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

1 Pinout Diagram

The pinout diagram for INTRANEX is shown below in Figure 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.

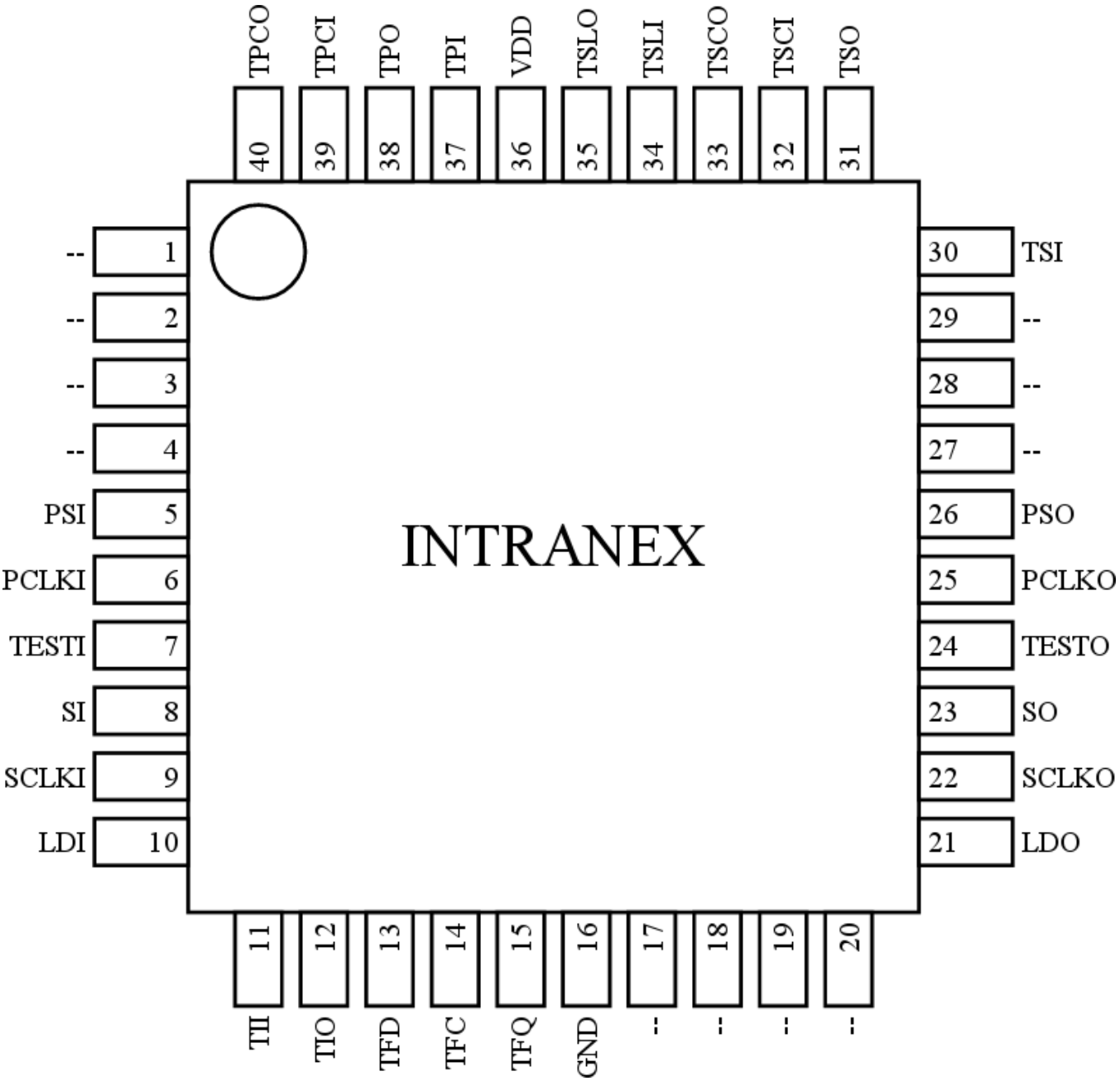

Figure 1: Pinout Diagram

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	–	-	–
2	–	-	–
3	–	-	–
4	–	-	–
5	PSI	I	PIN serial input
6	PCLKI	I	PIN clock input
7	TESTI	I	Test Mode enable input
8	SI	I	Serial input
9	SCLKI	I	Serial clock input
10	LDI	I	Parallel load input
11	TII	I	Test inverter input
12	TIO	O	Test interter output
13	TFD	I	Test flip-flop D input
14	TFC	I	Test flip-flop clock input
15	TFQ	O	Test flop-flop Q output
16	GND	P	–
17	–	-	–
18	–	-	–
19	–	-	–
20	–	-	–
21	LDO	O	Parallel load output
22	SCLKO	O	Serial clock output
23	SO	O	Serial output
24	TESTO	O	Test Mode enable output
25	PCLKO	O	PIN clock output
26	PSO	O	PIN serial output
27	–	-	–
28	–	-	–
29	–	-	–
30	TSI	I	Test shift slice serial input
31	TSO	O	Test shift slice serial output
32	TSCI	I	Test shift slice clock input
33	TSCO	O	Test shift slice clock output
34	TSLI	I	Test shift slice load input
35	TSLO	O	Test shift slice load output
36	VDD	P	–
37	TPI	I	Test pin slice serial input
38	TPO	O	Test pin slice serial output
39	TPCI	I	Test pin slice clock input
40	TPCO	O	Test pin slice clock output

Table 1: Pin Descriptions

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.

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 labels and will be explained further in later sections.

