INTRANEX

INTRANEX is a **programmable interconnect network** that accepts a N bit input W and produces a N bit output Z. The interconnect can be programmed to realize any mapping from W to Z.

University of Cincinnati - EECE6080
Fall 2013

Max Thrun
973 919 6593
`max.thrun@gmail.com`
(Coordinator)

Xiaohu Qi
513 652 2075
`qixiaohuihaha@gmail.com`

Contents

List of Figures

List of Tables

Listings

Chapter 1

Part 1

1.1 Pinout Diagram

The pinout diagram for INTRANEX is shown below in Figure 1.1. Pins that are currently unutilized will be assigned to various internal logic signals once the floorplan is finalized. Note the symmetry of the core functionality. This was done so that multiple INTRANEX chip can be directly chained together with minimal routing effort during PCB layout.

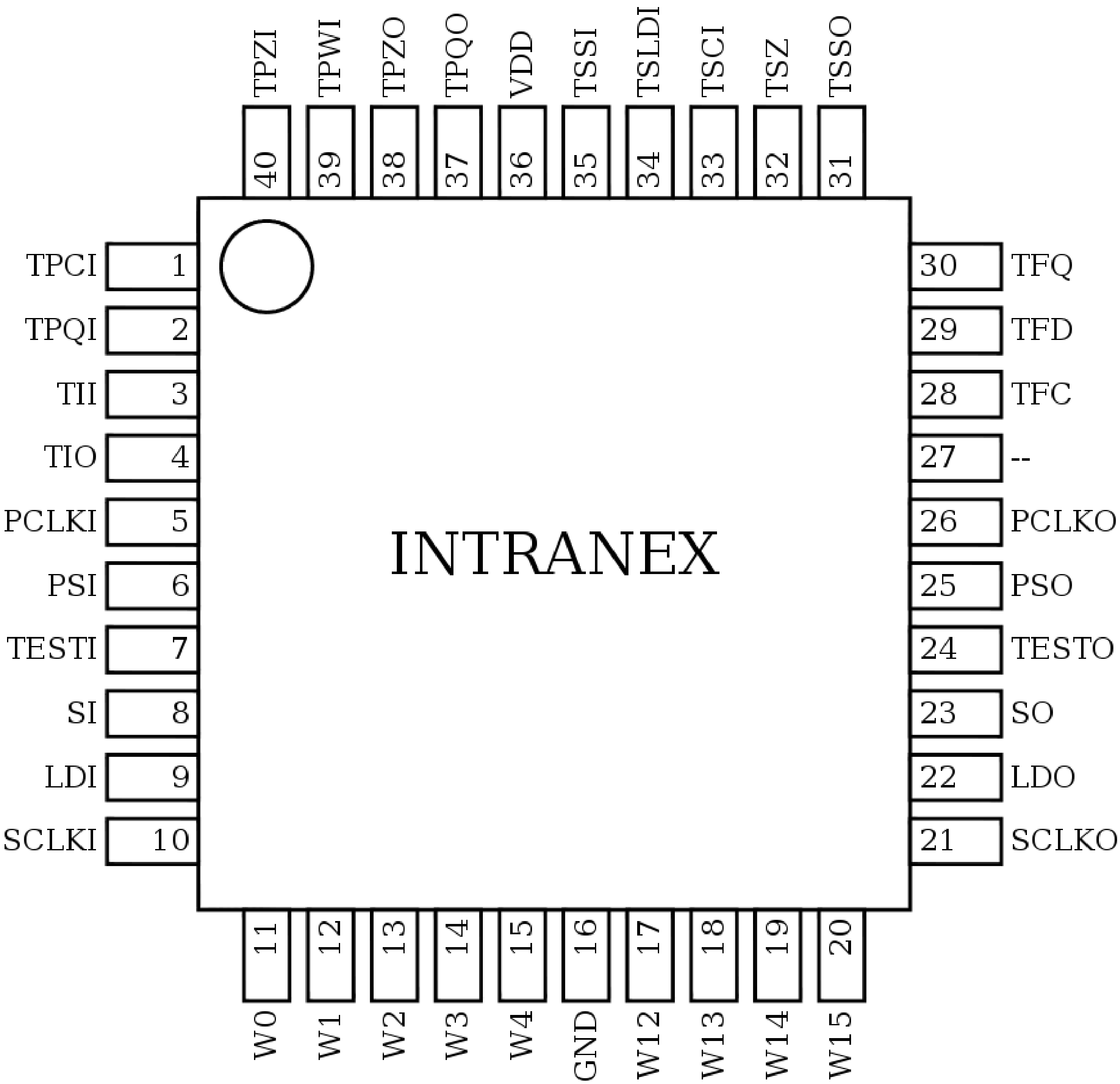

Figure 1.1: Pinout Diagram

The table below shows each pin and its corresponding name, type, and a brief description of its functionality. Type is of either I (Input), O (Output), or P (Power).

Pin #	Name	Type	Description
1	TPCI	I	Test pin slice clock input
2	TPQI	I	Test pin slice serial input
3	TII	I	Test inverter input
4	TIO	O	Test interter output
5	PCLKI	I	PIN clock input
6	PSI	I	PIN serial input
7	TESTI	I	Test Mode enable input
8	SI	I	Serial input
9	LDI	I	Parallel load input
10	SCLKI	I	Serial clock input
11	W0	O	W0 Debug Output
12	W1	O	W0 Debug Output
13	W2	O	W0 Debug Output
14	W3	O	W0 Debug Output
15	W4	O	W0 Debug Output
16	GND	P	–
17	W12	O	W0 Debug Output
18	W13	O	W0 Debug Output
19	W14	O	W0 Debug Output
20	W15	O	W0 Debug Output
21	SCLKO	O	Serial clock output
22	LDO	O	Parallel load output
23	SO	O	Serial output
24	TESTO	O	Test Mode enable output
25	PSO	O	PIN serial output
26	PCLKO	O	PIN clock output
28	TFC	I	Test flip-flop clock input
29	TFD	I	Test flip-flop D input
30	TFQ	O	Test flop-flop Q output
31	TSSO	O	Test shift slice serial output
32	TSZ	I	Test shift slice parallel input
33	TSCI	I	Test shift slice clock input
34	TSLDI	I	Test shift slice load input
35	TSSI	I	Test shift slice serial input
36	VDD	P	–
37	TPQO	O	Test pin slice serial output
38	TPZO	O	Test pin slice row output
39	TPWI	I	Test pin slice coloumn
40	TPZI	I	Test pin slice row input

Table 1.1: Pin Descriptions

1.2 Chip Functionality

The major function of this chip is to take an N bit input and translate any bit position to any other position. This allows for commonly desired functionality such as bit reversing or nibble swapping. To accomplish this we use a N by N bit interconnect network known as the PIN (Programmable Interconnect Network). The PIN is configured to perform the desired bit mappings by clocking in the mappings using the `PSI` (PIN Shift Input) and `PCLKI` (PIN Clock Input) pins. The value to be manipulated, called the Input Value, Shift Value or Shifter Value, is then clocked in serially using the `SI` (Shifter Input) and `SCLKI` (Shifter Clock Input) pins. To obtain the result the `LDI` (Load Input) pin is pulled high and the `SCLKI` pin is pulsed to latch the result in to the shift register. Once the result is latched the `LDI` pin is de-asserted and the result can be clocked out of the `SO` (Shifter Output) pin. Note that the input value is clocked in MSB first and the output value is clocked out MSB first as well.

1.2.1 Configuring the Programmable Interconnect Network

A timing diagram illustrating the PIN configuration process for a 3-Bit INTRANEX is shown below. For a 3-Bit input value a 3x3 grid is required resulting in a PIN configuration vector of 9 Bits. The mapping for each of these bits is also labeled and will be explained further in later sections.

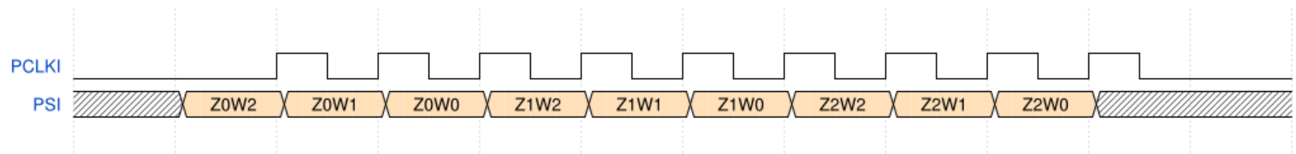

Figure 1.2: PIN Configuration

1.2.2 Loading and reading a value

Loading an input value is achieved by clocking the value in on the `SI` pin using the `SCLKI` pin. The `LDI` pin must be held low during this operation. The diagram below illustrates this process and shows the bit definitions of the value being clocked in.

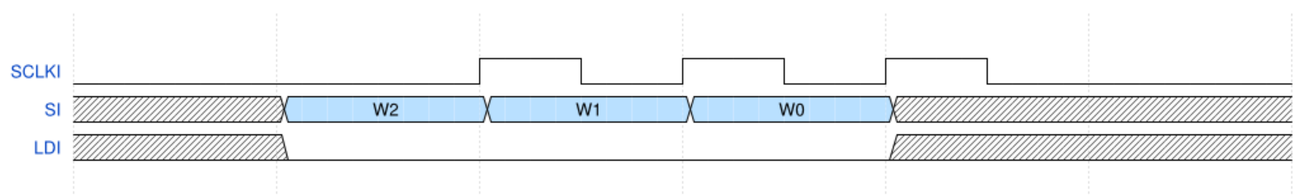

Figure 1.3: Loading a value

After the value has been loaded in the result is clocked out in a similar fashion. To first latch the result the `LDI` pin needs to be held high and the `SCLKI` pin pulsed. The MSB of the result is now available on the `SO` pin. The `LDI` pin should now be held low while clocking out the remaining result bits.

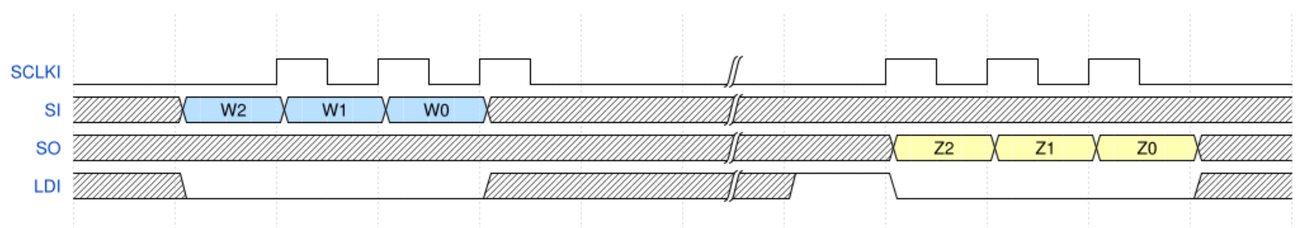

Figure 1.4: Loading a value and reading the result

1.2.3 Test Mode

Test Mode is enabled by pulling the `TESTI` pin high. When this occurs the output of the internal input value shift register is rerouted to connect to the input of the PIN network bypassing its normal `PSI` input. Additionally the `SCLKO` signal is also routed to the PIN bypassing its normal `PCLKI` signal. Finally the `PSO` signal is routed to the `SO` pin. This allows values that are clocked in via the `SI` pin to propagate through the shifter and then through the PIN and then out the `SO` pin. The fact that the values come out the `SO` pin allows multiple INTRANEX chips to be directly chained and tested in circuit using only the `SI` and `SCLKI` pins of the first chip in the chain. Note that the `LDI` pin must be held low during this entire operation in order to ensure proper shifting through the input value shift register.

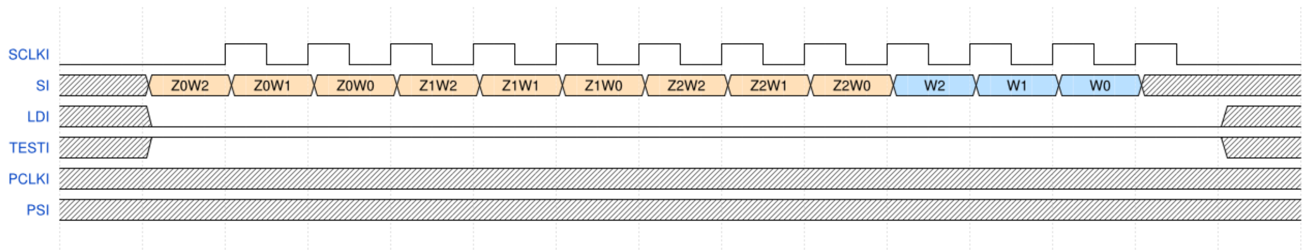

Figure 1.5: Enabling test mode and loading all DFFs

1.3 Design Decisions

When evaluating design concepts and possible solutions we prioritized a few key factors that we wanted to achieve. The first is a fully bit-sliced solution where each slice can directly connect to the next with minimal wiring overhead and zero additional logic. This will allow us to utilize Magics `Array` functionality to quickly build up our chip and allow us to easily scale to any desired size. As we see in later sections we were able to achieve a fully bit-sliced design with zero logic overhead.

In order to achieve totally minimized wiring overhead it would be necessary to design two different slice layouts, one of which is mirrored and flipped. This would allow each slice row in the PIN to share a power rail with the rows above and below it and also minimize the length of the row-to-row wiring. This design however greatly increases the complexity of the VHDL design as wiring the rows together becomes trickier. Additionally we would have to maintain two different versions of the PIN slices. We decided to instead go with a design where all slices are exactly identical and the interconnect between them is linear. This allows for easier calculation of PIN configuration values as every row has the same index order. The only real disadvantage to this design is that we will require long interconnects between slices. We are assuming for now that even with the added capacitance of these long interconnects we will still be able to achieve max clock speeds of greater than 50Mhz. By progress report 2 we will have layout simulation results to confirm this.

As stated earlier an important goal for us was to be able to directly chain multiple INTRANEX chips together. Our current design achieves this and an example chain showing 3 INTRANEXs chained together is shown below. Note that the pin layout in this diagram matches that of the actual layout we plan on implementing.

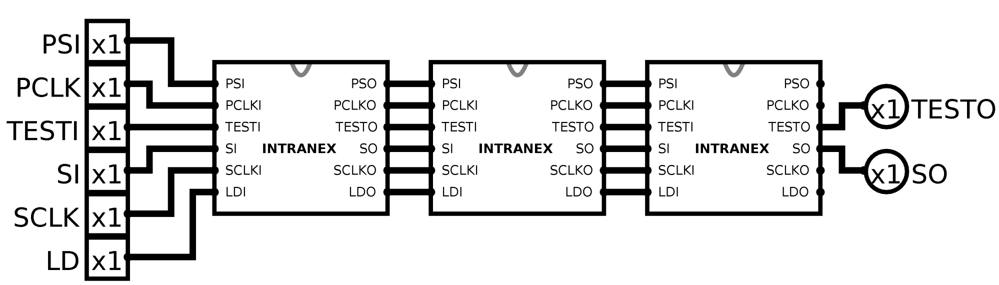

Figure 1.6: 3 INTRANEX Chain

1.4 Block Diagrams

1.4.1 Top Level

A top level block diagram for a 3-bit INTRANEX is shown below. The top module is the PIN and the bottom module is the parallel load shift register. Test mode logic has been excluded to more clearly illustrate the core functionality.

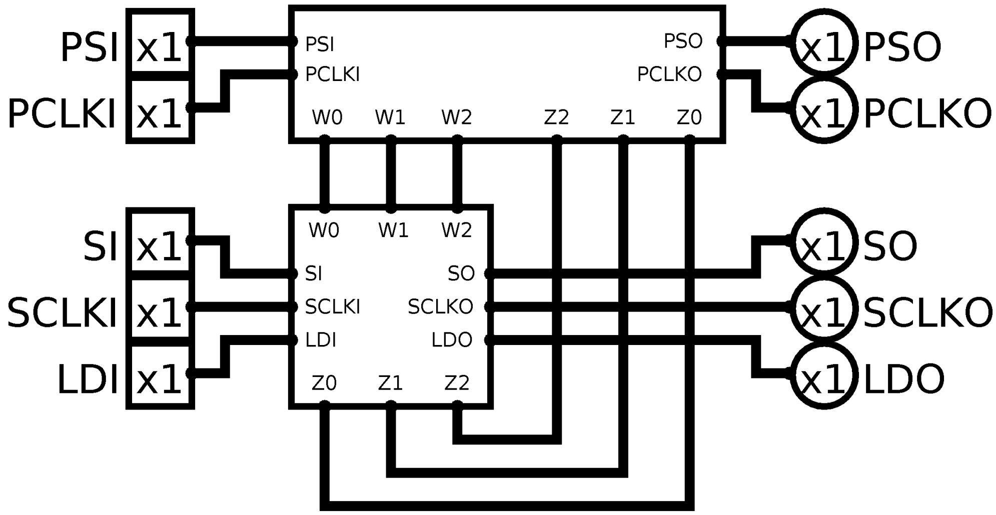

Figure 1.7: Top Level Block Diagram (3-Bit Configuration)

1.4.2 Top Level With Test Mode

The same top level diagram is shown with the addition of the test mode logic. The test mode logic simply consists of 3 2:1 multiplexers that redirect the output of the shift register to the input of the PIN and the output of the PIN to what is normally the output of the shift register. In other words, it wires in the PIN between the shifter and the shifters normal output pins.

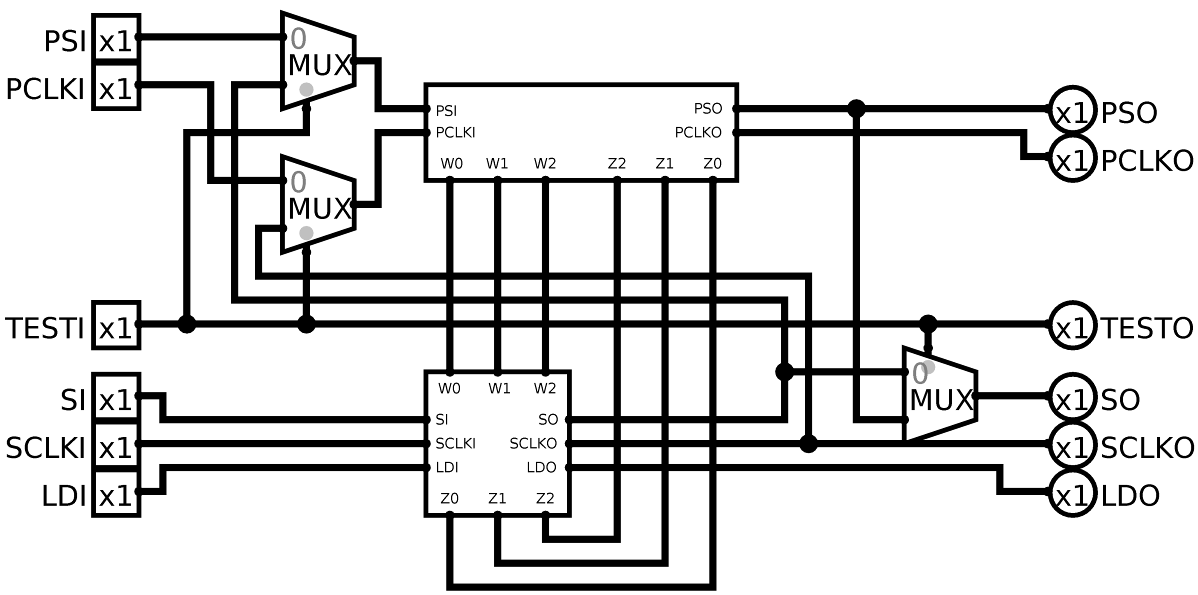

Figure 1.8: Top Level Block Diagram Showing Test Mode Logic (3-Bit Configuration)

1.4.3 Top Level Bit Sliced

The diagram below shows a bit sliced version of the top level diagram shown in Figure 1.7. We can see how each slice is directly connected together with zero interfacing logic as well as the long row-to-row connections as discussed earlier.

