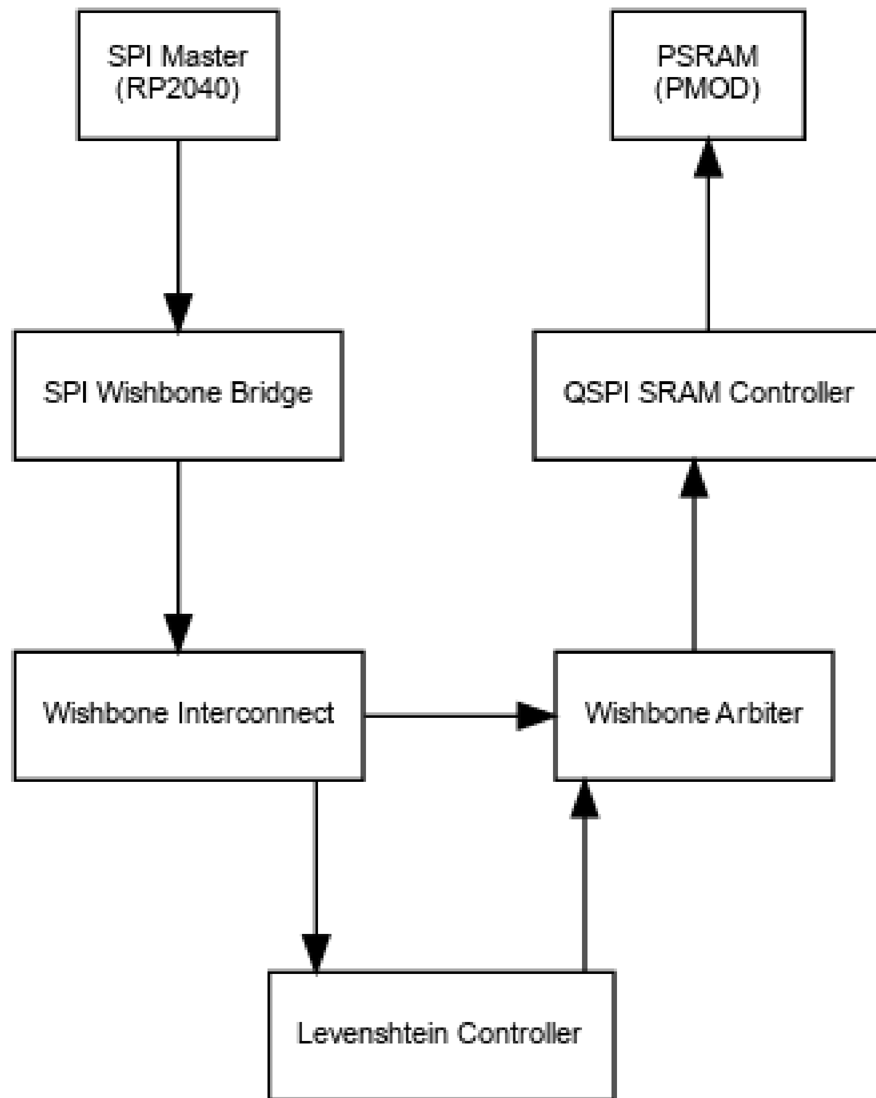


Figure 51: image

Bits	Size	Access	Description
0-7	8	R/W	Word length minus 1

Used to indicate the length of the search word. Note that the word cannot be empty and it cannot exceed 16 characters.

MAX_LENGTH

Bits	Size	Access	Description
0-7	8	R/O	Max word length supported minus 1

This field allows for applications to dynamically detect the size of the bit vector.

DISTANCE

When the engine has finished executing, this address contains the levenshtein distance of the best match.

INDEX

When the engine has finished executing, this address contains the index of the best word from the dictionary in big endian byte order.

VECTORMAP

The vector map must contain the corresponding bitvector for each input byte in the alphabet.

If the search word is `application`, the bit vectors will look as follows:

Letter	Index	Bit vector
`a`	`0x61`	`16'b00000000_01000001` (`a_____a____`)
`p`	`0x70`	`16'b00000000_00000110` (`_pp________`)
`l`	`0x6C`	`16'b00000000_00001000` (`___l_______`)
`i`	`0x69`	`16'b00000001_00010000` (`____i___i__`)
`c`	`0x63`	`16'b00000000_00100000` (`_____c_____`)
`t`	`0x74`	`16'b00000000_10000000` (`_______t___`)
`o`	`0x6F`	`16'b00000010_00000000` (`_________o_`)
`n`	`0x6E`	`16'b00000100_00000000` (`__________n`)
`*`	`*`	`16'b00000000_00000000` (`___________`)

Each vector is 16 bits in bit endian byte order.

The vectormap is stored in SRAM so the values are indetermined at power up and must be cleared.

DICT

The word list.

The word list is stored of a sequence of words, each encoded as a sequence of 8-bit characters and terminated by the byte value `0x00`. The list itself is terminated with the byte value `0x01`.

Note that the algorithm doesn't care about the particular characters. It only cares if they are identical or not, so even though the algorithm doesn't support UTF-8 and is limited to a character set of 254 characters, ignoring Asian alphabets, a list of words usually don't contain more than 254 distinct characters, so you can practially just map lettters to a value between 2 and 255.

How to test

You can compile the client as follows:

```
mkdir -p build
cmake -G Ninja -B build .
cmake --build build
```

Next, you can run the test tool:

```
# Machdyne QQSPI PSRAM
./build/client/client --interface tt --test --verify-dictionary --verify-

# mole99 PSRAM
./build/client/client --interface tt --cs cs2 --test --verify-dictionary
```

This will load 1024 words of random length and characters into the SRAM and then perform a bunch of searches, verifying that the returned result is correct.

External hardware

To operate, the device needs a QSPI PSRAM PMOD. The design is tested with the QQSPI PSRAM PMOD from Machdyne, but any memory PMOD will work as long as it supports:

- WRITE QUAD with the command `0x38` in 1S-4S-4S mode and no latency

- FAST READ QUAD with the command `0xE8` in 1S-4S-4S mode and 6 wait cycles
- 24-bit addresses
- Uses pin 0, 6, or 7 for `SS#`.
- Must be able to run at half the clock speed of the TT chip.

Note that this makes it incompatible with the spi-ram-emu project for the RP2040.

Pinout

#	Input	Output	Bidirectional
0			SRAM QSPI CS
1			SRAM QSPI SIO0/MOSI
2			SRAM QSPI SIO1/MISO
3			SRAM QSPI SCK
4	SPI SS#		SRAM QSPI SIO2
5	SPI SCK		SRAM QSPI SIO3
6	SPI MOSI		SRAM QSPI CS2
7		SPI MISO	SRAM QSPI CS3

TT09Ball GDS Art [905]

- Author: Rebecca G. Bettencourt
- Description: THE STRONGEST ROM and GDS art
- GitHub repository
- HDL project
- Mux address: 905
- Extra docs
- Clock: 0 Hz

How it works

An octal counter is connected to THE STRONGEST 8-byte ROM, which is connected to the dedicated output, which is connected to the seven-segment display, to provide a carrier for THE STRONGEST GDS art.

You can bypass the counter by setting `ui_in[3]` high and putting the address on `ui_in[2:0]`.

How to test

1. Set `ui_in[3]` low.
2. Set reset low and pulse clk.
3. Set reset high and pulse clk to change the LED display.
4. You should see, in order: `9` blank `C` `I`   `0` blank.
5. Set `ui_in[3]` high.
6. Set `ui_in[2:0]` to the values 0 through 7.
7. You should see, in order: `9` blank `C` `I`   `0` blank.

External hardware

Seven-segment LED display and/or chip decapping tools, depending on how destructive you want to be.

#	Input	Output	Bidirectional

Pinout

#	Input	Output	Bidirectional
0	A0	a	D0
1	A1	b	D1
2	A2	c	D2
3	address mode	d	D3
4		e	D4
5		f	D5
6		g	D6
7		dp	D7

Simon Says memory game [907]

- Author: Uri Shaked
- Description: Repeat the sequence of colors and sounds to win the game
- GitHub repository
- HDL project
- Mux address: 907
- Extra docs
- Clock: 50000 Hz

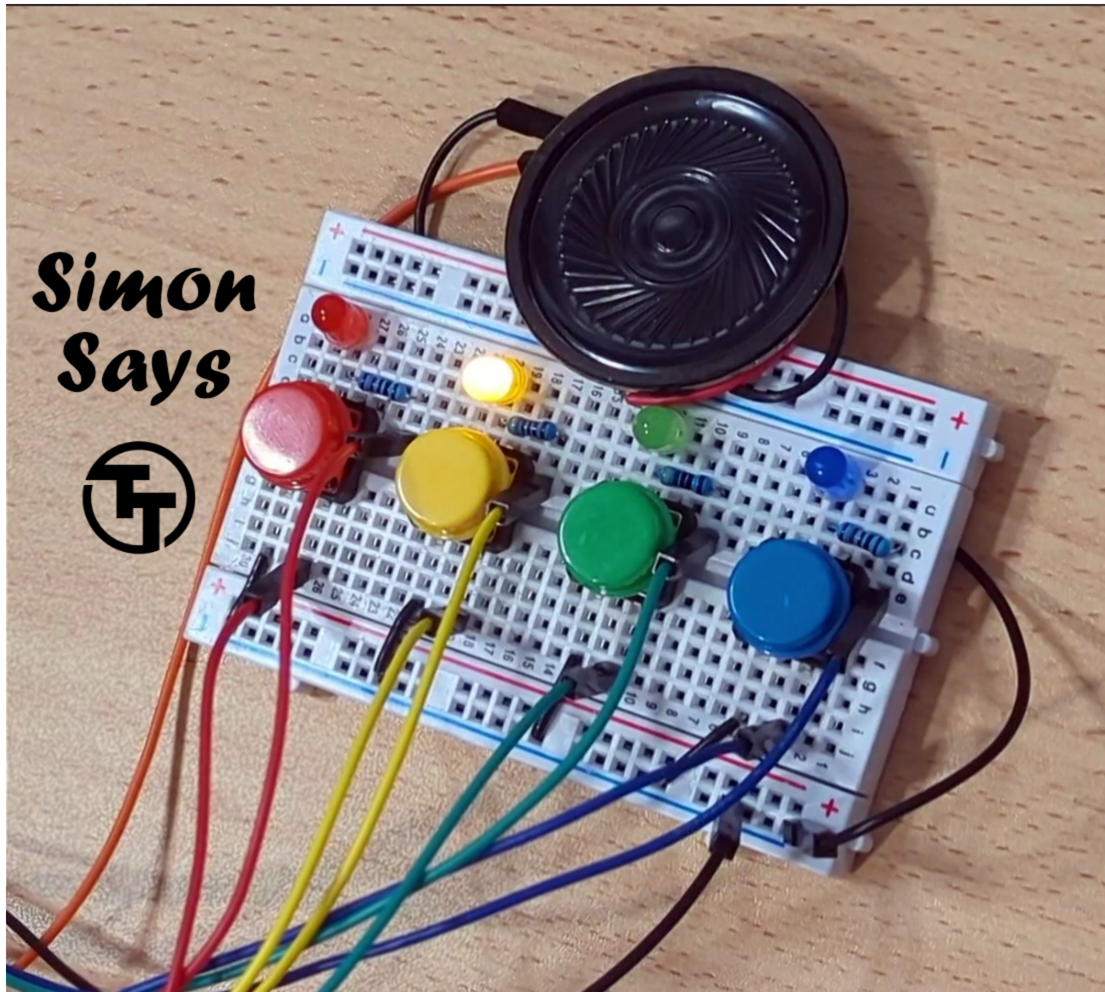


Figure 52: Simon Says Game

How it works

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

In each turn, the game will play the sequence, and then wait for the user to repeat the sequence by pressing the buttons according to the color sequence. If the user repeated the sequence correctly, the game will play a "leveling-up" sound, add a new color at the end of the sequence, and move to the next turn.