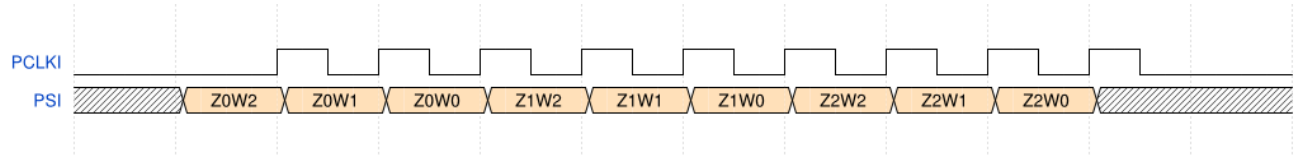

Figure 2: PIN Configuration

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.

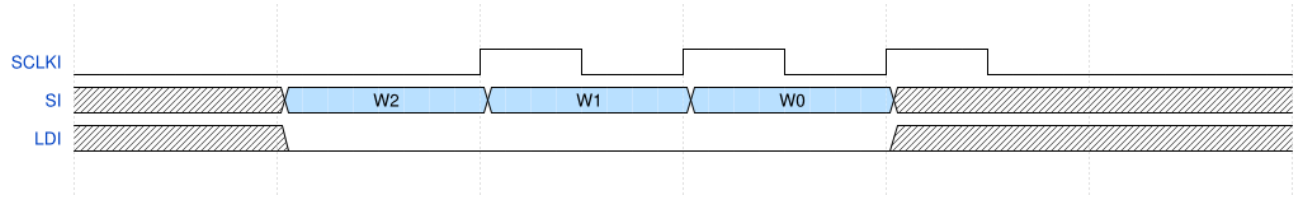

Figure 3: Loading a value

After the value has been loaded in the result is clocked on 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`. The `LDI` pin should now be held low while clocking out the remaining result bits.

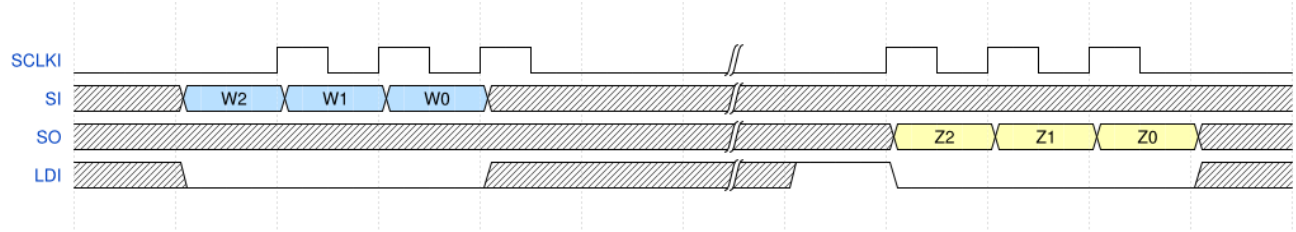

Figure 4: Loading a value and reading the result

2.3 Test Mode

Test Mode is enabled by pulling the `TESTI` pin high. When this occurs the output of the internal 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 shift register.

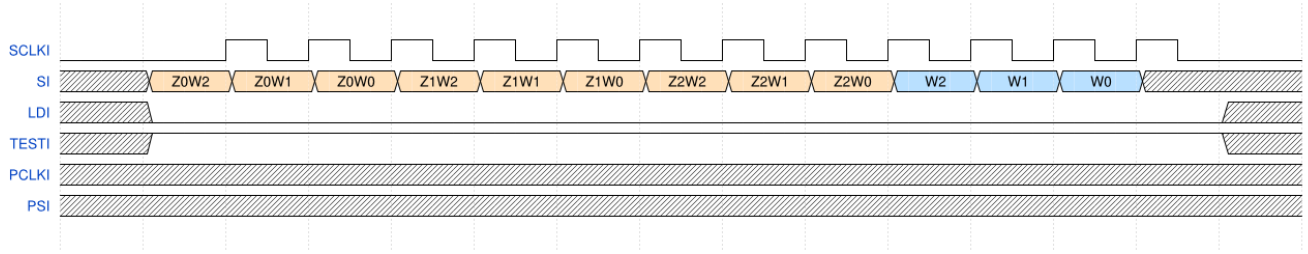

Figure 5: Enabling test mode and loading all DFFs

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 row in the PIN to share a power rail with the adjacent row 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 use 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.

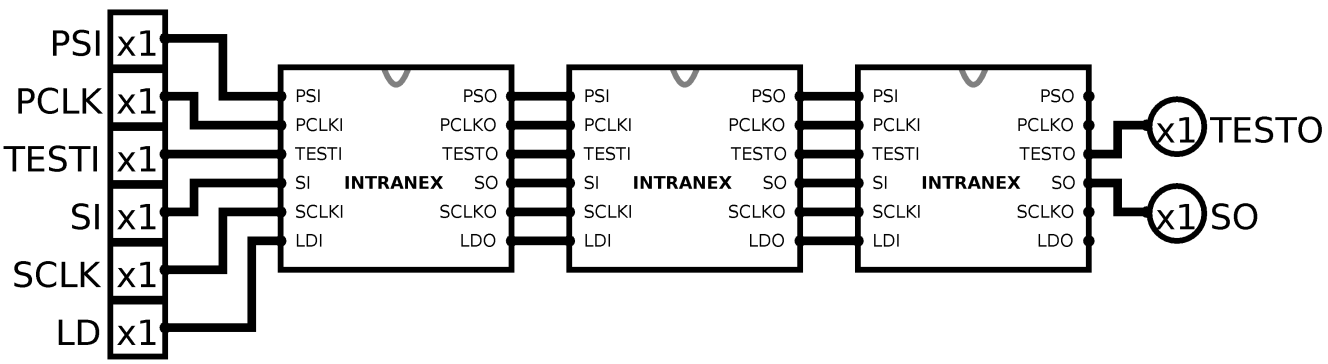

Figure 6: 3 INTRANEX Chain

4 Block Diagrams

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. Test mode logic has been excluded to illustrate only core functionality.

