Architectural level design of an FPGA

PROJECT REPORT
submitted towards the partial fulfillment of the
requirements for the award of the degree
of

BACHELOR OF TECHNOLOGY

in

ELECTRONICS AND COMMUNICATION ENGINEERING

Submitted by

SUMIT KUMAR YADAV
15117068

Under the guidance of:

Dr. ANAND BULUSU

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE – 247667 (INDIA)
May, 2019

STUDENT'S DECLARATION

I declare that the work presented in this report with title "**Architectural level design of an FPGA**" towards the fulfillment of the requirement for the award of the degree of **Bachelor of Technology** in **Electronics & Communication Engineering** submitted in the **Department of Electronics & Communication Engineering**, **Indian Institute of Technology, Roorkee**, India is an authentic record of my own work carried out during the period **from August 2018 to May 2019** under the supervision of **Dr. Anand Bulusu**, Associate Professor, Dept. of ECE, IIT Roorkee. The content of this report has not been submitted by me for the award of any other degree of this or any other institute.

DATE : ..... SIGNED: .....

PLACE: ..... (SUMIT KUMAR YADAV)
ENROL. NO.: 15117068

SUPERVISOR'S CERTIFICATE

This is to certify that the statement made by the candidate is correct to the best of my knowledge and belief.

DATE : .....

SIGNED: .....

(DR. ANAND BULUSU)
ASSOCIATE PROFESSOR
DEPT. OF ECE, IIT ROORKEE

ACKNOWLEDGEMENTS

I sincerely thank my B.Tech. Project advisor **Dr. Anand Bulusu** for his constant guidance and support throughout the project.

I would like to acknowledge the contribution of **Mr. Jaskirat Singh** & **Mr. Sanjay Yadav**, B. Tech students in ECE, IIT Roorkee for helping out with the necessary layouts and the ideation of the architecture. I am sincerely thankful to **Mr. Inder Choudhary**, research scholar in Microelectronics & VLSI, Department of ECE, IIT Roorkee for his ideas and motivation. I also thank all the research scholars and project staff of SMDP Lab, Department of ECE for helpful technical discussions at every stage of the project. I thank **Department of Electronics and Communication Engineering, IIT Roorkee** for providing the lab and other resources required for carrying out my project work.

Overall, working on this project has been a knowledgeable experience and provided me with a great experience in the area of FPGAs & computer architecture.

ABSTRACT

Minimal FPGAs have huge promise for implementing small interfacing and other logic designs at cheap costs. The minimal FPGAs available in the market are still too big for certain applications and thus we try to solve this problem. We developed the flow for designing a minimal Island-Style FPGA using VTR[3] experimentation and Cadence Schematic level simulations. We first develop models for area and delay of a CLB and choose suitable a LUT size. After that, we experiment with different switch matrix configurations and routing widths to settle for the optimum. The I/O block for the FPGA is then designed in Cadence Schematic L and tested for functionality. SKILL language is used to automatically generate decoder schematics. Smaller modules are used to create the top level schematic of the FPGA. A flow is created to program the bitstream using stimulus files and some scripts. The FPGA is tested for functionality and the delay path is then analyzed. The breakdown of the delay is provided for reference. All the simulations and tests were performed in Cadence ADE L using spectre at room temperature. SCL 180nm PDK was used in tt18 configuration with minimal sizing unless explicitly stated. The thesis ends by concluding the results and discussing future directions.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE **Page**

CHAPTER 1

INTRODUCTION

1.1 Motivation

With advancements in digital VLSI, chips which once could never be imagined can be fabricated. Making a digital ASIC involves many rigorous steps with feedback among different steps. Therefore, it is only human to expect several iterations of architecture, circuit and layout design. Even though ASICS produced in bulk are quite cheap, test ASICs that need to be fabricated while prototyping are very costly. FPGAs solve this issue by providing a fabric to test any digital circuit for functionality.

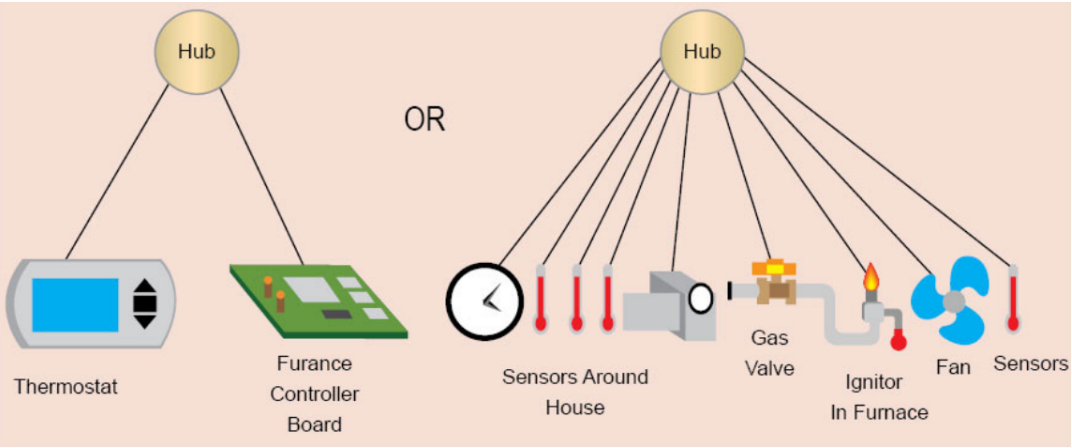


Figure 1.1: Small FPGA hubs in smart-home solutions

FPGAs work on the simple principles of boolean algebra and the use of flip-flops to

implement state machines. Apart from facilitating cheap prototyping, FPGAs can also be used by itself in a product eliminating the need to incur the cost of ASIC design and fabrication at the cost of some granularity and overheads. This is a particularly attractive option for modest digital circuits or when you don't have enough input capital. ASICs have been shown to be two or three orders of magnitude better than general purpose processors for some applications. FPGAs can come in handy to reap a big part of these benefits with minimal input capital. Therefore, FPGAs are also popular as accelerators these days. In many digital systems, a very minimal FPGA is required to implement a modest logic like interfacing etc (see Fig. 1.1). But even the smallest micro-controllers or FPGAs in the market (see Fig. 1.2) are an overkill for such applications. Also, not many FPGA projects are completely open-source.

