EECE6080 - HW 3

Max Thrun & Xiaohui Qi

October 4, 2013

Part 1

The objective of this part was to write a VHDL model of a parameterized n-bit FUN generator which accepts a generic parameter `n` and two n-bit vectors. The VHDL model we came up with is shown below:

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

entity fun is
    generic(
        n : integer := 8
    );
    port(
        a   : in  std_logic_vector(n-1 downto 0);
        b   : in  std_logic_vector(n-1 downto 0);
        f   : out std_logic_vector(n-1 downto 0)
    );
end entity;

architecture rtl of fun is
begin
    process (a,b) begin
        if(a<b) then
            f <= a;
        else
            f <= b;
        end if;
    end process;
end rtl;
```

Listing 1: Generic n-bit FUN generator

In order to test the functionality of our module we decided on an exhaustive test that checked every possible input. Given the small size of the design this was a reasonable approach as the execution time was negligible and it proved 100% correct functionality in the design. By using `assert` statements to check the output of the FUN module for each input condition the test is completely automated. The testbench used to drive the simulation is shown below. Note that the same testbench was used for both 3 and 8 bit-widths with the only change being the constant `n`.

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

entity fun_3_tb is
end fun_3_tb;

architecture tb_rtl of fun_3_tb is

    constant n : integer := 3;

    signal a : std_logic_vector(n-1 downto 0);
    signal b : std_logic_vector(n-1 downto 0);
    signal f : std_logic_vector(n-1 downto 0);

    component fun
        generic(
            n : integer := 8
        );
```

```
21          port(
22              a : in  std_logic_vector(n-1 downto 0);
23              b : in  std_logic_vector(n-1 downto 0);
24              f : out std_logic_vector(n-1 downto 0)
25          );
26      end component;
27
28  begin
29
30      uut : fun
31      generic map(
32          n => n
33      )
34      port map(
35          a => a,
36          b => b,
37          f => f
38      );
39
40      process begin
41          for aa in 0 to (2**n)-1 loop
42              for bb in 0 to (2**n)-1 loop
43
44                  a <= std_logic_vector(to_unsigned(aa, a'length));
45                  b <= std_logic_vector(to_unsigned(bb, b'length));
46                  wait for 10 ns;
47                  if (a < b) then
48                      assert f = a report "F not equal A" severity error;
49                  else
50                      assert f = b report "F not equal B" severity error;
51                  end if;
52
53              end loop;
54          end loop;
55          report "Test Complete" severity note;
56          wait;
57      end process;
58
59  end tb_rtl;
```

Listing 2: FUN Testbench

While our testbench is exhaustive and reported no failures we still spot checked the waveforms to ensure it was working as expected. The two screenshots below show that each configuration is performing as expected with the correct functionality.

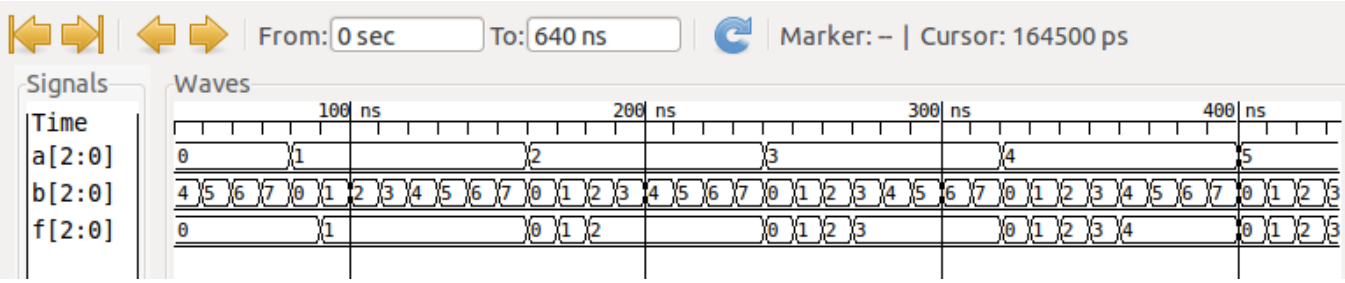

Figure 1: 3-Bit FUN Generator Simulation Waveform

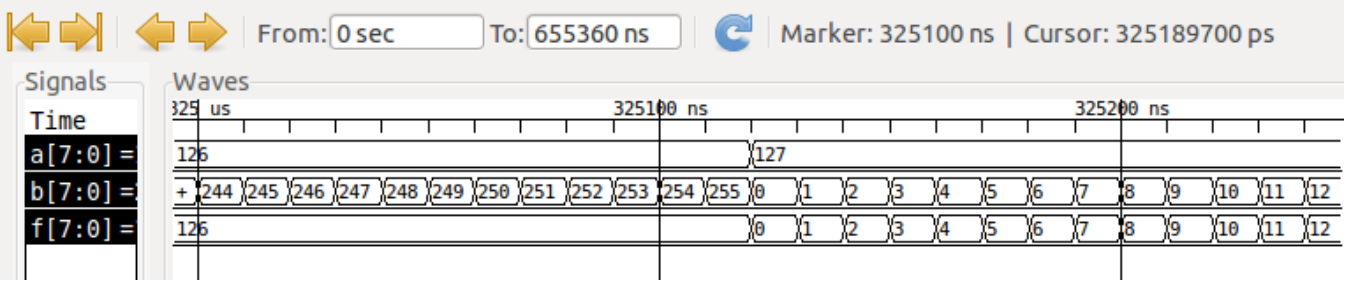

Figure 2: 8-Bit FUN Generator Simulation Waveform

Part 2

The objective of this part was to redesign the FUN module to utilize 1-bit slices. In order to do this we first came up with a slice design which we think is fairly optimized. It is composed of two main parts: a magnitude comparator, and a 2:1 multiplexer. The idea is that you can chain the magnitude comparators together and loop back the output of the last comparator to the select lines of each mux. A schematic representation of this design is shown below.

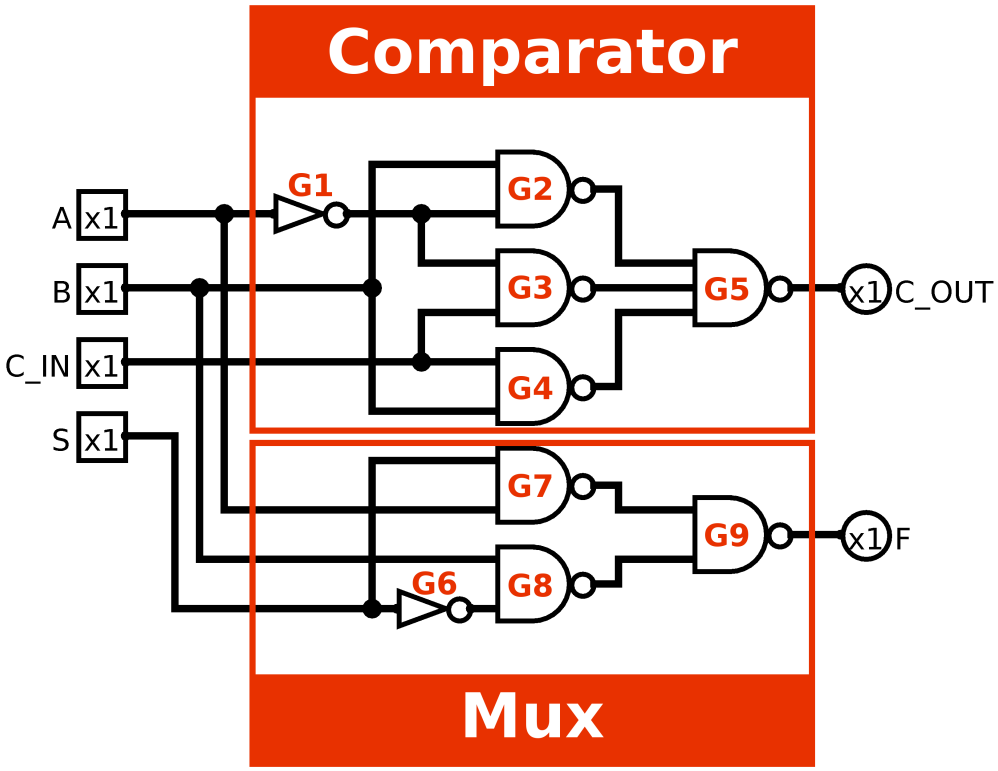

Figure 3: Gate Level Slice

An example circuit showing 4 slices chained together is shown below to illustrate the connections.

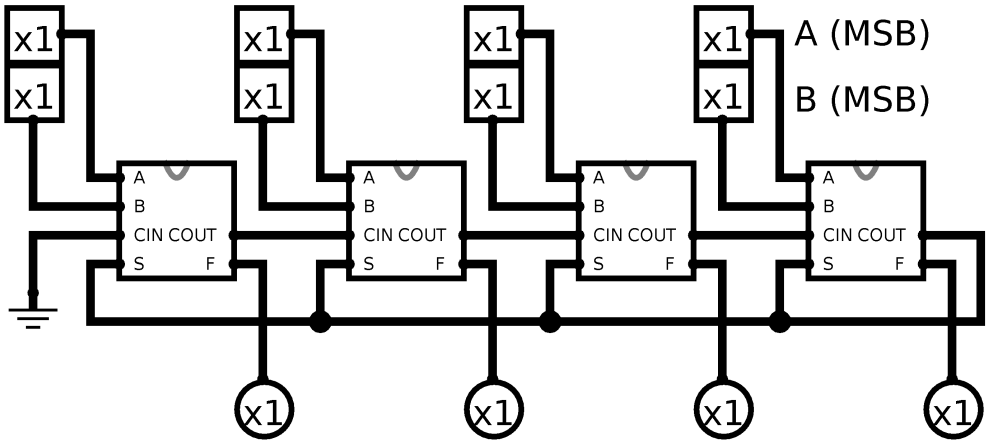

Figure 4: 4 Slice Chain

With the slice designed we translated it into a behavioral VHDL module shown below.