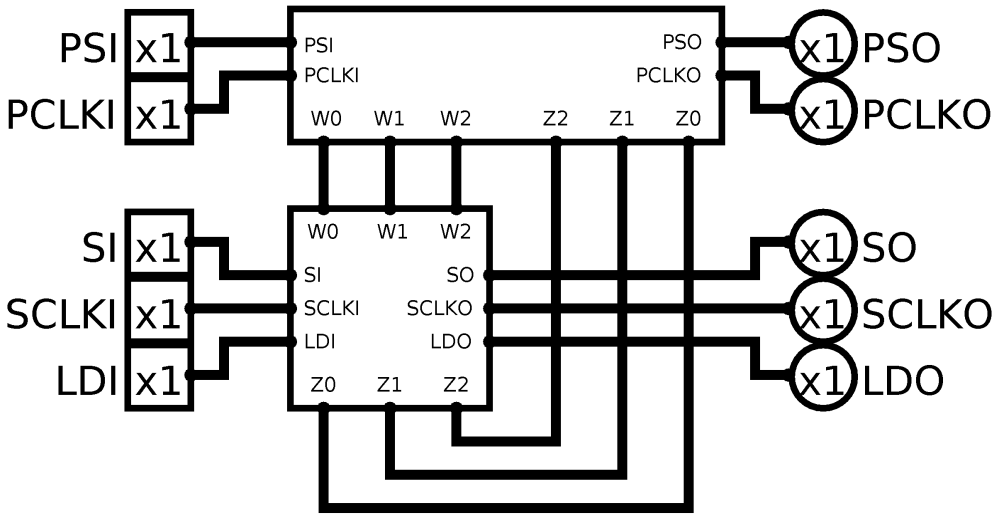

Figure 7: Top Level Block Diagram (3-Bit Configuration)

4.2 Top Level With Test Mode

The same top level diagram is shown below except this time with 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.

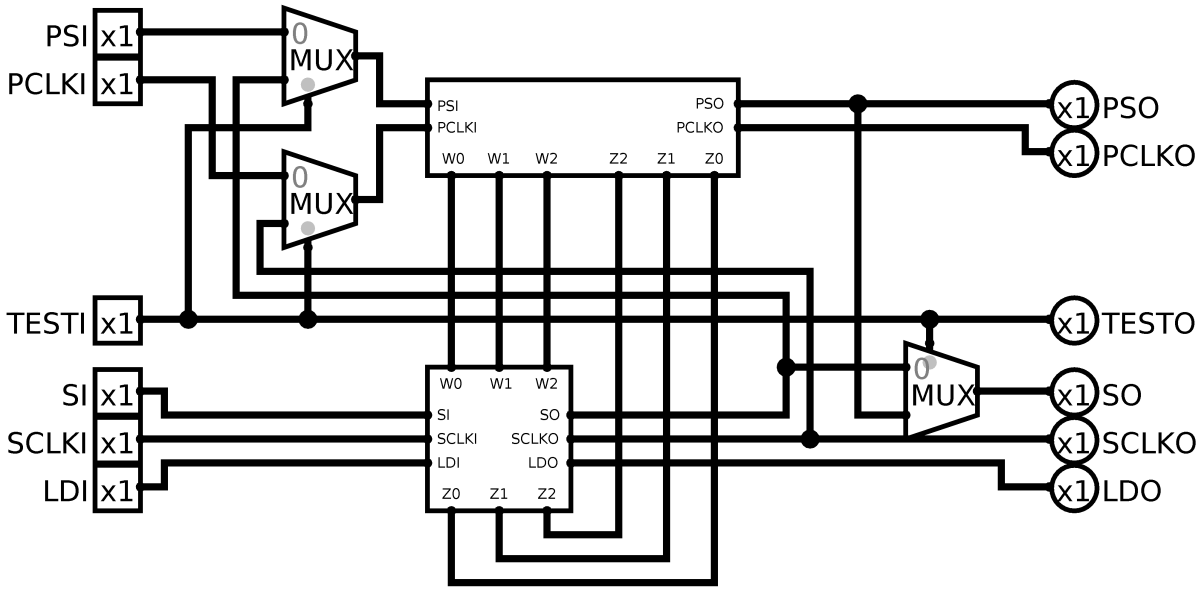

Figure 8: Top Level Block Diagram Showing Test Mode Logic (3-Bit Configuration)

4.3 Top Level Bit Sliced

The diagram below shows a bit sliced version of Figure 7. We can see how each slice is directly connected together with zero interfacing logic. We can also see the long row-to-row connections as discussed earlier.