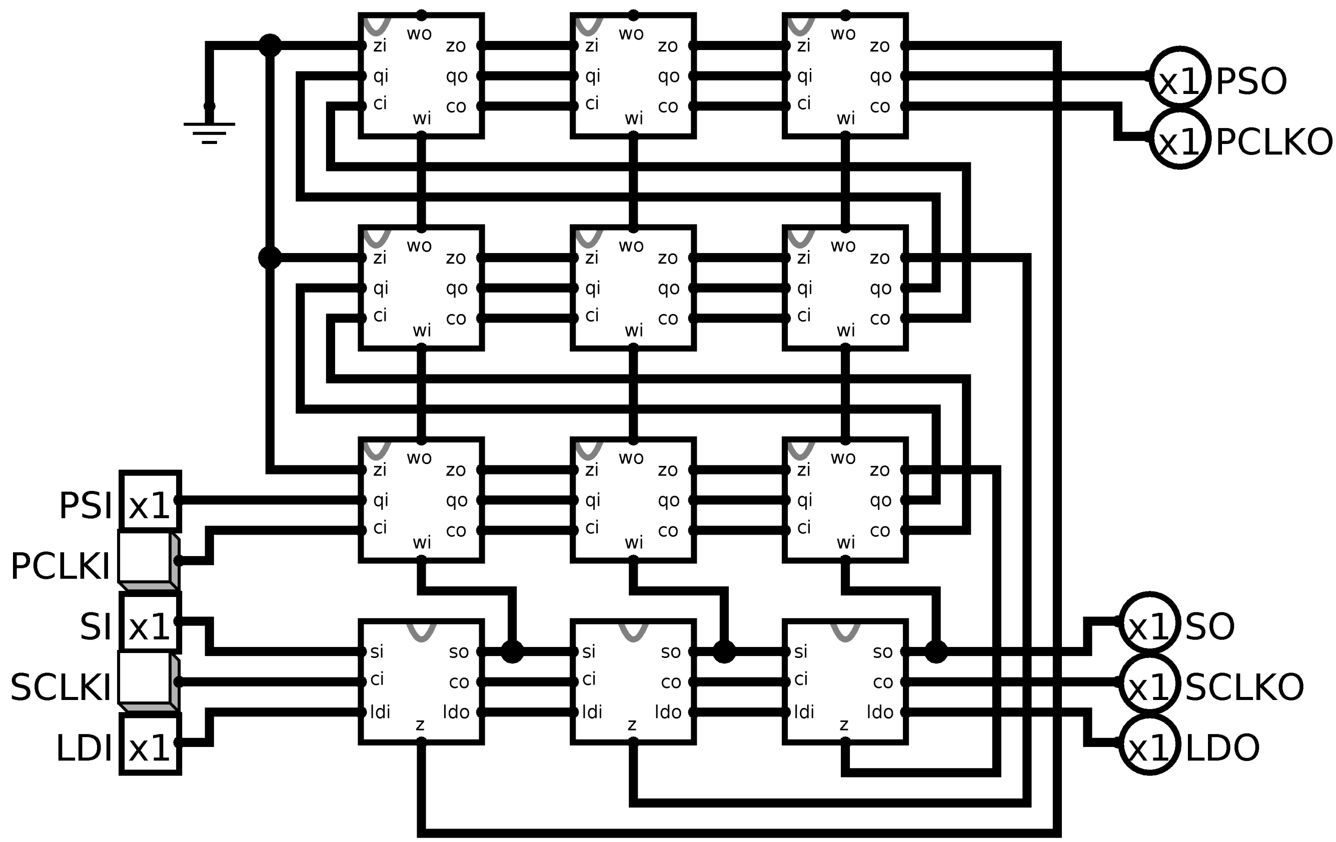

Figure 1.9: Top Level Bit Sliced Block Diagram (3-Bit Configuration)

1.4.4 Parallel Load Shift Register

Bit-slicing Scheme

Looking at just the shift register we can see that it is a parallel load parallel output shifter that is easily extendable by simply tacking on additional slices.

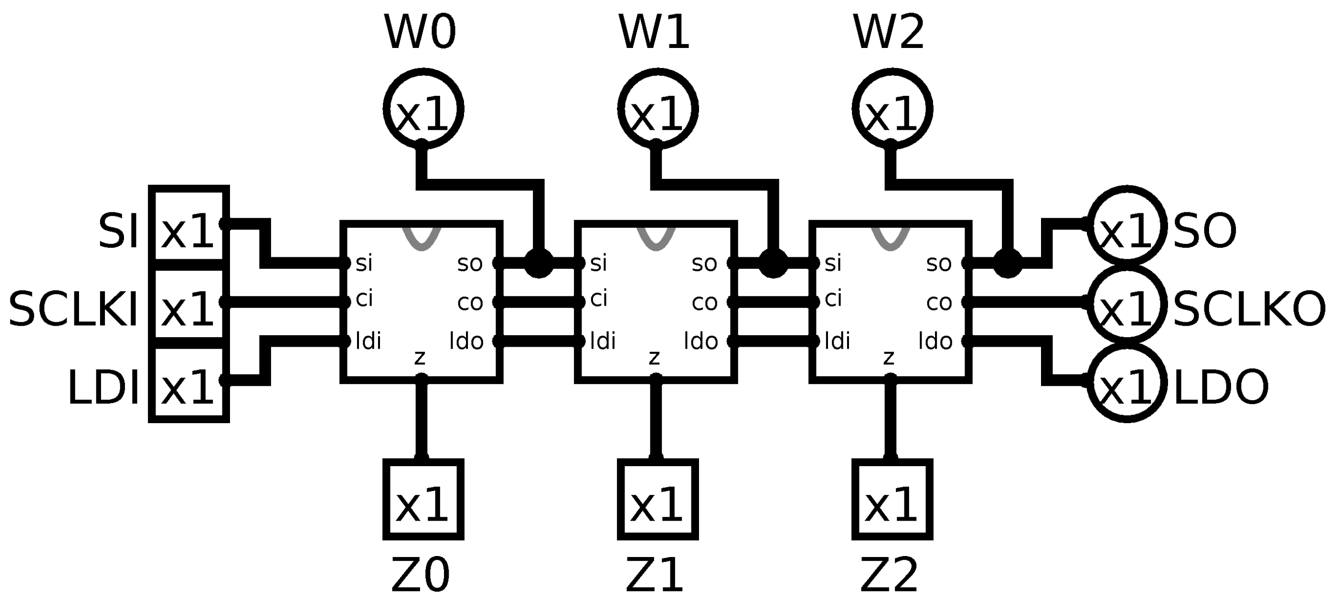

Figure 1.10: Parallel Load Bit-Sliced Shifter Register (3-Bit Configuration)

Bit-Slice

Looking at the internals of a single shift slice we can see that is is just a 2:1 multiplexer and a D Flip Flop. The multiplexer determines if the slice should load either the value from the previous slice (`SI`) or the parallel input (`Z`). When `LDI` is 0 it uses the value of the previous slice and when it is a 1 it uses the parallel load value.

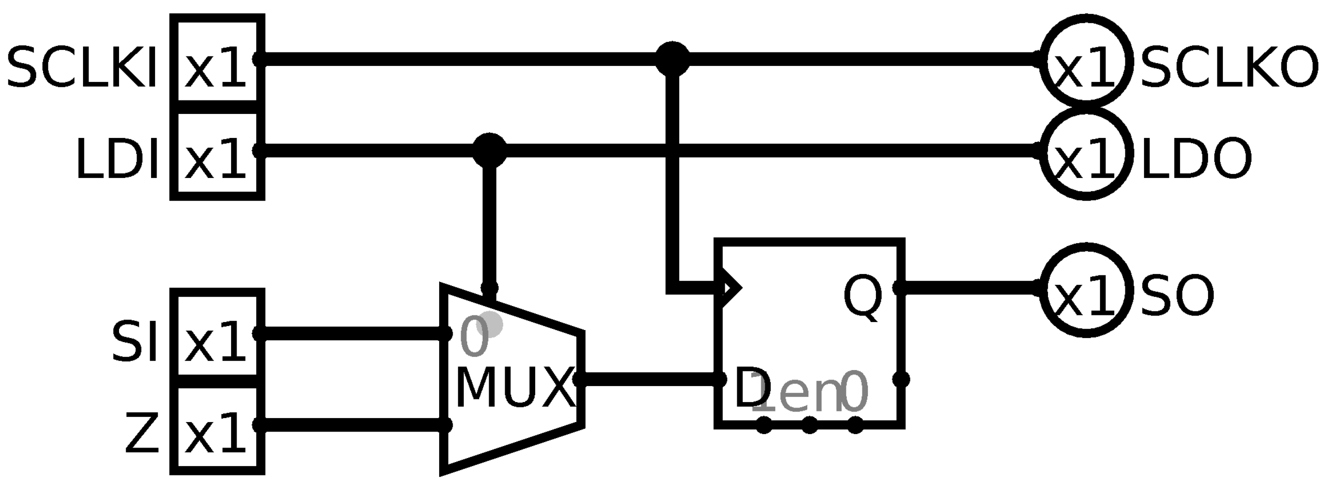

Figure 1.11: Parallel Load Shifter Register Bit-Slice

1.4.5 Programmable Interconnect Network

Bit-slicing Scheme

The diagram below showns just the PIN in bit-slice form. One of the design decisions made while determining the slice interconnects was to also pass the `PCLK` from slice to slice. The alternative was to simply connect each slices `PCLKI` to the main `PCLKI` pin at a higher level. We wanted to avoid as much manual layout as possible so it determined to be easier and cleaner to route the clock in such as way that it would be automatically connected when we layout the slice array.

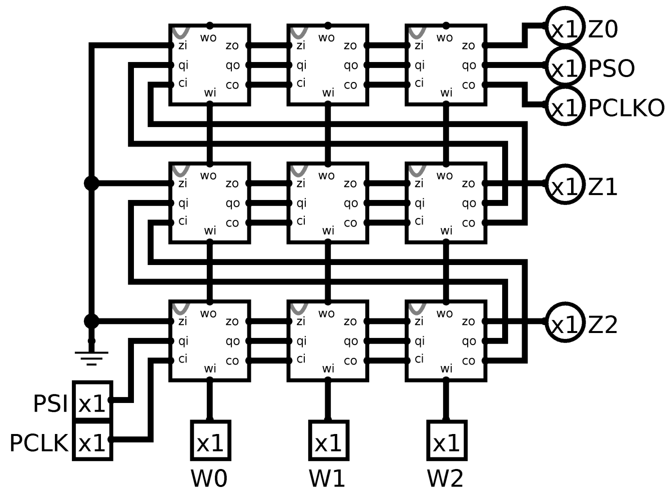

Figure 1.12: Bit-Sliced Programmable Interconnect Network (3-Bit Configuration)

Bit-Slice

The PIN bit-slices, one of which is shown below, is what drive the whole functionality of our chip. `WI` is the input values bit for the current column. If that bit is set and this slice is configured as 'connected' we want to output a logic high on the `Z` bus simultaneously. We cannot, however, just simply `AND` these two values together and attach it to the bus as this would allow for multiple slices to drive or sink the bus. To avoid this we use an `OR` gate to determine if the slice behind us is outputting a 1. If so we just pass it along. If we want to output a 1 it is also no problem as the `OR` will accommodate us as well.

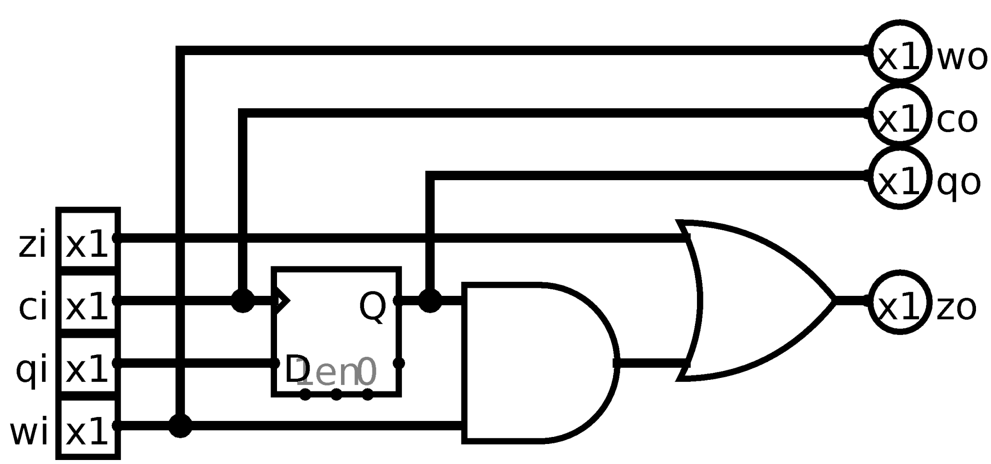

Figure 1.13: Programmable Interconnect Network Bit-Slice

1.5 VHDL Models

1.5.1 Top Level

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity top is
    generic(
        n : integer := 3
    );
    port(
        psi  : in  std_logic;
        pso  : out std_logic;
        pclk : in  std_logic;
        si   : in  std_logic;
        so   : out std_logic;
        sclk : in  std_logic;
        ld   : in  std_logic;
        test : in  std_logic
    );
end top;

architecture rtl of top is

    -- output of pin
    signal z : std_logic_vector((n-1) downto 0) := (others => '0');
    -- parallel output of shifter
    signal w : std_logic_vector((n-1) downto 0) := (others => '0');

    signal pin_clk : std_logic;
    signal pin_psi : std_logic;
    signal pin_pso : std_logic;
    signal shift_out : std_logic;

begin

    -- test mode mux connects shifter and pin together
    test_mux_1 : entity work.mux2x1 port map(pclk,      sclk,      test, pin_clk);
    test_mux_2 : entity work.mux2x1 port map(psi ,      shift_out, test, pin_psi);
    test_mux_3 : entity work.mux2x1 port map(shift_out, pin_pso,   test, so);

    pin : entity work.pin
    generic map(
        n => n
    )
    port map(
        clk => pin_clk,
        psi => pin_psi,
        pso => pin_pso,
        z => z,
        w => w
    );

    pso <= pin_pso;

    shifter : entity work.shift
    generic map(
        n => n
    )
    port map(
        clk => sclk,
        si => si,
        so => shift_out,
        ld => ld,
        z => z,
        w => w
    );

end rtl;
```

Listing 1.1: Top Level VHDL Module

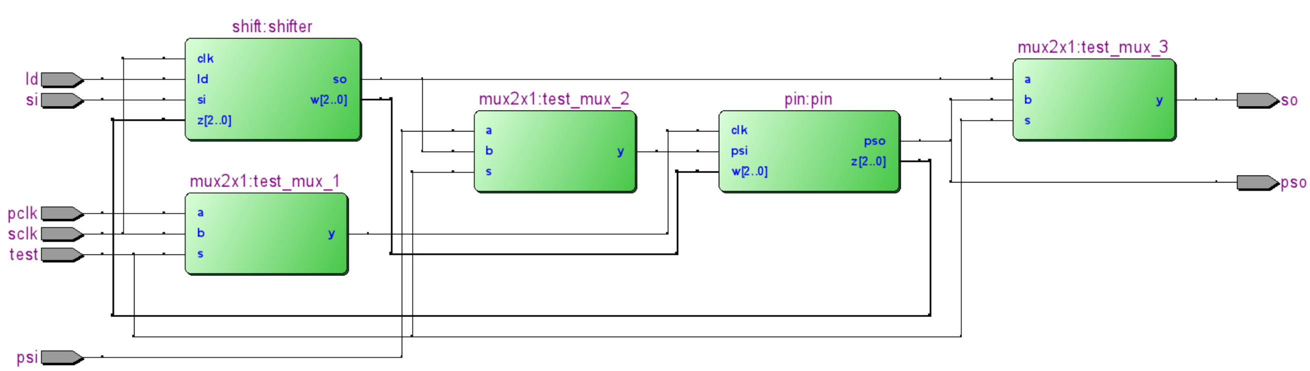

Figure 1.14: Top Level Generated RTL Diagram

1.5.2 PIN

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity pin is
    generic(
        n : integer := 3
    );
    port(
        clk  : in  std_logic;
        psi  : in  std_logic;
        pso  : out std_logic;
        z    : out std_logic_vector((n-1) downto 0);
        w    : in  std_logic_vector((n-1) downto 0)
    );
end pin;

architecture rtl of pin is

    component pin_slice is
        port(
            zi : in std_logic;
            qi : in std_logic;
            wi : in std_logic;
            ci : in std_logic;
            zo : out std_logic;
            qo : out std_logic;
            wo : out std_logic;
            co : out std_logic
        );
    end component;

    -- carray_array(row, col)
    type carry_array is array (0 to n, 0 to n) of std_logic;
    signal zc : carry_array;
    signal cc : carry_array;
    signal wc : carry_array;
    signal qc : carry_array;

    begin

    -- setup first and last inputs for each row
    z_connect : for i in 0 to n-1 generate
        zc(i, 0) <= '0';
        z(i) <= zc(i, n);
    end generate;

    -- setup first inputs for each column
    w_connect : for i in 0 to n-1 generate
        wc(0, i) <= w(i);
    end generate;

    -- setup row transfer
    -- (last output of row to first input of next row)
    r_connect : for i in 0 to n-2 generate
        qc(i, 0) <= qc(i+1, n);
        cc(i, 0) <= cc(i+1, n);
    end generate;

    -- connect external inputs
    qc(n-1, 0) <= psi;
    cc(n-1, 0) <= clk;
    pso <= qc(0, n);

    -- generate the grid of slices
    pin_z_gen : for zz in 0 to n-1 generate
        pin_w_gen : for ww in 0 to n-1 generate
            pin_i: pin_slice port map(
                zi => zc(zz, ww),
                qi => qc(zz, ww),
                wi => wc(zz, ww),
                ci => cc(zz, ww),
                zo => zc(zz, ww+1),
                qo => qc(zz, ww+1),
                wo => wc(zz+1, ww),
                co => cc(zz, ww+1)
            );
        end generate;
    end generate;

end rtl;
```

Listing 1.2: PIN VHDL Module

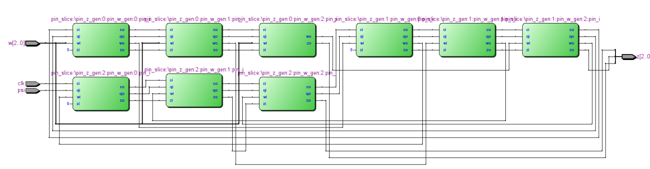

Figure 1.15: Pin Generated RTL Diagram

1.5.3 PIN Slice

```
library ieee;
use ieee.std_logic_1164.all;

entity pin_slice is
    port(
        zi : in std_logic;
        qi : in std_logic;
        wi : in std_logic;
        ci : in std_logic;
        zo : out std_logic;
        qo : out std_logic;
        wo : out std_logic;
        co : out std_logic
    );
end pin_slice;

architecture rtl of pin_slice is

    signal g1_o : std_logic := '0';
    signal g2_o : std_logic := '0';

begin

    g1 : entity work.dffposx1 port map(ci, qi, g1_o);
    g2 : entity work.aoi21x1  port map(wi, g1_o, zi, g2_o);
    g3 : entity work.invx1    port map(g2_o, zo);

    -- pass through
    co <= ci;
    wo <= wi;
    qo <= g1_o;

end rtl;
```