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

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

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

#	Input	Output	Bidirectional
2	btn3	led3	seg_c
3	btn4	led4	seg_d
4	seginv	speaker	seg_e
5		dig1	seg_f
6		dig2	seg_g
7			

TT09Ball VGA Screensaver [909]

- Author: Rebecca G. Bettencourt; Uri Shaked
- Description: THE STRONGEST DVD style screen saver (640x480, TinyVGA Pmod)
- GitHub repository
- HDL project
- Mux address: 909
- Extra docs
- Clock: 0 Hz

How it works

Displays THE STRONGEST bouncing logo on the screen, with animated color gradient.

Figure 53: THE STRONGEST screensaver

How to test

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

- `tile` (ui_in[0]) to repeat the logo and tile it across the screen,
- `solid_color` (ui_in1) to use a solid color instead of an animated gradient.

External hardware

TinyVGA PMOD

Pinout

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

ChatGPT-generated Spiking Neural Network with Delays [910]

- Author: Paola Vitolo
- Description: ChatGPT-generated Spiking Neural Network with Delays
- GitHub repository
- HDL project
- Mux address: 910
- Extra docs
- Clock: 50000000 Hz

Overview

How it works

This project implements 18 programmable digital LIF neurons with programmable delays and a total of 144 synapsis. The neurons are arranged in 3 layers (8 inputs + FC (8 neurons) + FC (8 neurons) + FC (2 neurons) +2 outputs). Spikes_in directly maps to the inputs of the first layer neurons. When an input spike is received, it is first multiplied by an 2-bit weight, programmable from an SPI interface, 1 per input neuron. This value is then added to the membrane potential of the respective neuron. When the first layer neurons activate, its pulse is routed to each of the 8 neurons in the next layer. There are 144 (8x8+8x8+8x2) programmable weights describing the connectivity between the input spikes and the first layer (64 weights=8x8), the first and second layers (64 weights=8x8), and the second and third layers (16 weights=8x2).

Through a configurable selection signal via SPI, it is possible to read any of the membrane potentials from any neuron in any layer, or the output spikes from any layer.

How to test

After reset, program the neuron threshold, decay rate, and refractory period. Additionally program the first, second, and third layer weights and delays. Once programmed activate spikes_in to represent input data, track spikes_out synchronously.

Memory Map Overview Each parameter (decay, refractory period, membrane potential threshold, weights, and delays) and each configuration signal (value for the configurable clock divider and output select signal) is accessible via SPI in specific byte addresses. The memory is organized as follows:

Parameter	Bit Range / Byte	Address (Hex)	Address (Decimal)	Description
decay	5:0 bits in 2nd byte	0x00	0	Decay configuration parameter
refractory_period	5:0 bits in 3rd byte	0x01	1	Refractory period parameter
threshold	5:0 bits in 4th byte	0x02	2	Membrane potential threshold
div_value	5th byte	0x03	3	Division value for clock divider
weights	36 bytes (5th to 40th)	0x04 - 0x27	4 - 39	Synaptic weights
delays	72 bytes (41st to 112th)	0x28 - 0x6F	40 - 111	Synaptic delay
output_config	8 bits in 113th byte	0x70	112	Output select signal

Simulations

External hardware

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

Pinout

#	Input	Output	Bidirectional
0	input_spike[0]	output[0]	CS
1	input_spike1	output1	MOSI
2	input_spike2	output2	MISO
3	input_spike[3]	output[3]	SCLK
4	input_spike[4]	output[4]	input_ready
5	input_spike[5]	output[5]	output_ready
6	input_spike[6]	output[6]	SNN_en
7	input_spike[7]	output[7]	spi_instruction_done

32x8 LED Matrix Animation [911]

- Author: Ayla Lin, Pavit Thakur, Lauren Low
- Description: An animation using a 32x8 matrix, switching between a beaver logo and the letters 'BWSI'
- GitHub repository
- HDL project
- Mux address: 911
- Extra docs
- Clock: 33000000 Hz

How it works

This project contains 3 components:

- SPI for sending instructions and bitmaps to MAX7219/7221.
- ROM for storing bitmaps to display.
- Controller for instructing SPI and ROM.

How to test

Test using FPGA and a breadboard

External hardware

- 32x8 LED Matrix.
- MAX7219/7221.
- 5V battery.

Pinout

#	Input	Output	Bidirectional
0		spi_clk	
1		spi_cs_n	
2		spi_mosi	
3			
4			
5			

#	Input	Output	Bidirectional
6			
7			

8b10b decoder and multiplier [961]

- Author: Mike Bell
- Description: Test for high speed cell library - 8b10b decoder and multiplier
- GitHub repository
- HDL project
- Mux address: 961
- Extra docs
- Clock: 0 Hz

What is it?

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

How it works

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

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

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

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

How to test

Send 8b10b encoded data streams, check the outputs.

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

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

External hardware

None required

Pinout

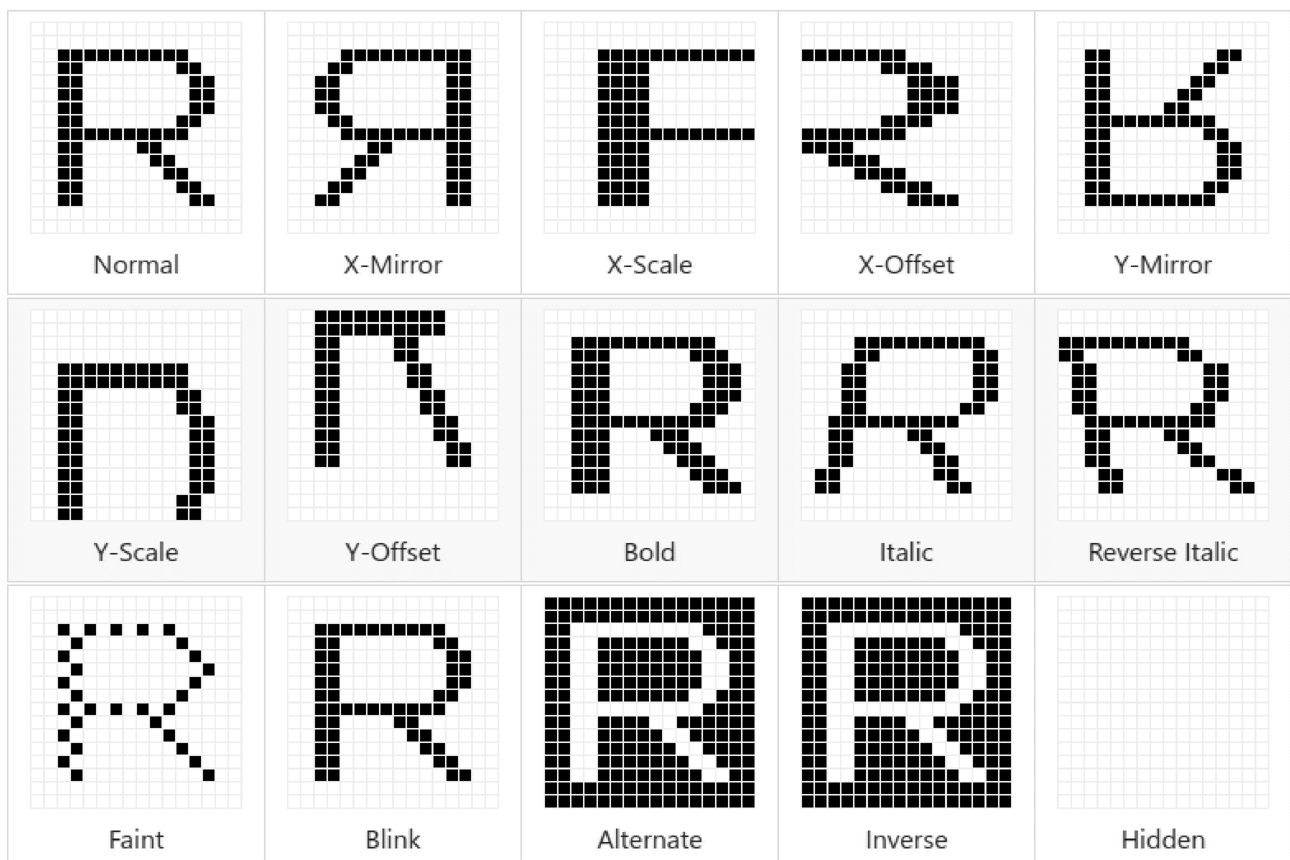
#	Input	Output	Bidirectional
0	A 8b10b in	Out 0	Out 8
1	B 8b10b in	Out 1	Out 9
2	Decoder status	Out 2	Out 10
3	Multiply result	Out 3	Out 11
4	Multiply result (update gated)	Out 4	Out 12
5	Decoded values (registered)	Out 5	Out 13
6	Decoded values (unregistered)	Out 6	Out 14
7	Bidir output enable	Out 7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