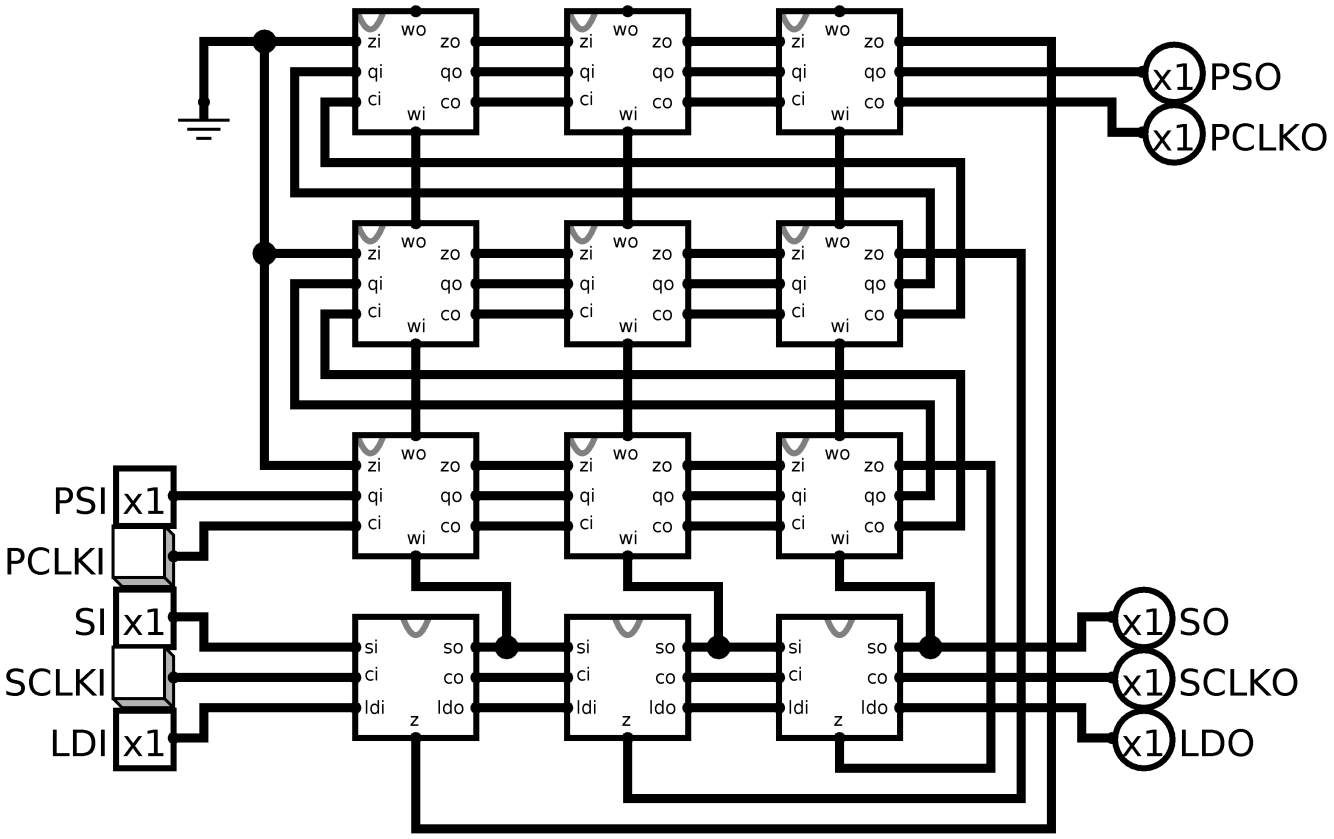

Figure 9: Top Level Bit Sliced Block Diagram (3-Bit Configuration)

4.4 Parallel Load Shift Register

4.4.1 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.

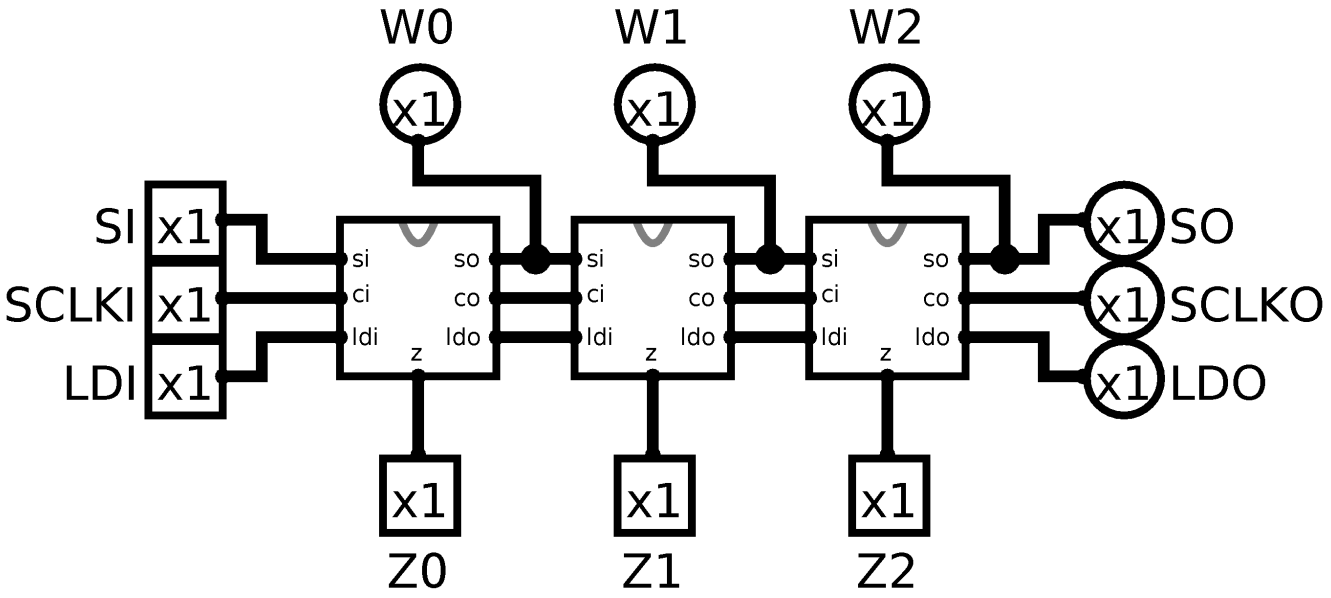

Figure 10: Parallel Load Bit-Sliced Shifter Register (3-Bit Configuration)

4.4.2 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.