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

entity slice is
    port(
        a : in std_logic;
        b : in std_logic;
        s : in std_logic;
        f : out std_logic;
        c_i : in std_logic;
        c_o : out std_logic
    );
end slice;

architecture rtl of slice is begin

    process(a, b, s, c_i) begin

        -- comparator
        if (a = '1') and (b = '0') then
            c_o <= '0';
        elsif (a = '0') and (b = '1') then
            c_o <= '1';
        else
            c_o <= c_i;
        end if;

        -- mux
        if (s = '1') then
            f <= a;
        else
            f <= b;
        end if;

    end process;

end rtl;
```

Listing 3: Single Slice

We then transformed the FUN module to use `generate` in order to automatically chain `n` slices together. The FUN module is fairly simple with just a single generate block and a vector to hold the carries between slices.

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

entity fun is
    generic(
        n : integer := 8
    );
    port(
        a : in  std_logic_vector((n-1) downto 0);
        b : in  std_logic_vector((n-1) downto 0);
        f : out std_logic_vector((n-1) downto 0)
    );
end fun;
```

```
14
15 architecture rtl of fun is
16
17     component slice
18         port(
19                 a : in std_logic;
20                 b : in std_logic;
21                 s : in std_logic;
22                 f : out std_logic;
23                 c_i : in std_logic;
24                 c_o : out std_logic
25             );
26     end component;
27
28     -- vector to hold carry values between slices
29     signal carry : std_logic_vector(n downto 0) := (others => '0');
30
31     begin
32     fun_gen : for i in 0 to n-1 generate
33         slice_i: slice port map(
34             a => a(i),
35             b => b(i),
36             f => f(i),
37             c_o => carry(i+1),  -- i+1 because carry(0) is initial
38             c_i => carry(i),    -- carry(i) is c_o of previous slice
39             s => carry(n)       -- mux uses output of last slice
40         );
41     end generate;
42
43 end rtl;
```

Listing 4: FUN module using 'generate'

In order to ensure that the `generate` block was mapping the slices correctly we synthesized a 3-bit version of our design in Quartus and viewed the resulting RTL block diagram. From the diagram below we can see that our slices are properly linked and the carry out of the final slice is connected to the mux input of each slice. We can also see that the carry in to the first slice is a 0 which is the expected configuration.

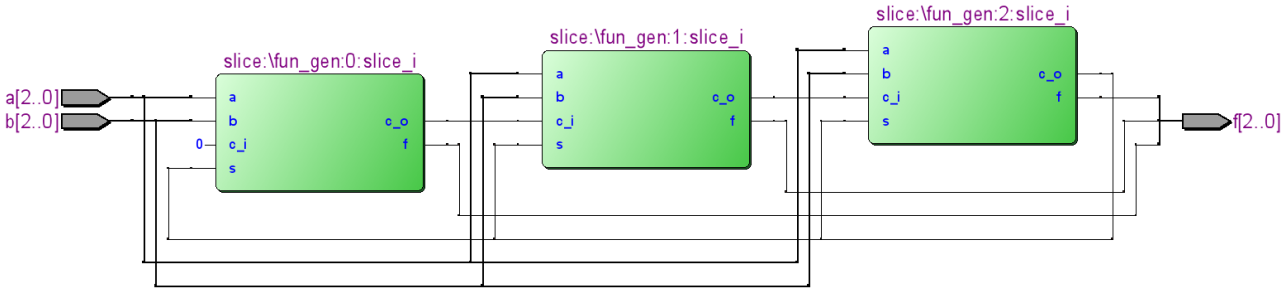

Figure 5: RTL block diagram of generated slices

Once we verified that our generate block was correct we ran it through the same simulations from Parts 1&2. Both simulations completed successfully and spot checks of the resulting waveforms also confirmed correct functionality. Given that we ran the exact same exhaustive testbench as Parts 1&2 we can say with certainty that our sliced design is functionally equivalent.

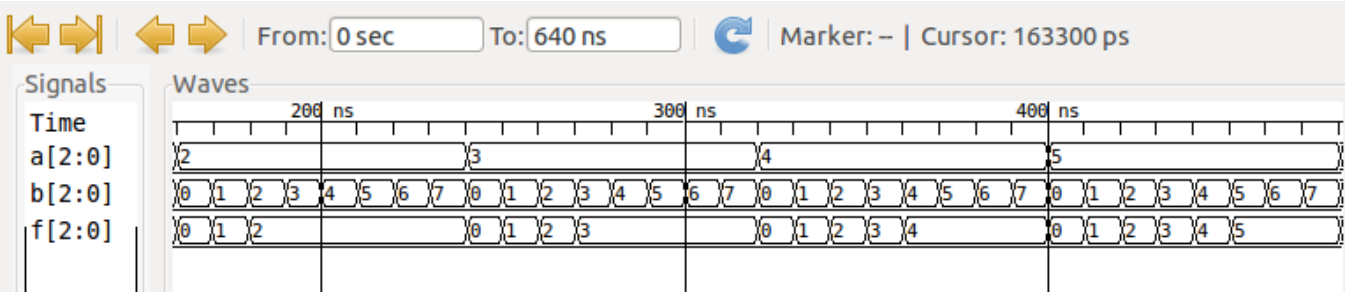

Figure 6: 3-Bit FUN Generator

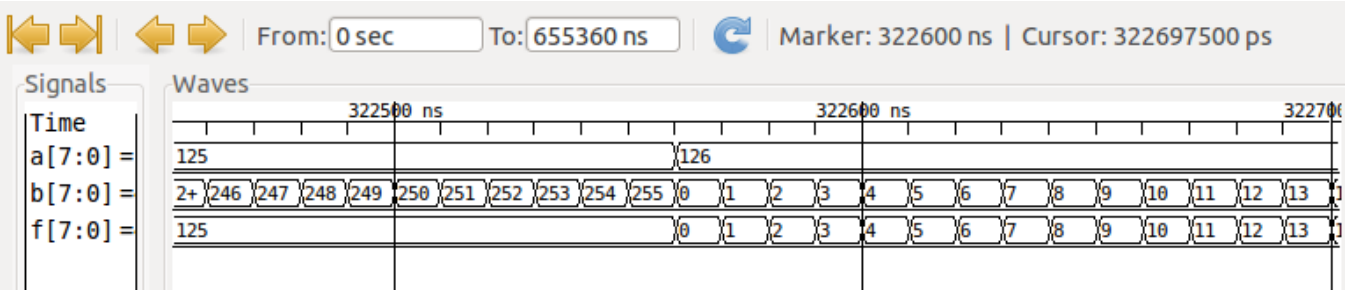

Figure 7: 8-Bit FUN Generator

Part 3

In this section we were required to each design a layout based on the slice design from Part 2. Since we already had a gate level representation of our slice it was fairly easy to convert it to a full CMOS representation using FETs. A diagram illustrating this representation is shown below with all transistors labeled. The size of the every transistor for both designs is tabulated below:

Parameter	Size
length	0.6u
n width	1.2u
p width	1.2u

Table 1: Transistor Dimensions

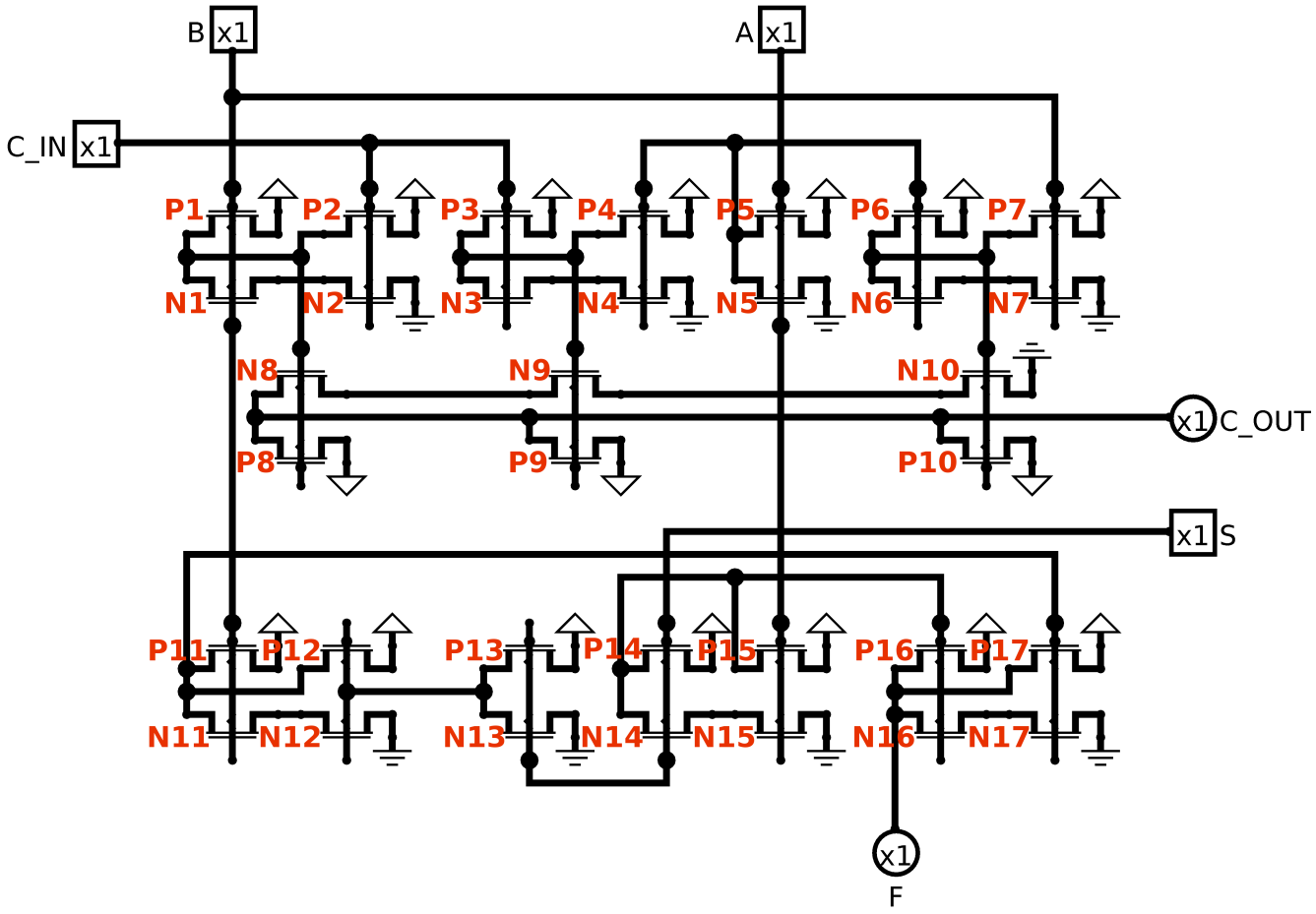

Figure 8: Transistor Diagram

Design A (Qi)