	LP Series (Low Power)				HX Series (High Performance)		
Feature	LP384	LP1K	LP4K	LP8K	HX1K	HX4K	HX8K
Look Up Tables (LUTs)	384	1280	3520	7680	1280	3520	7680
Embedded RAM Bits	0	64K	80K	128K	64K	80K	128K
Phase-Locked Loops	0	1	2	2	1	2	2
Typical Core Current (µA)	21	100	360	360	267	667	1100

Figure 1.2: Lattice's family of miniature FPGAs[1]

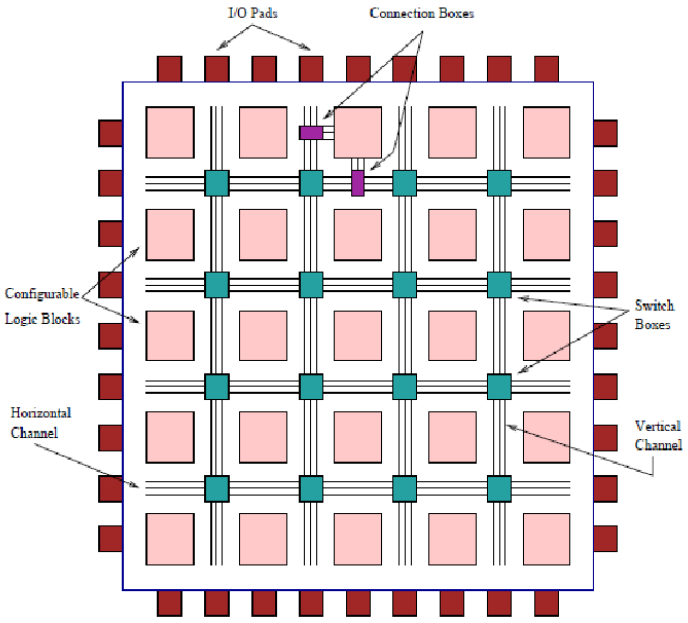


Figure 1.3: Island-Style FPGA architecture

Similarly, small and standard digital ASICs have overkill specifications for most use cases and thus a tiny FPGAs can emulate these ASICs quite comfortably. Island-style FPGAs (see Fig. 1.3) are the most common with small LUTs (Look Up Tables) interconnected together with a mesh of programmable routing.

We wish to develop a small island-style FPGA starting from scratch and my work tackles with the architectural-level design of the same. We first theoretically study the island-style FPGA in the light of SCL-180nm and derive useful results for the architecture of the FPGA. Furthermore, we have developed and tested (at schematic level) an 8X8 LUT island-style FPGA in SCL-180nm technology node.

1.2 Problem Definition

The aim of this thesis is to architect a minimal Island-Style FPGA in SCL-180 nm technology node and test its functionality at schematic level using Cadence Virtuoso. The steps taken in this regard are as follows:

- Understanding working of Island-Style FPGAs
- Deciding on the CLB
- Deciding on the Switch Matrix and I/O Block
- Deciding on the size of CLB Grid
- Designing the programmability of the FPGA
- Designing the address decoders
- Designing the schematic in Virtuoso
- Testing the circuit for functionality in virtuoso

1.3 Contributions

The objective of this thesis is to architect an Island-Style FPGA through various smaller steps as explained above. In designing this architecture and schematic, various

tools/scripts have been prepared for automate schematic generation, stimulus generation, pipeline to program the FPGA schematic with a bitstream in spectre, etc. These contributions will be discussed in subsequesnt chapters.

1.4 Thesis Organization

- **Chapter 1** gives an introduction to Island-Style FPGAs and the need for developing an open-source flow for minimal FPGA design.
- **Chapter 2** discusses the previous work done in this area
- **Chapter 3** discusses the design of CLB
- **Chapter 4** discusses the design of Switch Matrix
- **Chapter 5** discusses the design of I/O Block
- **Chapter 6** discusses the programmability of the FPGA
- **Chapter 7** discusses the complete system level design of an 8x8 FPGA along with simulations
- **Chapter 8** Conclusion

CHAPTER 2

LITERATURE REVIEW

2.1 Previous Work

Jonathan Rose[2] has done some work on architecting FPGAs. He first develops concepts of island-style FPGAs and then demonstrates different routing methods. He further provides a software tool to automate the layout generation of an FPGA by iteratively placing tiles of a CLB and Switch Matrix. This tool isnot compatible with modern PDKs and the tool isn't even available publically. Only a final netlist of their FPGA is provided which is not easy to scale or modify. [3] analyzes various parameters of an island-style FPGA architecture from theoretical derivations of area and delay but the models are approximate. The various area models used in the derivations are shown in Fig. 2.1. Equations 1.1 to 1.4 give approximations of total area required to layout an FPGA using these approximations.

$$A_{Tile} = A_L + A_R \tag{2.1}$$

$$A_L^K = A_b \cdot 2^K + A_f \tag{2.2}$$

$$A_R^K = A_b \cdot W_K^2 + 2 \cdot \sqrt{A_b} \cdot \sqrt{A_L^K} \cdot W_K \tag{2.3}$$

$$A_{Total}^K = A_{Tile}^K \cdot N_K = (\sqrt{A_b} \cdot W_K + \sqrt{A_L^K})^2 \cdot N_K \tag{2.4}$$

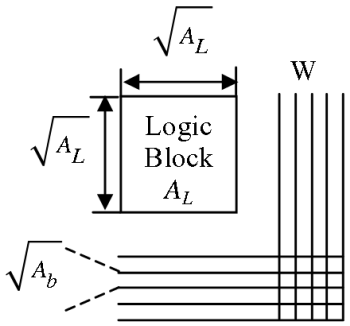


Figure 2.1: Area model used in [3]

A_{Tile} : Area of a single tile A_L : Area of Logic Block per tile A_R : Area of Routing per tile A_b : Area of an SRAM cell K : Number of Inputs to one CLB N_K : Number of tile required with K-size CLBs