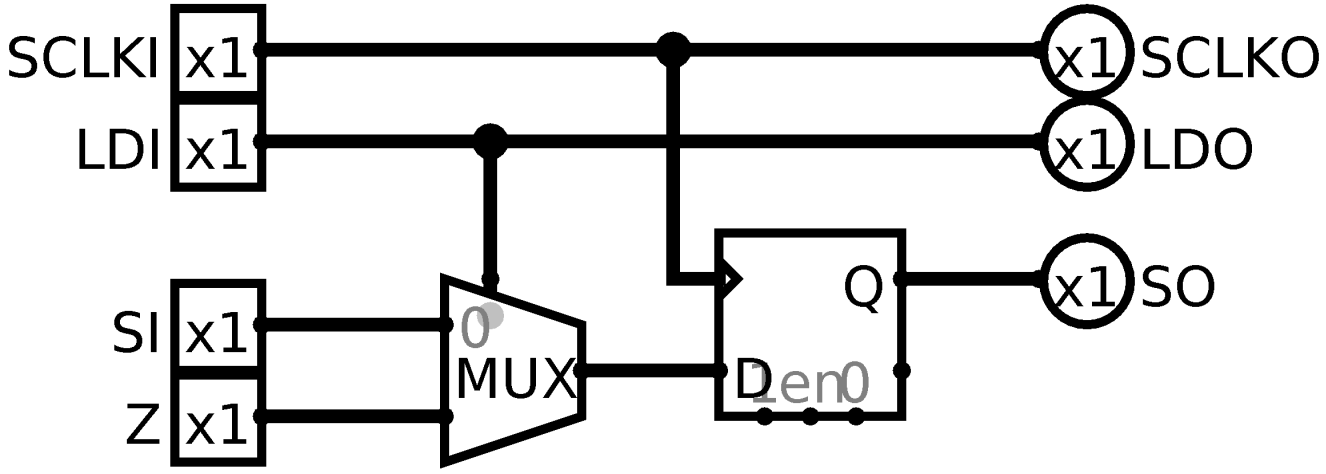

Figure 11: Parallel Load Shifter Register Bit-Slice

4.5 Programmable Interconnect Network

4.5.1 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 array.

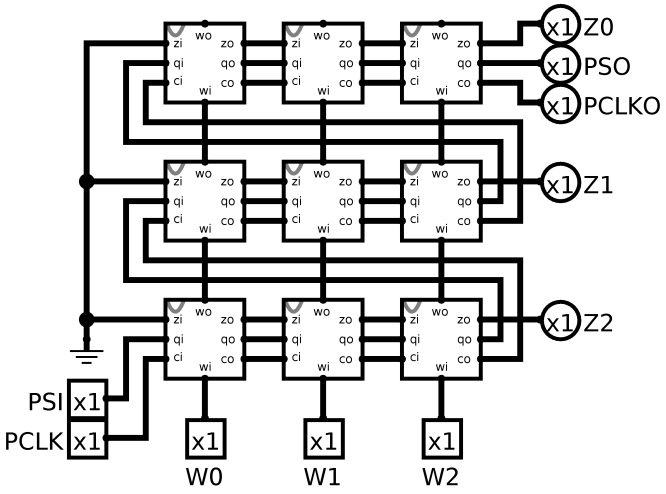

Figure 12: Bit-Sliced Programmable Interconnect Network (3-Bit Configuration)

4.5.2 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 row. If that bit is set and this slice is configured as 'connected' we want to output a logic high on the `Z` bus. 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.