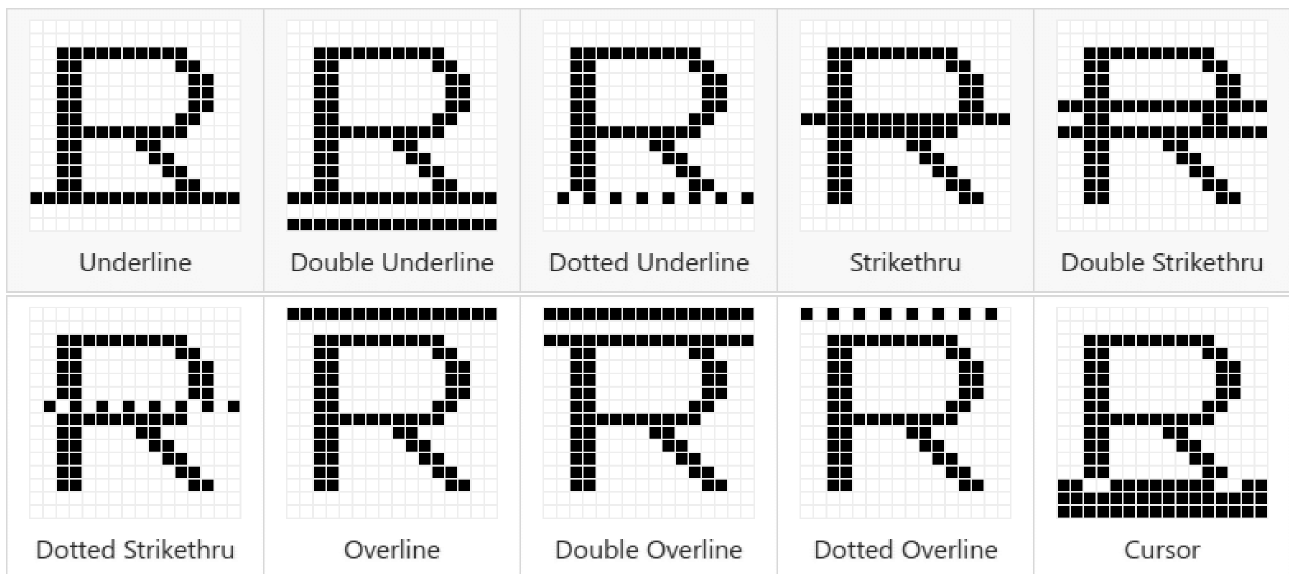
Styler [963]

- Author: Rebecca G. Bettencourt
- Description: 16x16 bitmap manipulation based on text mode attributes.
- GitHub repository
- HDL project
- Mux address: 963
- Extra docs
- Clock: 0 Hz

How it works

The styler chip is used to transform a 16x16 character glyph bitmap based on a set of text mode attributes. It consists of a 4-bit scanline register, an 8-bit control register, a 16-bit bitmap register, and a 25-bit attribute register. Additionally, three independent input lines are used to control polarity of faint text (even or odd pixels), text and cursor blink rate, and cursor position.





Typical use of the styler chip follows these steps:

1. Set output enable (input 6) HIGH and write enable (input 7) LOW.
2. Set the address (inputs 0-2) to 0.
3. Set the bidirectional pins to the physical scanline number.
4. Pulse `clk`.
5. Set output enable (input 6) LOW and write enable (input 7) HIGH.
6. Read the logical scanline number from the bidirectional pins.
7. Set output enable (input 6) HIGH and write enable (input 7) LOW.
8. Set the address (inputs 0-2) to 2.
9. Set the bidirectional pins to the right half of the row of the character bitmap corresponding to the logical scanline number.
10. Pulse `clk`.
11. Set the address (inputs 0-2) to 3.
12. Set the bidirectional pins to the left half of the row of the character bitmap corresponding to the logical scanline number.
13. Pulse `clk`.
14. Set output enable (input 6) LOW and write enable (input 7) HIGH.
15. Set the address (inputs 0-2) to 2.
16. Read the right half of the final character bitmap from the bidirectional pins.
17. Set the address (inputs 0-2) to 3.
18. Read the left half of the final character bitmap from the bidirectional pins.

You can also read from the dedicated output pins without changing output enable or write enable.

The register layout is as follows:

Address	Bits	Description
0	0-3	Input: physical scanline number; output: logical scanline number.
0	4-7	Input: ignored; output: 0.
1	0	Show cursor at bottom of character cell.
1	1	Show cursor at top of character cell.
1	2	Enable cursor blink.
1	3	Enable cursor.
1	4	Enable character underline, strikethrough, overline attributes.
1	5	Enable character blink, alternate attributes.
1	6	Reserved.
1	7	Reserved.
2	0-7	Right half of character glyph bitmap.
3	0-7	Left half of character glyph bitmap.
4	0	X offset. (Determines which half of a double-width character.)
4	1	Double width.
4	2	Y offset. (Determines which half of a double-height character.)
4	3	Double height.
4	4	X premirror (flip input bitmap horizontally).
4	5	X postmirror (flip output bitmap horizontally).
4	6	Y premirror (invert physical scanline).
4	7	Y postmirror (invert logical scanline).
5	0	Bold.
5	1	Faint.
5	2	Italic.
5	3	Reverse italic.
5	4	Blink (text only, VT100-style).
5	5	Alternate (text and background, Apple II-style).
5	6	Inverse.
5	7	Hidden.
6	0	Underline.
6	1	Double underline.
6	2	Dotted underline.
6	3	Strikethrough.
6	4	Double strikethrough.
6	5	Dotted strikethrough.
6	6	Overline.
6	7	Double overline.
7	0	Dotted overline.
7	1-7	Input: ignored; output: 0.

The input pin assignments are as follows:

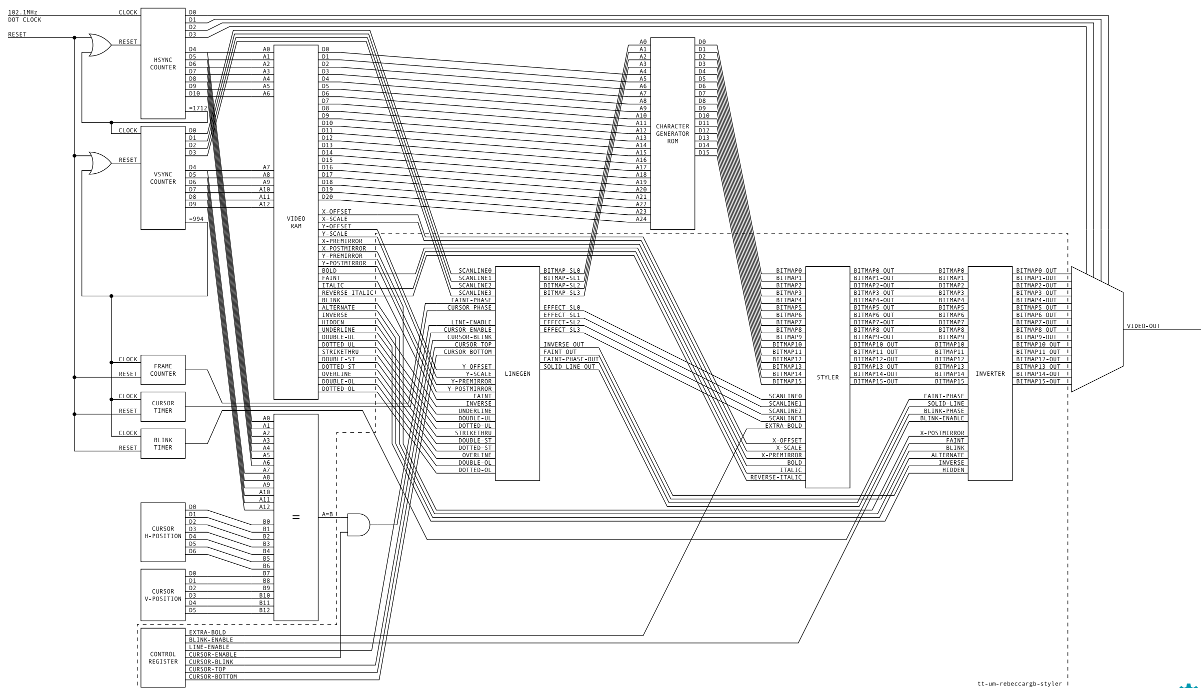
Pin	Description
0	A0 (address line 0).
1	A1 (address line 1).
2	A2 (address line 2).
3	Faint text polarity (even or odd pixels).
4	Blink phase.
5	Cursor enable.
6	/OE (output enable).
7	/WE (write enable).

How to test

The `test.py` file covers a variety of test cases.

External hardware

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



Pinout

#	Input	Output	Bidirectional
0	A0 (address)	D0	D0
1	A1 (address)	D1	D1
2	A2 (address)	D2	D2
3	faint text polarity	D3	D3
4	blink phase	D4	D4
5	cursor enable	D5	D5
6	/OE (output enable)	D6	D6
7	/WE (write enable)	D7	D7

VGA Timing Experiments [965]

- Author: Rebecca G. Bettencourt
- Description: Configurable VGA signal generator for experimentation purposes.
- GitHub repository
- HDL project
- Mux address: 965
- Extra docs
- Clock: 0 Hz

How it works

Generates VGA signals. All signal timings (display area, front porch, back porch, hsync, vsync, polarity) are fully configurable and several test patterns are included to enable experimentation.

How to test

Connect to a VGA monitor. Set `ui_in[3:0]` all LOW and pulse `ui_in[7]` to set signal timings to a "known good" configuration of 640×480 at 60Hz. Observe the vertical color bars. Set either `ui_in[0]` or `ui_in[1]` HIGH and pulse `ui_in[7]` to change the displayed test pattern.

Set `ui_in[3:0]` to a register address, set `{ui_in[6:4], uio_in}` to a register value, and pulse `ui_in[7]` to change individual timing values. (When setting hsync width or vsync height, set `ui_in[6]` HIGH for positive polarity or LOW for negative polarity.)

Address	Description	Default
0	Reset.	
1	Next pattern.	
2	Previous pattern.	
3	Pattern number.	31
4	Horizontal visible width.	640
5	Horizontal front porch (right border).	16
6	Horizontal sync width (polarity on `ui_in[6]`).	96
7	Horizontal back porch (left border).	48
8	Vertical visible height.	480
9	Vertical front porch (bottom border).	10
10	Vertical sync height (polarity on `ui_in[6]`).	2

Address	Description	Default
11	Vertical back porch (top border).	33
12	Pattern color.	0
13	Next color.	
14	Previous color.	
15	Reset.	

Pattern	Description
0	Solid color.
1	1×1 pixel checkerboard.
2	2×2 pixel checkerboard.
3	4×4 pixel checkerboard.
4	8×8 pixel checkerboard.
5	16×16 pixel checkerboard.
6	32×32 pixel checkerboard.
7	64×64 pixel checkerboard.
8	8×8 pixel grid.
9	16×16 pixel grid.
10	32×32 pixel grid.
11	64×64 pixel grid.
12	1×1 pixel color table.
13	2×2 pixel color table.
14	4×4 pixel color table.
15	8×8 pixel color table.
16	16×16 pixel color table.
17	32×32 pixel color table.
18	1×1 pixel color antidiagonal lines.
19	2×2 pixel color antidiagonal lines.
20	4×4 pixel color antidiagonal lines.
21	8×8 pixel color antidiagonal lines.
22	16×16 pixel color antidiagonal lines.
23	32×32 pixel color antidiagonal lines.
24	1×1 pixel color diagonal lines.
25	2×2 pixel color diagonal lines.
26	4×4 pixel color diagonal lines.
27	8×8 pixel color diagonal lines.
28	16×16 pixel color diagonal lines.
29	32×32 pixel color diagonal lines.
30	Horizontal color bars.
31	Vertical color bars.