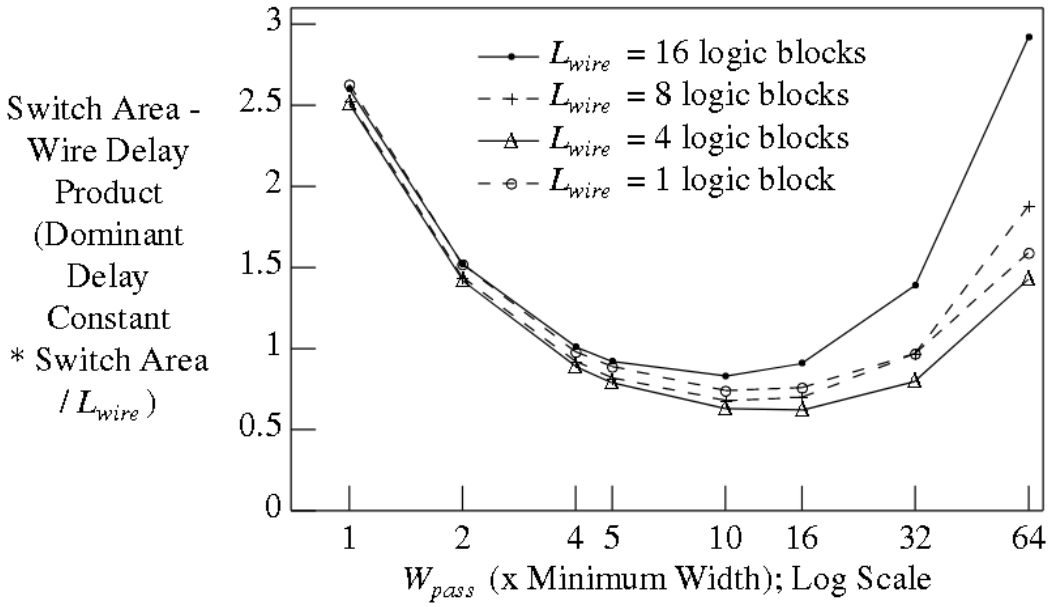


Figure 2.2: Sizing pass transistors of the switch matrices[5]

The FPGA research community has developed the VTR Project[4]. VTR(Verilog to Routing) is a tool chain which facilitates FPGA architecture research. The tool needs a description of the FPGA architecture in a special pre-defined XML format along with the Verilog file which is to be implemented on the FPGA. The tool will first determine the grid size and routing width needed to implement the circuit and then perform coarse and fine grained routing to produce a netlist. This tool is not very well supported in the schematic level circuit simulation softwares as well as layout tools.

[5] proposes methodologies to size interconnect circuits as shown in Fig. 2.2 but does not consider the effect of CLB circuit on the sizing by assuming a buffer in between which limits it to a first hand approximation. We use these suggested values as a starting point in the project.

CHAPTER 3

DESIGNING THE CLB

3.1 Overview

This chapter provides the architecture design process of a CLB. The area estimation models of CLB are developed in 3.2. The area estimation models of CLB are developed in 3.3. Section 3.4 develops the complete schematic of the CLB. Section 3.5 describe the simulation environment while section 3.6 summarizes the chapter.

3.2 CLB architecture: area estimation

The architecture wireframe of the CLB used is shown in Fig 3.1.

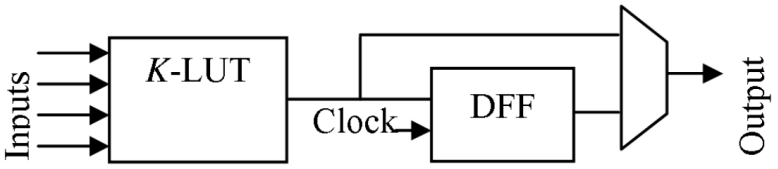


Figure 3.1: CLB Architecture Wireframe

Now, we have to determine the optimum LUT size to be kept in each CLB to balance area and delay requirements for our technology. If the LUT size(K) is too small, a large

number of CLBs will be needed to implement a given design which will incur a lot of routing and thus might use more area and produce a lot of delay. If K is too large, the granularity of logic will be coarse and thus even small boolean functions will take up a whole CLB. The routing area and number of CLBs will be less but the area of one CLB will be more. Thus, we need to strike a good balance for K to have reasonable granularity, area and delays. For this, we develop a model of CLB area similar to [3]. Fig 3.2 describes a tile which we use iteratively to produce the whole FPGA.

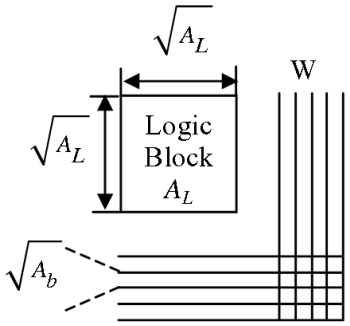


Figure 3.2: Area model

The area of this tile can be expressed as the sum of the CLB area(or Logic area) and the area of the routing tracks as expressed below.

$$A_{Tile}^K = A_L^K + A_R^K \tag{3.1}$$

Now, the logic area can be expressed as the sum of (i) 2^K bits to store the K-input truth table, (ii) area of one Flip-Flop, (iii) area of input crossbar, (iv) area of output crossbar and the area of configuration bits. We note that the configuration bits will be of the order of O($K.\log_2 K$+$\log_2 W$) which is negligible to O(2^K) of truth table bits and thus can be ignored in an approximation. The input and output crossbars can be assumed of the same height as that of the LUT. Thus, we get the following approximate estimate of CLB area:

$$A_L^K = A_{bit} \cdot 2^K + A_{FF} + \sqrt{A_{bit}} \cdot (\sqrt{A_{bit} \cdot 2^K} \cdot \log_2 K + \sqrt{A_{bit} \cdot 2^K} \cdot \log_2 W_K) \tag{3.2}$$

Now, the routing area can be expressed as the sum of 2 rectangles and a square as follows:

$$A_R^K = A_{bit} \cdot W_K^2 + 2 \cdot \sqrt{A_{bit}} \cdot W_K \cdot \sqrt{A_L^K} \tag{3.3}$$

Adding logic and routing areas, we get the following area per tile:

$$A_{Tile} = (\sqrt{A_{bit}} \cdot W_K + \sqrt{A_L^K})^2 + A_{bit} \cdot 2^{K/2} \cdot \log_2 (K W_K) \tag{3.4}$$

To find the area of the whole FPGA, we need to multiply the tile area with the number of CLBs needed in total. For that, we need an estimate of the number of CLBs required to implement a given design. The Rent-rule[6] proposes an empirical relationship b/w No. of input pins(T) and No. of logic blocks(N) given below:

$$T = \lambda \cdot N^p \tag{3.5}$$

where λ is a proportionality constant and p is the Rent-exponent shown to be around 0.75 for FPGA devices in [7]. [3] uses the following two equations to approximate W_k and N_k with N_2 to be 10^4 for K=2:

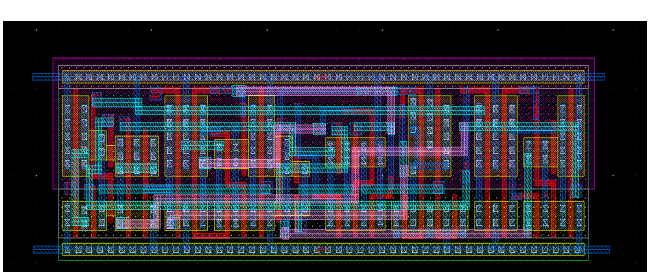
$$\frac{N_2}{N_k} = (\frac{K+1}{3})^{\frac{1}{p}} \tag{3.6}$$

$$W_K = \frac{K+1}{2} \cdot \frac{2}{3} \cdot \frac{1-p}{p-0.5} \cdot ((N_k)^{p-0.5} - 1) \tag{3.7}$$

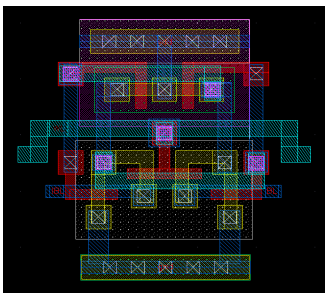
Now, the total area of the FPGA becomes N_k time the area of one tile plus some extra routing area to account for two edges of the formed FPGA:

$$A_{Total} = A_{Tile} \cdot N_k + 2 \cdot \sqrt{N_k} \cdot (A_{bit} \cdot W_K^2 + \sqrt{A_{bit} \cdot A_L^K} \cdot W_K) \tag{3.8}$$

We use the layouts shown in Fig. 3.3 for area estimates of a Flip-Flop(168.27 um sq.) and SRAM cell(16 um sq.).



(a) Layout of DFF



(b) Layout of SRAM cell

Figure 3.3: Layouts for area estimates

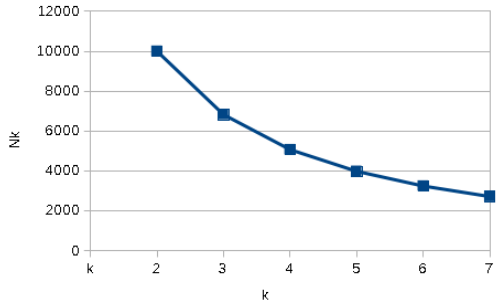
Figure 3.4 presents areas corresponding to different values of k.

Figure 3.5 explains the tradeoff between total area and k.

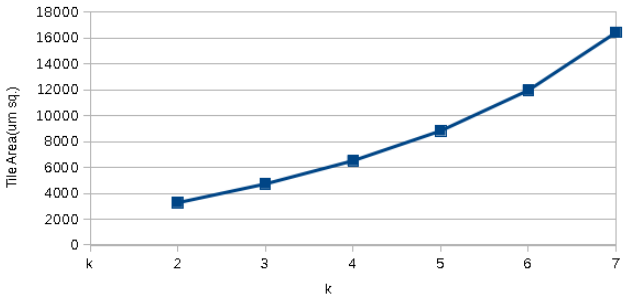
We find that K=3 gives the least area estimates for implementing a given logic circuit on our FPGA. But we also observe that K=4 is also not bad and moreover is a power of 2 which makes it very attractive to divide large circuits into smaller circuits.

k (LUT size)	Nk (No. of CLBs required)	Wk (Routing Width)	Logic Area per tile	Routing Area per tile	Tile Area	Total Area	Normalized Total Area
2.0	10000.0	9.0	448.4	2820.2	3268.7	33098182.5	0.987
3.0	6814.9	10.8	654.2	4064.9	4719.1	32649285.7	0.973
4.0	5061.5	12.4	970.8	5544.0	6514.7	33543206.2	1.000
5.0	3969.4	13.9	1480.3	7349.1	8829.4	35704611.1	1.064
6.0	3232.1	15.3	2337.4	9625.6	11963.0	39424897.4	1.175
7.0	2705.1	16.6	3836.8	12595.4	16432.2	45334026.2	1.352

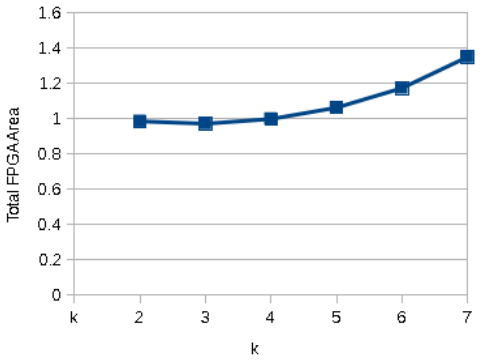
Figure 3.4: Area estimates in um^2 for different values of k



(a) Nk vs k



(b) Tile Area vs k



(c) Total Area vs k

Figure 3.5: Tradeoff in area for different k values

3.3 CLB architecture: delay estimation

Now, we move on to study the delay of implementing a given logic circuit on an FPGA of K-size CLBs. We first use the following equation derived in [3] to estimate the number of K-input LUTs required to implement an N-input design:

$$M_K = 2^{N-K} \cdot (1 + \frac{1}{K} + \frac{1}{K^2} + ...) \approx 2^{N-K} \cdot \frac{1}{1 - 1/K} \tag{3.9}$$

And the depth of tree to implement N-input logic using K-input LUTs as derived in [3] is:

$$D_K = \frac{N-K}{x(K)} + 1 \tag{3.10}$$

where x(K) is the solution of:

$$K = x + 2^{x-1} \tag{3.11}$$

After dividing by technology dependent scaling factors, the dependence of t_K (delay of implementing a given circuit on K-input LUTs) on k is shown in the Fig. 3.6. Here, $\mu = C_{R3}/C_{L3}$ increases with better technology nodes as routing capacitance starts becoming dominant compared to logic capacitance.