Listing 1.3: PIN Slice VHDL Module

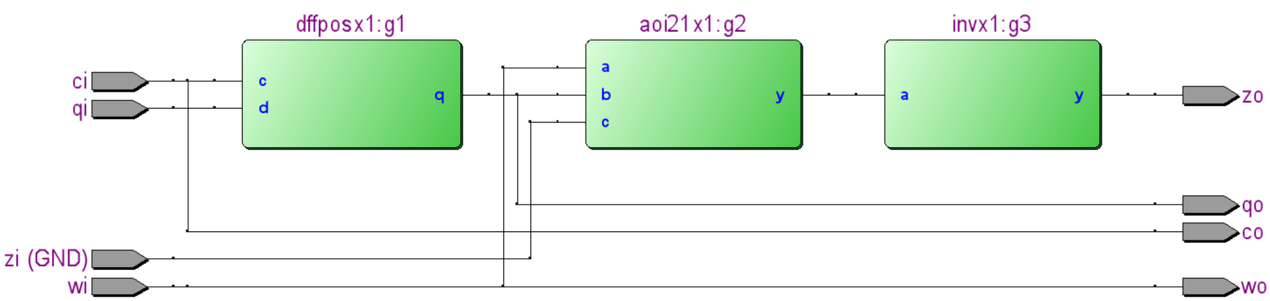

Figure 1.16: Pin Slice Generated RTL Diagram

1.5.4 Shifter

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity shift is
    generic(
        n : integer := 3
    );
    port(
        clk  : in  std_logic;
        ld   : in  std_logic;
        si   : in  std_logic;
        so   : out std_logic;
        z    : in  std_logic_vector((n-1) downto 0);
        w    : out std_logic_vector((n-1) downto 0)
    );
end shift;

architecture rtl of shift is

    component shift_slice
        port(
            clki   : in  std_logic;
            clko   : out std_logic;
            ldi    : in  std_logic;
            ldo    : out std_logic;
            si     : in  std_logic;
            so     : out std_logic;
            z      : in  std_logic
        );
    end component;

    -- vector to hold  values between slices
    signal c_so : std_logic_vector(n downto 0) := (others => '0');
    signal c_clk : std_logic_vector(n downto 0) := (others => '0');
    signal c_ld : std_logic_vector(n downto 0) := (others => '0');

    begin

    -- input of slice 0 comes from module input
    c_so(0) <= si;
    c_ld(0) <= ld;
    c_clk(0) <= clk;

    -- final shift output comes from output of last slice
    so <= c_so(n);

    -- generate N slices
    shift_gen : for i in 0 to n-1 generate
        shift_i: shift_slice port map(
            clki => c_clk(i),
            clko => c_clk(i+1),
            ldi => c_ld(i),
            ldo => c_ld(i+1),
            si => c_so(i),
            so => c_so(i+1),
            z => z(i)
        );
    end generate;

    -- connect the output of each slice to parallel output vector
    connect : for i in 0 to n-1 generate
        w(i) <= c_so(i+1);
    end generate;

end rtl;
```

Listing 1.4: Parallel Load Shifter VHDL Module

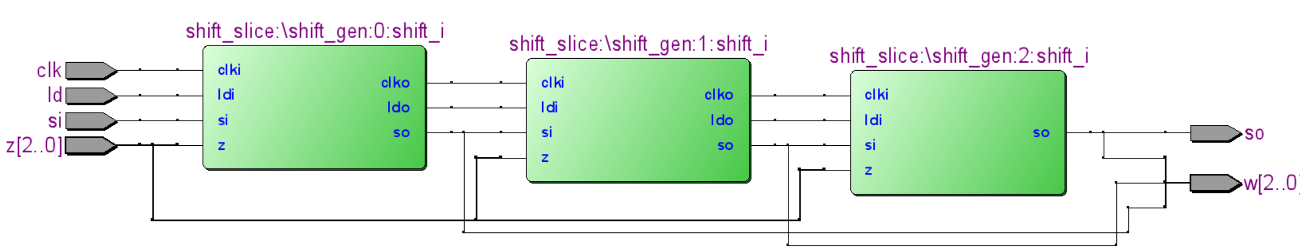

Figure 1.17: Shifter Generated RTL Diagram

1.5.5 Shifter Slice

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity shift_slice is
    port(
        clki : in  std_logic;
        clko : out std_logic;
        ldi    : in  std_logic;
        ldo    : out std_logic;
        si     : in  std_logic;
        so     : out std_logic;
        z      : in  std_logic
    );
end shift_slice;

architecture rtl of shift_slice is

    signal g1_o : std_logic := '0';

begin

    g1 : entity work.mux2x1   port map(si, z, ldi, g1_o);
    g2 : entity work.dffposx1 port map(clki, g1_o, so);

    -- pass through
    clko <= clki;
    ldo <= ldi;

end rtl;
```

Listing 1.5: Parallel Load Shifter Slice VHDL Module

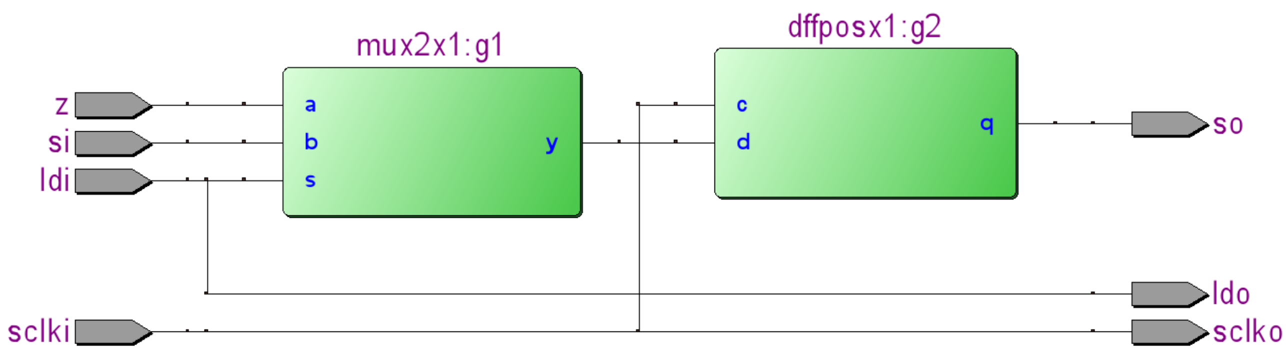

Figure 1.18: Shifter Slice Generated RTL Diagram

1.5.6 Gates

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity aoi21x1 is
    generic(delay : time := 0 ps);
    port(
        a : in std_logic;
        b : in std_logic;
        c : in std_logic;
        y : out std_logic
    );
end aoi21x1;

architecture rtl of aoi21x1 is begin
    process(a, b, c) begin
        y <= not ((a and b) or c) after delay;
    end process;
end rtl;
```

Listing 1.6: AOI21X1 VHDL Module

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity dffposx1 is
    generic(delay : time := 0 ps);
    port(
        c : in std_logic;
        d : in std_logic;
        q : out std_logic := '0'
    );
end dffposx1;

architecture rtl of dffposx1 is begin
    process(c) begin
        if rising_edge(c) then
            q <= d after delay;
        end if;
    end process;
end rtl;
```

Listing 1.7: DFFPOSX1 VHDL Module

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity invx1 is
    generic(delay : time := 0 ps);
    port(
        a : in std_logic;
        y : out std_logic
    );
end invx1;

architecture rtl of invx1 is begin
    y <= not a after delay;
end rtl;
```

Listing 1.8: INVX1 VHDL Module

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity mux2x1 is
    generic(delay : time := 0 ps);
    port(
        a : in std_logic;
        b : in std_logic;
        s : in std_logic;
        y : out std_logic
    );
end mux2x1;

architecture rtl of mux2x1 is begin
    process(a, b, s) begin
        if (s = '1') then
            y <= b after delay;
        else
            y <= a after delay;
        end if;
    end process;
end rtl;
```

Listing 1.9: MUX2X1 VHDL Module

1.6 VHDL Test Benches

1.6.1 Top Level Functional

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use work.txt_util.all;

entity top_tb is
    generic(
        stim_file : string := "vectors_3_bit.sim"
    );
end top_tb;

architecture tb_rtl of top_tb is

    constant n : integer := 3;

    signal psi  : std_logic := '0';
    signal pso  : std_logic;
    signal pclk : std_logic := '0';
    signal si   : std_logic := '0';
    signal so   : std_logic;
    signal sclk : std_logic := '0';
    signal ld   : std_logic := '0';
    signal test : std_logic := '0';

    component top
        generic(
            n : integer := 3
        );
        port(
            psi  : in  std_logic;
            pso  : out std_logic;
            pclk : in  std_logic;
            si   : in  std_logic;
            so   : out std_logic;
            sclk : in  std_logic;
            ld   : in  std_logic;
            test : in  std_logic
        );
    end component;

    signal pin_vector    : std_logic_vector((n*n)-1 downto 0);
    signal shift_vector  : std_logic_vector(n-1 downto 0);
    signal result_vector : std_logic_vector(n-1 downto 0);

    file stimulus : TEXT open read_mode is stim_file;

begin

    uut : top
    generic map(
        n => n
    )
    port map(
        psi  => psi,
        pso  => pso,
        pclk => pclk,
        si   => si,
        so   => so,
        sclk => sclk,
        ld   => ld,
        test => test
    );

    process

        procedure clock_shifter is begin
            sclk <= '1';
            wait for 20 ns;
            sclk <= '0';
            wait for 20 ns;
        end procedure clock_shifter;

        procedure clock_pin is begin
            pclk <= '1';
            wait for 20 ns;
            pclk <= '0';
            wait for 20 ns;
        end procedure clock_pin;

        variable l: line;
        variable pin_str: string(1 to n*n);
        variable shf_str: string(1 to n);

    begin

        while not endfile(stimulus) loop

            -- load stimulus for this test
            readline(stimulus, l); read(l, pin_str);
            pin_vector <= to_std_logic_vector(pin_str);

            readline(stimulus, l); read(l, shf_str);
            shift_vector <= to_std_logic_vector(shf_str);

            readline(stimulus, l); read(l, shf_str);
            result_vector <= to_std_logic_vector(shf_str);

            wait for 100 ns;

```

```
            -- clock in the pin
            for i in 0 to (n*n)-1 loop
                psi <= pin_vector(i);
                wait for 20 ns;
                clock_pin;
            end loop;

            -- clock in the value
            for i in 0 to n-1 loop
                si <= shift_vector(i);
                wait for 20 ns;
                clock_shifter;
            end loop;

            -- pull latch high so the first result
            -- loop will trigger the latch
            ld <= '1';
            wait for 20 ns;

            -- clock out result and check it
            for i in 0 to n-1 loop
                clock_shifter;
                assert so = result_vector(i) report "Test Failed!";
                ld <= '0';
                wait for 20 ns;
            end loop;

        end loop;

        report "Test Complete" severity note;
        wait;

    end process;

end tb_rtl;
```

Listing 1.10: Top Level VHDL Test Bench

We decided to write a small Python script to generate the expected output vector for all possible PIN configurations and input values. Our test bench then runs through all of these vectors and checks if the output vector from our VHDL design matches the known output.

```
N = 3

out = open("vectors_%d_bit.sim"%N, "w")

# loop through all possible pins and shift values
for pin in range(2**(N*N)):
    for shift in range(2**N):

        # get bit strings
        pin_bits = bin(pin).replace("0b", "").zfill(N*N)
        shift_bits = bin(shift).replace("0b", "").zfill(N)
        result_bits = ["0"]*N

        # calculate the expected result
        for z in range(N):
            for w in range(N):
                if pin_bits[(N-1-z)*N+w] == "1" and shift_bits[w] == "1":
                    result_bits[z] = "1"

        # writeout results
        out.write("%s\n" % pin_bits)
        out.write("%s\n" % shift_bits)
        out.write("%s\n" % "".join(result_bits))

out.close()
```

Listing 1.11: Python Vector Generator

1.6.2 Top Level Test Mode

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity top_test_tb is
end top_test_tb;

architecture tb_rtl of top_test_tb is

    constant n : integer := 3;

    signal psi  : std_logic := '0';
    signal pso  : std_logic;
    signal pclk : std_logic := '0';
    signal si   : std_logic := '0';
    signal so   : std_logic;
    signal sclk : std_logic := '0';
    signal ld   : std_logic := '0';
    signal test : std_logic := '0';

    component top
        generic(
            n : integer := n
        );
        port(
            psi  : in  std_logic;
            pso  : out std_logic;
            pclk : in  std_logic;
            si   : in  std_logic;
            so   : out std_logic;
            sclk : in  std_logic;
            ld   : in  std_logic;
            test : in  std_logic
        );
    end component;

begin

    uut : top
    generic map(
        n => n
    )
    port map(
        psi  => psi,
        pso  => pso,
        pclk => pclk,
        si   => si,
        so   => so,
        sclk => sclk,
        ld   => ld,
        test => test
    );

    process
        procedure clock is begin
            sclk <= '1';
            wait for 20 ns;
            sclk <= '0';
            wait for 20 ns;
        end procedure clock;
    begin

        wait for 20 ns;

        -- pull test line high to enable test mode
        test <= '1';

        -- clock in a '1'
        si <= '1';
        wait for 20 ns;
        clock;

        -- clock in a '0'
        si <= '0';
        wait for 20 ns;
        clock;

        -- push the pulse through till just before the last FF
        -- (n*n)+n == number of flip flops
        -- 2 == we already did two clocks
        -- 1 == we want to stop before before final output
        for i in 1 to (n*n)+n-2-1 loop
            clock;
        end loop;

        -- check to make sure the bit in front of the pulse is 0
        assert so = '0' report "Bit leading pulse not 0";
        clock;
        -- check to make sure the pulse is 1
        assert so = '1' report "Pulse is not 1";
        clock;
        -- check to make sure the bit behind pulse is 0
        assert so = '0' report "Bit trailing pulse not 0";
        clock;

        report "Test Complete" severity note;
        wait;

    end process;

end tb_rtl;
```

Listing 1.12: Top Level Test Mode VHDL Test Bench

1.6.3 PIN Slice

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity pin_slice_tb is
end pin_slice_tb;

architecture tb_rtl of pin_slice_tb is

    signal zi : std_logic := '0';
    signal qi : std_logic := '0';
    signal wi : std_logic := '0';
    signal ci : std_logic := '0';
    signal zo : std_logic;
    signal qo : std_logic;
    signal wo : std_logic;
    signal co : std_logic;

    component pin_slice
        port(
            zi : in std_logic;
            qi : in std_logic;
            wi : in std_logic;
            ci : in std_logic;
            zo : out std_logic;
            qo : out std_logic;
            wo : out std_logic;
            co : out std_logic
        );
    end component;

begin

    uut : pin_slice
    port map(
        zi => zi,
        qi => qi,
        wi => wi,
        ci => ci,
        zo => zo,
        qo => qo,
        wo => wo,
        co => co
    );

    process
        type pattern_type is record
            -- inputs
            zi, qi, wi : std_logic;
            -- output
            zo : std_logic;
        end record;

        type pattern_array is array (natural range <>) of pattern_type;
        constant patterns : pattern_array :=
        --zi  qi  wi   zo
        (('0','0','0', '0'),
         ('0','0','1', '0'),
         ('0','1','0', '0'),
         ('0','1','1', '1'),
         ('1','0','0', '1'),
         ('1','0','1', '1'),
         ('1','1','0', '1'),
         ('1','1','1', '1'));

    begin
        -- check each pattern
        for i in patterns'range loop

            -- set the inputs
            zi <= patterns(i).zi;
            qi <= patterns(i).qi;
            wi <= patterns(i).wi;
            wait for 10 ns;

            -- pulse the clock and check clock passthrough
            ci <= '1';
            wait for 10 ns;
            assert co = '1' report "CO does not equal 1" severity error;
            ci <= '0';
            wait for 10 ns;
            assert co = '0' report "CO does not equal 0" severity error;

            -- check the outputs
            assert qo = patterns(i).qi report "QI not equal QO" severity error;
            assert zo = patterns(i).zo report "ZO does not match pattern" severity error;

        end loop;

        report "Test Complete" severity note;
        wait;

    end process;

end tb_rtl;
```

Listing 1.13: PIN Slice VHDL Test Bench

1.6.4 Shifter Slice

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity shift_slice_tb is
end shift_slice_tb;

architecture tb_rtl of shift_slice_tb is

    signal clki  : std_logic := '0';
    signal clko  : std_logic;
    signal ldi   : std_logic := '0';
    signal ldo   : std_logic;
    signal si    : std_logic := '0';
    signal so    : std_logic;
    signal z     : std_logic := '0';

    component shift_slice is
        port(
            clki  : in  std_logic;
            clko  : out std_logic;
            ldi   : in  std_logic;
            ldo   : out std_logic;
            si    : in  std_logic;
            so    : out std_logic;
            z     : in  std_logic
        );
    end component;

begin

    uut : shift_slice
    port map(
        clki  => clki,
        clko  => clko,
        ldi   => ldi  ,
        ldo   => ldo  ,
        si    => si   ,
        so    => so   ,
        z     => z
    );

    process
        type pattern_type is record
            -- inputs
            ldi, z, si : std_logic;
            -- output
            so : std_logic;
        end record;

        type pattern_array is array (natural range <>) of pattern_type;
        constant patterns : pattern_array :=
        --ldi  z  si   so
        (('0','0','0', '0'),
         ('0','0','1', '1'),
         ('0','1','0', '0'),
         ('0','1','1', '1'),
         ('1','0','0', '0'),
         ('1','0','1', '0'),
         ('1','1','0', '1'),
         ('1','1','1', '1'));

    begin
        -- check each pattern
        for i in patterns'range loop

            -- set the inputs
            ldi <= patterns(i).ldi;
            z   <= patterns(i).z;
            si  <= patterns(i).si;
            wait for 10 ns;

            -- pulse the clock and check clock passthrough
            clki <= '1';
            wait for 10 ns;
            assert clko = '1' report "SCLKO does not equal 1" severity error;
            assert ldo = patterns(i).ldi report "SCLKO does not equal 1" severity error;
            clki <= '0';
            wait for 10 ns;
            assert clko = '0' report "SCLKO does not equal 0" severity error;
            assert ldo = patterns(i).ldi report "SCLKO does not equal 1" severity error;

            -- check the output
            assert so = patterns(i).so report "SO is incorrect" severity error;

        end loop;

        report "Test Complete" severity note;
        wait;

    end process;

end tb_rtl;
```

Listing 1.14: Parallel Load Shifter Slice VHDL Test Bench

1.7 VHDL Test Bench Results

1.7.1 Top Level Functional

While our top level functional testbench is completely automated and does an exhaustive test on all possible inputs an example waveform is shown below. We can first see that we clock in a PIN configuration vector of `000001000` which enables slice `Z1W2`. We then shift in a value of `001`. Given these input vectors we expect the output vector to be `010`. We can see from the waveform below that we achieve the expected result.

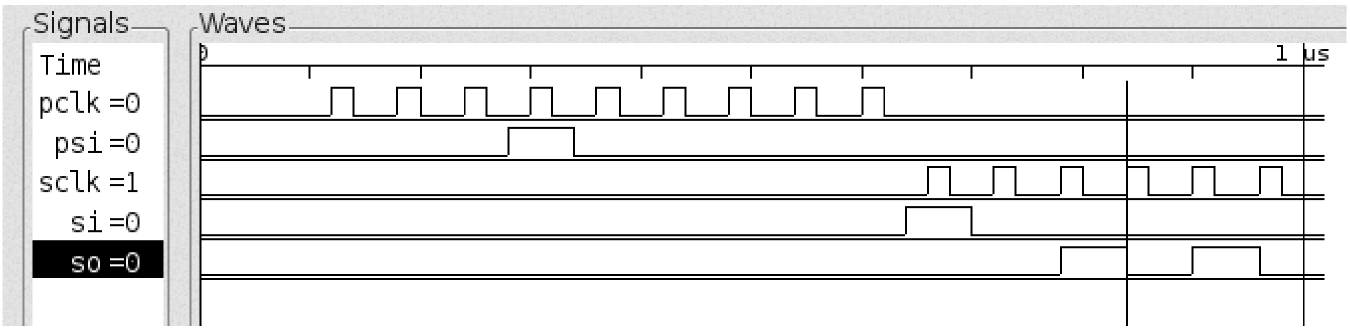

Figure 1.19: Top Level Functional Test Bench Waveform

1.7.2 Top Level Test Mode

Our top level test mode testbench is also completely automated. For this test we simply send a pulse through all the flip flops and count the number of clock cycles it takes for the pulse to come out the other end. For a 3-Bit configuration there are 3*3+3 flip flops so we expect the pulse to appear at the output after 12 clock pulses. We can see from the waveform below that we achieve the expected output.