External hardware

TinyVGA PMOD

Pinout

#	Input	Output	Bidirectional
0	A0	R1	D0
1	A1	G1	D1
2	A2	B1	D2
3	A3	VSync	D3
4	D8	R0	D4
5	D9	G0	D5
6	D10	B0	D6
7	WE	HSync	D7

Universal Binary to Segment Decoder [967]

- Author: Rebecca G. Bettencourt
- Description: Decodes various binary codes to various segmented displays.
- GitHub repository
- HDL project
- Mux address: 967
- Extra docs
- Clock: 0 Hz

How it works

This project is composed of four modules:

- A BCD to seven segment decoder with a wide variety of options for customizing the appearance of digits
- An ASCII to seven segment decoder with two different "fonts"
- A dual BCD to Cistercian numeral decoder
- A BCV (binary-coded *vigesimal*) to Kaktovik numeral decoder

BCD to seven segment decoder

This mode converts a decimal digit in BCD to its representation on a standard seven segment display. There are inputs that affect the display of the digits 6, 7, and 9, and eight different options for handling out-of-range values. These inputs allow this decoder to match the behavior of just about any other BCD to seven segment decoder, making it *universal*.

0000 0001 0010 0011 0100 0101 0110 0111 1000 1001

1010 1011 1100 1101 1110 1111

V0=0 V1=0 V2=0

V0=1 V1=0 V2=0

V0=0 V1=1 V2=0

V0=1 V1=1 V2=0

1010 1011 1100 1101 1110 1111

V0=0 V1=0 V2=1

V0=1 V1=0 V2=1

V0=0 V1=1 V2=1

V0=1 V1=1 V2=1

The signals used in this mode are:

- **/AL** - Active low. If HIGH, outputs will be HIGH when lit. If LOW, outputs will be LOW when lit.
- **/BI** - Blanking input. If LOW, all segments will be blank regardless of other inputs, including **/LT**.
- **/LT** - Lamp test. When **/BI** is HIGH and **/LT** is LOW, all segments will be lit.
- **/RBI** - Ripple blanking input. If the BCD value is zero and **/RBI** is LOW, all segments will be blank.
- **V0**, **V1**, **V2** - Selects the output when the BCD value is out of range.
- **X6** - When HIGH, the extra segment **a** will be lit on the digit 6.
- **X7** - When HIGH, the extra segment **f** will be lit on the digit 7.
- **X9** - When HIGH, the extra segment **d** will be lit on the digit 9.
- **A**, **B**, **C**, **D** - BCD input from least significant bit **A** to most significant bit **D**.
- **a**, **b**, **c**, **d**, **e**, **f**, **g** - Outputs for a seven segment display.
- **/RBO** - Ripple blanking output. HIGH when BCD value is nonzero or **/RBI** is HIGH.

The pin assignments in this mode are:

	Dedicated Input	Dedicated Output	Bidirectional
0	A	Segment a	Input - X6

	Dedicated Input	Dedicated Output	Bidirectional
1	B	Segment b	Input - X7
2	C	Segment c	Input - X9
3	D	Segment d	Input - /LT
4	V0	Segment e	Input - /BI
5	V1	Segment f	Input - /AL
6	V2	Segment g	Input - LOW
7	/RBI	/RBO	Input - LOW

ASCII to seven segment decoder

This mode converts an ASCII character to a representation on a standard seven segment display. Like with the BCD decoder, there are inputs that affect the display of the digits 6, 7, and 9. There are also two choices of "font" and the option to display lowercase letters as uppercase or as lowercase.

FS=0:

0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111

D6=0 D5=1 D4=0

D6=0 D5=1 D4=1

D6=1 D5=0 D4=0

D6=1 D5=0 D4=1

D6=1 D5=1 D4=0

D6=1 D5=1 D4=1

FS=1:

	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
D6=0 D5=1 D4=0																
D6=0 D5=1 D4=1																
D6=1 D5=0 D4=0																
D6=1 D5=0 D4=1																
D6=1 D5=1 D4=0																
D6=1 D5=1 D4=1																

The signals used in this mode are:

- **/AL** - Active low. If HIGH, outputs will be HIGH when lit. If LOW, outputs will be LOW when lit.
- **/BI** - Blanking input. If LOW, all segments will be blank regardless of other inputs.
- **FS** - Font select. Selects one of two "fonts."
- **LC** - Lower case. If LOW, lowercase letters will appear as uppercase.
- **X6** - When HIGH, the extra segment **a** will be lit on the digit 6.
- **X7** - When HIGH, the extra segment **f** will be lit on the digit 7.
- **X9** - When HIGH, the extra segment **d** will be lit on the digit 9.
- **D0**...**D6** - ASCII input from least significant bit **D0** to most significant bit **D6**.
- **a**, **b**, **c**, **d**, **e**, **f**, **g** - Outputs for a seven segment display.
- **/LTR** - Letter. LOW when the input is a letter (A...Z or a...z).

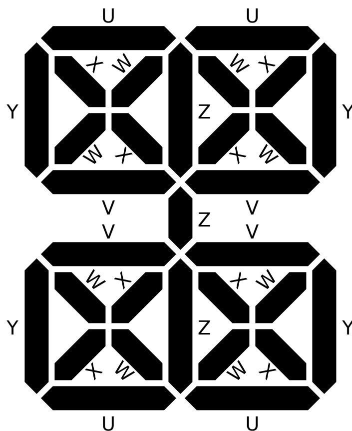
The pin assignments in this mode are:

	Dedicated Input	Dedicated Output	Bidirectional
0	D0	Segment a	Input - X6
1	D1	Segment b	Input - X7
2	D2	Segment c	Input - X9
3	D3	Segment d	Input - FS
4	D4	Segment e	Input - /BI

	Dedicated Input	Dedicated Output	Bidirectional
5	D5	Segment f	Input - /AL
6	D6	Segment g	Input - HIGH
7	LC	/LTR	Input - LOW

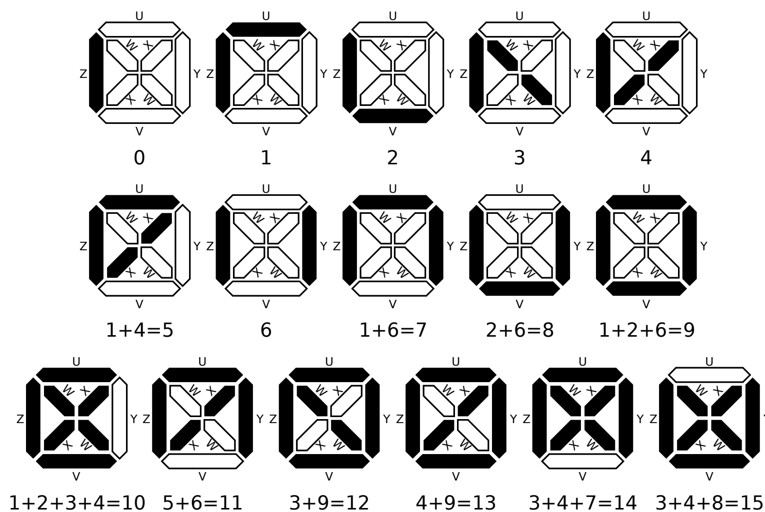
Dual BCD to Cistercian numeral decoder

This mode converts two decimal digits in BCD to their representations on the segmented display for Cistercian numerals shown below.



The patterns produced for each input value are shown below.

Patterns as seen in top right (units) position:



The signals used in this mode are:

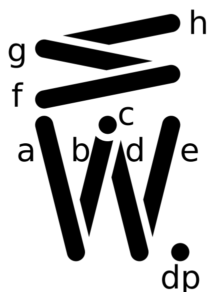
- **/AL** - Active low. If HIGH, outputs will be HIGH when lit. If LOW, outputs will be LOW when lit.
- **/BI** - Blanking input. If LOW, all segments will be blank regardless of other inputs, including **/LT1** and **/LT2**.
- **/LT1** - Lamp test for digit 1. When **/BI** is HIGH and **/LT1** is LOW, all segments for digit 1 will be lit.
- **/LT2** - Lamp test for digit 2. When **/BI** is HIGH and **/LT2** is LOW, all segments for digit 2 will be lit.
- **A1**, **B1**, **C1**, **D1** - BCD input for digit 1 from least significant bit **A1** to most significant bit **D1**.
- **A2**, **B2**, **C2**, **D2** - BCD input for digit 2 from least significant bit **A2** to most significant bit **D2**.
- **U1**, **V1**, **W1**, **X1**, **Y1** - Outputs for digit 1 on a Cistercian segmented display.
- **U2**, **V2**, **W2**, **X2**, **Y2** - Outputs for digit 2 on a Cistercian segmented display.