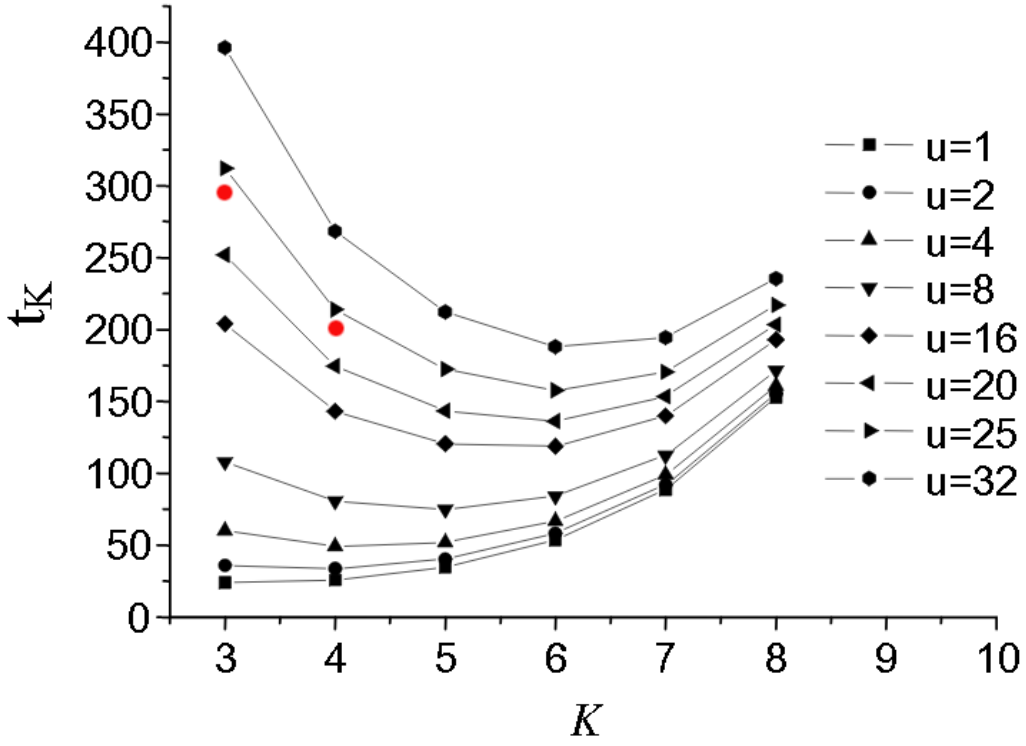


Figure 3.6: t_K vs k for different values of μ

For SCL 180nm technology node, we find a rough estimate of μ using schematic level captab analysis in Cadence ADE L to be μ = 20.5/8.25 = 2.5. As observed from the Fig. 3.6, delay in case of K=3 is roughly 1.3 times the delay for K=4. This important observation dictates to select K=4 for optimal design in an Area-Delay Product sense.

3.4 CLB schematic

The following schematic instances are used in the schematic of a CLB:

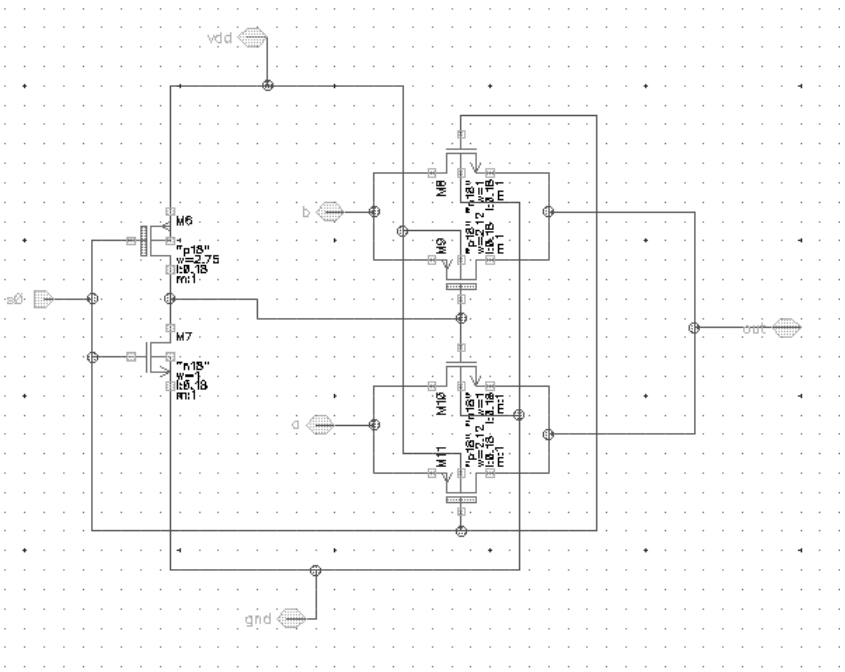


Figure 3.7: 2-to-1 MUX Schematic

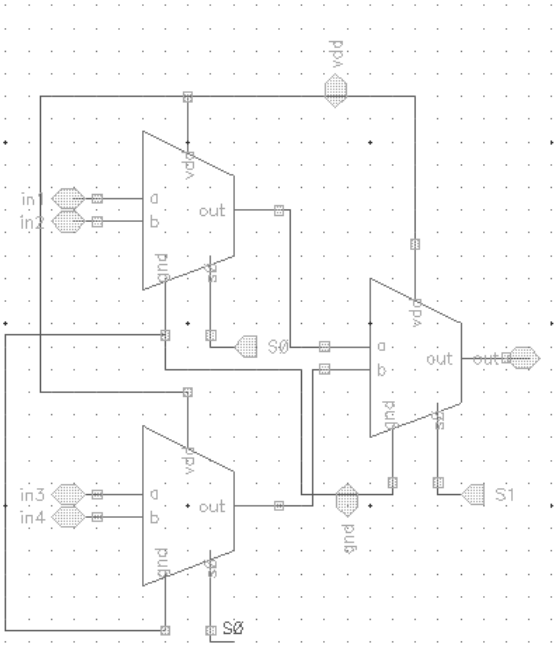


Figure 3.8: 4-to-1 MUX Schematic

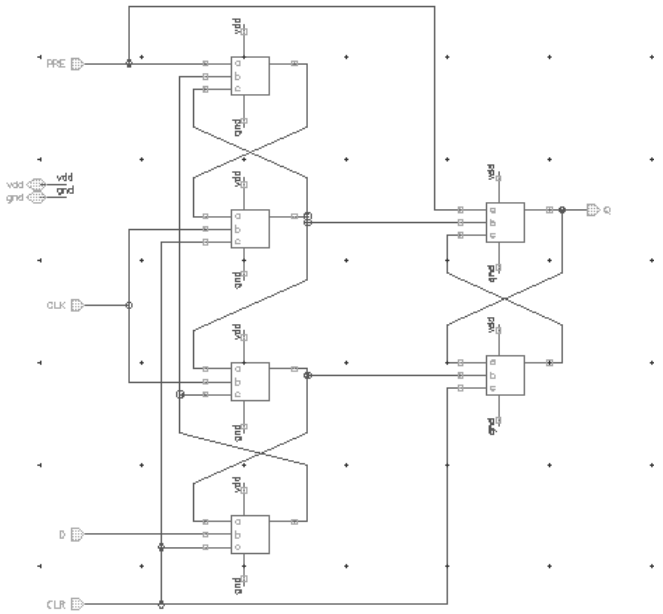


Figure 3.9: D Flip-Flop Schematic

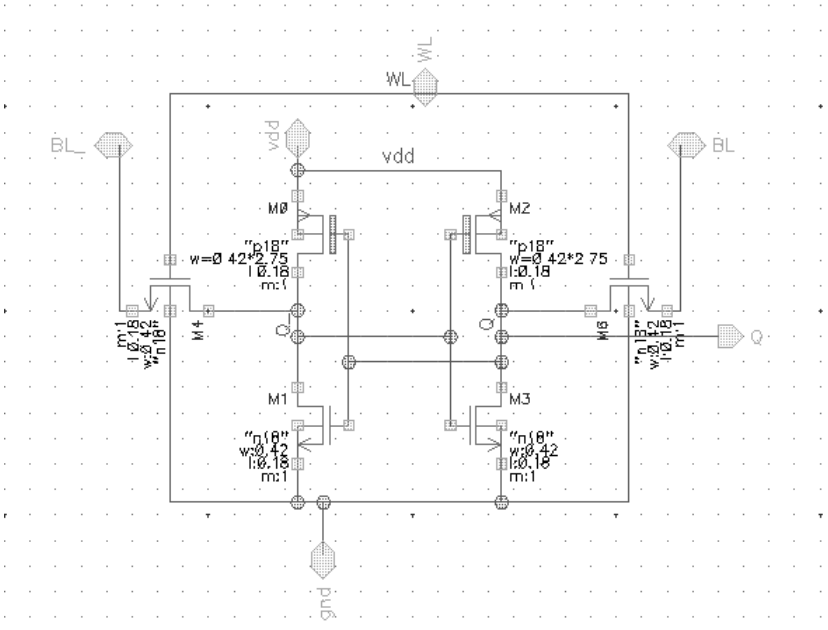


Figure 3.10: SRAM cell Schematic

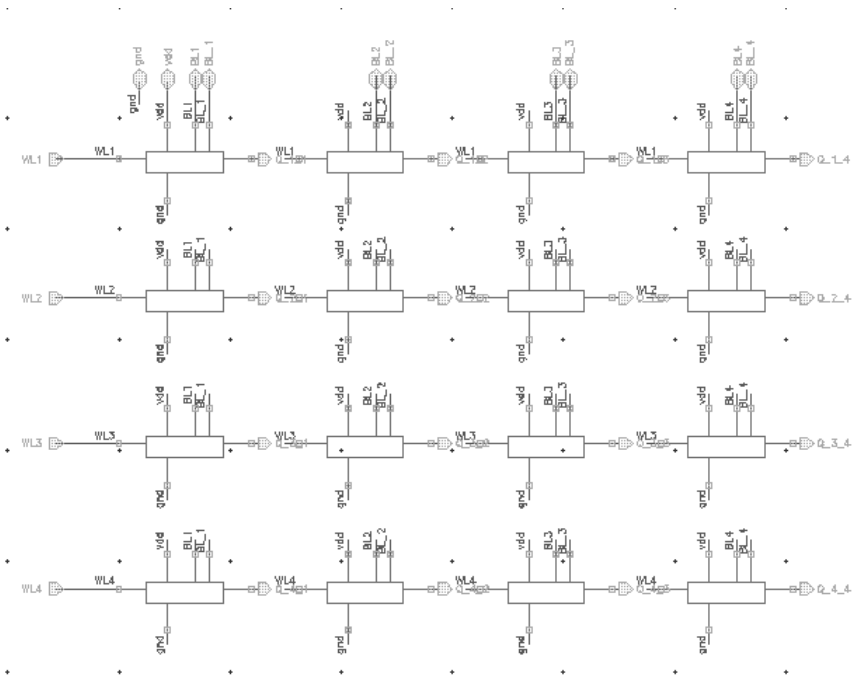


Figure 3.11: 4 X 4 SRAM array Schematic

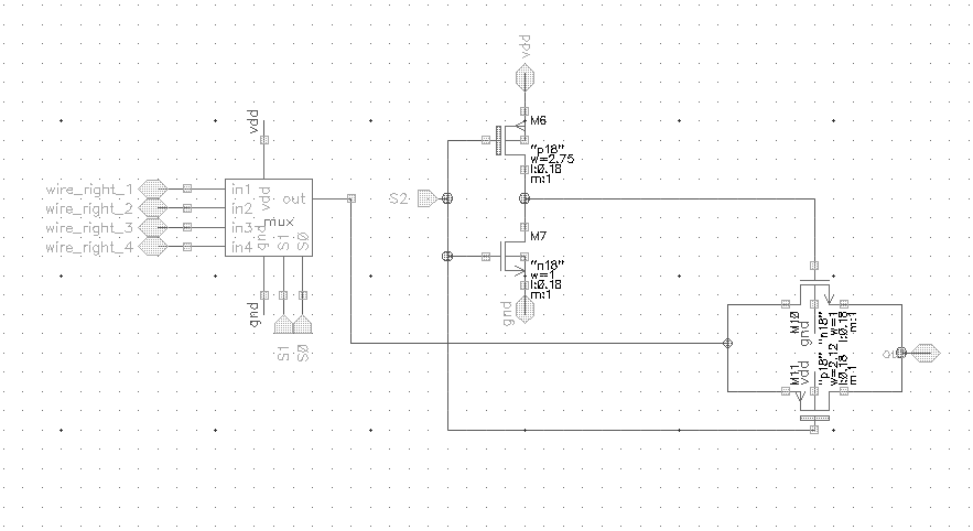


Figure 3.12: 4 X 1 Crossbar Schematic

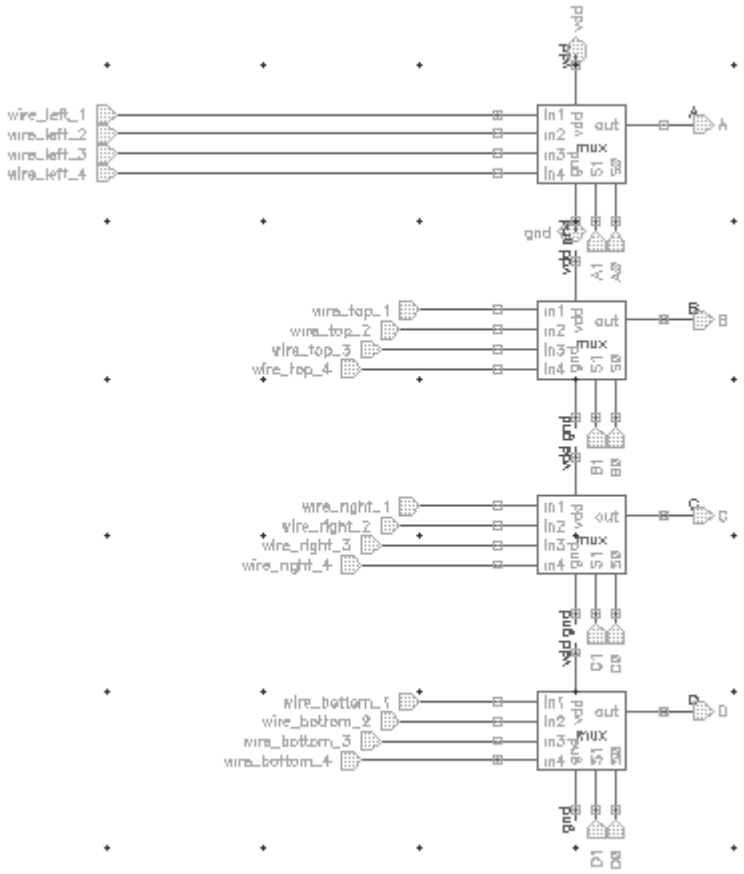


Figure 3.13: 4 X 4 Crossbar Schematic

The components used in the schematic of CLB are described below:

- **4X4 LUT :** A 16 bit LUT arranged in a 4X4 6T-SRAM array to hold the truth table of a 4-input boolean function to be implemented. The Q-outputs of all the 16 SRAM cells are exposed.
- **4 X 1 Decoders :** A MUX tree of 5 4X1 decoder MUXes is formed to decode the 16 inputs to one output using 4 select lines.
- **4X4 input crossbar :** A full crossbar to connect the 4 select lines to routing wires on left, right , top and bottom.
- **4X1 output crossbar :** A full crossbar to connect the CLB output to one of the 4 routing wires on the CLB's right side.

- **2X1 DFF MUX :** A mux to select whether to feed calculated boolean result in the DFF or retain its previous state.

- **2X1 OUT MUX :** A mux to select whether to send the DFF value or the calculated boolean result out from the CLB.

- **Inverter :** To invert the FF-initial-state

- **NAND gates :** 2 NAND gates are used to form PRE and CLR signals for the DFF when config is HIGH

- **D Flip-Flop :** Flip Flop provides 1-bit storage per CLB which can either be used to store current result of the LUT or to send out a previously stored value for further use in other parts of the FPGA.

- **4X4 Configuration Memory :** A 16-bit SRAM arranged in a 4X4 6T-SRAM array to hold the configuration bits of the whole CLB.

  - 1-bit for FF-initial-state
  - 2-bits for the two MUXes
  - 8-bits for the 4X4 input crossbar
  - 3-bits for the 4X1 output crossbar
  - 2-bits left unused for future feature updates

- **Bitlines :** 8 pairs of BL and $\overline{BL}$ go into each CLB for programming the two 4 X 4 arrays of SRAMs. The cells in the same column in an array share the same pair of bitlines.

- **Wordlines :** 4 WL go into each CLB for programming the two 4 X 4 arrays of SRAMs. The row 1 of both 4 X 4 grids share the same WL and similar follows for other three rows.

Figure 3.14 shows the schematic of the CLB designed in Schematic L.