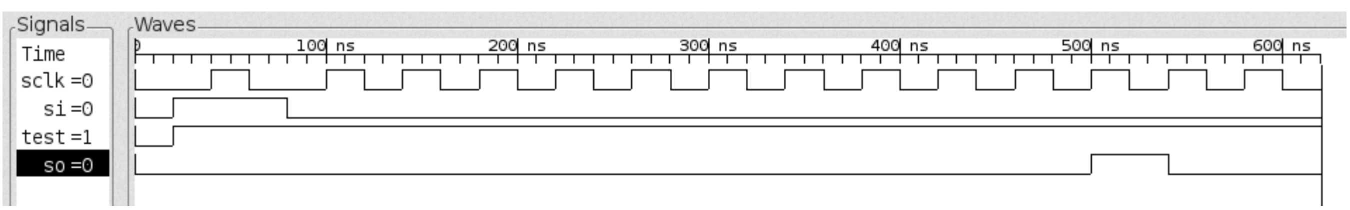

Figure 1.20: Top Level Test Mode Test Bench Waveform

1.8 Work Division

Task	Person
Pinout Diagram	Both
Explanation of Functionality	Both
Design Decisions	Both
Top Level Block Diagrams	Both
Shifter Block Diagrams	Qi
PIN Block Diagrams	Thrun
VHDL Shifter+TB	Qi
VHDL PIN+TB	Qi
VHDL Top+TB	Thrun
VHDL Top Test Mode TB	Thrun

Table 1.2: Task Assignment

Chapter 2

Part 2

2.1 Slice Layouts

2.1.1 PIN Slice Layout

The PIN slice layout consists of 3 cells from the provided library. They were arranged as to provided maximum material density and uniformity among the power rails. The connections in and out of the slice are arranged such that slices can be directly patterned together with little to no additional connections at a higher level.

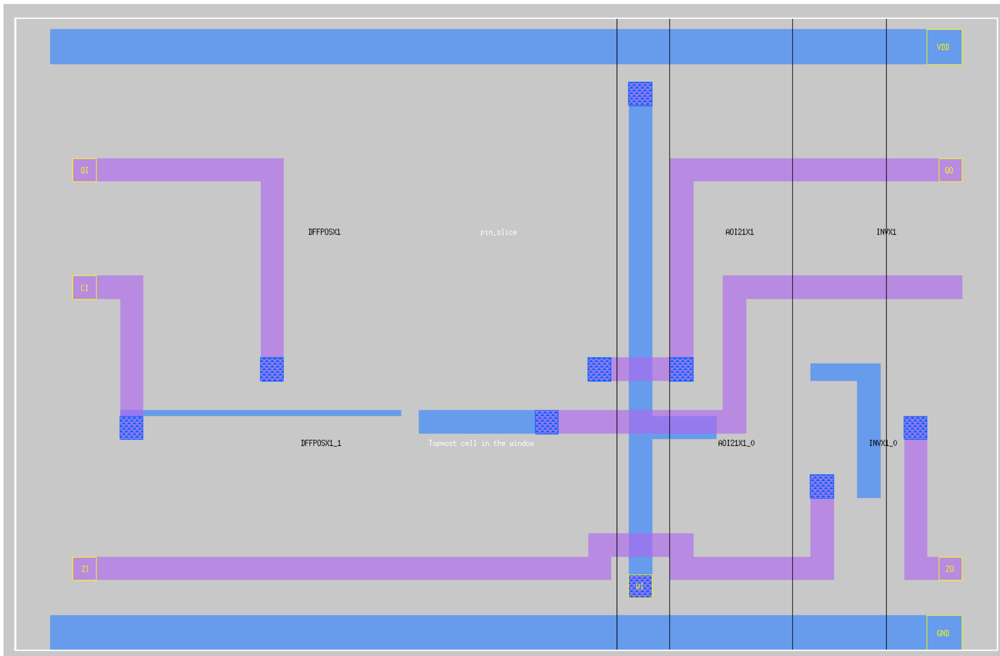

Figure 2.1: PIN Slice Layout

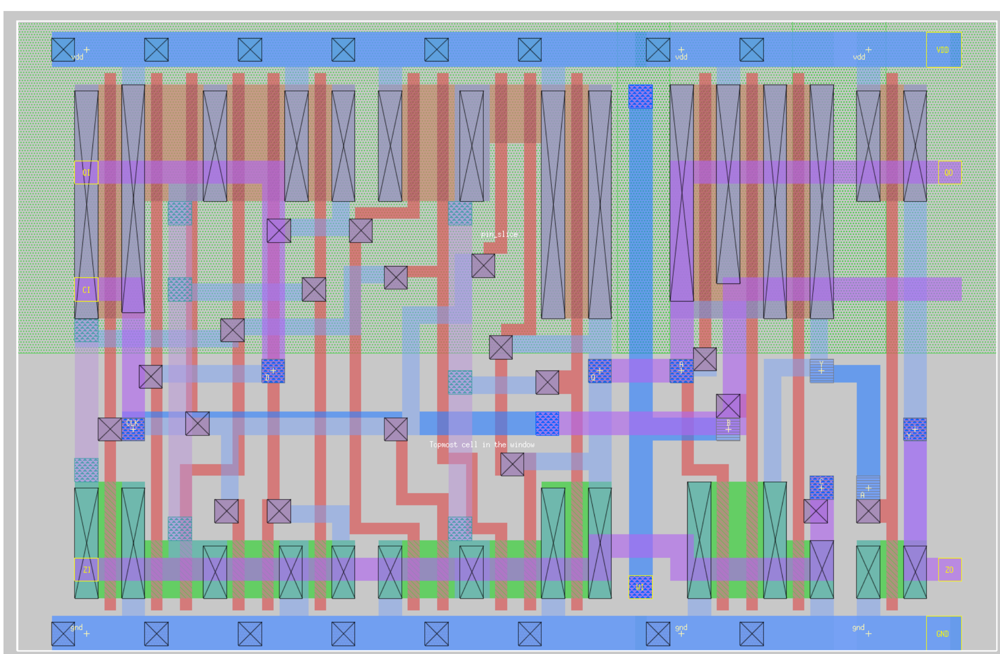

Figure 2.2: PIN Slice Layout Internal

2.1.2 Shift Slice Layout

Like the PIN slice, the Shift slice layout consists of 3 cells from the provided library. Again, we chose to keep a linear layout to maintain a uniform power rail between slices. Also, like the PIN slice, the connections in and out of this slice are laid out in such a manner that allows for direct patterning of slices with no additional work required.

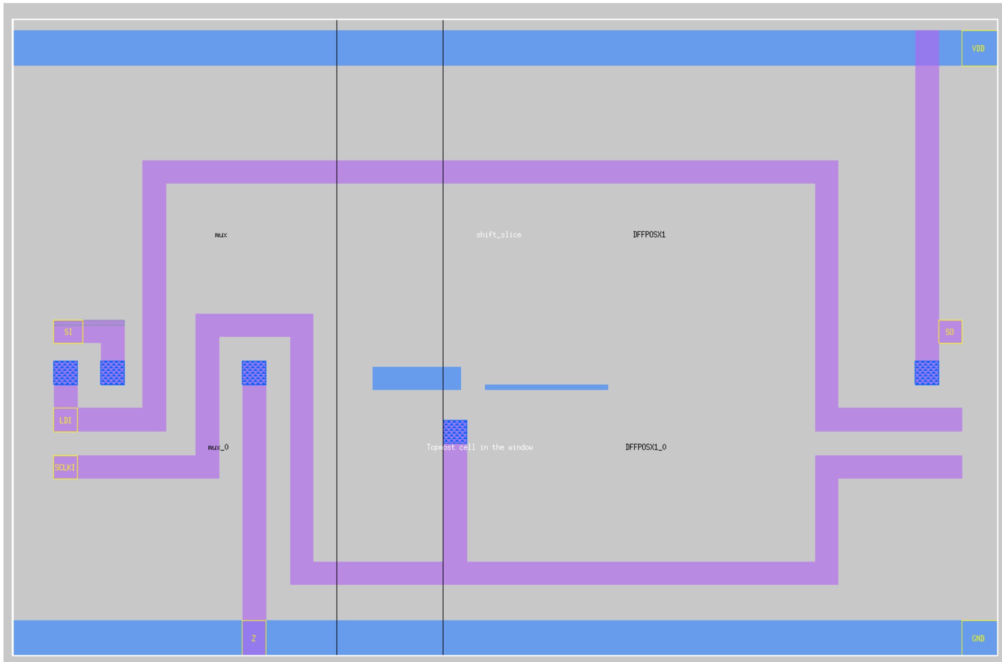

Figure 2.3: PIN Slice Layout

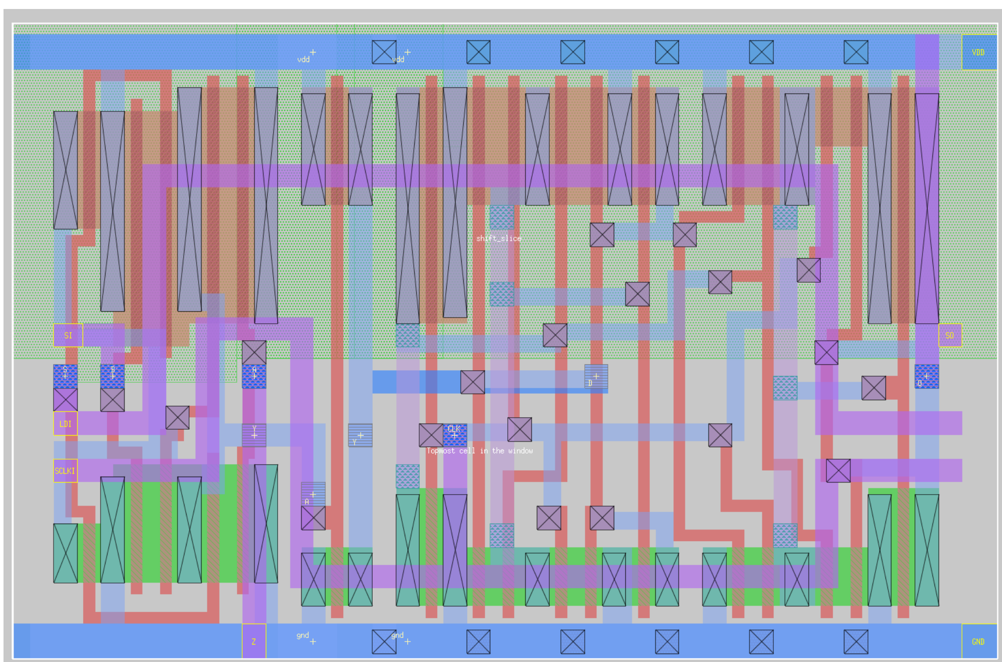

Figure 2.4: PIN Slice Layout Internal

2.2 Slice IRSIM Results

2.2.1 PIN Slice IRSIM Results

In order to test the PIN slice functionally a Python script was developed that translates our VHDL testbench patterns into a IRSIM command file. The output CMD file is not included here because of its length and the fact that it can be inferred from the Python script shown below.

```python
with open("pin_slice.cmd", "w") as f:

    f.write("stepsize 10\n")
    f.write("logfile pin_slice.log\n")
    f.write("h VDD\n")
    f.write("l GND\n")
    f.write("vector CLK CI\n")
    f.write("clock CLK 0 1\n")
    f.write("ana CI ZI QI WI ZO QO\n")
    f.write("w   CI ZI QI WI ZO QO\n")

    # zi  qi  wi   zo
    patterns = (
        ('l','l','l', 'l'),
        ('l','l','h', 'l'),
        ('l','h','l', 'l'),
        ('l','h','h', 'h'),
        ('h','l','l', 'h'),
        ('h','l','h', 'h'),
        ('h','h','l', 'h'),
        ('h','h','h', 'h')
    )

    for zi, qi, wi, zo in patterns:
        f.write("%s ZI\n" % zi)
        f.write("%s QI\n" % qi)
        f.write("%s WI\n" % wi)
        f.write("c CLK\n")
```

Listing 2.1: Python PIN Slice IRSIM CMD File Generator

The resulting IRSIM waveform, shown below, illustrates that we achieve correct functional behavior for our slice layout.

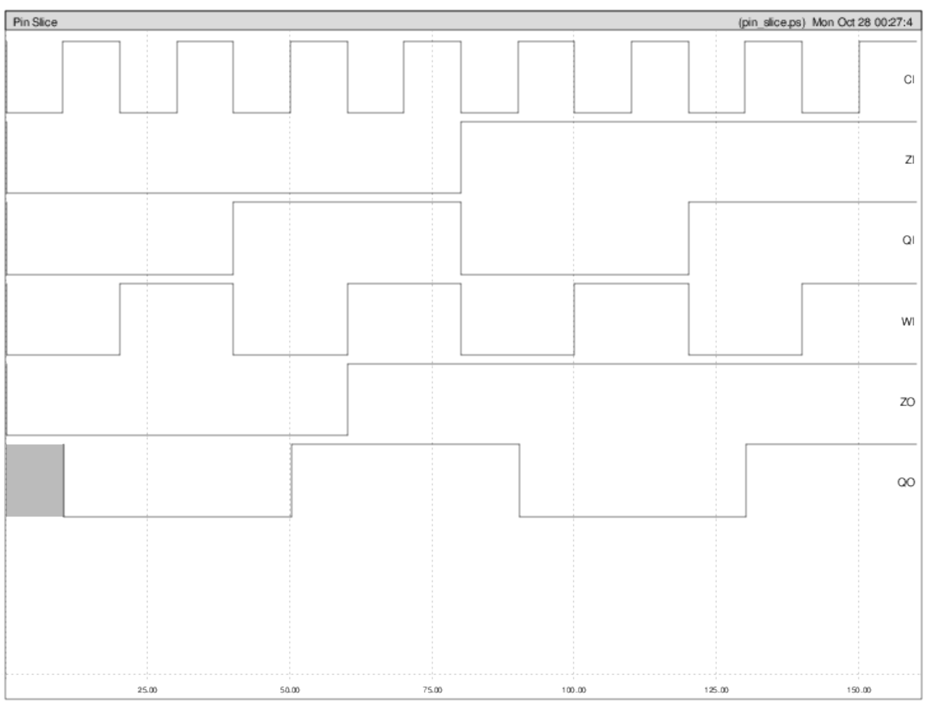

Figure 2.5: PIN Slice IRSIM Functional Results

The critical path through the PIN slice is highlighted in red in the diagram below. Knowing this path we constructed a simple CMD file to toggle the `qi` pin high and low on two consecutive clock cycles. This gives us a rising and falling edge through the critical path that we were able to measure using the `PATH` command in IRSIM.

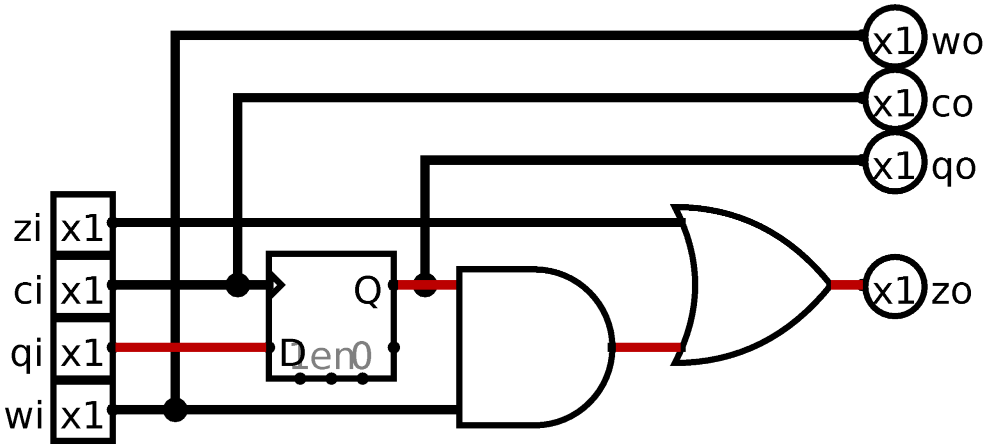

Figure 2.6: PIN Slice Critical Path

The textual output from IRSIM is shown below:

```
QO=0 WI=1 QI=0 ZI=0 ZO=0 CI=1
time = 20.000ns
QO=1 WI=1 QI=1 ZI=0 ZO=1 CI=1
time = 40.000ns
critical path for last transition of ZO:
  CI -> 1 @ 30.000ns , node was an input
  DFFPOSX1_1/a_66_6# -> 0 @ 30.141ns   (0.141ns)
  QO -> 1 @ 30.230ns   (0.089ns)
  INVX1_0/A -> 0 @ 30.285ns   (0.055ns)
  ZO -> 1 @ 30.286ns   (0.001ns)
QO=0 WI=1 QI=0 ZI=0 ZO=0 CI=1
time = 60.000ns
critical path for last transition of ZO:
  CI -> 1 @ 50.000ns , node was an input
  DFFPOSX1_1/a_2_6# -> 0 @ 50.039ns   (0.039ns)
  DFFPOSX1_1/a_66_6# -> 1 @ 50.198ns   (0.159ns)
  QO -> 0 @ 50.290ns   (0.092ns)
  INVX1_0/A -> 1 @ 50.347ns   (0.057ns)
  ZO -> 0 @ 50.348ns   (0.001ns)
```

Figure 2.7: PIN Slice IRSIM Critical Path Delay

Looking at the output we can see the two delays of the critical path. The first of the two delays is the output value going from a `0` to a `1` and the second is the output going from a `1` to a `0`. The table below tabulates the two delays and indicates that the falling edge delay was the worse of the two.

State Change	Delay	
0	0.286n	
1	0.348n	WORST

Table 2.1: PIN Slice IRSIM Critical Path Delays

2.2.2 Shift Slice IRSIM Results

Similarly to the PIN slice, for the Shift Slice we used the same Python script to generate a CMD file that matched our VHDL testbench patterns. The script is shown below:

```python
with open("shift_slice.cmd", "w") as f:

    f.write("stepsize 10\n")
    f.write("logfile shift_slice.log\n")
    f.write("h VDD\n")
    f.write("l GND\n")
    f.write("vector CLK SCLKI\n")
    f.write("clock CLK 0 1\n")
    f.write("ana SCLKI LDI SI Z SO\n")
    f.write("w  SCLKI LDI SI Z SO\n")

    # ldi  z  si   so
    patterns = (
        ('l','l','l', 'l'),
        ('l','l','h', 'h'),
        ('l','h','l', 'l'),
        ('l','h','h', 'h'),
        ('h','l','l', 'l'),
        ('h','l','h', 'l'),
        ('h','h','l', 'h'),
        ('h','h','h', 'h')
    )

    for ldi, z, si, so in patterns:
        f.write("%s LDI\n" % ldi)
        f.write("%s Z\n" % z)
        f.write("%s SI\n" % si)
        f.write("c CLK\n")
```

Listing 2.2: Python Shift Slice IRSIM CMD File Generator

Looking at the results we can see that our Shift Slice performs as expected and matches our VHDL simulations.