The pin assignments in this mode are:

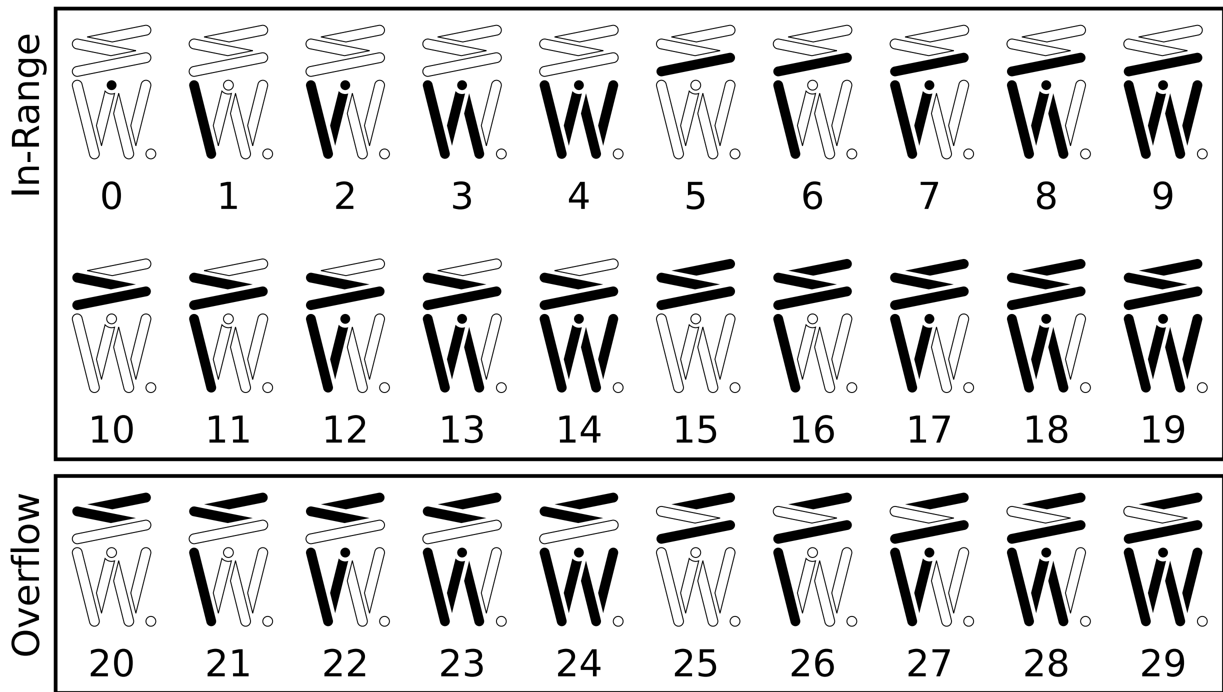
	Dedicated Input	Dedicated Output	Bidirectional
0	A1	Segment U1	Output - Y1
1	B1	Segment U2	Output - Y2
2	C1	Segment V1	Input - /LT1
3	D1	Segment V2	Input - /LT2
4	A2	Segment W1	Input - /BI
5	B2	Segment W2	Input - /AL
6	C2	Segment X1	Input - LOW
7	D2	Segment X2	Input - HIGH

BCV to Kaktovik numeral decoder

This mode converts a *vigesimal* (base 20) digit in BCV (binary-coded vigesimal) to its representation on the segmented display for Kaktovik numerals shown below.



The patterns produced for each input value are shown below.



The signals used in this mode are:

- **/AL** - Active low. If HIGH, outputs will be HIGH when lit. If LOW, outputs will be LOW when lit.
- **/BI** - Blanking input. If LOW, all segments will be blank regardless of other inputs, including **/LT**.
- **/LT** - Lamp test. When **/BI** is HIGH and **/LT** is LOW, all segments will be lit.
- **/RBI** - Ripple blanking input. If the BCV value is zero and **/RBI** is LOW, all segments will be blank.
- **/VBI** - Overflow blanking input. If the BCV value is out of range and **/VBI** is LOW, all segments will be blank.
- **A**, **B**, **C**, **D**, **E** - BCV input from least significant bit **A** to most significant bit **E**.
- **a**, **b**, **c**, **d**, **e**, **f**, **g**, **h** - Outputs for a Kaktovik segmented display.
- **/RBO** - Ripple blanking output. HIGH when BCV value is nonzero or **/RBI** is HIGH.
- **V** - Overflow. HIGH when BCV value is out of range (greater than or equal to 20).

The pin assignments in this mode are:

	Dedicated Input	Dedicated Output	Bidirectional
0	A	Segment a	Output - h
1	B	Segment b	Output - V
2	C	Segment c	

	Dedicated Input	Dedicated Output	Bidirectional
3	D	Segment d	Input - /LT
4	E	Segment e	Input - /BI
5		Segment f	Input - /AL
6	/VBI	Segment g	Input - HIGH
7	/RBI	/RBO	Input - HIGH

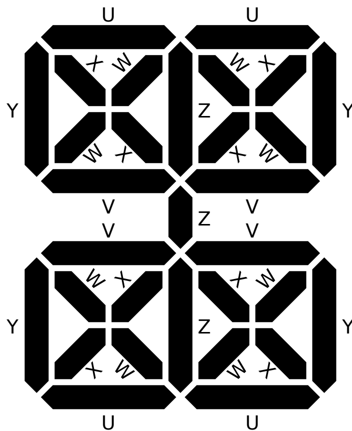
How to test

The `test` directory includes extensive tests for each of the four modules.

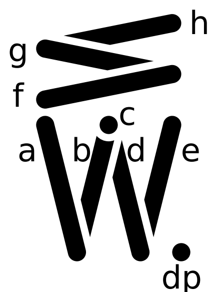
External hardware

For the BCD and ASCII modes, a standard seven-segment display is used.

For the Cistercian mode, a segmented display like the one below is used. There are design files for such a display here.



For the Kaktovik mode, a segmented display like the one below is used. There are design files for such a display here.



Pinout

#	Input	Output	Bidirectional
0	A; D0; A1; A	Segment a; U1; a	X6; X6; Y1; h
1	B; D1; B1; B	Segment b; U2; b	X7; X7; Y2; V
2	C; D2; C1; C	Segment c; V1; c	X9; X9; /LT1; -
3	D; D3; D1; D	Segment d; V2; d	/LT; FS; /LT2; /LT
4	V0; D4; A2; E	Segment e; W1; e	/BI (blanking input)
5	V1; D5; B2; -	Segment f; W2; f	/AL (active low)
6	V2; D6; C2; /VBI	Segment g; X1; g	M0 (mode select)
7	/RBI; LC; D2; /RBI	/RBO; /LTR; X2; /RBO	M1 (mode select)

INTERCAL ALU [969]

- Author: Rebecca G. Bettencourt
- Description: An ALU for the five operators of the INTERCAL programming language.
- GitHub repository
- HDL project
- Mux address: 969
- Extra docs
- Clock: 0 Hz

How it works

As an educational project, it is inevitable that Tiny Tapeout would attract various pedagogical examples of common logic circuits, such as ALUs. While ALUs for common operations such as addition, subtraction, and binary bitwise logic are surprisingly common, it is much rarer to encounter one that can calculate the five operations of the INTERCAL programming language. Due to either the cost-prohibitive nature of Warmenhovian logic gates or general lack of interest, such a feat has never been performed until now. With chip production finally within reach of the average person, all it takes is one person who has more dollars than sense to design the fabled INTERCAL ALU (Arrhythmic Logic Unit).

The pin assignments for this design are roughly as follows. The /OE (output enable) and /WE (write enable) signals are active low, so should be set HIGH by default.

#	Dedicated Input	Dedicated Output	Bidirectional I/O
0	A0 (address)	D0 (output only)	D0 (input and output only)
1	A1 (address)	D1 (output only)	D1 (input and output only)
2	S0 (selector)	D2 (output only)	D2 (input and output only)
3	S1 (selector)	D3 (output only)	D3 (input and output only)
4	S2 (selector)	D4 (output only)	D4 (input and output only)
5	S3 (selector)	D5 (output only)	D5 (input and output only)
6	/OE (output enable)	D6 (output only)	D6 (input and output only)
7	/WE (write enable)	D7 (output only)	D7 (input and output only)

This ALU has two 32-bit registers, B and A (in no particular order). (These may also be thought of as four 16-bit registers, AL, AH, BL, and BH.) To write a byte to a register, set A0 and A1 to the byte address, set S0 LOW for the A register or HIGH for the B register, set S1 through S3 LOW, set the bidirectional I/O pins to the byte

value, set /WE LOW, then set /WE HIGH again. (Do not set S1 through S3 HIGH when writing, or else something unpredictable will happen, most likely nothing.)

To read a register or result, set A0 and A1 to the byte address, set S0 through S3 to the desired operation, set /OE LOW, read the byte value from the bidirectional I/O pins, then set /OE HIGH. Results can also be read from the dedicated outputs; the dedicated outputs are not affected by the /OE signal, as they do not need to care about your feelings.

The operations supported are listed below. An attempt was made to make it understandable.

						Address				
						A	3	2	1	0
Selector						**A1**	1	1	0	0
S	**S3**	**S2**	**S1**	**S0**	**Operation**	**A0**	1	0	1	0
0	0	0	0	0	`A`		AH		AL	
1	0	0	0	1	`B`		BH		BL	
2	0	0	1	0	`AND16`		& AH		& AL	
3	0	0	1	1	`AND32`		& A			
4	0	1	0	0	`OR16`		V AH		V AL	
5	0	1	0	1	`OR32`		V A			
6	0	1	1	0	`XOR16`		? AH		? AL	
7	0	1	1	1	`XOR32`		? A			
8	1	0	0	0	`MINGLE16L`		AL $ BL			
9	1	0	0	1	`MINGLE16H`		AH $ BH			
10	1	0	1	0	`SELECT16`		AH~BH		AL~BL	
11	1	0	1	1	`SELECT32`		A ~ B			

Operations 0 and 1 simply return the current value of the A or B register, respectively. This corresponds with the values of S0 through S3 used in write mode. This is not unintentional. This might also explain why S1 through S3 must be LOW in write mode.

Operations 2 through 7 correspond to INTERCAL's unary AND, unary OR, and unary XOR operators, represented by ampersand (&), book (V), and what (?), respectively. From the INTERCAL manual:

These operators perform their respective logical operations on all pairs of adjacent bits, the result from the first and last bits going into the first bit of the result. The effect is that of rotating the operand one place to the right and ANDing, ORing, or XORing with its initial value. Thus, #&77 (binary = 1001101) is binary 0000000000000100 = 4, #V77 is binary 1000000001101111 = 32879, and #?77 is binary 1000000001101011 = 32875.

Operations 2, 4, and 6 work on the 16-bit halves of the A register independently, while operations 3, 5, and 7 work on the 32-bit whole of the A register.

Operations 8 and 9 correspond to INTERCAL's *interleave* (also called *mingle*) operator, represented by big money ($). From the INTERCAL manual:

The interleave operator takes two 16-bit values and produces a 32-bit result by alternating the bits of the operands. Thus, #65535$#0 has the 32-bit binary form 101010....10 or 2863311530 decimal, while #0$#65535 = 0101....01 binary = 1431655765 decimal, and #255$#255 is equivalent to #65535.

Operation 8 returns the interleave of the lower halves of A and B, while operation 9 returns the interleave of the upper halves of A and B. (Should the chip fabrication process allow for it, operation 8½ will, of course, return the interleave of the middle halves of A and B.)

Operations 10 and 11 correspond to INTERCAL's *select* operator, represented by sqiggle (~). From the INTERCAL manual:

The select operator takes from the first operand whichever bits correspond to 1's in the second operand, and packs these bits to the right in the result. Both operands are automatically padded on the left with zeros. [...] For example, #179~#201 (binary value 10110011~11001001) selects from the first argument the 8th, 7th, 4th, and 1st from last bits, namely, 1001, which = 9. But #201~#179 selects from binary 11001001 the 8th, 6th, 5th, 2nd, and 1st from last bits, giving 10001 = 17. #179~#179 has the value 31, while #201~#201 has the value 15.