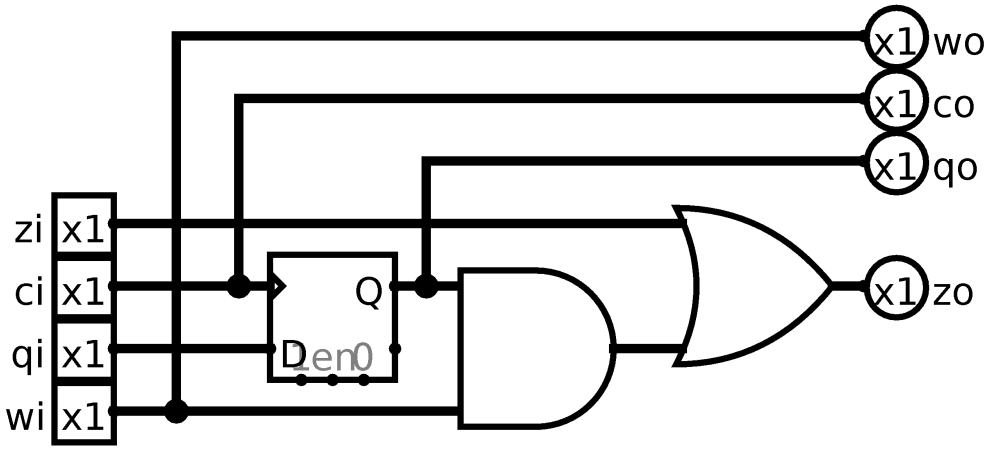

Figure 13: Programmable Interconnect Network Bit-Slice

5 VHDL Models

5.1 Top Level

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity top is
5      generic(
6          n : integer := 3
7      );
8      port(
9          psi   : in  std_logic;
10         pso   : out std_logic;
11         pclk  : in  std_logic;
12         si    : in  std_logic;
13         so    : out std_logic;
14         sclk  : in  std_logic;
15         ld    : in  std_logic;
16         test  : in  std_logic
17     );
18 end top;
19
20 architecture rtl of top is
21
22     -- output of pin
23     signal z : std_logic_vector((n-1) downto 0) := (others => '0');
24     -- parallel output of shifter
25     signal w : std_logic_vector((n-1) downto 0) := (others => '0');
26
27     signal pin_clk : std_logic;
28     signal pin_psi : std_logic;
29     signal pin_pso : std_logic;
30     signal shift_out : std_logic;
31
32 begin
33
34     -- test mode mux connects shifter and pin together
35     test_mux_1 : entity work.mux2x1 port map(pclk,      sclk,      test, pin_clk);
36     test_mux_2 : entity work.mux2x1 port map(psi ,      shift_out, test, pin_psi);
37     test_mux_3 : entity work.mux2x1 port map(shift_out, pin_pso,   test, so);
38
39     pin : entity work.pin
40     generic map(
41         n => n
42     )
43     port map(
44         clk => pin_clk,
45         psi => pin_psi,
46         pso => pin_pso,
47         z => z,
48         w => w
49     );
50
51     pso <= pin_pso;
52
53     shifter : entity work.shift
54     generic map(
55         n => n
56     )
57     port map(
58         clk => sclk,
59         si => si,
60         so => shift_out,
61         ld => ld,
62         z => z,
63         w => w
64     );
65
66 end rtl;
```

Listing 1: Top Level VHDL Module

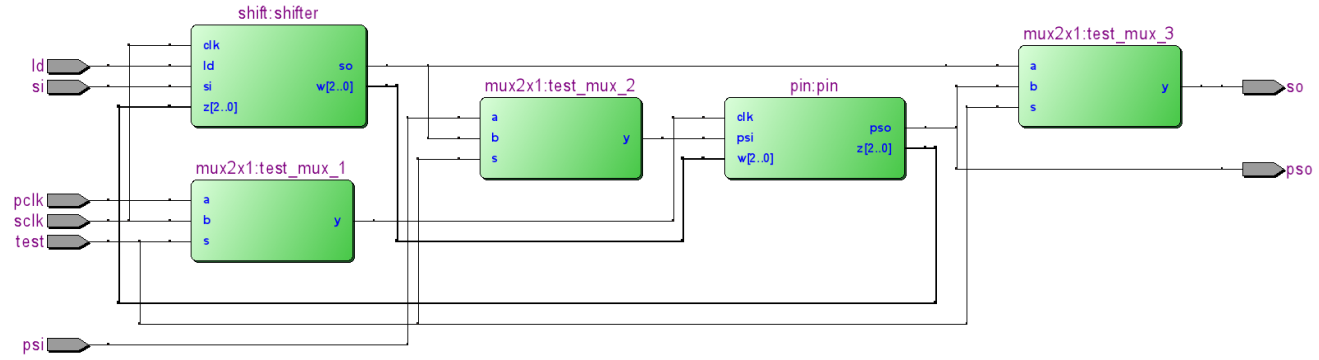

Figure 14: Top Level Generated RTL Diagram

5.2 PIN

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity pin is
5      generic(
6          n : integer := 3
7      );
8      port(
9          clk : in  std_logic;
10         psi : in  std_logic;
11         pso : out std_logic;
12         z   : out std_logic_vector((n-1) downto 0);
13         w   : in  std_logic_vector((n-1) downto 0)
14     );
15 end pin;
16
17 architecture rtl of pin is
18
19     component pin_slice is
20         port(
21             zi : in std_logic;
22             qi : in std_logic;
23             wi : in std_logic;
24             ci : in std_logic;
25             zo : out std_logic;
26             qo : out std_logic;
27             wo : out std_logic;
28             co : out std_logic
29         );
30     end component;
31
32     -- carray_array(row, col)
33     type carry_array is array (0 to n, 0 to n) of std_logic;
34     signal zc : carry_array;
35     signal cc : carry_array;
36     signal wc : carry_array;
37     signal qc : carry_array;
38
39     begin
40
41     -- setup first and last inputs for each row
42     z_connect : for i in 0 to n-1 generate
43         zc(i, 0) <= '0';
44         z(i) <= zc(i, n);
45     end generate;
46
47     -- setup first inputs for each column
48     w_connect : for i in 0 to n-1 generate
49         wc(0, i) <= w(i);
50     end generate;
51
52     -- setup row transfer
53     -- (last output of row to first input of next row)
54     r_connect : for i in 0 to n-2 generate
55         qc(i, 0) <= qc(i+1, n);
56         cc(i, 0) <= cc(i+1, n);
57     end generate;
58
59     -- connect external inputs
60     qc(n-1, 0) <= psi;
61     cc(n-1, 0) <= clk;
62     pso <= qc(0, n);
63
64     -- generate the grid of slices
65     pin_z_gen : for zz in 0 to n-1 generate
66         pin_w_gen : for ww in 0 to n-1 generate
67             pin_i: pin_slice port map(
68                 zi => zc(zz, ww),
69                 qi => qc(zz, ww),
70                 wi => wc(zz, ww),
71                 ci => cc(zz, ww),
72                 zo => zc(zz, ww+1),
73                 qo => qc(zz, ww+1),
74                 wo => wc(zz+1, ww),
75                 co => cc(zz, ww+1)
76             );
77         end generate;
78     end generate;
79
80 end rtl;
```