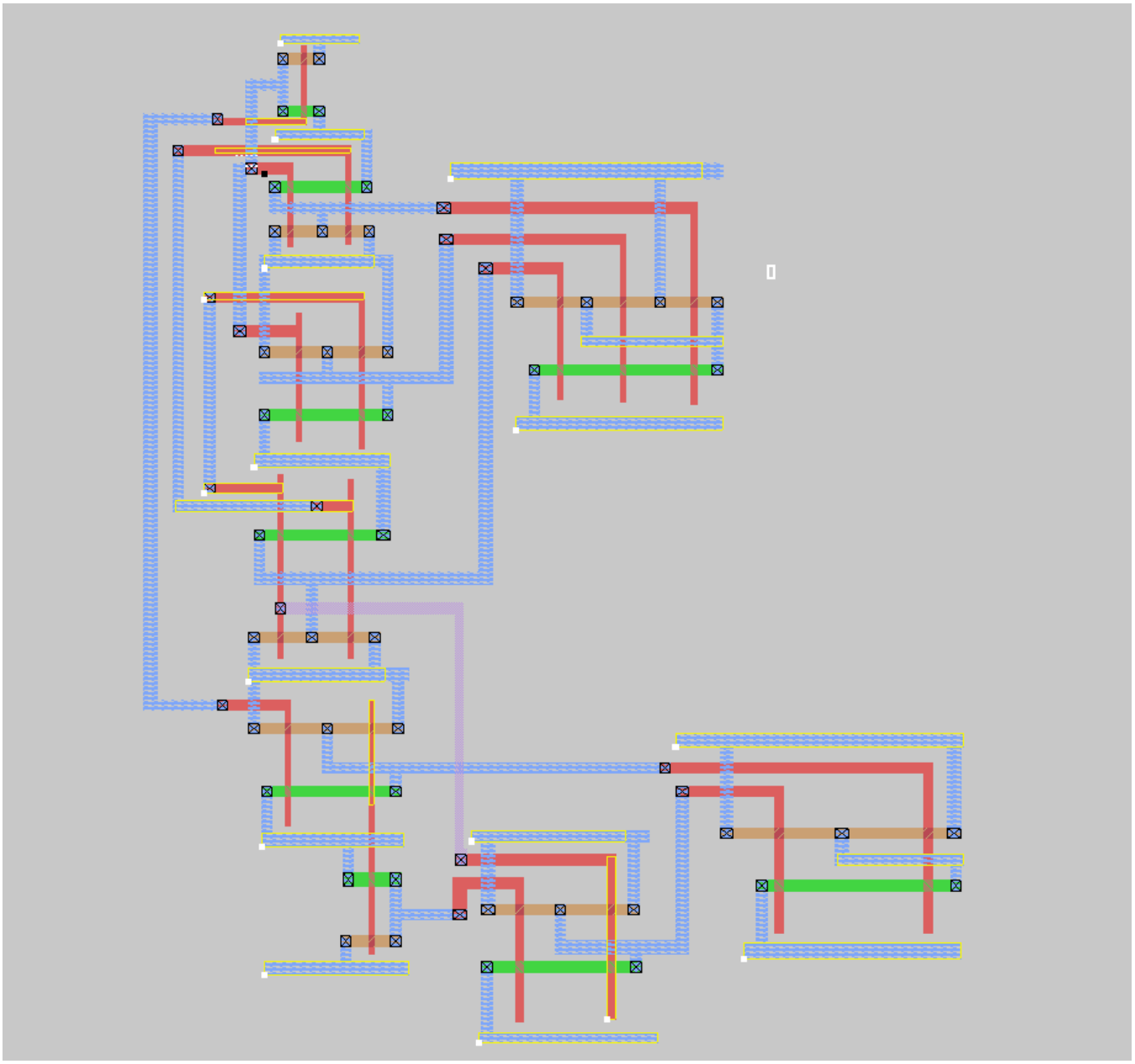

Figure 9: Design A: Layout

```
stepsize 10
analyzer A B cin S F cout
vector vec1 A B cin S
setvector vec1 1100
s
setvector vec1 0011
s
setvector vec1 1111
s
setvector vec1 0111
s
setvector vec1 1000
s
```

Listing 5: Design A: IRSIM Command File

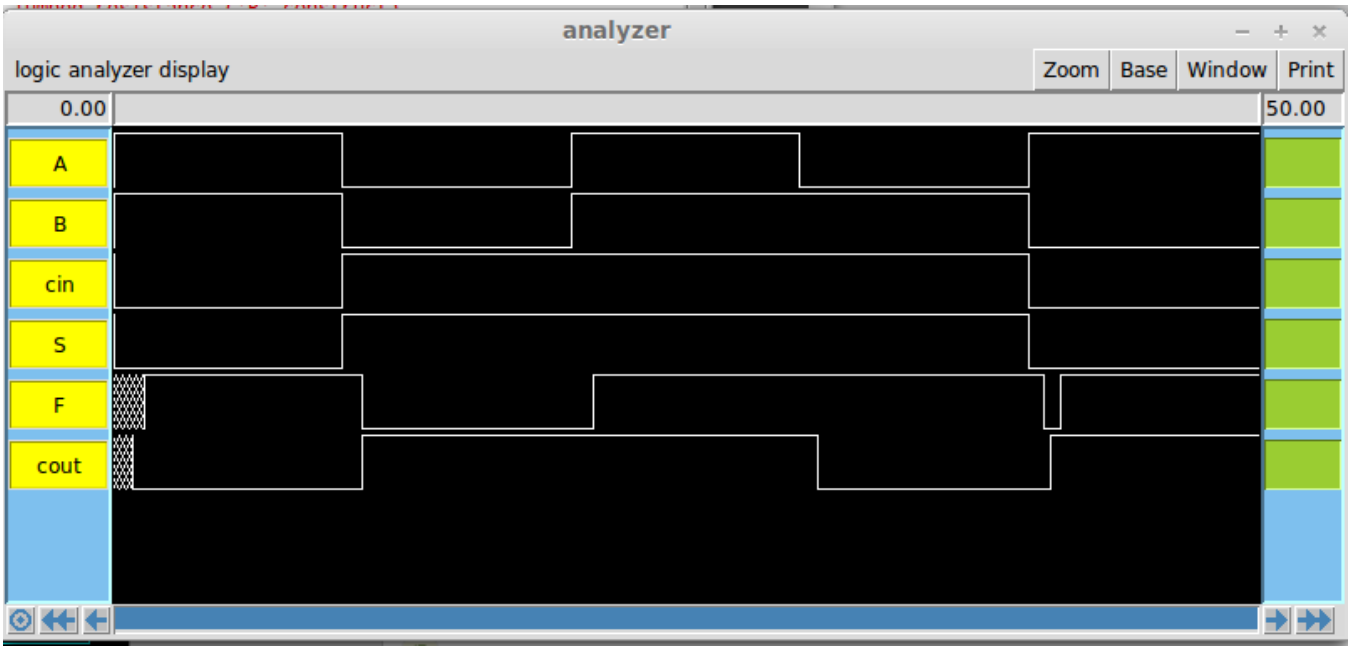

Figure 10: Design A: IRSIM Output

```
*********

.include ../../../models/model_t36s.sp
.include bitslice.spice

VDD vdd gnd 5v
Va A gnd PWL(0 5v 4.9n 5v 5n 0v 9.9n 0v 10n 5v 14.9n 5v 15n 0v 19.9n 0v 20n 5v 24.9n 5v)
Vb B gnd PWL(0 5v 4.9n 5v 5n 0v 9.9n 0v 10n 5v 19.9n 5v 20n 0v 24.9n 0v)
Vcin cin gnd PWL(0 0v 4.9n 0v 5n 5v 19.9n 5v 20n 0v 24.9n 0v)
Vs S gnd PWl(0 0v 4.9n 0v 5n 5v 19.9n 5v 20n 0v 24.9n 0v)
.option post
.tran 0.1n 25n
.end
```

Listing 6: Design A: Spice Simulation File

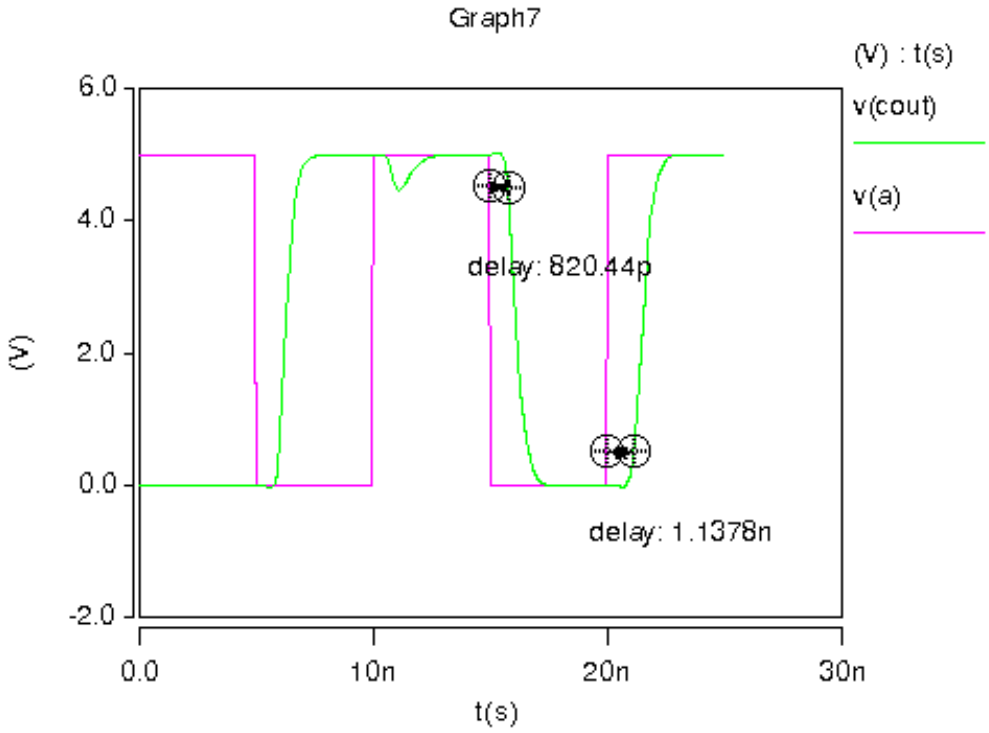

Figure 11: Design A: A to C_OUT Delay

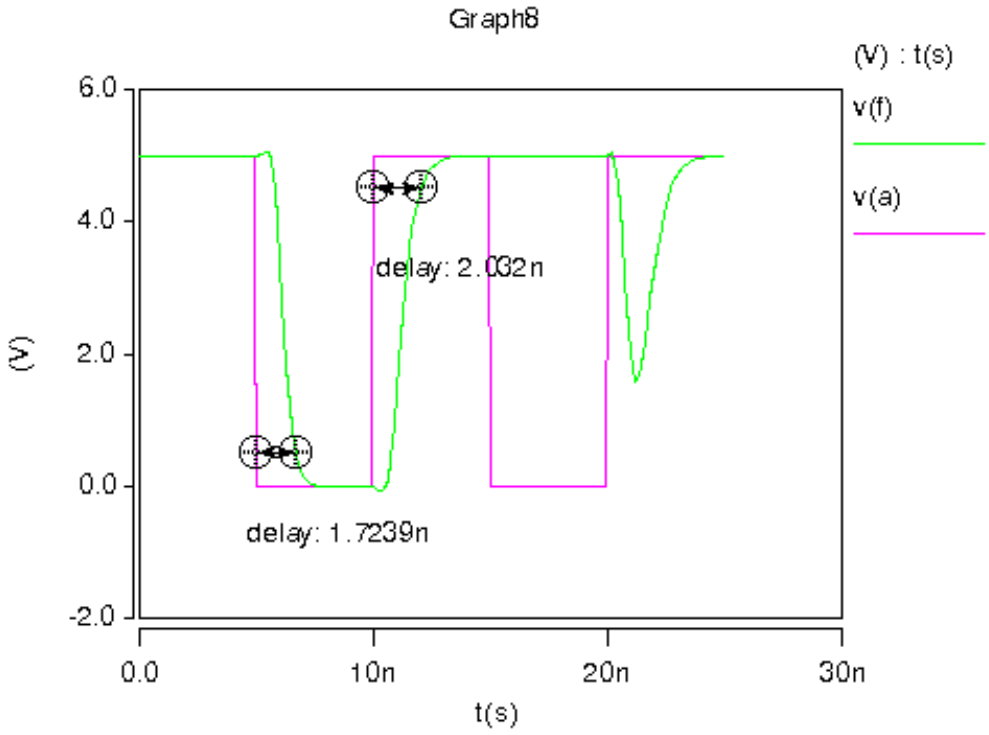

Figure 12: Design A: A to F Delay

Design B (Thrun)