To help understand the select operator, the INTERCAL manual also provides a helpful circuitous diagram.

Use of operations 12 and above is not recommended, unless undefined behavior is required.

How to test

The following example calculations found in the INTERCAL manual should be particularly illuminating.

S	A	B	F
MINGLE16L (8)	0	256	65536
MINGLE16L (8)	65535	0	2863311530
MINGLE16L (8)	0	65535	1431655765
MINGLE16L (8)	255	255	65535
SELECT16 (10)	51	21	5 *
SELECT16 (10)	179	201	9
SELECT16 (10)	201	179	17
SELECT16 (10)	179	179	31
SELECT16 (10)	201	201	15
AND16 (2)	77		4
OR16 (4)	77		32879
XOR16 (6)	77		32875

These test cases are included in the (unfortunately Python and not INTERCAL) `test.py` file. As these are likely more INTERCAL operations than any sensible person will ever perform, they should be sufficient for testing purposes. However, for curiosity's sake, an extensive set of additional test cases have also been included.

- Not found in the INTERCAL manual.

External hardware

The ALU may be used without external hardware, although seeing the output values may present a challenge. Instead, it is recommended to use a microcontroller of some sort to drive the inputs and read the outputs, as microcontrollers are designed to do. The implementation of the rest of the INTERCAL language is left as an exercise for the reader.

Further reading

The INTERCAL Programming Language Revised Reference Manual by Donald R. Woods and James M. Lyon with revisions by Louis Howell and Eric S. Raymond (can recommend highly enough)

Pinout

#	Input	Output	Bidirectional
0	A0 (address)	D0	D0
1	A1 (address)	D1	D1
2	S0 (selector)	D2	D2
3	S1 (selector)	D3	D3
4	S2 (selector)	D4	D4
5	S3 (selector)	D5	D5
6	/OE (output enable)	D6	D6
7	/WE (write enable)	D7	D7

Simple PWM Module [971]

- Author: Tobi McKellar
- Description: PWM for LED control.
- GitHub repository
- HDL project
- Mux address: 971
- Extra docs
- Clock: 0 Hz

How it works

A basic PWM controller. ui[5:0] control the reference. When ui[6] is low, this reference is used to set the PWM duty cycle. when ui[6] is high, functionality changes from manual reference to a triangular reference generated internally. In this mode, ui[5:0] control the frequency of the triangular reference. ui[7] enables pwm output when high. PWM is output on uo[7].

How to test

Set ui[7] to high. Measure the output on uo[7]. FLip the other input switches and see what happens!

External hardware

None, but an LED and resistor would be nice.

Pinout

#	Input	Output	Bidirectional
0	Manual mode PWM reference control	PWM output	
1	Manual mode PWM reference control		
2	Manual mode PWM reference control		
3	Manual mode PWM reference control		
4	Manual mode PWM reference control		
5	Manual mode PWM reference control		
6	Toggle PWM breathe or manual mode		
7	Enable PWM output		

freqSweep [973]

- Author: Jesus Minguillon
- Description: Frequency sweeper
- GitHub repository
- HDL project
- Mux address: 973
- Extra docs
- Clock: 5000000 Hz

How it works

The project (src/project.v) implements a clock frequency sweeper in Verilog. It uses the Tiny Tapeout clock as input to generate a clock signal at the output uo[0] whose frequency is divided by 2 every 15 clock cycles. It goes from 1/2 to 1/16 of the input clock frequency and starts again. Input ui[0] is used as internal enable (active high).

How to test

For simulation, use the test bench (test/tb.v), which includes a module (src/periodCount.v) for measuring the frequency (or period) ratio between input and output clocks. Use the Python script (test/test.py) if you want to perform unit tests using cocotb. For hardware testing, make sure the internal enable (input ui[0]) is high and check the output clock at output uo[0] using an oscilloscope.

Gate level simulation (5 MHz input clock) The frequency (or period) ratio between the input clock (clk) and the output clock (uo_out[0]) is given by clk_factor register of periodCount module:

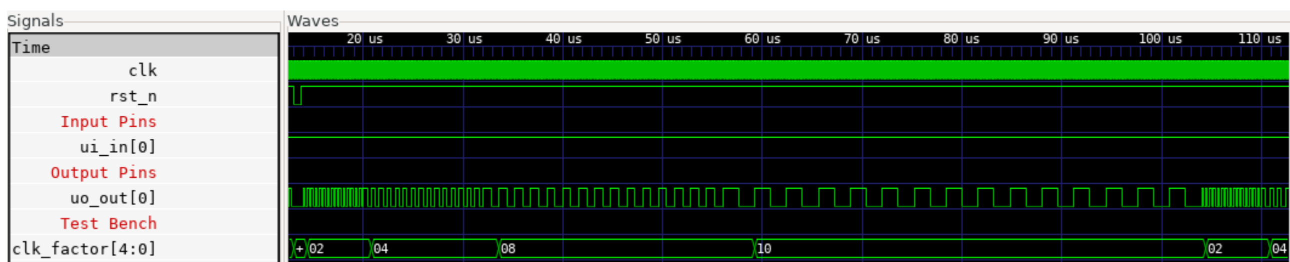


Figure 54: image info

Zoom in:

Unit tests:

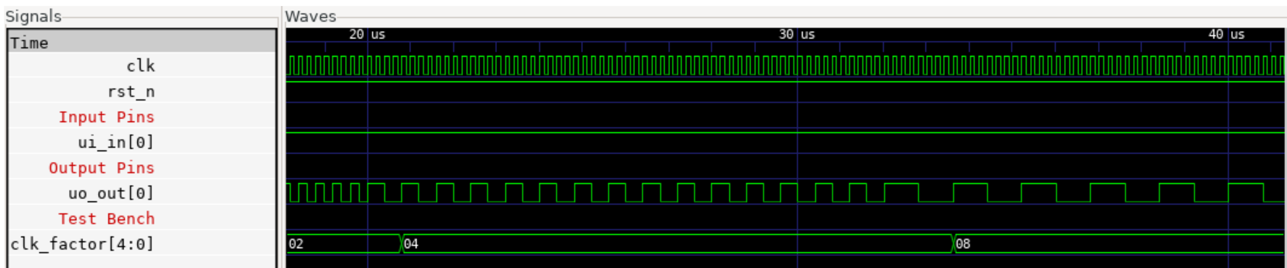


Figure 55: image info

```
(venv) jesus@ws131571:~/tt09-verilog-freqSweep/test$ make -B GATES=yes
rm -f results.xml
"make" -f Makefile results.xml
make[1]: Entering directory '/home/jesus/tt09-verilog-freqSweep/test'
mkdir -p sim_build/gl
/usr/bin/iverilog -o sim_build/gl/sim.vvp -D COCOTB_SIM=1 -s tb -g2012 -D
rm -f results.xml
MODULE=test TESTCASE= TOPLEVEL=tb TOPLEVEL_LANG=verilog \
         /usr/bin/vvp -M /home/jesus/ttsetup/venv/lib/python3.12/site-pac
     -.--ns INFO     gpi                                ..mbed/gpi_embed.
     -.--ns INFO     gpi                                ../gpi/GpiCommon.
     0.00ns INFO     cocotb                             Running on Icarus
     0.00ns INFO     cocotb                             Running tests wit
     0.00ns INFO     cocotb                             Seeding Python ra
     0.00ns INFO     cocotb.regression                  Found test test.t
     0.00ns INFO     cocotb.regression                  Found test test.t
     0.00ns INFO     cocotb.regression                  Found test test.t
     0.00ns INFO     cocotb.regression                  Found test test.t
     0.00ns INFO     cocotb.regression                  running test_star
     0.00ns INFO     cocotb.tb                          Startup test with
     0.00ns INFO     cocotb.tb                          rst_n = 0, ui_in[
VCD info: dumpfile tb.vcd opened for output.
     2.00ns INFO     cocotb.tb                          uo_out[0] = 0
  2000.00ns INFO     cocotb.tb                          rst_n = 1, ui_in[
  2202.00ns INFO     cocotb.tb                          uo_out[0] = 0
  4000.00ns INFO     cocotb.tb                          rst_n = 1, ui_in[
  4202.00ns INFO     cocotb.tb                          uo_out[0] = 1
  4402.00ns INFO     cocotb.tb                          uo_out[0] = 0
  4402.00ns INFO     cocotb.tb                          Wait until 6 us
  6000.00ns INFO     cocotb.tb                          End of startup te
  6000.00ns INFO     cocotb.regression                  test_startup pass
  6000.00ns INFO     cocotb.regression                  running test_rese
  6000.00ns INFO     cocotb.tb                          Reset test
```

```
  6000.00ns INFO     cocotb.tb                           rst_n = 0, ui_in[
  6002.00ns INFO     cocotb.tb                           uo_out[0] = 0
  6002.00ns INFO     cocotb.tb                           clk_factor = 1
  6202.00ns INFO     cocotb.tb                           uo_out[0] = 0
  6202.00ns INFO     cocotb.tb                           clk_factor = 1
  8000.00ns INFO     cocotb.tb                           uo_out[0] = 0
  8000.00ns INFO     cocotb.tb                           clk_factor = 1
  8000.00ns INFO     cocotb.tb                           rst_n = 1, ui_in[
  8202.00ns INFO     cocotb.tb                           uo_out[0] = 1
  8202.00ns INFO     cocotb.tb                           clk_factor = 1
  8402.00ns INFO     cocotb.tb                           uo_out[0] = 0
  8402.00ns INFO     cocotb.tb                           clk_factor = 1
  8402.00ns INFO     cocotb.tb                           Wait until 10 us
 10000.00ns INFO     cocotb.tb                           End of reset test
 10000.00ns INFO     cocotb.regression                   test_reset passed
 10000.00ns INFO     cocotb.regression                   running test_inte
 10000.00ns INFO     cocotb.tb                           Internal enable t
 10000.00ns INFO     cocotb.tb                           rst_n = 1, ui_in[
 11600.00ns INFO     cocotb.tb                           rst_n = 1, ui_in[
 11802.00ns INFO     cocotb.tb                           uo_out[0] = 1
 11802.00ns INFO     cocotb.tb                           clk_factor = 2
 11802.00ns INFO     cocotb.tb                           Wait until 13 us
 13000.00ns INFO     cocotb.tb                           Reset for 800 ns
 14000.00ns INFO     cocotb.tb                           End of internal e
 14000.00ns INFO     cocotb.regression                   test_internal_ena
 14000.00ns INFO     cocotb.regression                   running test_peri
 14000.00ns INFO     cocotb.tb                           Period count test
 14402.00ns INFO     cocotb.tb                           clk_factor = 2
 20802.00ns INFO     cocotb.tb                           clk_factor = 4
 33602.00ns INFO     cocotb.tb                           clk_factor = 8
 59202.00ns INFO     cocotb.tb                           clk_factor = 16
 59202.00ns INFO     cocotb.tb                           Wait until 200 us
200000.00ns INFO     cocotb.tb                           End of period cou
200000.00ns INFO     cocotb.regression                   test_period_count
200000.00ns INFO     cocotb.regression                   ****************
                                                         ** TEST
                                                         ****************
                                                         ** test.test_star
                                                         ** test.test_rese
                                                         ** test.test_inte
                                                         ** test.test_peri
                                                         ****************
```

```
** TESTS=4 PASS=4
*****************
```

```
make[1]: Leaving directory '/home/jesus/tt09-verilog-freqSweep/test'
```