Listing 2: PIN VHDL Module

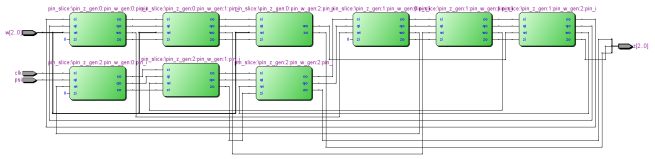

Figure 15: Pin Generated RTL Diagram

5.3 PIN Slice

```vhdl
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 3: PIN Slice VHDL Module

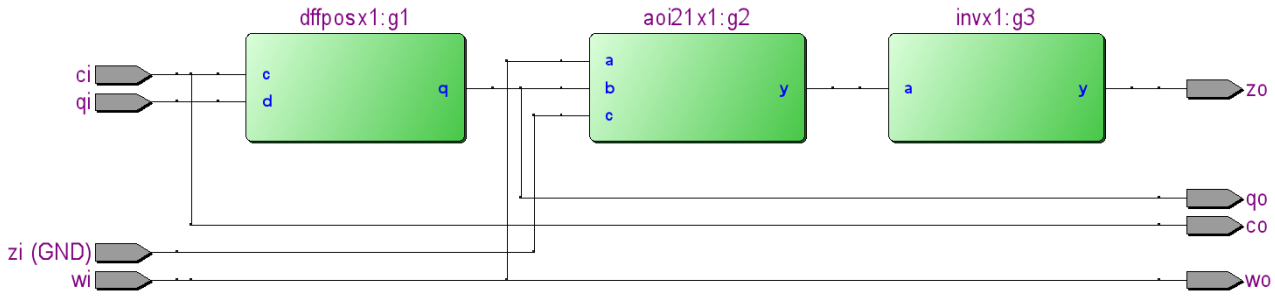

Figure 16: Pin Slice Generated RTL Diagram

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 4: Parallel Load Shifter VHDL Module

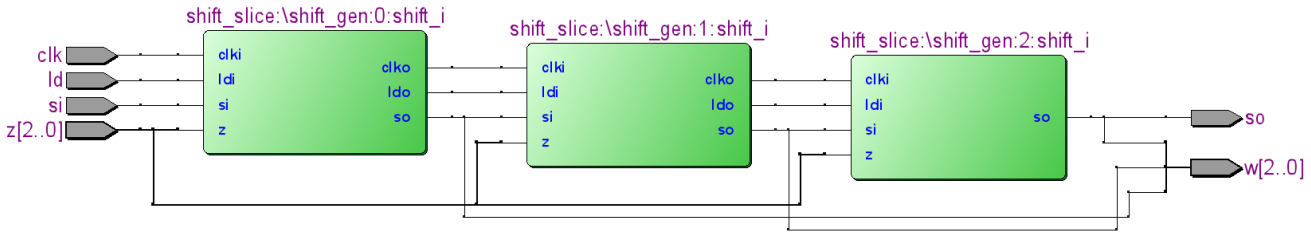

Figure 17: Shifter Generated RTL Diagram

5.5 Shifter Slice

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity shift_slice is
5      port(
6          clki : in  std_logic;
7          clko : out std_logic;
8          ldi  : in  std_logic;
9          ldo  : out std_logic;
10         si   : in  std_logic;
11         so   : out std_logic;
12         z    : in  std_logic
13     );
14 end shift_slice;
15
16 architecture rtl of shift_slice is
17
18     signal g1_o : std_logic := '0';
19
20 begin
21
22     g1 : entity work.mux2x1   port map(si, z, ldi, g1_o);
23     g2 : entity work.dffposx1 port map(clki, g1_o, so);
24
25     -- pass through
26     clko <= clki;
27     ldo <= ldi;
28
29 end rtl;
```

Listing 5: Parallel Load Shifter Slice VHDL Module

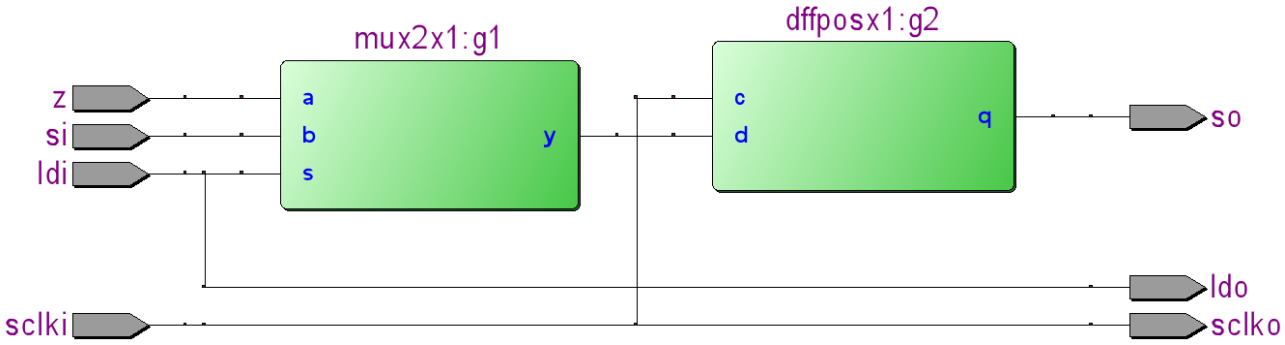

Figure 18: Shifter Slice Generated RTL Diagram

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 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 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 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 9: MUX2X1 VHDL Module