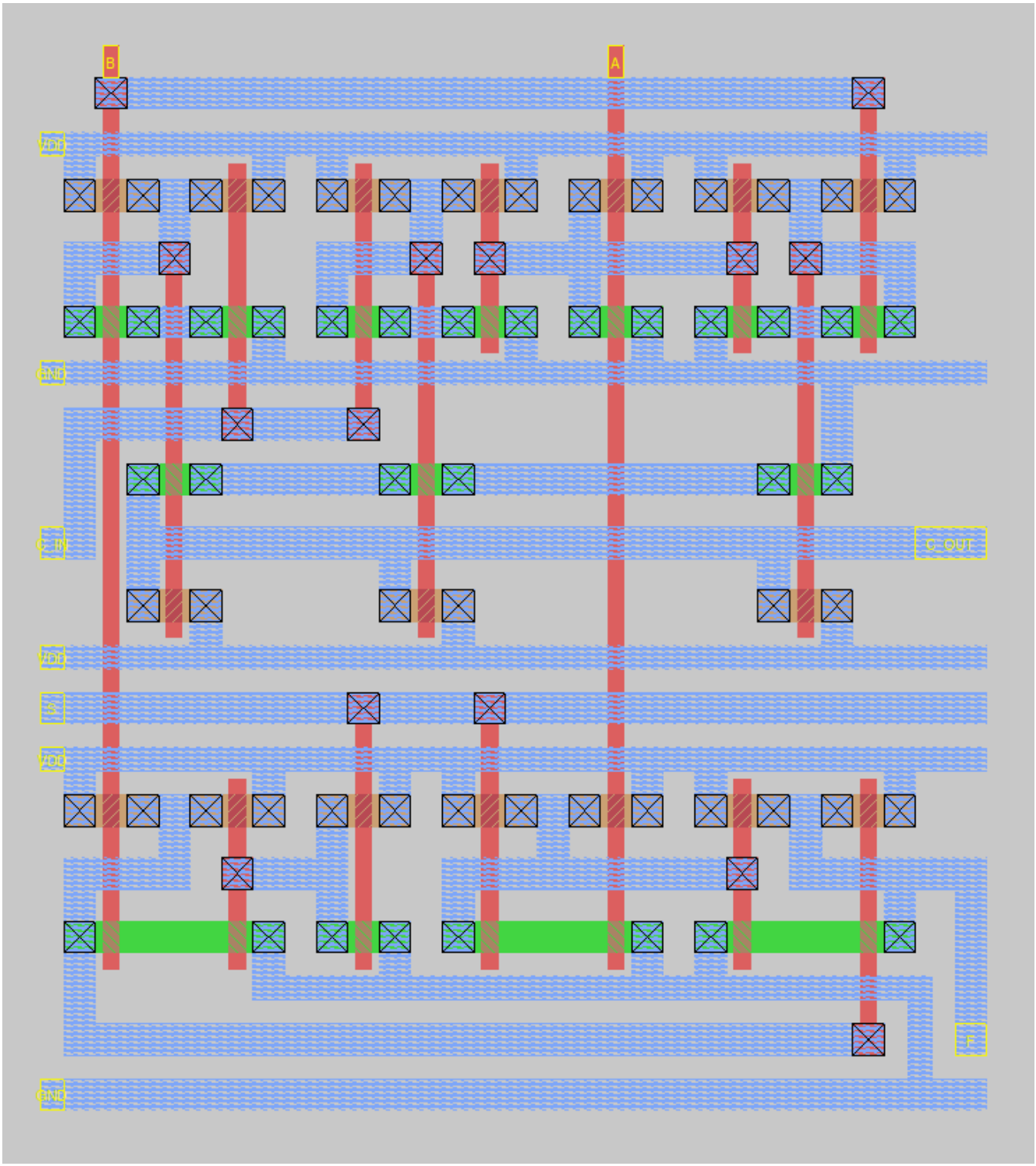

Figure 13: Design B: Layout

```
stepsize 50

logfile irsim_f.log

h VDD
l GND

l A B C_IN S
vector in A B C_IN S
ana A B C_IN S C_OUT F
w A B C_IN S C_OUT F

setvector in 0000
s
path F
setvector in 0001
s
path F
setvector in 0010
s
```

Listing 7: Design B: IRSIM Command File (truncated)

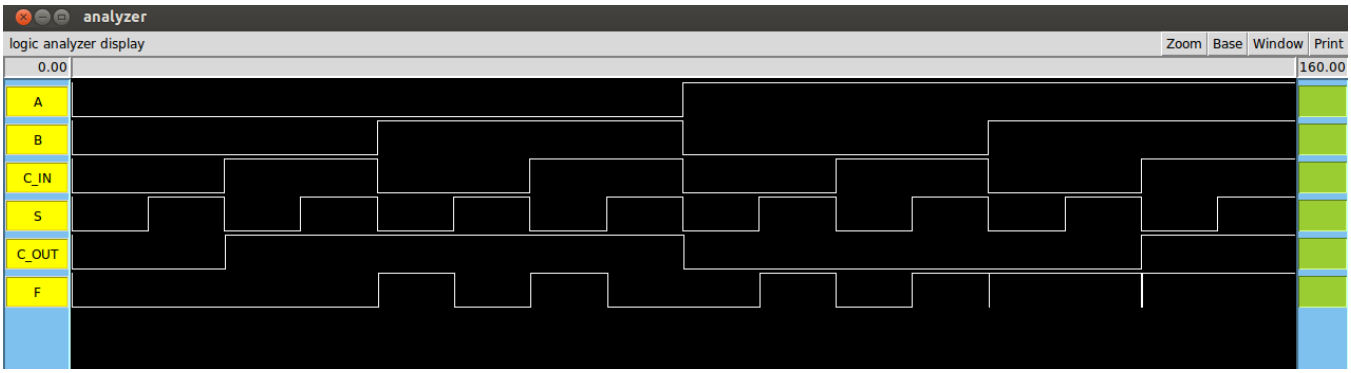

Figure 14: Design B: IRSIM Output

```
*
* MODELS
*

.include ../../../models/model_t36s.sp
.include slice.spice

*
* CIRCUIT
*

VDD vdd gnd 5V

*
* SIMULATION
*

* delay B to F
*va  A     gnd PWL(0n 0V)
```

```
*vb  B     gnd PWL(0n 0V 4.9n 0V 5n 5V 9.9n 5V 10n 0V)
*vc  C_IN  gnd PWL(0n 5V 4.9n 5V 5n 0V 9.9n 0V 10n 5V)
*vs  S     gnd PWL(0n 0V)

* delay B to C_OUT
va  A     gnd PWL(0n 0V)
vb  B     gnd PWL(0n 0V 4.9n 0V 5n 5V 9.9n 5V 10n 0V)
vc  C_IN  gnd PWL(0n 0V)
vs  S     gnd PWL(0n 0V)

.option post
.tran 0.01n 15n
.end
```