Test using Tang Nano 9K FPGA (4.5 MHz input clock) Input clock signal (yellow) and ouput clock signal (blue) acquired with an oscilloscope (analog inputs and passive probes):

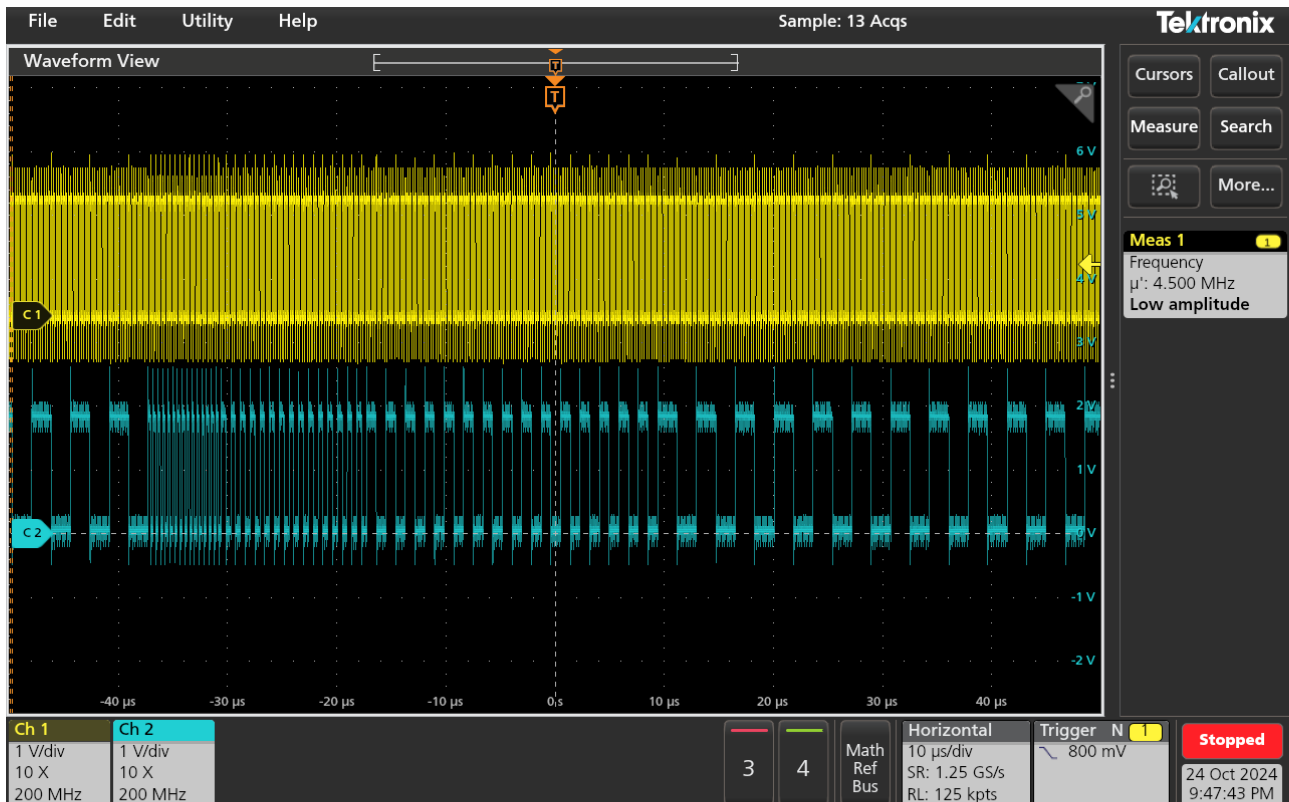


Figure 56: image info

Zoom in:

External hardware

No external hardware is needed.

Pinout

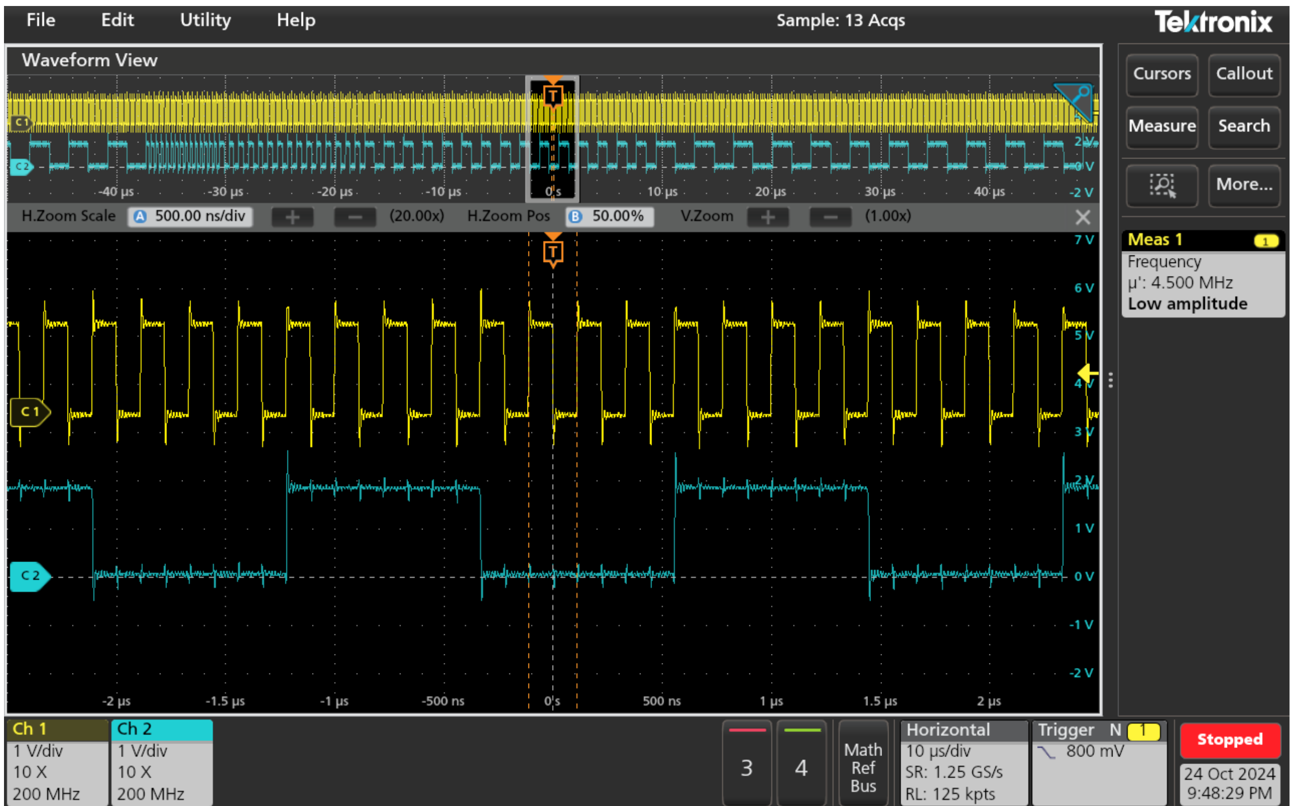


Figure 57: image info

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

Atari 2600 [974]

- Author: Renaldas Zioma
- Description: Replica of Atari 2600
- GitHub repository
- HDL project
- Mux address: 974
- 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 / Switch 1	R1	QSPI CS
1	DOWN / Switch 2	G1	QSPI SD0
2	LEFT / Switch BW	B1	QSPI SD1
3	RIGHT / Switch 3	VSync	QSPI SCK
4	FIRE	R0	QSPI SD2
5	Joystick 1 / 2	G0	QSPI SD3
6	Switches	B0	
7	RESET	HSync	Audio (PWM)

LED Bitserial Cipher [975]

- Author: simon cipher
- Description: A bitserial implementation of the LED cipher
- GitHub repository
- HDL project
- Mux address: 975
- Extra docs
- Clock: 0 Hz

How it works

tt09-led-serial is a nibble-serial implementation of the LED block cipher, proposed in 2012 and defined in The LED Block Cipher by J. Guo et. al. The cipher encrypts a 64-bit block of plaintext with a 128-bit key into a 64-bit block of ciphertext. The nibble-serial implementation enables a very compact implementation as most of the datapath logic can be reused over each nibble. The downside is that such nibble-serial implementations have a much larger latency. The nibble-serial architecture shown below was presented and analyzed earlier in Differential Fault Intensity Analysis on PRESENT and LED Block Ciphers by N. F. Galathy et. al.

Figure 58: image

To further reduce the I/O pinout constraints, this design also serializes the data-input (64 bit plaintext and 128 bit key) as well as the data-output (64 bit ciphertext).

Activity	Cycles
Load Plaintext	64
Load Key	128
Read Ciphertext	64
Encrypt	2045

The module is controlled through the bits of the input word ui_in. The serial data format is MSB to LSB. That is, given a block of plaintext `0x0123...`, the bits would be shift in as in the bitstring `0b0000000100100011....`

Bit	Name	Function
7-6	unused	NA
5	start	Assert to start encryption
4	getct	Assert to shift out ciphertext bit
3	loadkey	Assert to shift in key bit
2	loadpt	Assert to shift in plaintext bit
1	keyi	Key input bit
0	datai	Plaintext input bit

The results are generation on the output word uo_out.

Bit	Name	Function
7-2	unused	NA
1	done	1 indicates encryption complete
0	dataq	Ciphertext output bit

LIMITATIONS

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

How to test

This block could be tested with some integration on a Raspberry PI to control ui_in and uo_out. The typical sequence of operation is as follows.

1. Wait until done == 1, which indicates that the cipher is idle
2. Assert loadkey, and shift in key bits. Repeat 128 times. De-assert loadkey.
3. Assert loadpt, and shift in plaintext bits. Repeat 64 times. De-assert loadpt.
4. Assert start for one clock cycle.
5. Wait until done == 1.
6. Assert getct and shift out ciphertext bits. Repeat 64 times. De-assert getct.