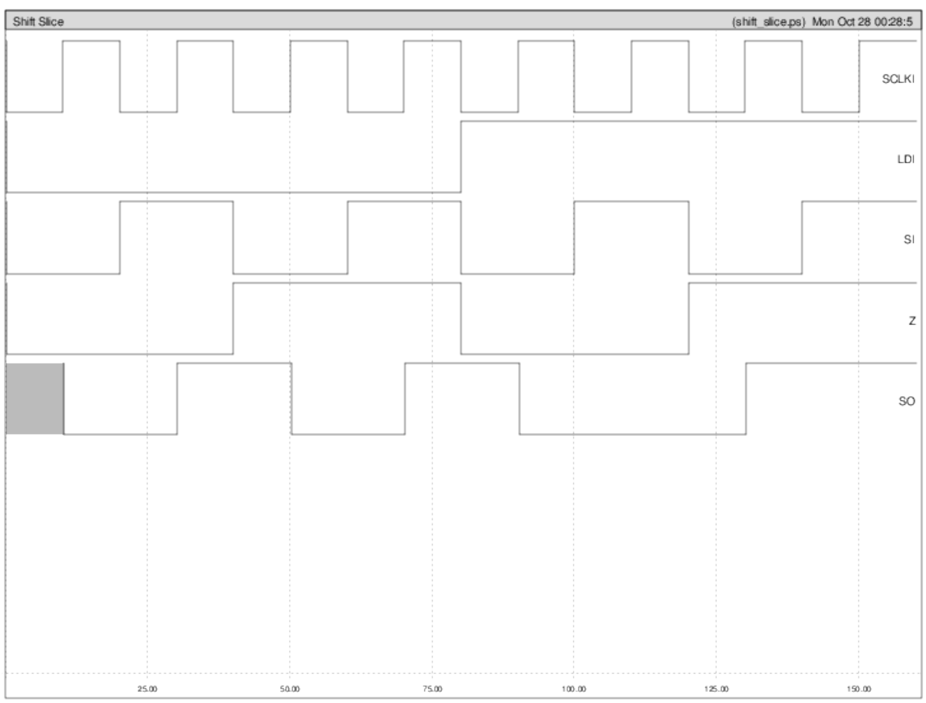

Figure 2.8: Shift Slice IRSIM Functional Results

The critical path through the shift slice is highlighted in red in the diagram below. Again, knowing this path we constructed a simple CMD file to drive a pulse through the path.

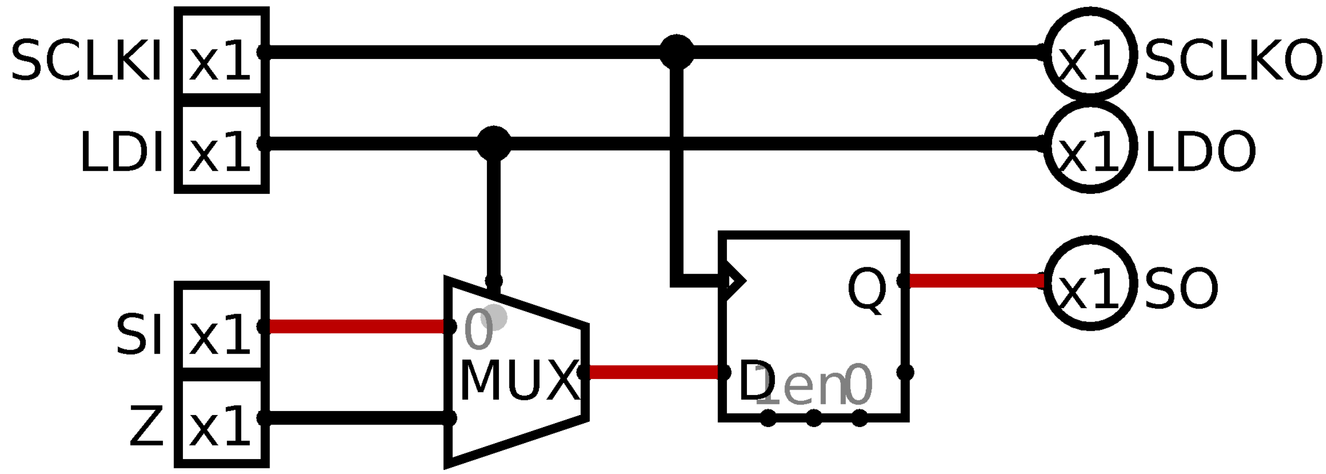

Figure 2.9: Shift Slice Critical Path

The textual output from IRSIM, shown below, provides us with two critical path delays one for the rising edge of the output and one for the falling edge.

```
SO=0 Z=0 SI=0 LDI=0 SCLKI=1
time = 20.000ns
SO=1 Z=0 SI=1 LDI=0 SCLKI=1
time = 40.000ns
critical path for last transition of SO:
  SCLKI -> 1 @ 30.000ns , node was an input
  DFFPOSX1_0/a_66_6# -> 0 @ 30.141ns   (0.141ns)
  SO -> 1 @ 30.181ns   (0.040ns)
SO=0 Z=0 SI=0 LDI=0 SCLKI=1
time = 60.000ns
critical path for last transition of SO:
  SCLKI -> 1 @ 50.000ns , node was an input
  DFFPOSX1_0/a_2_6# -> 0 @ 50.040ns   (0.040ns)
  DFFPOSX1_0/a_66_6# -> 1 @ 50.200ns   (0.160ns)
  SO -> 0 @ 50.242ns   (0.042ns)
```

Figure 2.10: Shift Slice IRSIM Critical Path Delay

Looking at the output we can see that again the falling edge has a greater propagation delay through the path. The table below summarizes the results for this slice.

State Change	Delay	
0	0.181n	
1	0.242n	WORST

Table 2.2: Shift Slice IRSIM Critical Path Delays

2.3 Slice Spice Results

2.3.1 PIN Slice Spice Results

We wanted to be able to functionally test our slices in HSpice in addition to analyzing the propagation delay. To do this another Python script was written that took our test patterns and wrote out the required Piecewise Linear (PWL) statements to generate them.

```
# zi  qi  wi   zo
patterns = (
    ('0','0','0', '0'),
    ('0','0','5', '0'),
    ('0','5','0', '0'),
    ('0','5','5', '5'),
    ('5','0','0', '5'),
    ('5','0','5', '5'),
    ('5','5','0', '5'),
    ('5','5','5', '5')
)

with open("pin_slice_all.sp", "w") as f:

    f.write("* Pin Slice Test All\n")

    f.write(".include ../../models/model_t36s.sp\n")
    f.write(".include ../magic/pin_slice.spice\n")

    for n in ("zi", "ci", "qi", "wi", "zo", "qo"):
        f.write(".ic v(%s) = 0\n" % n)

    f.write("VDD vdd gnd 5V\n")

    f.write("Vsclki ci gnd PULSE(0V 5V 10n 0 0 10n 20n)\n")

    o_zi = ""
    o_qi = ""
    o_wi = ""

    for i, (zi, qi, wi, zo) in enumerate(patterns):
        o_zi += "%dn %sV %fn %sV " % (i*20, zi, (i+1)*20-0.001, zi)
        o_qi += "%dn %sV %fn %sV " % (i*20, qi, (i+1)*20-0.001, qi)
        o_wi += "%dn %sV %fn %sV " % (i*20, wi, (i+1)*20-0.001, wi)

    f.write("Vzi zi gnd PWL(%s)\n" % o_zi)
    f.write("Vqi qi gnd PWL(%s)\n" % o_qi)
    f.write("Vwi wi gnd PWL(%s)\n" % o_wi)

    f.write(".option post\n")
    f.write(".tran 0.01n %dn\n" % (len(patterns)*20))
    f.write(".end\n")
```

Listing 2.3: Python PIN Slice Spice File Generator

In the figure below we can see the result of our functional Spice test. As you can see, it again matches our expected functionality once again proving the slice is operating correctly.

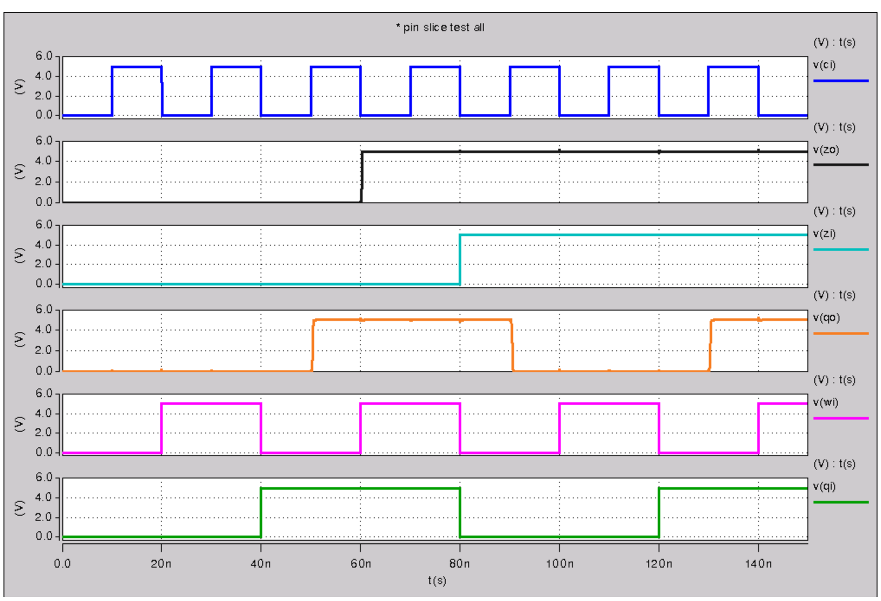

Figure 2.11: PIN Slice Spice Functional Results

In order to measure the critical path delay the patterns in the above Python program were modified to simply toggle the input line `QI`. The resulting output was then measured against the input clock `CI` to obtain the delays for both rising and falling edges.

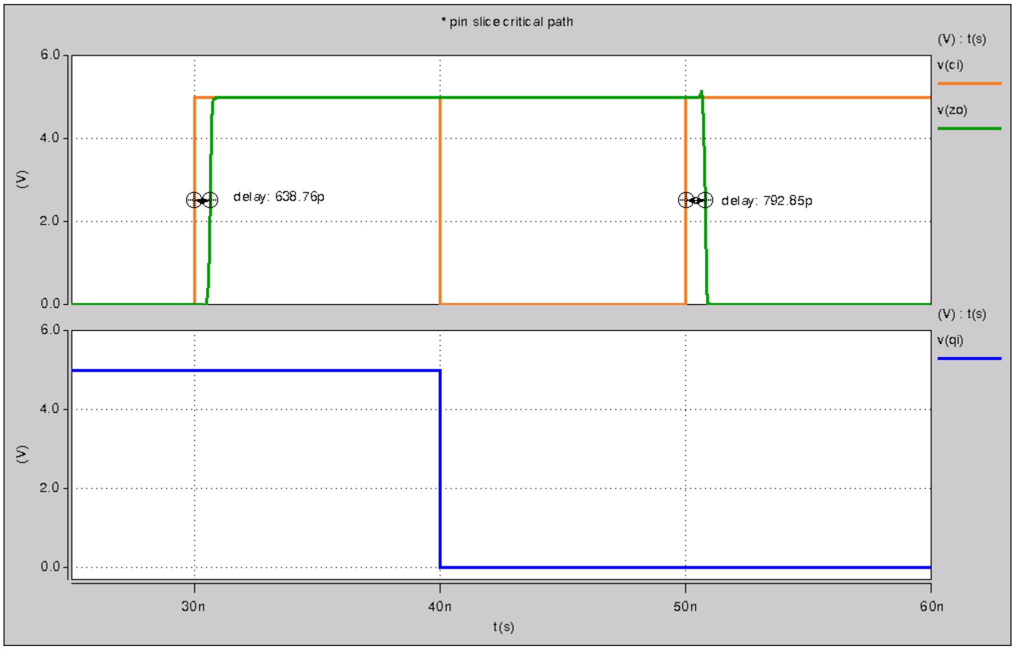

Figure 2.12: PIN Slice Spice Critical Path Delay

The delay times for each of these state changes is tabulated below.

State Change	Delay	
0	0.638n	
1	0.792n	WORST

Table 2.3: PIN Slice Spice Critical Path Delays

2.3.2 Shift Slice Spice Results

Similarly to the PIN slice spice tests we again wanted to be able to test the logical functionality of our slice in HSpice. The same Python script was utilized with a few tweaks to the variable names and pattern definitions in order to match this slice.

```
# ldi  z  si   so
patterns = (
    ('0','0','0', '0'),
    ('0','0','5', '5'),
    ('0','5','0', '0'),
    ('0','5','5', '5'),
    ('5','0','0', '0'),
    ('5','0','5', '0'),
    ('5','5','0', '5'),
    ('5','5','5', '5')
)

with open("shift_slice_all.sp", "w") as f:

        f.write("* Shift Slice Test All\n")

        f.write(".include ../../models/model_t36s.sp\n")
        f.write(".include ../magic/shift_slice.spice\n")

        for n in ("ldi", "z", "si", "so", "sclki"):
            f.write(".ic v(%s) = 0\n" % n)

        f.write("VDD vdd gnd 5V\n")

        f.write("Vsclki SCLKI gnd PULSE(0V 5V 10n 0 0 10n 20n)\n")

        o_ldi = ""
        o_z = ""
        o_si = ""

        for i, (ldi, z, si, so) in enumerate(patterns):
            o_ldi += "%dn %sV %fn %sV " % (i*20, ldi, (i+1)*20-0.001, ldi)
            o_z   += "%dn %sV %fn %sV " % (i*20, z,   (i+1)*20-0.001, z)
            o_si  += "%dn %sV %fn %sV " % (i*20, si,  (i+1)*20-0.001, si)

        f.write("Vldi ldi gnd PWL(%s)\n" % o_ldi)
        f.write("Vz   z   gnd PWL(%s)\n" % o_z)
        f.write("Vsi  si  gnd PWL(%s)\n" % o_si)

        f.write(".option post\n")
        f.write(".tran 0.01n %dn\n" % (len(patterns)*20))
        f.write(".end\n")
```

Listing 2.4: Python Shift Slice Spice File Generator

From the output waveform below we can see that our slice performed as we expected at a logical level.

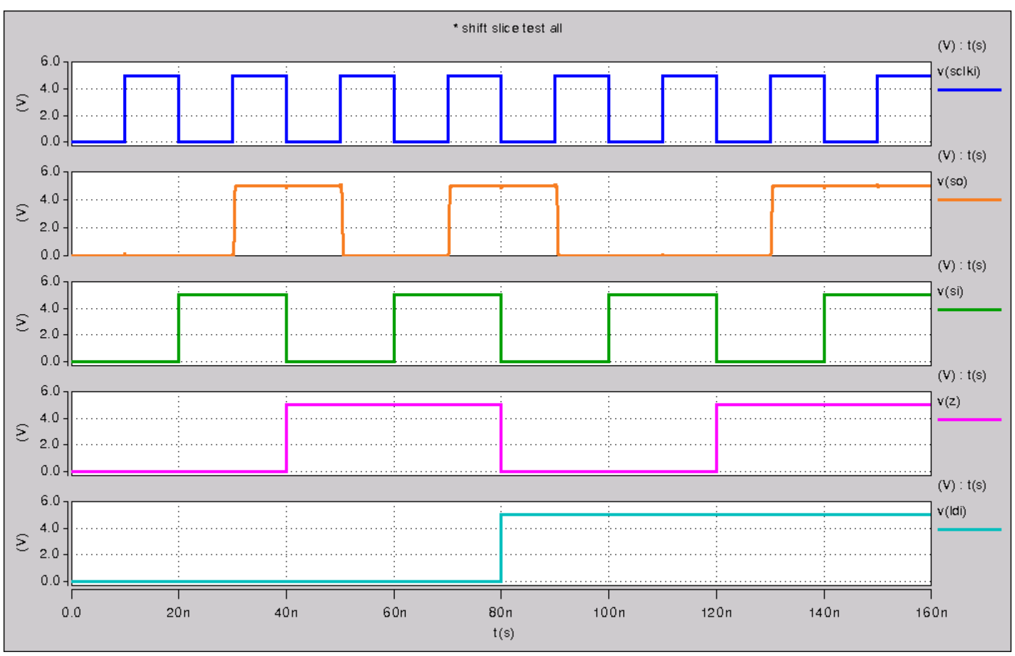

Figure 2.13: Shift Slice Spice Functional Results

To obtain the delays the input pattern was modified to send a single pulse through the critical path. The output was then referenced to the input clock in order to measure the propagation delay of each edge.

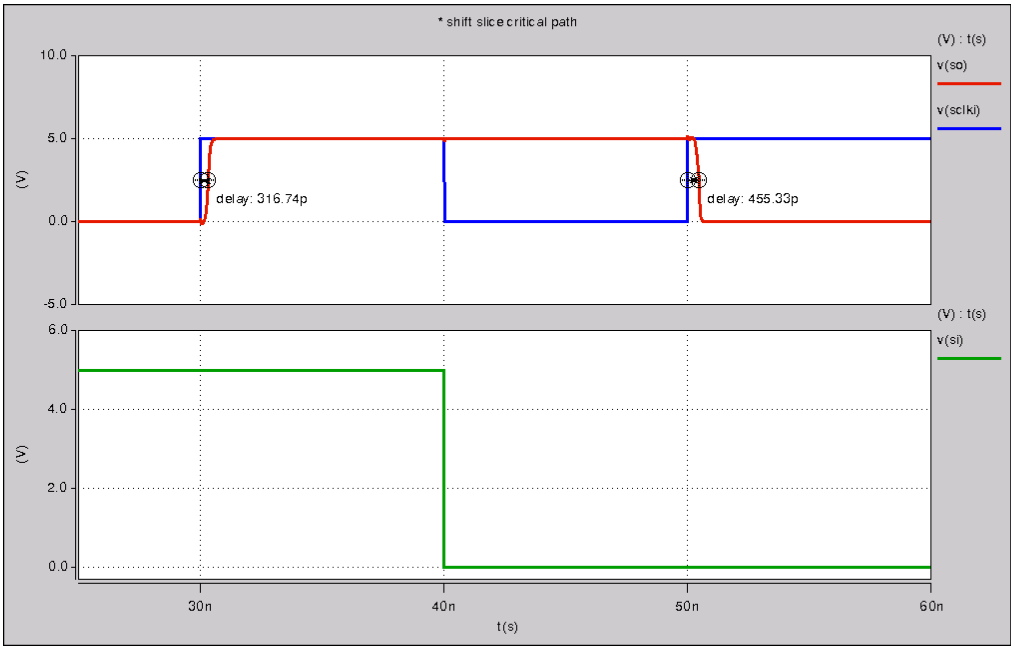

Figure 2.14: Shift Slice Spice Critical Path Delay

The delays of the rising and falling edges are tabulated below.

State Change	Delay	
0	0.316n	
1	0.455n	WORST

Table 2.4: Shift Slice Spice Critical Path Delays

2.4 Gate Spice Results

In order to figure out the worst case delay of each gate we generate an exhaustive list of input patterns that toggle the gate inputs in every such state that results in the outputs changing. With this we are trying to find the input state change that causes the worst case delay. We then measure each delay using the `.measure` directive and look for worst delay time.