Listing 8: Design B: Spice Simulation File

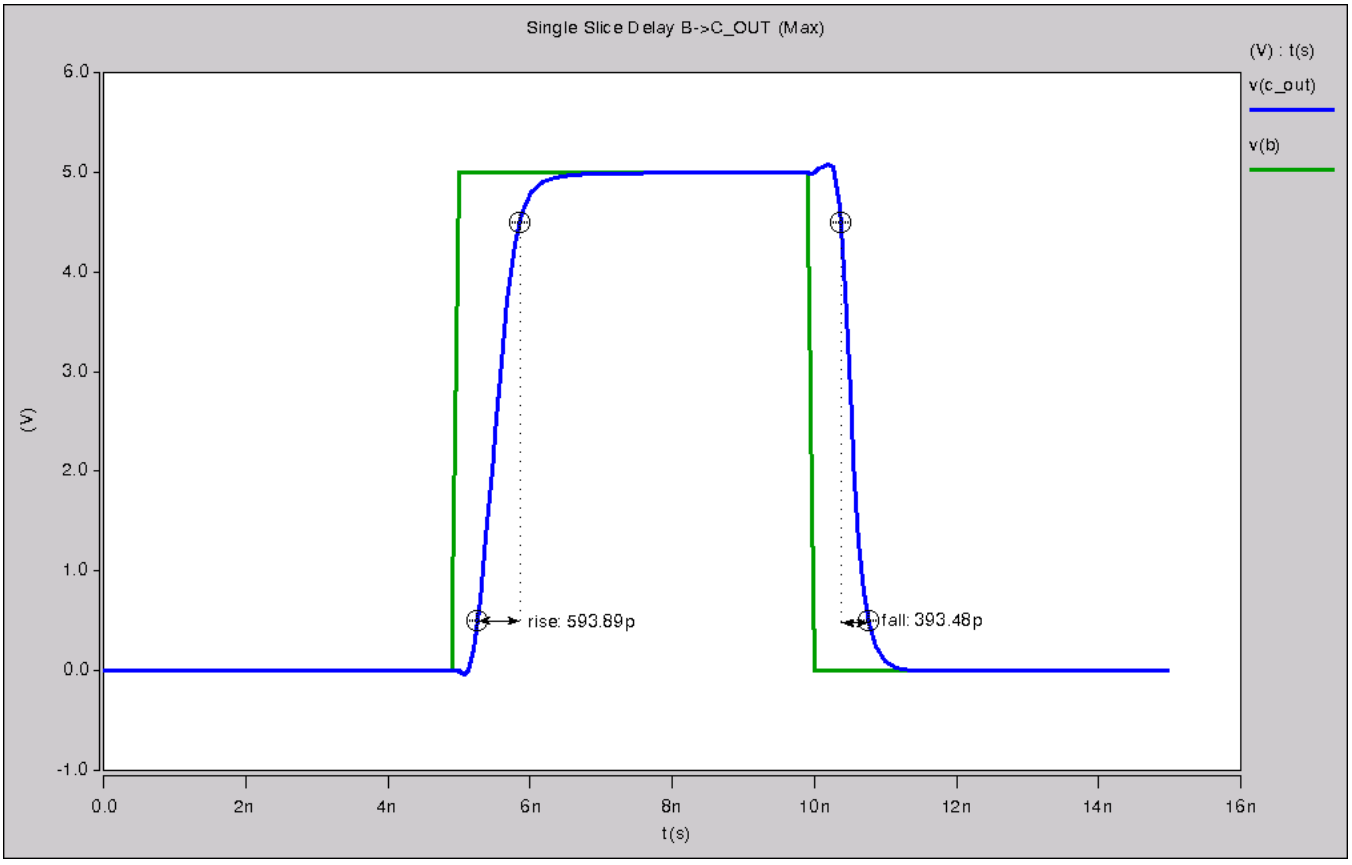

Figure 15: Design B: B to C_OUT Delay

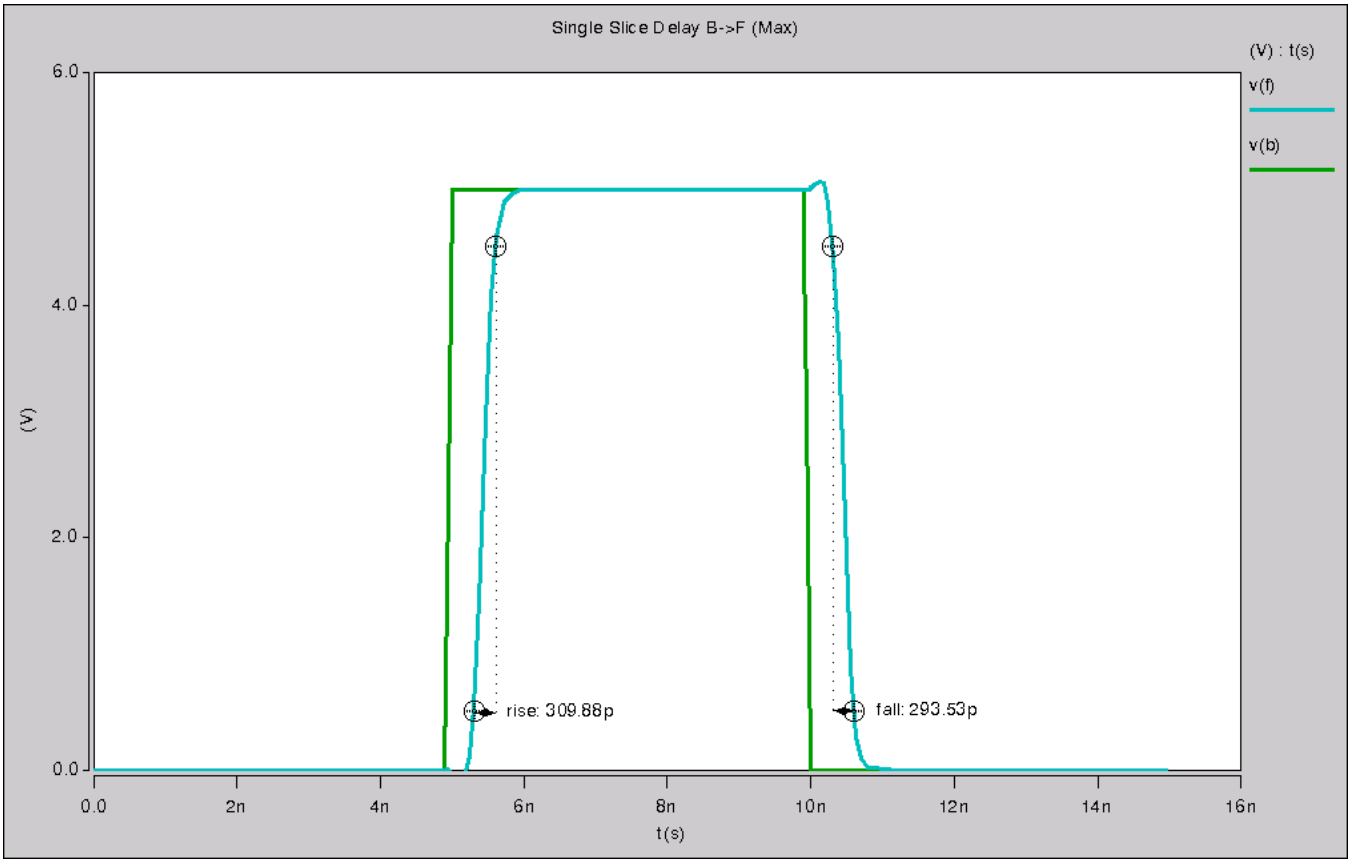

Figure 16: Design B: B to F Delay

Design Summary

A comparison of delays obtained from hspice for both designs is tabulated below. From the results we can see that design B is significantly faster. Given this result, combined with the fact that the layout of design B lends itself to easier patterning, we chose design B as the design to move forward with.

Parameter	Design A	Design B
A/B to C_Out (Fall)	820.44ps	393.48ps
A/B to C_Out (Rise)	1.1378ns	593.89ps
A/B to F (Fall)	1.7239ns	293.53ps
A/B to F (Rise)	2.0320ns	309.88ps

Table 2: Delay Comparison

Part 4

The objective of this part was to rewrite our slice architecture from Part 2 to use component instances of VHDL models of the static gates we used to implement the slice ensuring that we include the worst case delay. In order to achieve this the first thing we needed to do was simulate each gate and find the worst case delay times. The spice simulation file and the resulting output for each of the three different gates are shown below.

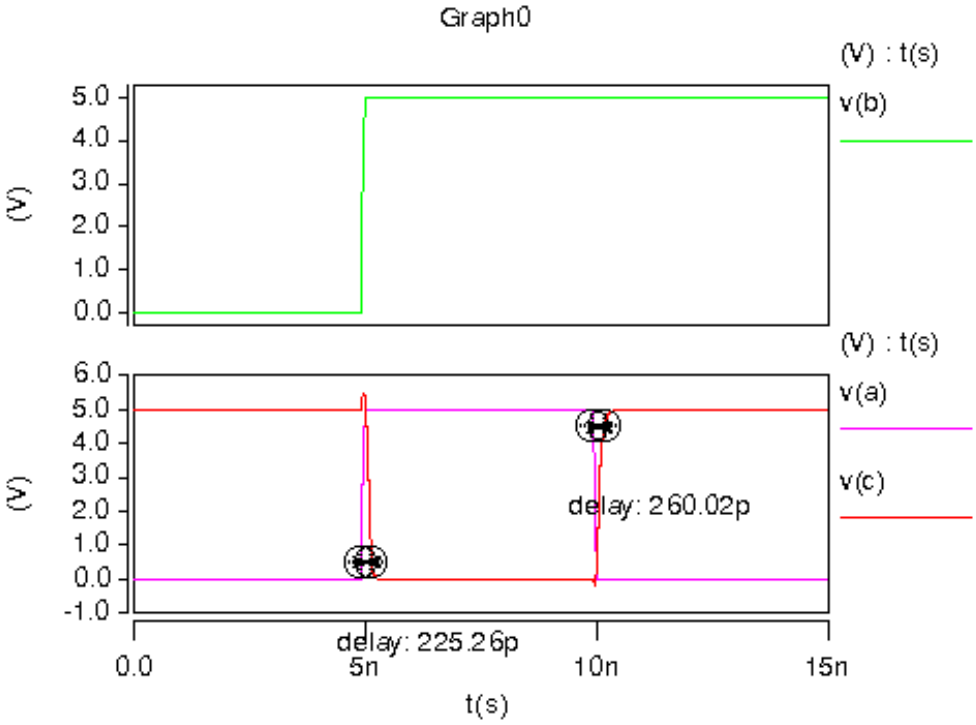

Figure 17: 2 Input NAND Gate

```
.include ../../../models/model_t36s.sp

.param n=4
.param wp=0.3u*n
.param wn=0.3u*n
.param hd=2u

mPMos1 c a vdd vdd pfet w=wp l=0.6u ad=hd*wp pd=2*hd+2*wp as=hd*wp ps=2*hd+2*wp
mPMos2 b a vdd vdd pfet w=wp l=0.6u ad=hd*wp pd=2*hd+2*wp as=hd*wp ps=2*hd+2*wp

mNMos1 c a d d nfet w=wn l=0.6u ad=hd*wn pd=2*hd+2*wn as=hd*wn ps=2*hd+2*wn
mNMos2 d b gnd gnd nfet w=wn l=0.6u ad=hd*wn pd=2*hd+2*wn as=hd*wn ps=2*hd+2*wn

VDD vdd gnd 5v
Va a gnd PWL(0 0v 4.9n 0v 5n 5v 9.9n 5v 10n 0v 14.9n 0v )
Vb b gnd PWL(0 0v 4.9n 0v 5n 5v 9.9n 5v 10n 5v 14.9n 5v )

.option post
.tran 0.1n 15n
.end
```

Listing 9: 2 Input NAND Gate Spice Simulation File

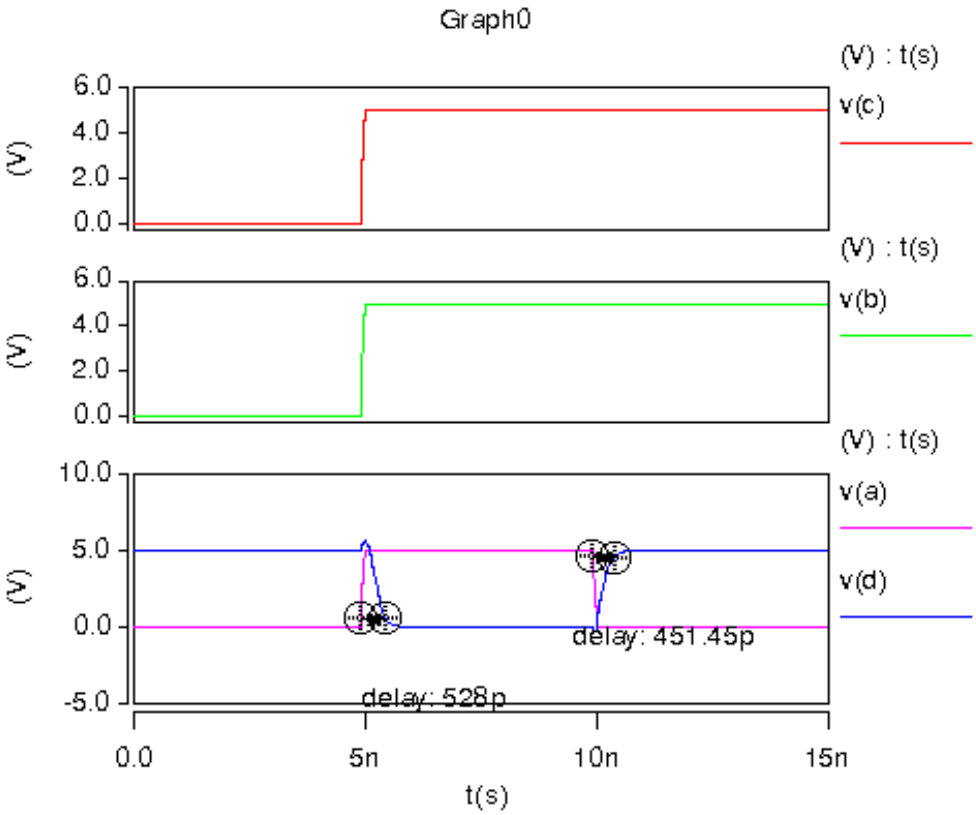

Figure 18: 3 Input NAND Gate

```
.include ../../../models/model_t36s.sp

.param n=4
.param wp=0.3u*n
.param wn=0.3u*n
.param hd=2u

mPMos1 d a vdd vdd pfet w=wp l=0.6u ad=hd*wp pd=2*hd+2*wp as=hd*wp ps=2*hd+2*wp
mPMos2 d b vdd vdd pfet w=wp l=0.6u ad=hd*wp pd=2*hd+2*wp as=hd*wp ps=2*hd+2*wp
mPMos3 d c vdd vdd pfet w=wp l=0.6u ad=hd*wp pd=2*hd+2*wp as=hd*wp ps=2*hd+2*wp

mNMos1 d a e e nfet w=wn l=0.6u ad=hd*wn pd=2*hd+2*wn as=hd*wn ps=2*hd+2*wn
mNMos2 e b f f nfet w=wn l=0.6u ad=hd*wn pd=2*hd+2*wn as=hd*wn ps=2*hd+2*wn
mNMos3 f c gnd gnd nfet w=wn l=0.6u ad=hd*wn pd=2*hd+2*wn as=hd*wn ps=2*hd+2*wn

VDD vdd gnd 5v
Va a gnd PWL(0 0v 4.9n 0v 5n 5v 9.9n 5v 10n 0v 14.9n 0v )
Vb b gnd PWL(0 0v 4.9n 0v 5n 5v 9.9n 5v 10n 5v 14.9n 5v )
Vc c gnd PWL(0 0v 4.9n 0v 5n 5v 9.9n 5v 10n 5v 14.9n 5v )

.option post
.tran 0.1n 15n
.end
```