Here are twotthree sample test vectors. Consult the testbench for additional test vectors.

Plaintext	Key	Ciphertext
0000000000000000	00000000000000000000000000000000	3decb2a0850cdba1
0123456789abcdef	00000000000000000000000000000000	da261393c73be9ce
12153524c0895e81	00000000000000000000000000000000	29db5fe262572f4e

External hardware

You will need external hardware to use the block cipher.

Pinout

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

Pinout

The chip is packaged in a 64-pin QFN package. The pinout is shown below.

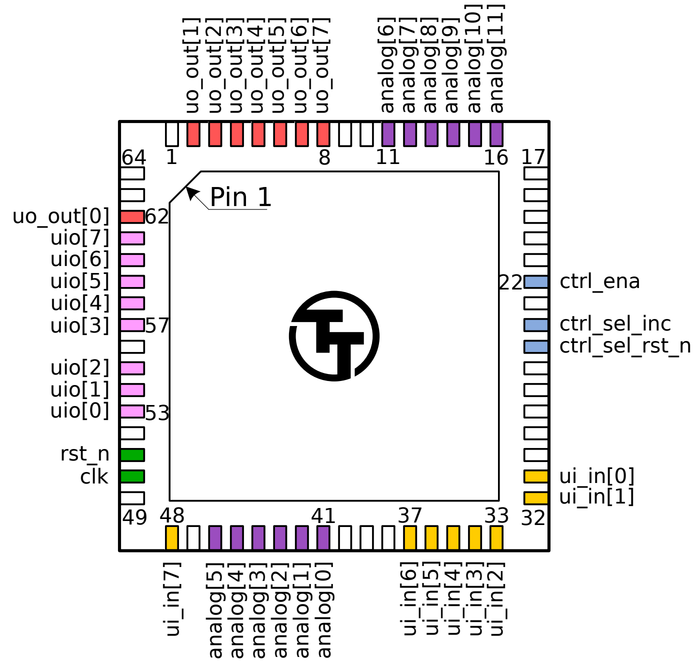


Figure 59: Pinout

Note: you will receive the chip mounted on a breakout board. The pinout is provided for advanced users, as most users will not need to solder the chip directly.

The Tiny Tapeout Multiplexer

Overview

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

It has the following features:

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

Operation

The multiplexer consists of three main units:

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

The Controller

The mux controller has 3 inputs lines:

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

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

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

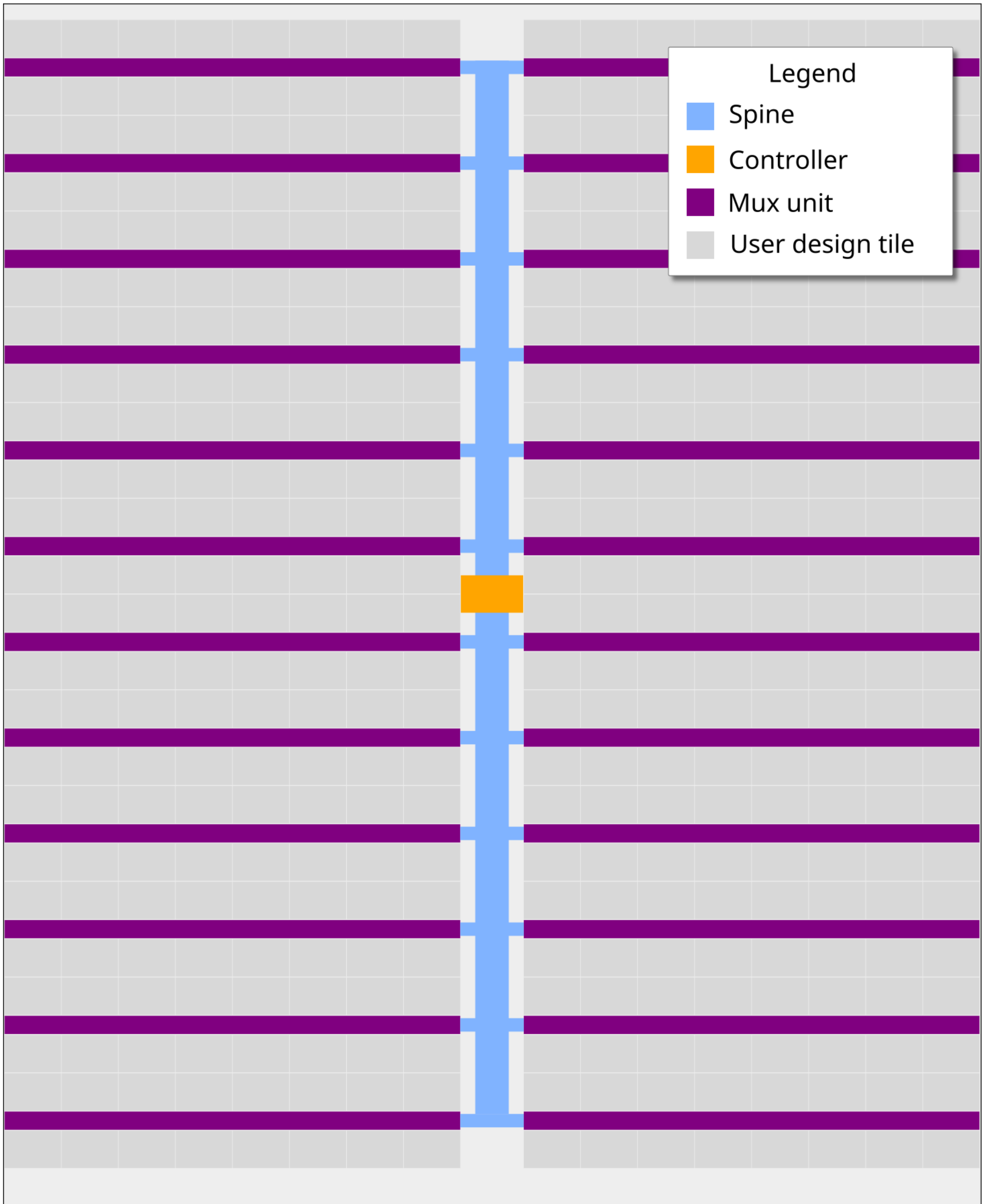


Figure 60: Mux Diagram

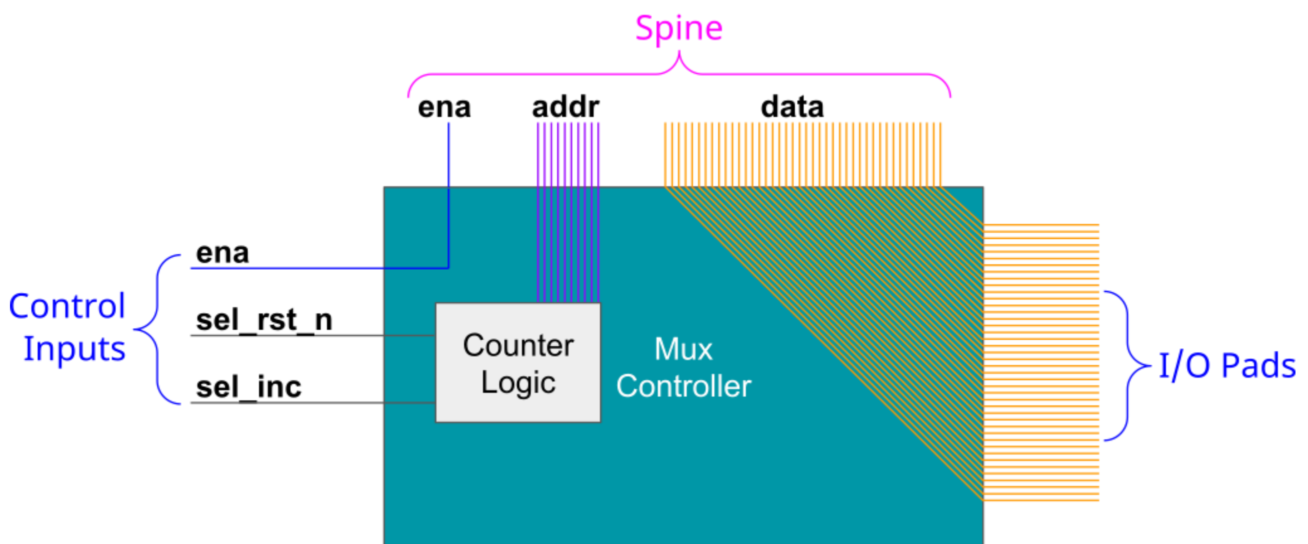


Figure 61: Mux Controller Diagram

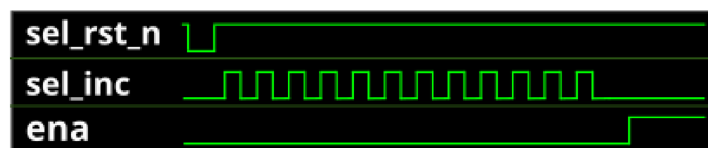


Figure 62: Mux signals for activating the design at address 12

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

The following Wokwi projects demonstrates this setup: https://wokwi.com/projects/3643478076 It contains an Arduino Nano that decodes the currently selected mux address and displays it on a 7-segment display. Click on the button labeled `RST_N` to reset the counter, and click on the button labeled `INC` to increment the counter.

The Spine

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

From controller to mux:

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

From mux to controller:

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

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

The Multiplexer (The Mux)

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

The mux implements the following logic:

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

For the active design:

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

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

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

Pinout

mprj_io pin	Function	Signal	QFN64 pin
0	Input	ui_in[0]	31
1	Input	ui_in1	32
2	Input	ui_in2	33
3	Input	ui_in[3]	34
4	Input	ui_in[4]	35

mprj_io pin	Function	Signal	QFN64 pin
5	Input	ui_in[5]	36
6	Input	ui_in[6]	37
7	Analog	analog[0]	41
8	Analog	analog1	42
9	Analog	analog2	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	uio1	54
18	Bidirectional	uio2	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_out1	2
26	Output	uo_out2	3
27	Output	uo_out[3]	4
28	Output	uo_out[4]	5
29	Output	uo_out[5]	6
30	Output	uo_out[6]	7
31	Output	uo_out[7]	8
32	Analog	analog[6]	11
33	Analog	analog[7]	12
34	Analog	analog[8]	13
35	Analog	analog[9]	14
36	Analog	analog[10]	15
37	Analog	analog[11]	16
38	Mux Control	ctrl_ena	22
39	Mux Control	ctrl_sel_inc	24
40	Mux Control	ctrl_sel_rst_n	25
41	Reserved	(none)	26
42	Reserved	(none)	27
43	Reserved	(none)	28

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

Sponsored by

Team

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

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