2.4.1 DFFPOSX1 Spice Results

```python
patterns = ("0", "5", "0")

with open("dffposx1_test.sp", "w") as f:

        f.write("* DFFPOSX1 Test\n")

        f.write(".include ../../models/model_t36s.sp\n")
        f.write(".include ../magic/DFFPOSX1.spice\n")

        for n in ("clk", "d", "q"):
            f.write(".ic v(%s) = 0\n" % n)

        f.write("V1 vdd gnd  5V\n")

        f.write("Vclk clk gnd PULSE(0V 5V 10n 0 0 10n 20n)\n")

        o_d = ""

        for i, d in enumerate(patterns):
            o_d += "%dn %sV %fn %sV " % (i*20, d, (i+1)*20-0.001, d)

        f.write("Vd d gnd PWL(%s)\n" % o_d)

        f.write(".option post\n")
        f.write(".tran 0.01n %dn\n" % (len(patterns)*20))
        f.write(".meas tran delay_0 when v(q)=2.5 td=5n cross=1\n")
        f.write(".meas tran delay_1 when v(q)=2.5 td=5n cross=2\n")
        f.write(".end\n")
```

Listing 2.5: Python DFFPOSX1 Spice File Generator

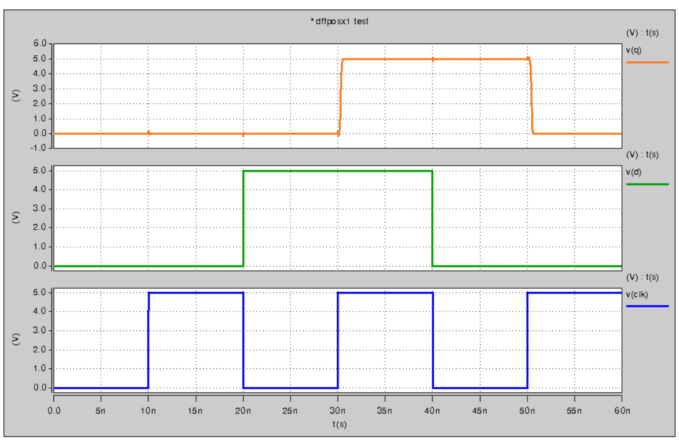

Figure 2.15: DFFPOSX1 Spice Results

State Change	Delay	
0	0.3011n	
1	0.4257n	WORST

Table 2.5: DFFPOSX1 Delays

2.4.2 AOI21X1 Spice Results

```python
import itertools

# C A B Y
patterns = (
    ('0','0','0', '5'),
    ('0','0','5', '5'),
    ('0','5','0', '5'),
    ('0','5','5', '0'),
    ('5','0','0', '0'),
    ('5','0','5', '0'),
    ('5','5','0', '0'),
    ('5','5','5', '0')
)

with open("aoi21x1_test.sp", "w") as f:

        f.write("* AOI21X1 Test\n")

        f.write(".include ../../models/model_t36s.sp\n")
        f.write(".include ../magic/AOI21X1.spice\n")

        for n in ("a", "b", "c", "y"):
            f.write(".ic v(%s) = 0\n" % n)

        f.write("V1 vdd gnd  5V\n")

        o_a = ""
        o_b = ""
        o_c = ""

        i = 0
        for state_1, state_2 in itertools.permutations(patterns, 2):
            # if this state change doesn't change the output, skip it
            if state_1[-1] == state_2[-1]: continue
            # if more than one input changed skip it
            if sum(1 for x, y in zip(state_1[:-1], state_2[:-1]) if x != y) > 1: continue
            # otherwise execute the state change
            print (state_1, state_2)
            for c, a, b, y in (state_1, state_2):
                o_a += "%dn %sV %fn %sV " % (i*20, a, (i+1)*20-0.00001, a)
                o_b += "%dn %sV %fn %sV " % (i*20, b, (i+1)*20-0.00001, b)
                o_c += "%dn %sV %fn %sV " % (i*20, c, (i+1)*20-0.00001, c)
                i += 1

        print i

        f.write("Va a gnd PWL(%s)\n" % o_a)
        f.write("Vb b gnd PWL(%s)\n" % o_b)
        f.write("Vc c gnd PWL(%s)\n" % o_c)

        f.write(".option post\n")
        f.write(".tran 0.01n %dn\n" % (i*20))

        # measure each crossing
        for n in range(0,(i/2)):
            f.write(".meas tran delay_%d when v(y)=2.5 td=%sn cross=1\n" % (n,(n*40)+10))

        f.write(".end\n")
```

Listing 2.6: Python AOI21X1 Spice File Generator

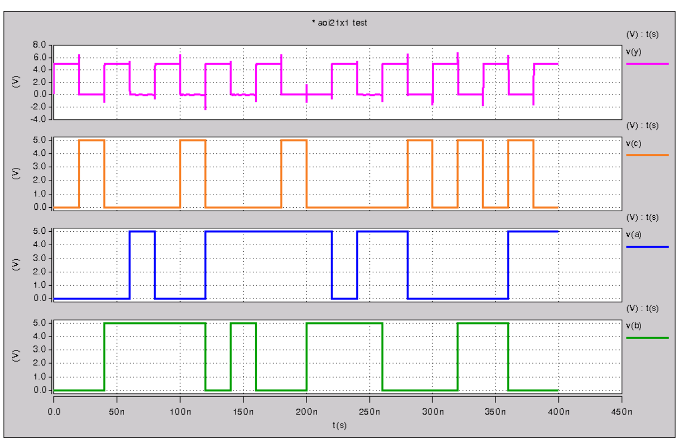

Figure 2.16: AOI21X1 Spice Results

State Change	Delay	
0	0.1075n	
1	0.1577n	WORST
2	0.1207n	
3	0.1434n	
4	0.1076n	
5	0.1457n	
6	0.1150n	
7	0.0491n	
8	0.0871n	
9	0.0580n	

Table 2.6: AOI21X1 Delays

2.4.3 MUX2X1 Spice Results

```python
import itertools

# S A B Y
patterns = (
    ('0','0','0', '5'),
    ('0','0','5', '0'),
    ('0','5','0', '5'),
    ('0','5','5', '0'),
    ('5','0','0', '5'),
    ('5','0','5', '5'),
    ('5','5','0', '0'),
    ('5','5','5', '0')
)

with open("mux2x1_test.sp", "w") as f:

        f.write("* MUX2X1 Test\n")

        f.write(".include ../../models/model_t36s.sp\n")
        f.write(".include ../magic/MUX2X1.spice\n")

        for n in ("s", "a", "b", "y"):
            f.write(".ic v(%s) = 0\n" % n)

        f.write("V1 vdd gnd  5V\n")

        o_a = ""
        o_b = ""
        o_s = ""

        i = 0
        for state_1, state_2 in itertools.permutations(patterns, 2):
            # if this state change doesn't change the output, skip it
            if state_1[-1] == state_2[-1]: continue
            # if more than one input changed skip it
            if sum(1 for x, y in zip(state_1[:-1], state_2[:-1]) if x != y) > 1: continue
            # otherwise execute the state change
            print (state_1, state_2)
            for s, a, b, y in (state_1, state_2):
                o_s += "%dn %sV %fn %sV " % (i*20, s, (i+1)*20-0.001, s)
                o_a += "%dn %sV %fn %sV " % (i*20, a, (i+1)*20-0.001, a)
                o_b += "%dn %sV %fn %sV " % (i*20, b, (i+1)*20-0.001, b)
                i += 1

        print i

        f.write("Vs s gnd PWL(%s)\n" % o_s)
        f.write("Va a gnd PWL(%s)\n" % o_a)
        f.write("Vb b gnd PWL(%s)\n" % o_b)

        f.write(".option post\n")
        f.write(".tran 0.01n %dn\n" % (i*20))

        # measure each crossing
        for n in range(0,(i/2)):
            f.write(".meas tran delay_%d when v(y)=2.5 td=%sn cross=1\n" % (n,(n*40)+10))

        f.write(".end\n")
```

Listing 2.7: Python MUX2X1 Spice File Generator

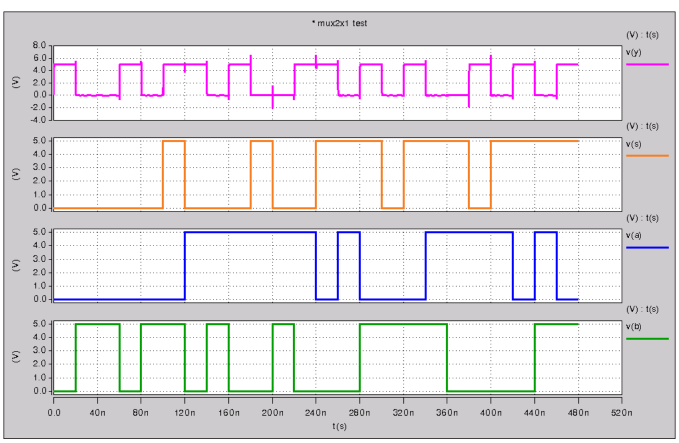

Figure 2.17: MUX2X1 Spice Results

State Change	Delay	
0	0.1536n	
1	0.1342n	
2	0.2569n	
3	0.1542n	
4	0.0832n	
5	0.1340n	
6	0.1390n	
7	0.2571n	WORST
8	0.1391n	
9	0.0788n	
10	0.1464n	
11	0.1467n	

Table 2.7: MUX2X1 Delays

2.4.4 INVX1 Spice Results

```python
patterns = ("0", "5", "0")

with open("invx1_test.sp", "w") as f:

        f.write("* INVX1 Test\n")

        f.write(".include ../../models/model_t36s.sp\n")
        f.write(".include ../magic/INVX1.spice\n")

        for n in ("a", "y"):
            f.write(".ic v(%s) = 0\n" % n)

        f.write("V1 vdd gnd 5V\n")

        o_a = ""

        for i, d in enumerate(patterns):
            o_a += "%dn %sV %fn %sV " % (i*20, d, (i+1)*20-0.001, d)

        f.write("Va a gnd PWL(%s)\n" % o_a)

        f.write(".option post\n")
        f.write(".tran 0.01n %dn\n" % (len(patterns)*20))
        # measure each crossing
        f.write(".meas tran delay_0 when v(y)=2.5 td=10n cross=1\n")
        f.write(".meas tran delay_1 when v(y)=2.5 td=30n cross=1\n")
        f.write(".end\n")
```

Listing 2.8: Python INVX1 Spice File Generator

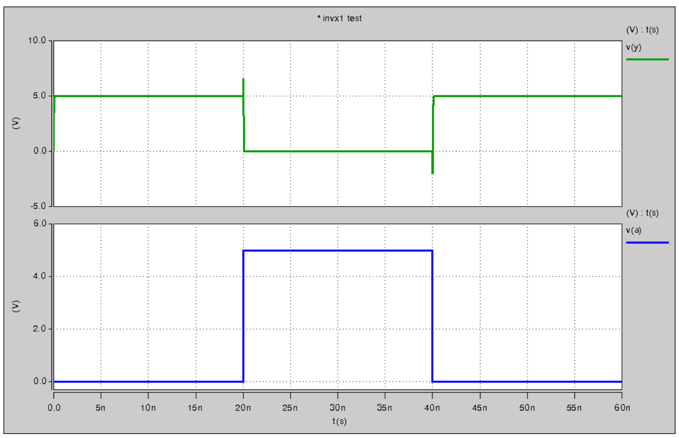

Figure 2.18: INVX1 Spice Results

State Change	Delay	
0	0.0550n	WORST
1	0.0407n	

Table 2.8: INVX1 Delays

2.4.5 Leaf Component Delay Summary

A table summarizing the worst delays for each gate is shown below.

Component	Worst Delay
AOI21X1	0.1577n
DFFPOSX1	0.4257n
INVX1	0.0550n
MUX2X1	0.2571n

Table 2.9: Worst Case Delay Summary

2.5 VHDL Models With Timing

Using the worst case leaf delays, found above, we updated our VHDL models in order to take the delay into account. All other modules and testbenches required no changes and were left as is.

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity aoi21x1 is
    generic(delay : time := 0.1577 ns);
    port(
        a : in std_logic;
        b : in std_logic;
        c : in std_logic;
        y : out std_logic
    );
end aoi21x1;

architecture rtl of aoi21x1 is begin
    process(a, b, c) begin
        y <= not ((a and b) or c) after delay;
    end process;
end rtl;
```

Listing 2.9: AOI21X1 VHDL Module With Delay

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity dffposx1 is
    generic(delay : time := 0.4257 ns);
    port(
        c : in std_logic;
        d : in std_logic;
        q : out std_logic := '0'
    );
end dffposx1;

architecture rtl of dffposx1 is begin
    process(c) begin
        if rising_edge(c) then
            q <= d after delay;
        end if;
    end process;
end rtl;
```

Listing 2.10: DFFPOSX1 VHDL Module With Delay

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity invx1 is
    generic(delay : time := 0.0550 ns);
    port(
        a : in std_logic;
        y : out std_logic
    );
end invx1;

architecture rtl of invx1 is begin
    y <= not a after delay;
end rtl;
```

Listing 2.11: INVX1 VHDL Module With Delay

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity mux2x1 is
    generic(delay : time := 0.2571 ns);
    port(
        a : in std_logic;
        b : in std_logic;
        s : in std_logic;
        y : out std_logic
    );
end mux2x1;

architecture rtl of mux2x1 is begin
    process(a, b, s) begin
        if (s = '1') then
            y <= b after delay;
        else
            y <= a after delay;
        end if;
    end process;
end rtl;
```

Listing 2.12: MUX2X1 VHDL Module With Delay

2.6 VHDL Testbench Results With Timing

2.6.1 VHDL Slice Testbench With Delays Waveform

Looking at the outputs of the slice test benches we can see that they perform as expected and match not only the original test benches but the IRSIM and HSpice functional test waveforms. Since we know the delays of each gate and which gates are in each cell there is no need to show a zoomed in waveform illustrating the delay in VHDL, we can simply add up the delays manually as there is no other delay introduced in the simulation.

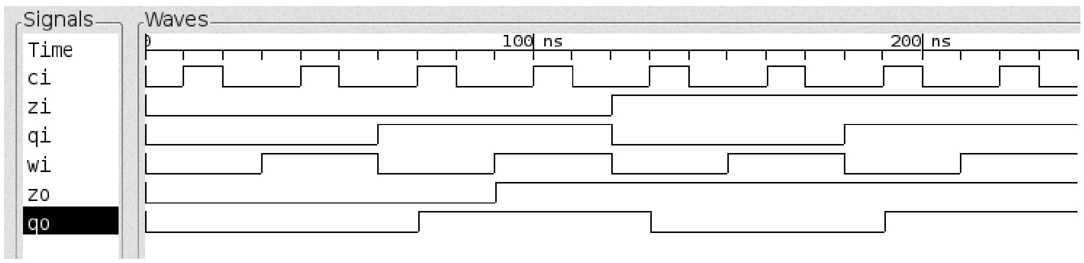

Figure 2.19: VHDL PIN Slice With Delays Waveform

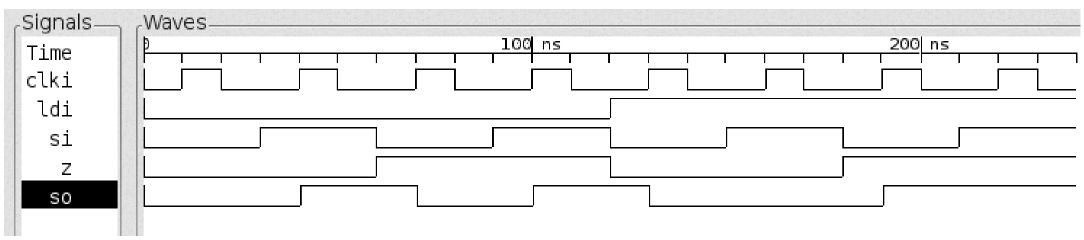

Figure 2.20: VHDL Shift Slice With Delays Waveform

2.6.2 VHDL Top Level Testbench With Delays Waveform

Looking at the results of the top level testbenches for both normal and test mode we can see that they are again identical to the waveforms captured without delays. Additionally, since our testbench is exhaustive and self-checking we can be certain that the delays did not introduce any corner cases that one might miss if they are spot checking manually.

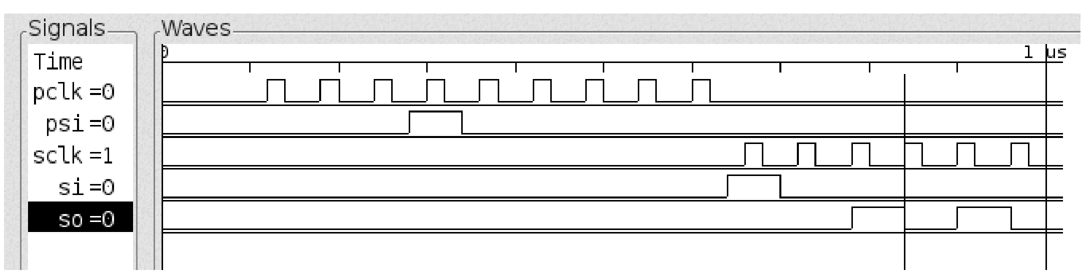

Figure 2.21: VHDL Top Level Functional Test Bench With Delays Waveform

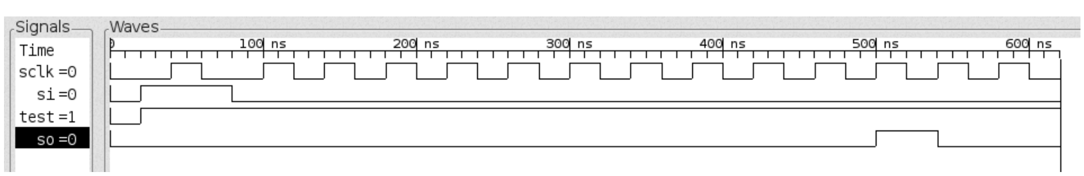

Figure 2.22: VHDL Top Level Test Mode Test Bench With Delays Waveform

2.7 Final Simulation Comparision

Taking a look at the final critical path delay summary we can see that there is a bit of discrepancy between the different simulations. We are not one hundred percent certain why this exists but our best guesses lead to explaining it as an artifact of the different simulation techniques used by each simulator and what parameters they take into account. While there are slight discrepancies all the worst case delays are under 1ns which gives us a theoretical **max clock speed of 1.3GHz**. Since our design is basically all shift registers we should also be able to achieve a throughput equal to the max clock rate. Once we simulate the full layout, which will introduce some long traces between slice rows, we will be able to determine a more accurate maximum clock and throughput rates.

Simulation	PIN	Shift
IRSIM	0.348n	0.242n
SPICE	0.792n	0.455n
VHDL	0.638n	0.682n

Table 2.10: Critical Path Delay Comparison

2.8 Floor Plan

The current floorplan that we plan on pursuing is shown below. From initial placement testing we believe we will be able to achieve a **16x16** grid of slices. The majority of the core functionality is contained in a nice, symmetrically sliced, square. The only additional components that fall outside of this model are the 3 MUXs that are required for test mode. The floor plan shown below indicates the planned location of all test slices and the major components of our design, namely the Shifter and the PIN.

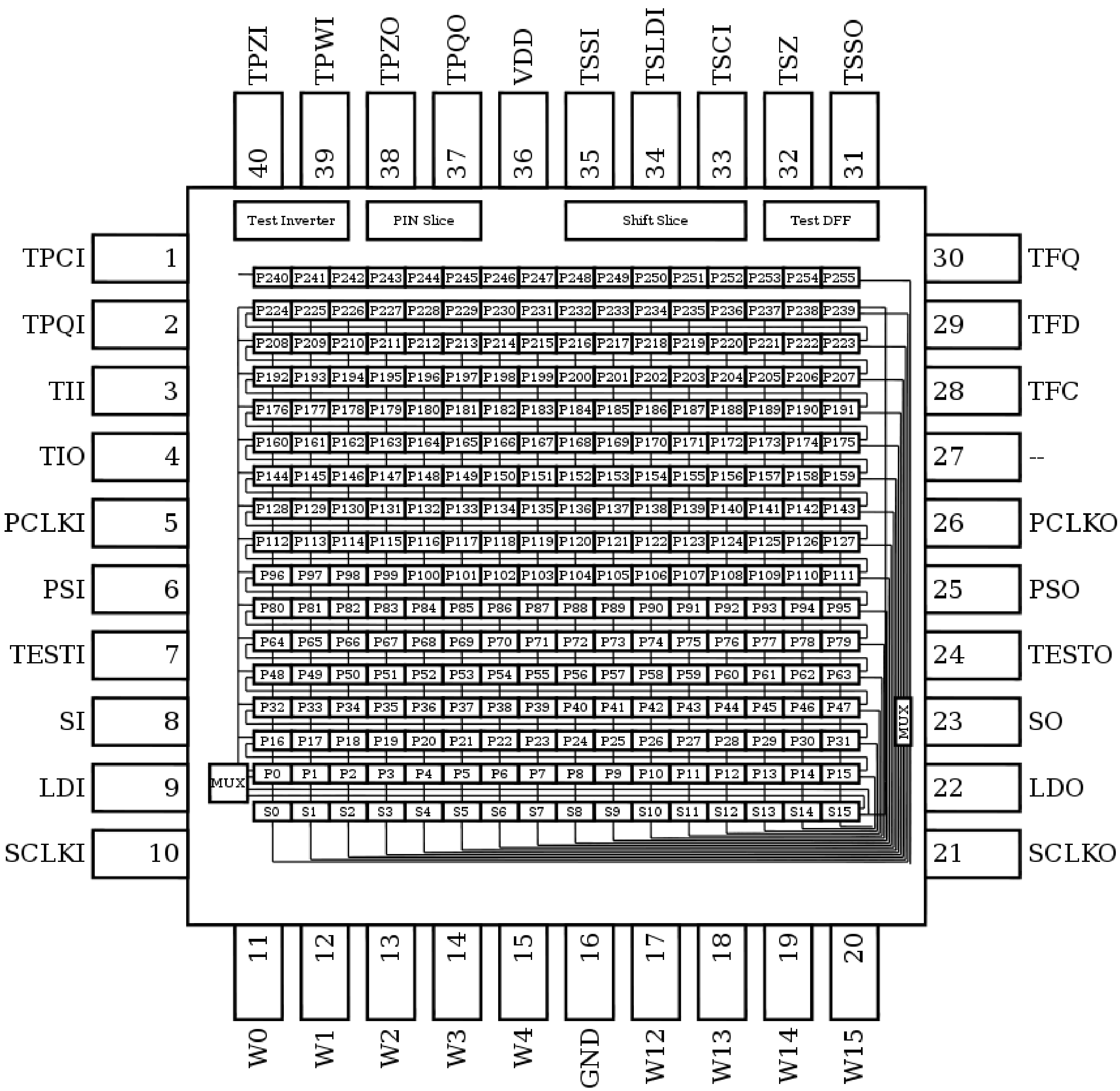

Figure 2.23: Floor Plan Diagram

2.9 Major Design Decisions

The major design decisions revolved mainly around the layout of each slice. We knew we wanted to come up with a design that would allow us to tile with minimal effort. Achieving this goal was not necessarily difficult but it required an iterative design process to break out each signal in such as way that would allow them to directly connect together.