Listing 10: 3 Input NAND Gate Spice Simulation File

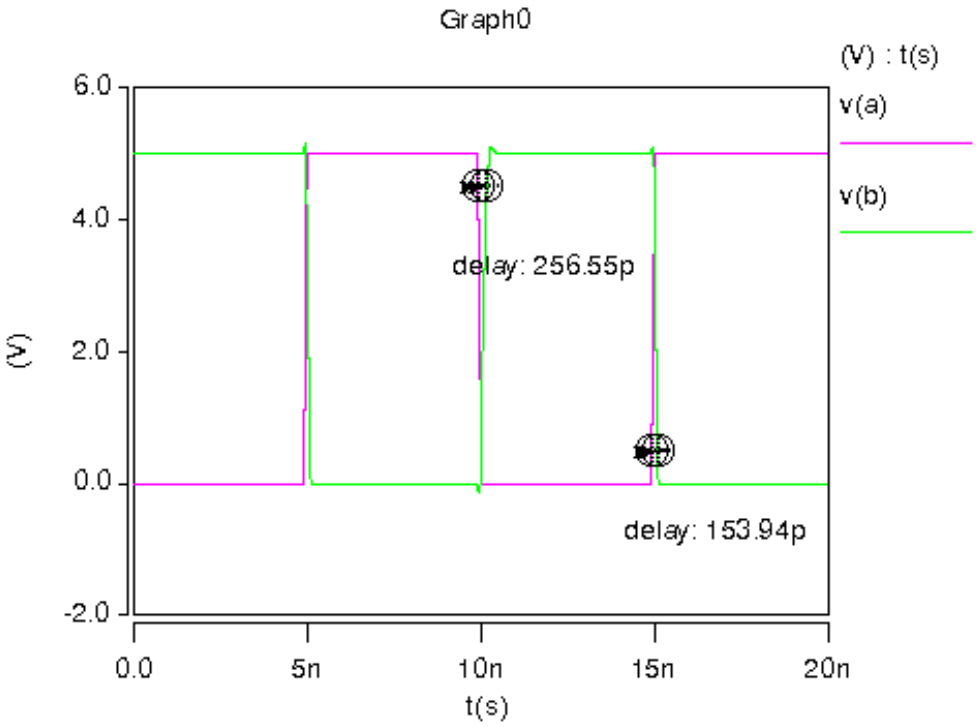

Figure 19: NOT Gate

```
.include ../../../models/model_t36s.sp

.param n=4
.param wp=0.3u*n
.param wn=0.3u*n
.param hd=2u

mPMos1 b a vdd vdd pfet w=wp l=0.6u ad=hd*wp pd=2*hd+2*wp as=hd*wp ps=2*hd+2*wp
mNMos2 b a gnd gnd nfet w=wn l=0.6u ad=hd*wn pd=2*hd+2*wn as=hd*wn ps=2*hd+2*wn

VDD vdd gnd 5v
Va a gnd PWL(0 0v 4.9n 0v 5n 5v 9.9n 5v 10n 0v 14.9n 0v 15n 5v 19.9n 5v)

.option post
.tran 0.1n 20n
.end
```

Listing 11: NOT Gate Spice Simulation File

The worst case delays for each of these gates is tabulated below.

Gate	Worst Case Delay
NAND-2	260.02ps
NAND-3	528ps
NOT	256.55ps

Table 3: Worst Case Gate Delay

Knowing the worst case delay for each gate we can construct each as a separate module in VHDL.

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

entity nandgate2  is
    port (
    a,b :in std_logic;
    c :out std_logic
);
end entity;

architecture arc of nandgate2 is
begin
    process(a,b) begin
        c <= not(a and b) after 260.02 ps;
    end process;
end architecture;
```

Listing 12: 2 Input NAND Gate Spice Simulation File

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

entity nandgate3  is
    port (
    a,b,c :in std_logic;
    d :out std_logic
);
end entity;

architecture arc of nandgate3 is
begin
    process(a,b,c) begin
        d <= not(a and b and c) after 528 ps;
    end process;
end architecture;
```

Listing 13: 3 Input NAND Gate Spice Simulation File

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

entity notgate is
    port (
              a :in std_logic;
              b :out std_logic
         );
end entity;

architecture arc of notgate is
begin
    process (a) begin
        b <= not a after 256.55 ps;
    end process;
end architecture;
```

Listing 14: NOT Gate Spice Simulation File

With each gate implemented we can rewrite our slice to match the original gate level design shown in part 2. The new slice module is shown below. Note that the FUN module remains the same as Part 2.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity slice is
5      port(
6          a : in std_logic;
7          b : in std_logic;
8          s : in std_logic;
9          f : out std_logic;
10         c_i : in std_logic;
11         c_o : out std_logic
12     );
13 end slice;
14
15 architecture rtl of slice is
16     -- comparator
17     signal g1_o : std_logic;
18     signal g2_o : std_logic;
19     signal g3_o : std_logic;
20     signal g4_o : std_logic;
21
22     -- mux
23     signal g6_o : std_logic;
24     signal g7_o : std_logic;
25     signal g8_o : std_logic;
26
27 begin
28
29     -- comparator
30     g1 : entity work.notgate   port map(a,   g1_o);
31     g2 : entity work.nandgate2 port map(b,   g1_o, g2_o);
32     g3 : entity work.nandgate2 port map(c_i, g1_o, g3_o);
33     g4 : entity work.nandgate2 port map(c_i, b,    g4_o);
34     g5 : entity work.nandgate3 port map(g2_o, g3_o, g4_o, c_o);
35
36     -- mux
37     g6 : entity work.notgate   port map(s, g6_o);
38     g7 : entity work.nandgate2 port map(s, a, g7_o);
39     g8 : entity work.nandgate2 port map(b, g6_o, g8_o);
40     g9 : entity work.nandgate2 port map(g7_o, g8_o, f);
41
42 end architecture;
```