6 VHDL Test Benches

6.1 Top Level Functional

```
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;

```

```
100             -- clock in the pin
101             for i in 0 to (n*n)-1 loop
102                 psi <= pin_vector(i);
103                 wait for 20 ns;
104                 clock_pin;
105             end loop;
106
107             -- clock in the value
108             for i in 0 to n-1 loop
109                 si <= shift_vector(i);
110                 wait for 20 ns;
111                 clock_shifter;
112             end loop;
113
114             -- pull latch high so the first result
115             -- loop will trigger the latch
116             ld <= '1';
117             wait for 20 ns;
118
119             -- clock out result and check it
120             for i in 0 to n-1 loop
121                 clock_shifter;
122                 assert so = result_vector(i) report "Test Failed!";
123                 ld <= '0';
124                 wait for 20 ns;
125             end loop;
126
127         end loop;
128
129         report "Test Complete" severity note;
130         wait;
131
132     end process;
133
134 end tb_rtl;
```

Listing 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 design matches the known output. If then indicates if they pass or fail.

```
1  N = 3
2
3  out = open("vectors_%d_bit.sim"%N, "w")
4
5  # loop through all possible pins and shift values
6  for pin in range(2**(N*N)):
7      for shift in range(2**N):
8
9          # get bit strings
10         pin_bits = bin(pin).replace("0b", "").zfill(N*N)
11         shift_bits = bin(shift).replace("0b", "").zfill(N)
12         result_bits = ["0"]*N
13
14         # loop through all rows and columns
15         for z in range(N):
16             for w in range(N):
17                 if pin_bits[(N-1-z)*N+w] == "1" and shift_bits[w] == "1":
18                     result_bits[z] = "1"
19
20         # writeout results
21         out.write("%s\n" % pin_bits)
22         out.write("%s\n" % shift_bits)
23         out.write("%s\n" % "".join(result_bits))
24
25 out.close()
```

Listing 11: Python Vector Generator

6.2 Top Level Test Mode

```
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 zero";
        clock;
        -- check to make sure the pulse is 1
        assert so = '1' report "Pulse is not zero";
        clock;
        -- check to make sure the bit behind pulse is 0
        assert so = '0' report "Pulse trailing pulse not zero";
        clock;

        report "Test Complete" severity note;
        wait;

    end process;

end tb_rtl;
```

Listing 12: Top Level Test Mode VHDL Test Bench

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 13: PIN Slice VHDL Test Bench

6.4 Shifter Slice

```
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 sclki : std_logic := '0';
    signal sclko : 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(
            sclki : in  std_logic;
            sclko : 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(
        sclki => sclki,
        sclko => sclko,
        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
            sclki <= '1';
            wait for 10 ns;
            assert sclko = '1' report "SCLKO does not equal 1" severity error;
            assert ldo = patterns(i).ldi report "SCLKO does not equal 1" severity error;
            sclki <= '0';
            wait for 10 ns;
            assert sclko = '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 14: Parallel Load Shifter Slice VHDL Test Bench

7 VHDL Test Bench Results

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.

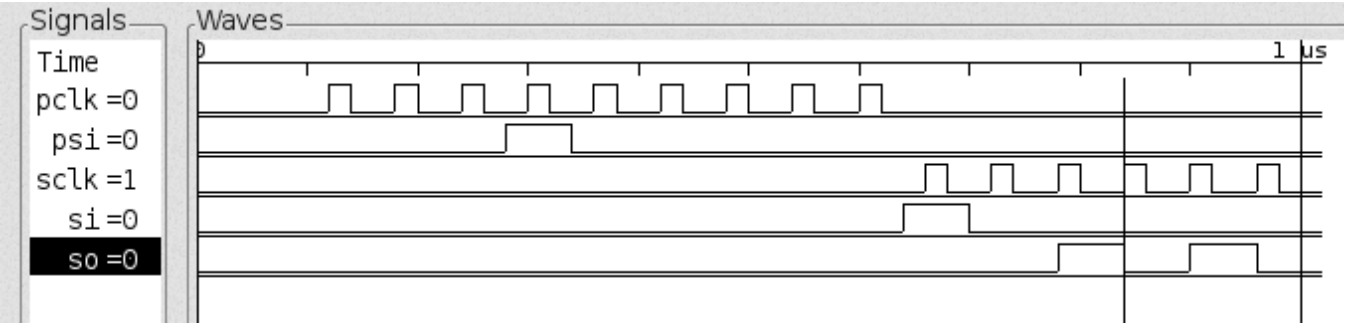

Figure 19: Top Level Functional Test Bench Waveform

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.

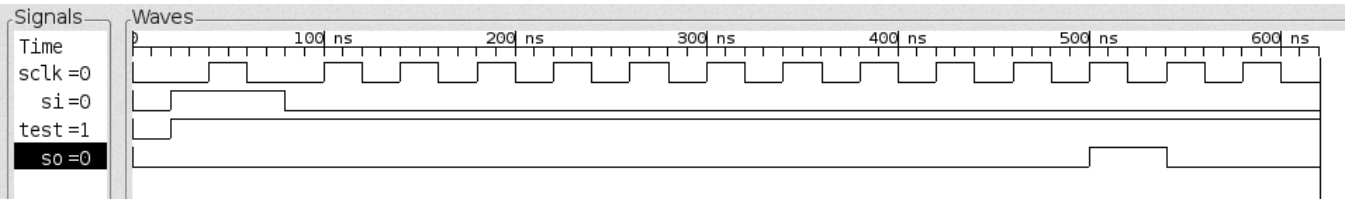

Figure 20: Top Level Test Mode Test Bench Waveform

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	Thrun
VHDL Top+TB	Both
VHDL Top Test Mode TB	Thrun

Table 2: Task Assignment