Another decision that was made early on was simply 'which cells should we use?'. While some parts of our slice, such as the D Flip Flop, were obvious choices others were more flexible. In the schematic representation of our PIN slice we show a 2 input AND gate feeding into a 2 input OR gate. As it turns out, the provided library has a cell which performs that function but with an inverted output which is easily mitigated by adding. After comparing two layouts, one using an AND and an OR cell and one using the AOI cell plus an INV cell it turned out that using the AOI and INV saved us some horizontal space which allowed use to fit an extra column of slices in bumping our PIN size to 16x16.

Design decisions revolving around the floor plan are derived from not only from our initial pin layout, which strives to provide chip-to-chip slicing, but also organically as we continue to place components in the frame and see how they fit together. As such, we have not completely finalized the pinout and many pins are still left unassigned. As stated in the first progress report, once we move further into finalizing placement of our Shifter and PIN in the frame we will start tapping off various interesting signals and routing them to close by unassigned pins.

2.10 Work Division

Task	**Person**
PIN Slice Layout	Thrun
Shift Slice Layout	Qi
PIN Slice IRSIM	Qi
Shift Slice IRSIM	Thrun
PIN Slice Spice	Thrun
Shift Slice Spice	Qi
Gate Spice	Both
VHDL Models	Both
VHDL Slice Tests	Thrun
VHDL Top Tests	QI
Simulation Comparison	Both
Floor Plan	Both
Design Decisions	Both

Table 2.11: Task Assignment

Chapter 3

Part 3

3.1 Chip Layout

3.1.1 PIN Layout

The layout for our 16x16 Programmable Interconnect Network is shown below. The component on the bottom left are two 2:1 muxes which are used for test mode.

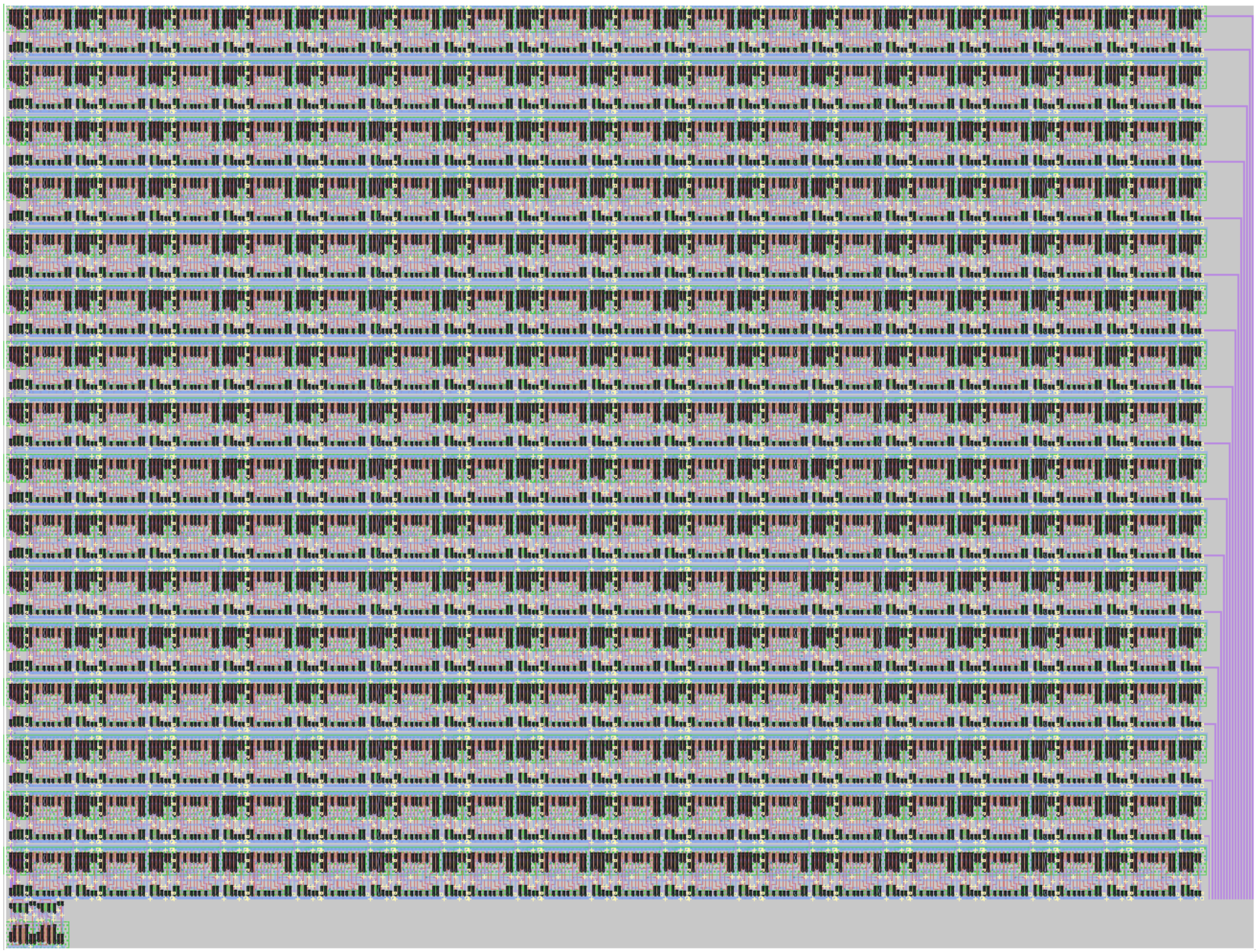

Figure 3.1: PIN Layout

3.1.2 Shifter Layout

The layout for our 16 bit parallel-load parallel-output shifter register is shown below. At 16 bits the shifter just fits within the frame of our chip. The component on the top right is a 2:1 mux which is used for test mode.

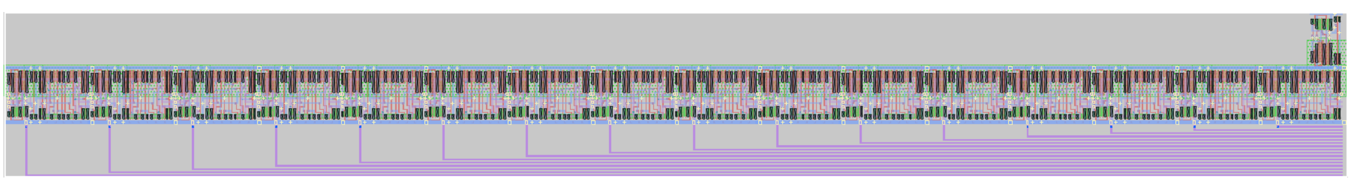

Figure 3.2: Shifter Layout

3.1.3 Overall Layout

The final layout for Intranex is shown below. While our design spans the full width of the frame we had quite a bit of empty vertical space. We used this extra area to include a fun logo.

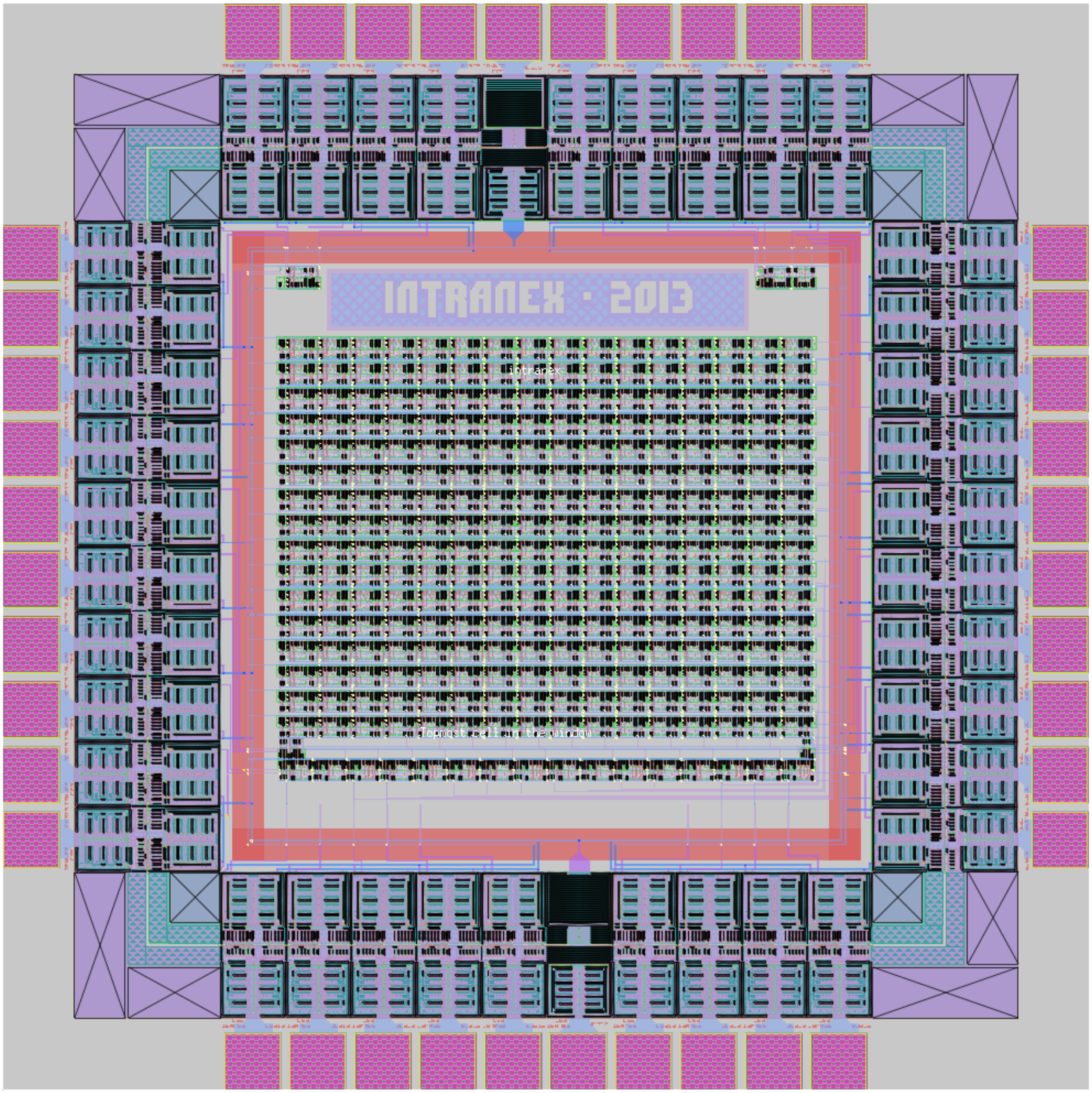

Figure 3.3: Overall Layout

The same layout in its flattened form is shown below.

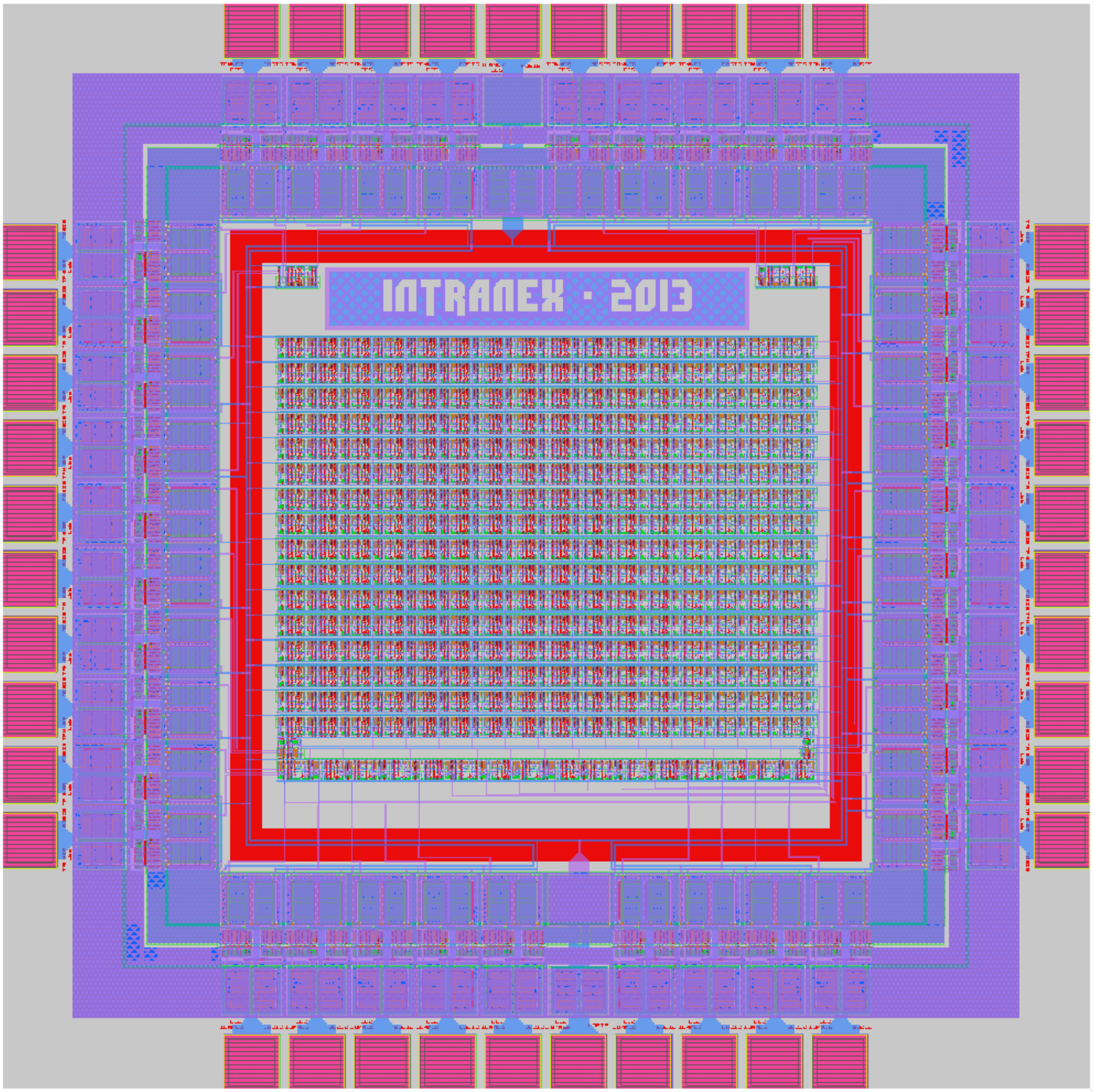

Figure 3.4: Overall Layout (Flattened)

3.2 Users Guide

The typical use case for Intranex is manipulating the order of bits in a vector. As an example lets see how we can use Intrnex to flip the bit order of a 16bit vector.

Lets choose 0b0111110000110001 as our input vector and its flipped form 0b1000110000111110 as our target result. In order to flip the vector we will need to configure the PIN mapping as such:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1    0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0    1
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0    2
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0    3
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0    4
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0    5
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0    6
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0    7  <--- Output value bit position (hex)
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0    8
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0    9
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0    A
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0    B
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0    C
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0    D
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0    E
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    F
-------------------------------
0 1 2 3 4 5 6 7 8 9 A B C D E F    <--- Input value bit posisition (hex)
```

As we can see the value at input bit position 0 will end up at output bit position 15 since that grid cell is marked as a 1, indicating a connection.

The PIN vector is shifted in on the `PSI` pin starting at the top-right of the network. For our example the first 48 bits shifted in would be: (note we are shifting the left-most bit in the following string first)

```
1000000000000000 0100000000000000 0010000000000000 ...
```

The timing diagram below illustrates clocking in the first 20 and last 3 bits of our PIN configuration.

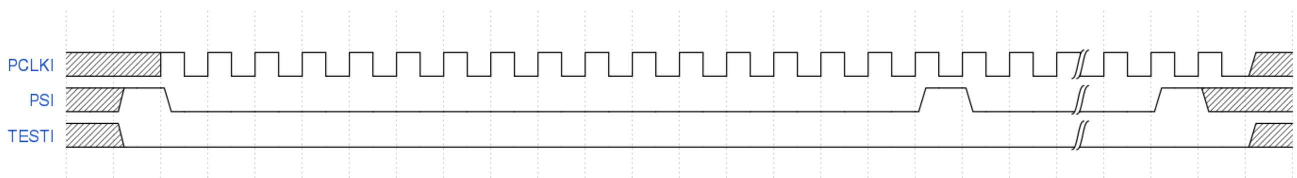

Figure 3.5: PIN Configuration Example

In order to clock in the input vector we simply use the `SCLKI` and `SI` pins to clock the vector in most significant bit first. After the input has been clocked in the `LDI` pin must be asserted and the `SCLKI` line pulse to latch the result vector. The MSB of the result is available immediately on `SO`. The remaining 15 bits of the result can then be clocked out. A diagram illustrating this process for our example is shown below.

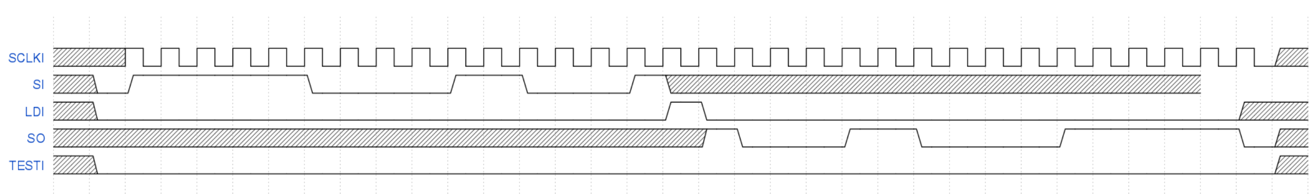

Figure 3.6: Loading the input vector and reading the result

3.3 Test Strategy

3.3.1 Independent Logic Gates

The are a few different strategies to go about testing our chip. The first involves the independent gates and slices placed at the top of our chip. The `Test Inverter`, found on pins `3` and `4`, along with the standard D-Flip Flop, found on pins `28-30`, can be used to ensure the die is structurally valid and that power is being delivered on the distribution rails.

3.3.2 Independent Slices

Secondly, there are two independent slices which can be used for verification. On pins `31-35` there are connections to a single `Shift Slice` which can be used to verify its operation. The expected operation of this slice has been discussed heavily in previous sections of this document. There is also an independent `PIN Slice` found on pins `37-40` and `1-2`. Like this `Shift Slice`, this slice can be used to prove the functionality of the design and layout.

3.3.3 Test Mode

To confirm that the main functional blocks are connected the `TESTI` pin can be asserted which will enabled the design to function as a scan chain. With `TESTI` asserted all 272 flip flops in the design are chained together. Clocking a pulse into the `SI` pin using the `SCLKI` clock pin should result in the same pulse appearing on the `SO` pin 272 clock cycles later.

3.3.4 Internal Signals

The output of nine rows of the PIN are brought out onto dedicated debugging pins `11-15` and `17-20`. These signals can be used to ensure that the PIN is operating correctly and can be used to confirm that the parallel load shifter is parallel loading the correct values.

3.3.5 Function Test

Finally, a standard functional test can be used with known vectors. For instance, a good vector to test is one that reverses the bit order of the input vector. This configuration is ideal as it allows for quick human validation. An example of this has been shown in previous sections.

3.4 Chip Architecture

As has been discussed the architecture of our chip is broken up into two major parts: the Programmable Interconnect Network, and the Parallel-Load Parallel-Output Shifter Register. Our design has remained fairly consistent throughout our project with the only major change between progress reports being a slight modification to our slices to squeeze out a few more lambdas. This optimization ultimately lead us to achieve a PIN of 16x16 which we are satisfied with as 16 is a nice clean number. With input vectors of 16 bits we can now accomplish interesting tasks such as switching the endianess of words.

In order to handle the large fanout of the PIN clock we added buffers to the beginning of each PIN row. Each buffer provides more than enough drive strength to clock the entire row. Additionally, we opted to not include a buffer on the shift register as the fanout is fairly low.

3.5 Simulation Results

3.5.1 Test Mode

For the overall simulation of the final chip the first thing we tested was Test Mode. A simple IRSIM CMD file was written, shown below, to send a pulse through and assert that it is received after 272 clock pulses, which is the number of flip-flops in our chain.