Listing 15: Slice With Static Gates

In our design the worst case delay is when a change in either A0 or B0 results in a change in F0. The change in A0 or B0 has to propagate through all the slices and then back through all the muxes until it finally reaches F0. Knowing this we can target our testbench to only simulate this condition. The testbench simulating a 3-bit configuration is shown below. Note that the testbench for the 8-bit configuration is exactly the same except the `n` constant is changed to 8 and the input vectors are also 8 bit.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity fun_3_tb is
5  end fun_3_tb;
6
7  architecture rtl of fun_3_tb is
8
9      constant n : integer := 3;
10
11     signal a : std_logic_vector(n-1 downto 0);
12     signal b : std_logic_vector(n-1 downto 0);
13     signal f : std_logic_vector(n-1 downto 0);
14
15     component fun
16         generic(
17             n : integer := 8
18         );
19         port(
20             a : in  std_logic_vector(n-1 downto 0);
21             b : in  std_logic_vector(n-1 downto 0);
22             f : out std_logic_vector(n-1 downto 0)
23         );
24     end component;
25
26 begin
27
28     uut : fun
29     generic map(
30         n => n
31     )
32     port map(
33         a => a,
34         b => b,
35         f => f
36     );
37
38     process begin
39         a<="001";
40         b<="001";
41         wait for 100 ns;
42         a<="000";
43         b<="001";
44         wait for 100 ns;
45         a<="001";
46         b<="001";
47         wait for 100 ns;
48         wait;
49     end process;
50
51 end rtl;
```

Listing 16: Static Gate Slice Testbench

Looking at the output waveform we can measure the delta between the input state change and the output state change. Measurements were taken for both a falling edge and rising edge change for both the 3 and 8 bit configurations. The four output waveforms are shown below.

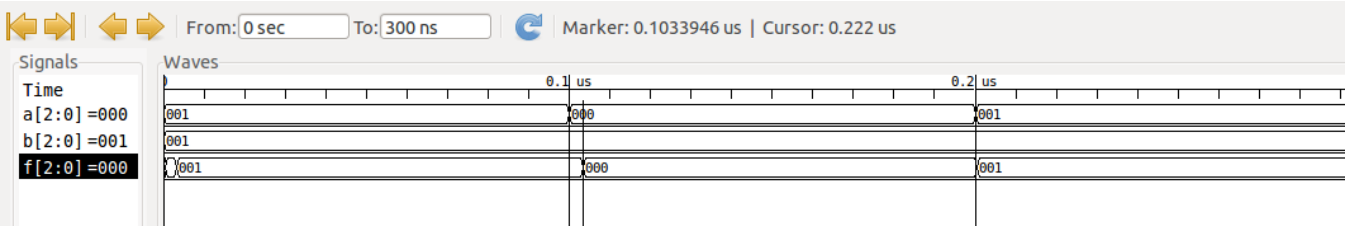

Figure 20: 3-Bit Fall Delay

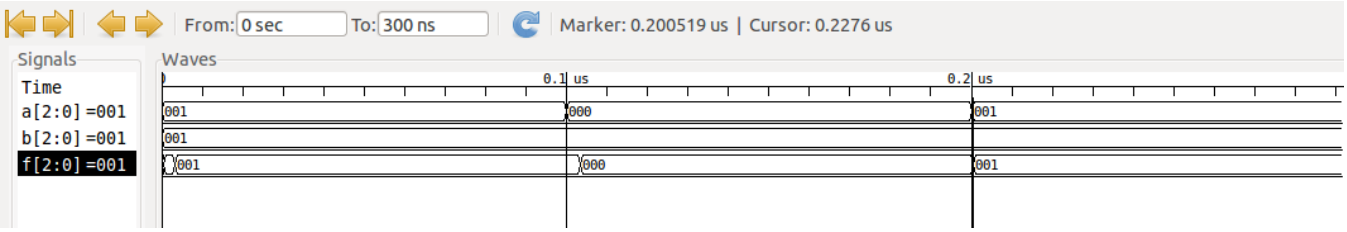

Figure 21: 3-Bit Rise Delay

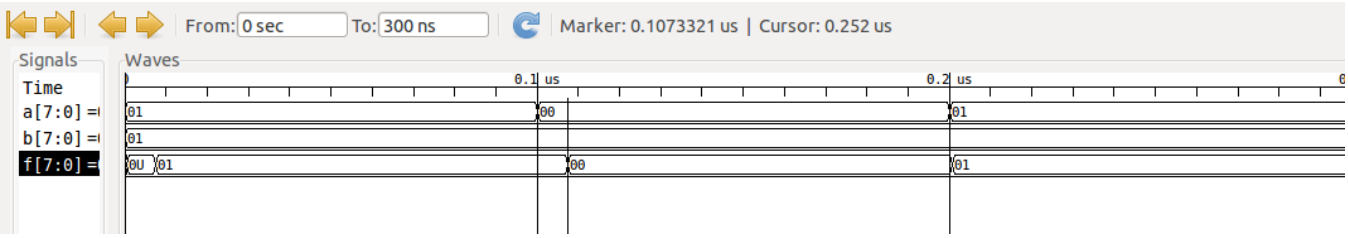

Figure 22: 8-Bit Fall Delay

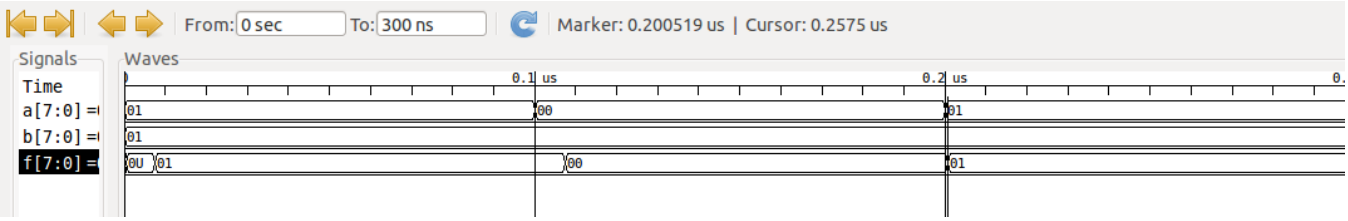

Figure 23: 8-Bit Rise Delay

All the delays are tabulated below.

Configuration	Fall Delay	Rise Delay
3-Bit	3.3946ns	0.519ns
8-Bit	7.3321ns	0.519ns

Table 4: Worst Case Delay Through Chain

Part 5

The objective of this section is to generate the layouts of a 3-bit and 8-bit FUN generator and determine their worst case delays. This was easily achieved by using the `array` command in magic. Since our slice tiles together perfect it was almost trivial to chain them. The only paint we had to add was to connect the last slices C_OUT to the last slices S. The layouts for both configurations are shown below.

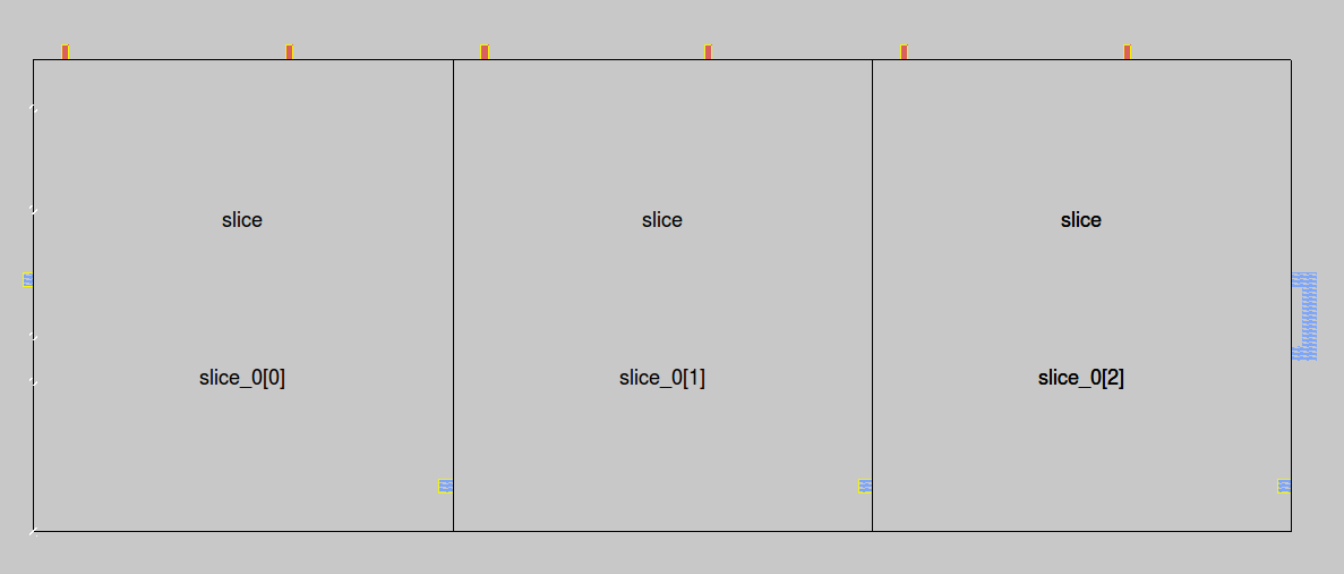

Figure 24: 3-Bit Layout

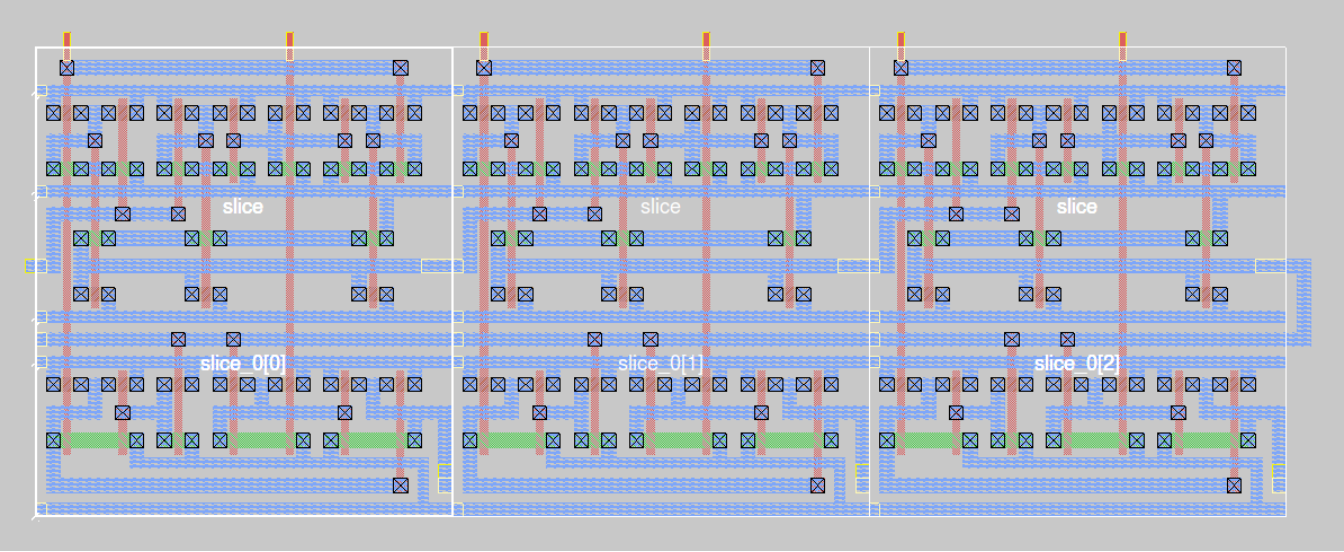

Figure 25: 3-Bit Layout Expanded

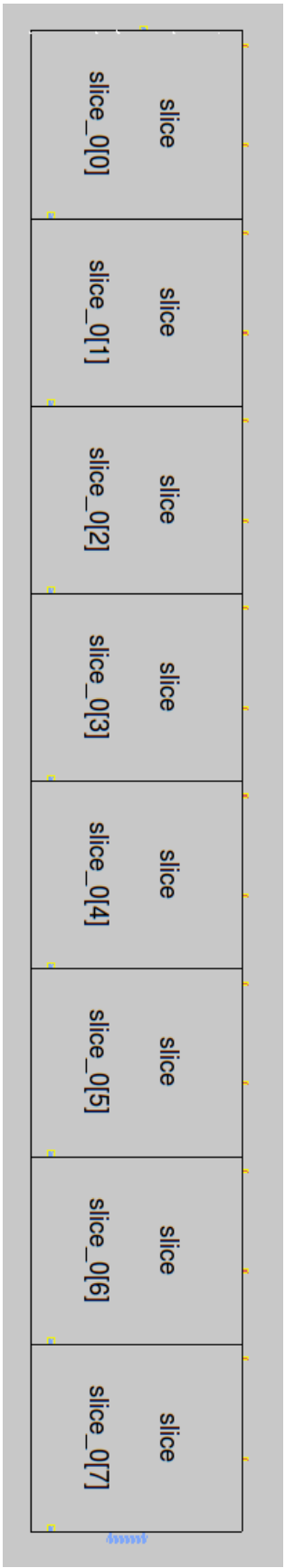

Figure 26: 8-Bit Layout

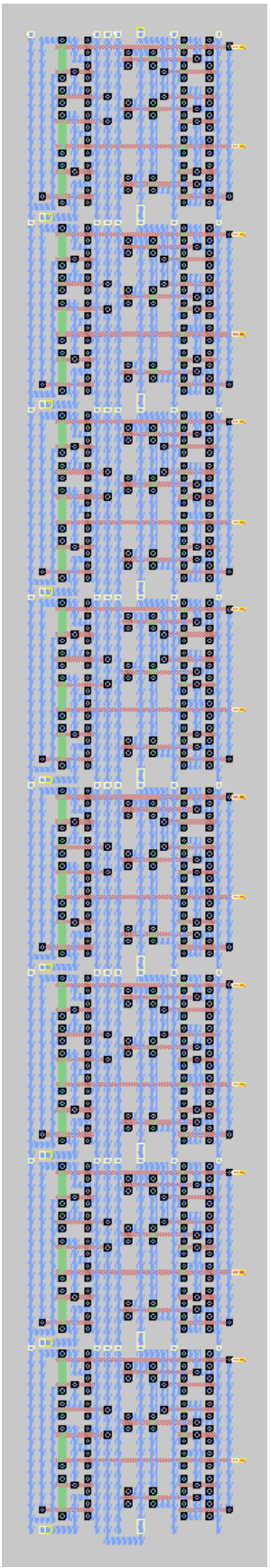

Figure 27: 8-Bit Layout Expanded

In order to test the functionality of our multi-bit FUN generators we used two different strategies. For the 3-bit version we simply did an exhaustive simulation and manually looked at the waveform. To generate all the input vectors a simple Python script was written. For the 8-bit version a similar strategy was used except instead of a completely exhaustive search we skip over chunks.