```
stepsize 10
logfile intranex_test.log
h VDD
l GND
vector CLK SCLKI
clock CLK 0 1
w SCLKI SI SO
ana SCLKI SI SO
l LDI
l PCLKI
l PSI

| enable test mode
h TESTI

| clock in a pulse (010)
l SI
c
h SI
c
l SI
c

| clock it until the pulse is just about to come out
| 16*16+16-3-1 == 268
|
| 16*16+16 == number of flip flops
| 3 == we already clocked three times
| 1 == stop right before the pulse comes out
c 268

| make sure we see the pulse
assert SO 0
c
assert SO 1
c
assert SO 0
c
```

Listing 3.1: Intranex Test Mode IRSIM CMD File

While the figure below provides little to no value it is included here for formality. The waveforms at this level are of little use to us since we are working with events that happen every couple hundred clocks. This is why we chose to use the assertion feature of IRSIM to check the output.

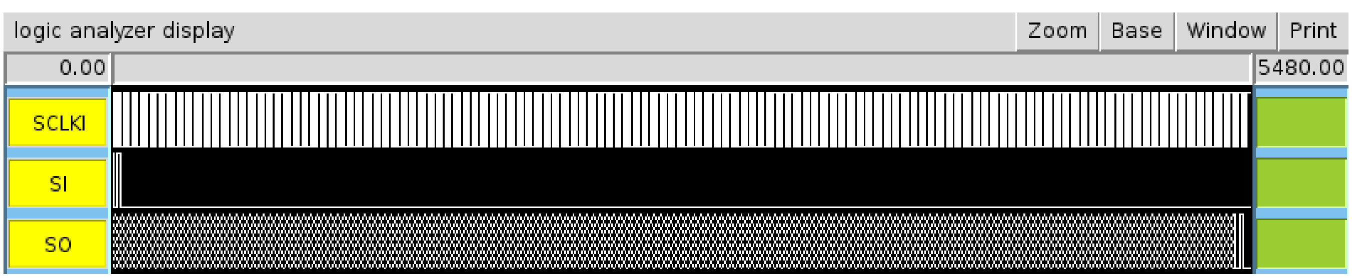

Figure 3.7: Intranex Test Mode IRSIM Result

3.5.2 Functional Mode

For a functional test we configured the PIN to simply flip the input value. In the high level waveform shown below we can see the PIN being configured followed by the input value be shifted in and the result shifted out.

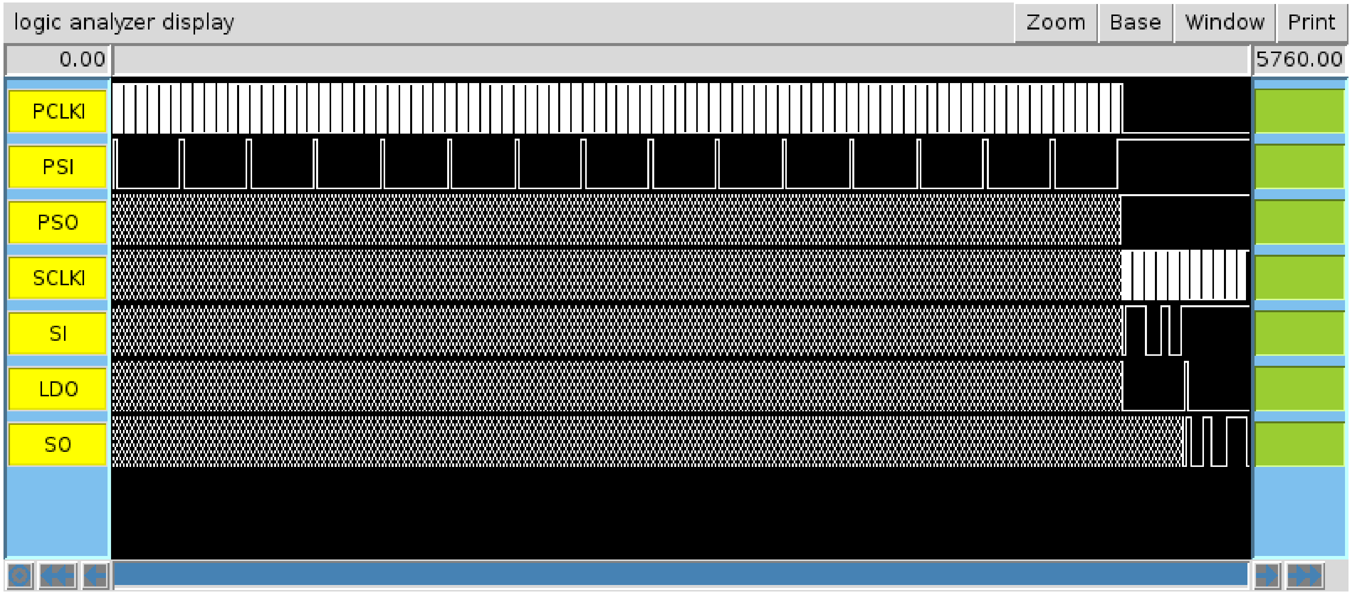

Figure 3.8: Intranex Functional Mode IRSIM Result

Taking a closer look at the input-output waveforms we can see that we clock in a value of `0111110000110001`. The `LDO` line, which is a passthrough from `LDI`, shows us that we then latched the result vector. The result vector is then shifted out and as we can see the expected output, `1000110000111110`, is achieved.

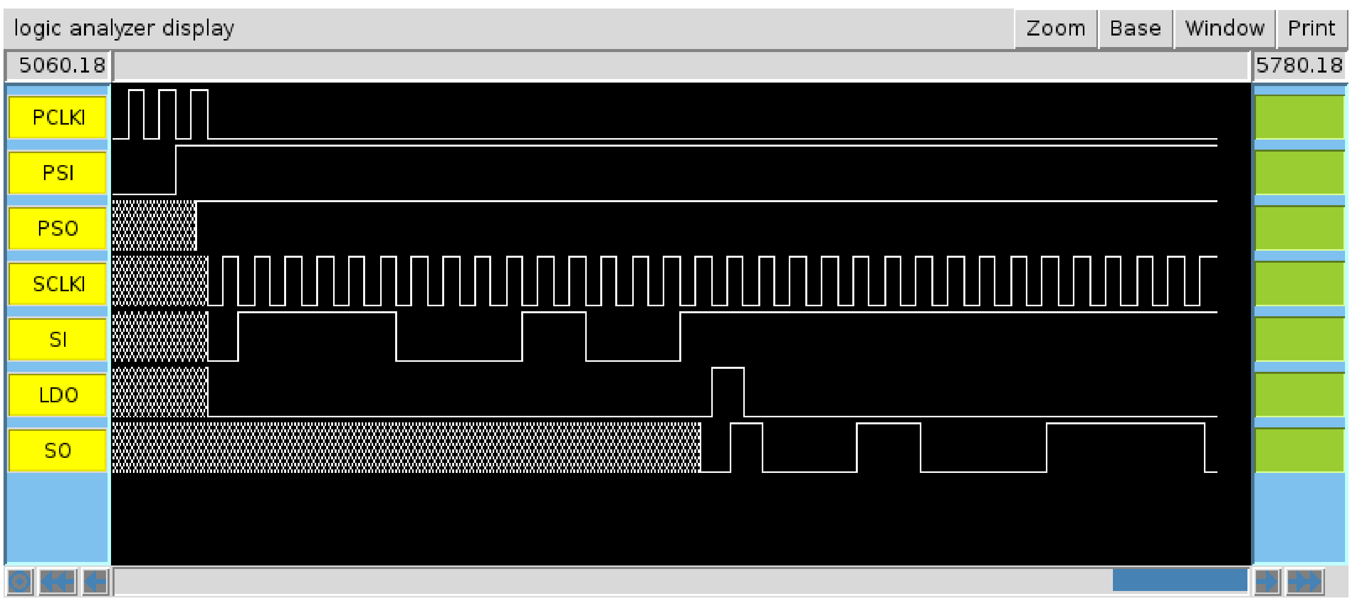

Figure 3.9: Intranex Functional Mode IRSIM Result (zoomed)

It would be impossible to exhaustively test our design at this scale so we settled for a random check over a few hundred input vectors. To achieve this a program was written to first generate a random input vector, a random permutation of the input vector, and the PIN configuration vector required to achieve the permutation. The program then writes the IRSIM `cmd` file for this setup which has assertions to check if the output is correct. Our program than launches IRSIM, which logs its output to a file, and then checks the resulting log for any assertion fails. If no assertion failed it continues to the next test. A screenshot showing the tester in action is shown below.

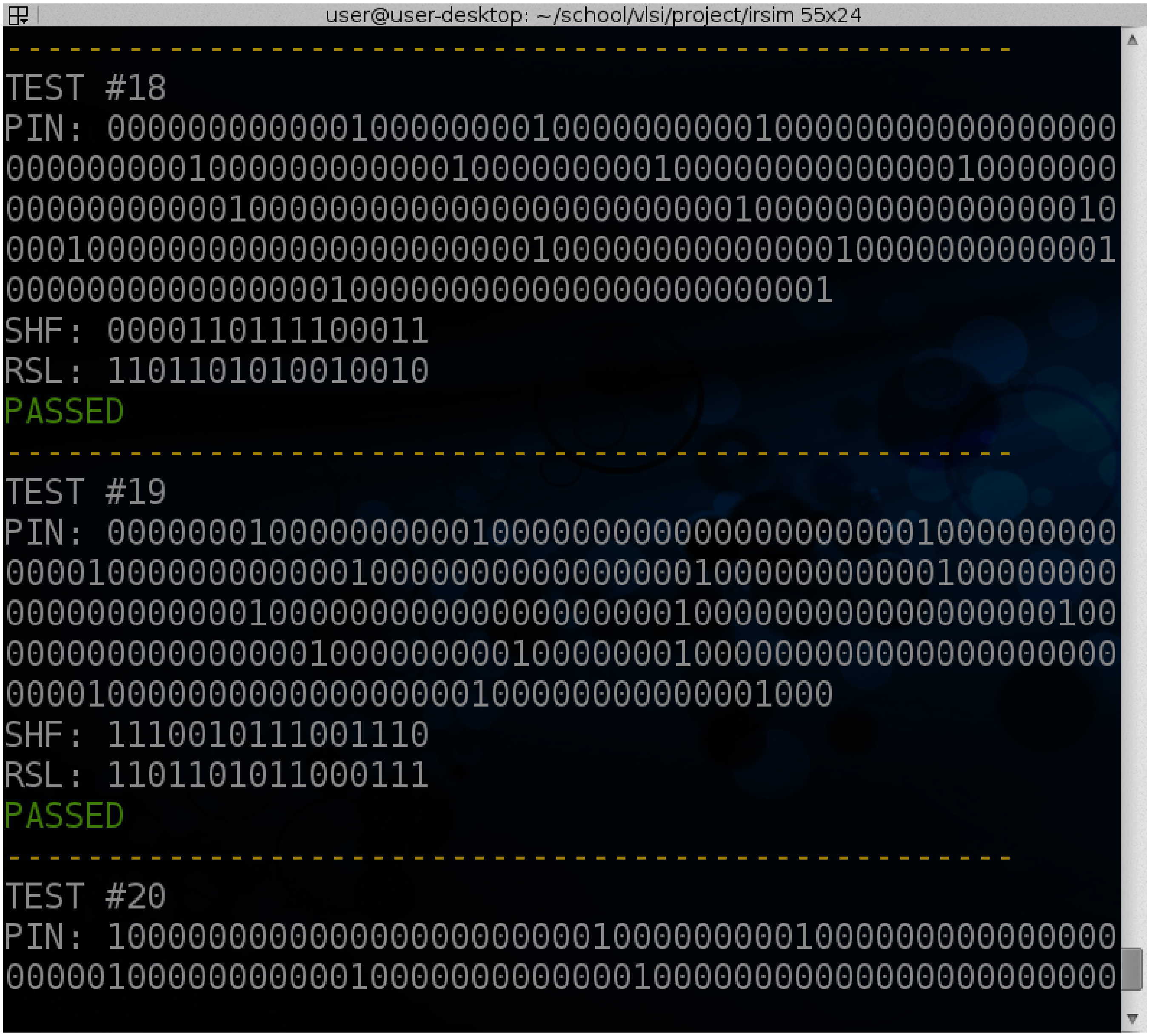

Figure 3.10: Automated Testing Screenshot

The source code for the automated tester is included below:

```python
import sys
import shlex
import random
import subprocess
from termcolor import colored

N = 16

class Pin:

    def __init__(self):
        self.array = [["0"]*N for i in range(N)]

    def set_xy(self, x, y):
        self.array[y][x] = "1"

    # return a bit vector of the PIN
    # MSB (top right of grid) on right
    #
    # 6 7 8
    # 3 4 5   =>  87654321
    # 0 1 2
    #
    def bits(self):
        rt = ""
        for line in self.array:
            rt += "".join(reversed(line))
        return rt

def get_bits(shift):

    # get bit strings
    shift_bits = bin(shift).replace("0b", "").zfill(N)
    result_bits = ["0"]*N
    pin = Pin()

    # generate a random ordering permutation
    order = range(N)
    #random.shuffle(order)
    order = order[::-1]

    # reorder the result bits
    for i in range(N):
        result_bits[i] = shift_bits[order[i]]

    # calculate PIN for this permutation
    for i in range(N):
        pin.set_xy(order[i], i)

    # return with MSB on left
    return (pin.bits(), shift_bits[::-1], "".join(result_bits)[::-1])

def write_cmd_file(pin_bits, shift_bits, result_bits, analyzer=False):

    with open("intranex.cmd", "w") as f:

        f.write("stepsize 10\n")
        f.write("logfile intranex.log\n")
        f.write("h VDD\n")
        f.write("l GND\n")
        f.write("vector SCLK SCLKI\n")
        f.write("vector PCLK PCLKI\n")
        if analyzer: f.write("ana PCLKI PSI PSO SCLKI SI LDO SO\n")
        f.write("w   PCLKI PSI PSO SCLKI SI LDO SO\n")
        f.write("l TESTI\n")

        # load the PIN
        f.write("clock PCLK 0 1\n")
        for i in pin_bits:
            if i == "1":
                f.write("h PSI\n")
            else:
                f.write("l PSI\n")
            f.write("c\n")

        # disable PCLK
        f.write("clock PCLK 0 0\n")

        # load the SHIFT
        f.write("l LDI\n")
        f.write("clock SCLK 0 1\n")
        for i in shift_bits:
            if i == "1":
                f.write("h SI\n")
            else:
                f.write("l SI\n")
            f.write("c\n")

        # latch the result
        f.write("h LDI\n")
        f.write("c\n")
        f.write("l LDI\n")

        f.write("assert SO %s\n" % result_bits[0])

        # shift out remaining result bits
        for i in range(1, N):
            f.write("c\n")
            f.write("assert SO %s\n" % result_bits[i])
            f.write("path SO\n")

        if not analyzer: f.write("exit\n")

if __name__ == "__main__":
```

```
    random.seed()

    completed = []
    for i in range(100):

        # find a unique permutation we havent tested yet
        while True:

            shift = random.randrange(0, 2**N)
            (pin_bits, shift_bits, result_bits) = get_bits(shift)

            if (pin_bits, shift_bits, result_bits) in completed:
                print "Duplicate test, skipping..."
                continue

            completed.append((pin_bits, shift_bits, result_bits))
            break

        print "TEST #%d" % i
        print "PIN: %s" % pin_bits
        print "SHF: %s" % shift_bits
        print "RSL: %s" % result_bits

        write_cmd_file(pin_bits, shift_bits, result_bits)

        irsim_cmd = "irsim -s ../../models/scmos30.prm ../magic/intranex_2.sim -intranex.cmd"
        subprocess.call(shlex.split(irsim_cmd))

        grep_cmd = "grep assertion intranex.log"
        grep_code = subprocess.call(shlex.split(grep_cmd))

        if grep_code == 0:
            print colored("TEST FAILED!!!", 'red')
            sys.exit(1)

        print colored("PASSED", 'green')
        print colored("-"*50, 'yellow')

    print "DONE"
```

Listing 3.2: Python Intranex Automated Tester

3.5.3 Timing Analysis

In order to determine the worst case delay through our chip we used a modified version of the automated test program which generated an IRSIM command file for a single input vector and also included a `PATH` statement. The `PATH` statement, which we also used to time the slices in Part 2, shows us the worst case delays for every input state change. After running our program we have an IRSIM log with the delays for each of the 16 state changes. The delays are unfortunately presented per net and are not automatically totalled. A small program, `find_delay.py`, was written to parse the log and add up the net delays for each input vector. It then returns the longest delay time. Doing this over various vectors we found our worst case delay to be consistently around **2.51ns**. This indicates a max clock speed of **398MHz**.

An example showing the analysis flow is shown below:

```
    $ python intranex_single.py \
    0001000000000000\
    0000000000000010\
    { 12 lines snipped }
    0000001000000000\
    0000000001000000 1010000011110001 1000011010011100
    $ python find_delay.py intranex.log
    Worst delay: 2.510000
```

The source code for the delay parser is shown below:

```
import re
import sys

f = open(sys.argv[1])

contents = f.read()
paths = contents.split("critical path")

time_regex = re.compile("@\s(.*?)ns\s")

deltas = []

for path in paths:
    times = re.findall(time_regex, path)
    if len(times) == 0: continue
    print times
    deltas.append(float(times[-1])-float(times[0]))

print "——"
print "Worst delay: %f" % sorted(deltas)[-1]
print "——"
```

Listing 3.3: Delay Parser

3.6 Work Division

Task	Person
PIN Layout	Thrun
Shift Layout	Qi
Overall Layout	Thrun
Users Guide	Qi
Testing Strategy	Qi
Chip Architecture	Thrun
Simulation Test Mode	Thrun
Simulation Functional	Both
Simulation Timing	Qi

Table 3.1: Task Assignment