```
for a in range(0, 2**3):
    for b in range(0, 2**3):
        print "setvector A %s" % (bin(a).replace("0b","").zfill(3))
        print "setvector B %s" % (bin(b).replace("0b","").zfill(3))
        print "s"
        print "path F"
```

Listing 17: Python Vector Generator

The output IRSIM waveforms for each configuration is shown below.

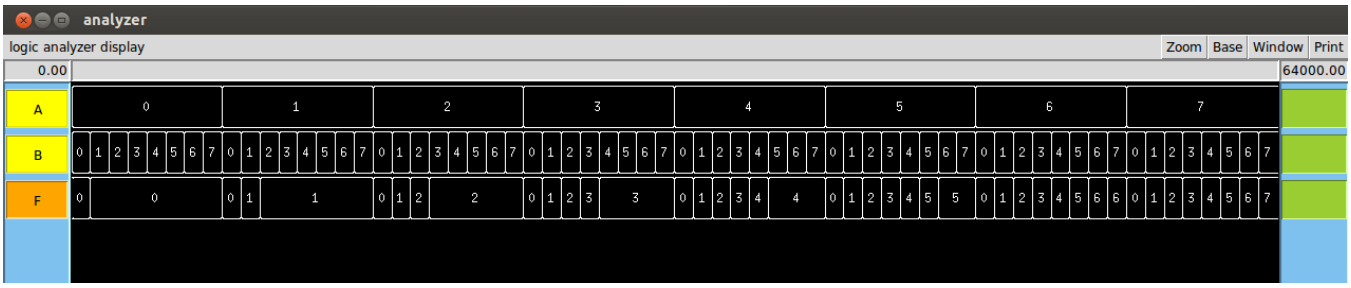

Figure 28: 3-Bit IRSIM Waveform

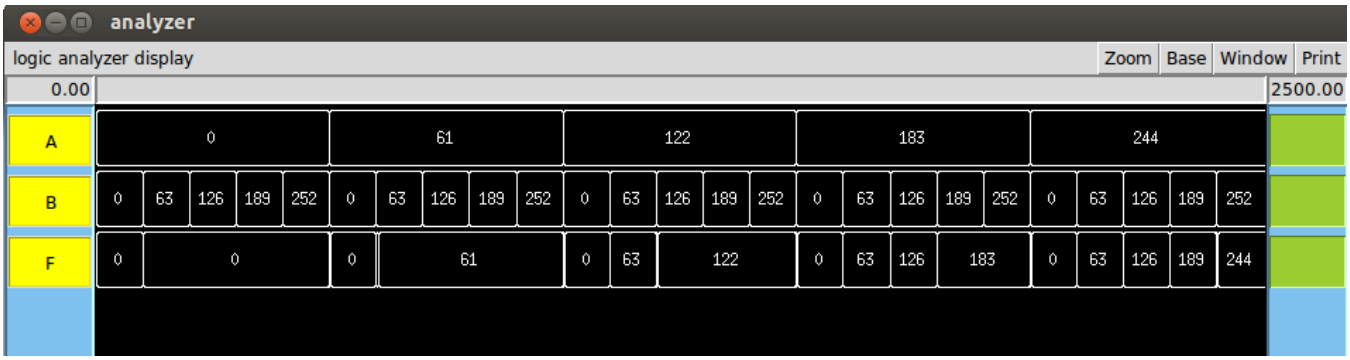

Figure 29: 8-Bit IRSIM Waveform

To get the worst case delay out of IRSIM we utilized the `PATH` command which tells you delay through the critical path. Interestingly it reported that our worst case delay was **0.003ns** for each configuration. This is hard to believe but it might be possible due to the fact that IRSIM says it's 'Ignoring lumped-resistance'. In regards to trying to increase the accuracy of the VHDL model it could be possible to account for additional capacitances introduced by the wiring of all the gates. Instead of introducing the delay per gate you could introduce the total delay per slice. A table comparing these results to the VHDL simulation is shown below.

Configuration	VHDL	IRSIM
3-Bit	3.3946ns	0.0030ns
8-Bit	7.3321ns	0.0030ns

Table 5: Worst Case Delay Comparison Between VHDL, IRSIM, and Spice

Part 6

The objective of this part was to simulate the layout level designs using Spice. Since we only wanted to focus on the worst case delay we chose an input transition on slice 0 that causes and output transition on slice 0. The simulation file to achieve this is shown below. Note that the simulation file for the 8-bit version is exactly the same except 5 more inputs are added, all pulled to ground.

```
*
* MODELS
*

.include ../../models/model_t36s.sp
.include ../part_5/fun_3.spice

*
* CIRCUIT
*

VDD vdd gnd 5V

*
* SIMULATION
*

va0  A0     gnd PWL(0n 5V 4.9n 5V 5n 0V 19.9n 0V 20n 5v)
va1  A1     gnd PWL(0n 0V)
va2  A2     gnd PWL(0n 0V)

vb0  B0     gnd PWL(0n 5V)
vb1  B1     gnd PWL(0n 0V)
vb2  B2     gnd PWL(0n 0V)

.option post
.tran 0.01n 25n
.end
```

Listing 18: 3-Bit FUN Generator Spice Simulation

The output from both simulations is shown below

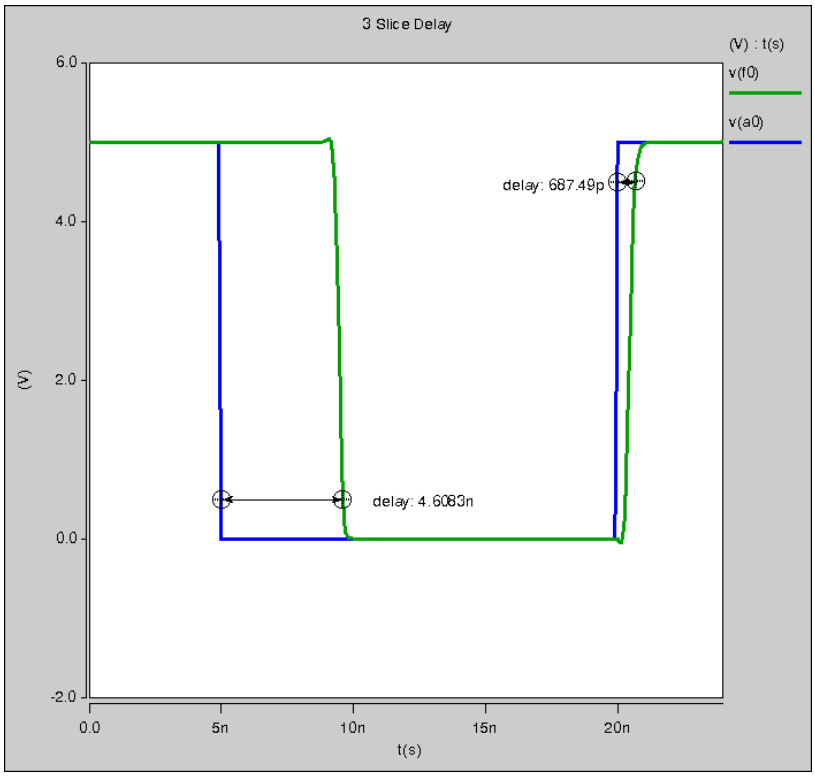

Figure 30: 3-Bit Spice Waveform

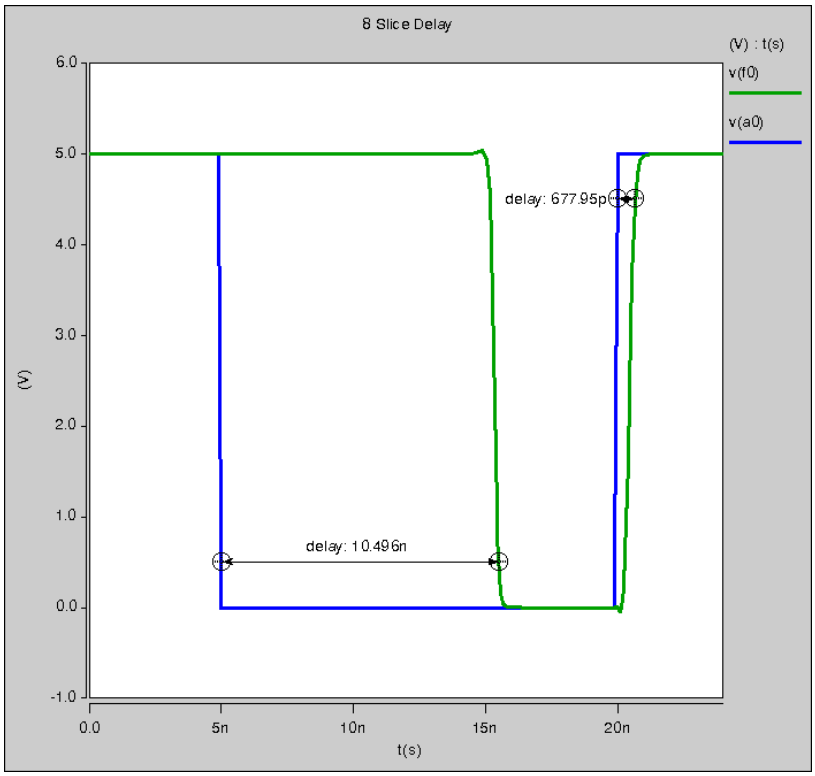

Figure 31: 8-Bit Spice Waveform

From the results we can see that our designs do seem to work correctly. Looking at a final comparison of all the delays between VHDL, IRSIM, and Spice we can clearly see that the VHDL simulation and Spice simulation are in agreement. The IRSIM simulation results are unfortunately not in line with the other two simulations. This leads us to believe that our method of obtaining the worst case delay with IRSIM is incorrect.

Configuration	VHDL	IRSIM	Spice
3-Bit	3.3946ns	0.0030ns	4.6083ns
8-Bit	7.3321ns	0.0030ns	10.496ns

Table 6: Worst Case Delay Comparison Between VHDL, IRSIM, and Spice

There are a few things we can take with us into future design processes. The first of which is spending some time upfront to figure out why IRSIM does not seem to report the correct delays. At the moment it seems that the easiest method to obtain an accurate simulation of our design is to export the Spice netlist from Magic of either an individual gate or a functional block such as a slice. The spice simulation can be simple as we just want to know the worst case delay. Once we obtain the delay we can add it to our VHDL model. Using the VHDL model provides much more flexibility in terms of functional testing as we can totally automate the testbench as we did in this lab. Hopefully, with this insight we will be able to accomplish future design tasks in a more efficient manner.

Task Breakdown

Task	Person
Part 1	Qi
Part 2	Thrun
Part 3	Both
Part 4	Qi
Part 5	Thrun
Part 6	Qi (with Thrun input on conclusion)

Table 7: Task Assignment