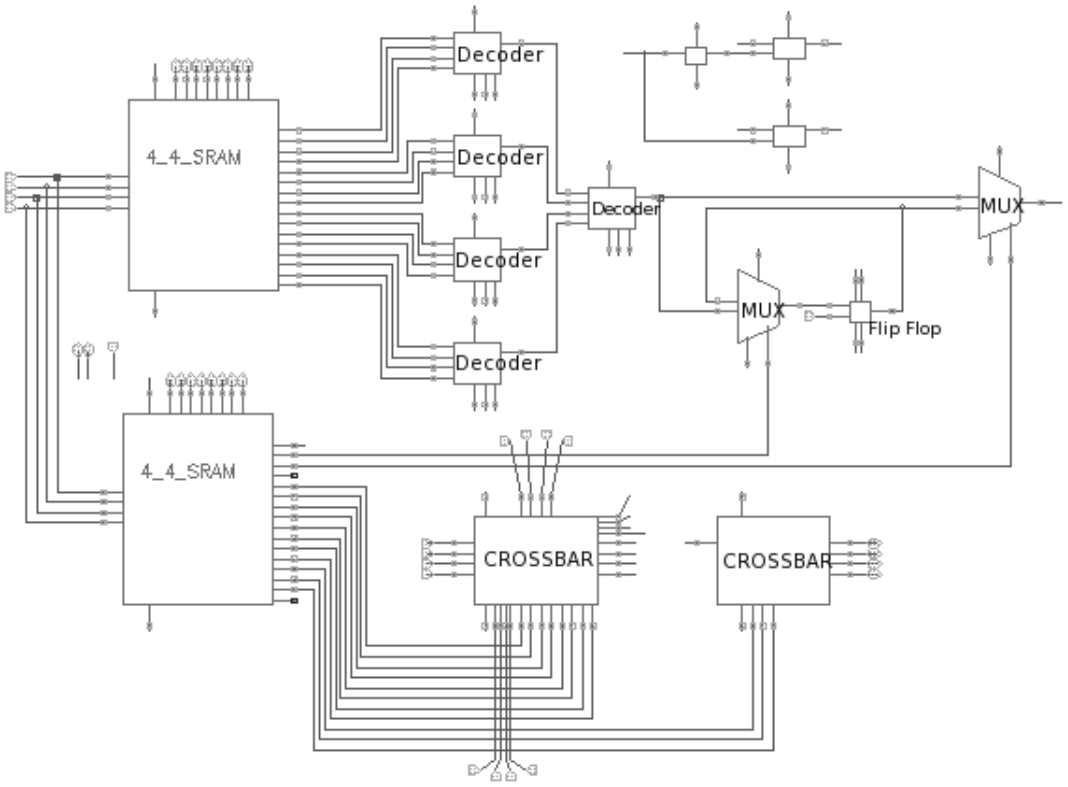


Figure 3.14: CLB Schematic

3.5 Simulation Setup

We are using Cadence Schematic L for designing the schematic and spectre is used in Cadence ADE L to test the designs. SCL 180nm PDK has been used at tt_18 configuration throughout the design. PMOS-to-NMOS ratio of 2.75 has been used at most places to get symmetric rise and fall characteristics with minimal sizing unless explicitly specified. This CLB is tested in ADE L using spectre for correct functionality.

3.6 Summary

This chapter presented the design of CLB along with the various derivations and modeling of area and delay to decide on an optimum LUT size in area-delay-product sense. We also decided on the total bits required for LUT and configuration.

CHAPTER 4

DESIGNING THE SWITCH MATRIX

4.1 Overview

In the previous chapter, the CLB was designed based on theoretical derivations regarding area and delay. This chapter will focus on architecting and designng the Switch Matrix that will provide routability on the FPGA fabric.

4.2 Switch Matrix type

A lot of switch matrices have been proposed[8] in the academia till now including the universal switch matrix, wilton switch matrix[9] and disjoint switch matrix.

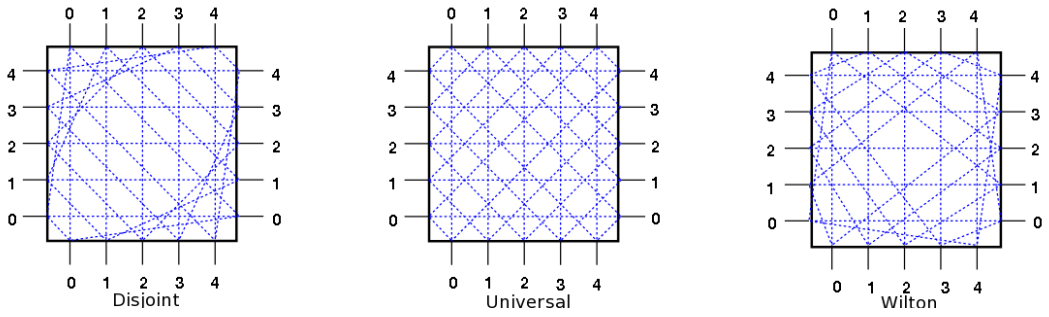


Figure 4.1: Different types of switch matrices

- **Universal :** Universal switch matrix proposed by Chang et.al in [CWW96a, CWW96b] uses 6W switches and any two nets can be connected to each other provided that each side has less than W signals. This method is superior in routing performance but lags behind other designs in terms of area and delay.

- **Wilton :** Wilton switch matrix proposed by Wilton[Wil97] solves one key problem of most other SM designs. Most designs try to restrict a signal in a particular track and thus perform poorly with sparse connected designs of memories. This is because memories use pins connected on specific tracks only. Wilton proposed a solution which arbitrated the track of a signal after every SM it encountered. This makes sure that all the tracks are used evenly and thus congestion is relieved.

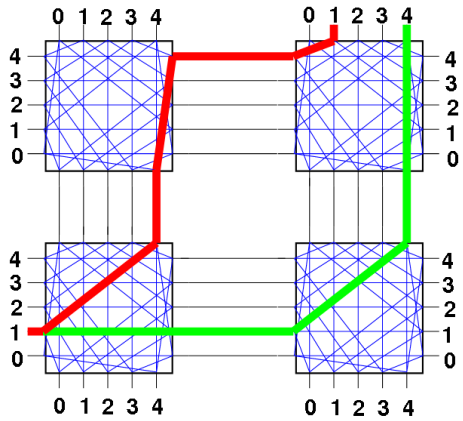


Figure 4.2: Routing possibilities with Wilton design

- **Disjoint :** Disjoint switch matrix was first proposed and used by Xilinx. The design can connect a net to 4 different possibilities, one on each side. This block also suffers from the same problem of using the same track throughout leading to congestion in many areas.

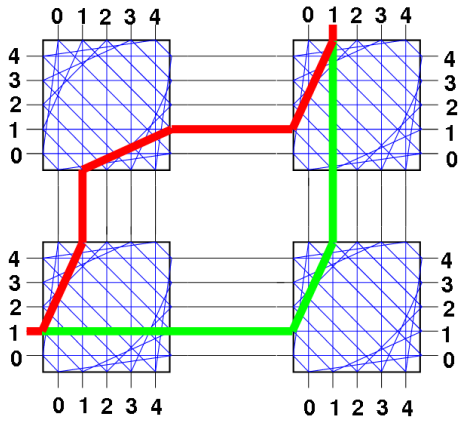


Figure 4.3: Routing possibilities with Disjoint design

Due to its versatility and robustness, we stick with Wilton switch matrix design for our design and analysis purposes.

4.3 Describing the FPGA in VTR

To study the effects of connectivity fractions, orientation of input pins and output pins, routing width etc., we used VTR[4] tool. Verilog to Routing needs the architecture of the FPGA to be described in an XML format with specifications given in the relevant tags. We described our architecture in the prescribed format and kept the grid size as variable depending on the design to be implemented. We settled with MUX based one way tracks in our switch matrices to save routing area as our FPGA is only designed for minimal applications.

4.4 Deciding CLB pinout

The 4 inputs of the CLB can be taken from any of the neighbouring routing wires of the CLB. There are 2 cases to put the pins as discussed below:

- **All inputs of left :** This design (Fig. 4.4) will connect the input crossbar to the 4 wires passing from the left side of the CLB. This method forces all 4 signal to be brought to the left 4 wires before they can be used to perform boolean functions. This takes up all the 4 wires on a CLB's left. So the signal going from top to bottom are hindered and thus have to travel some extra distance horizontally.

This phenomenon accumulates congestion quickly and thus needs larger routing widths(W) to route any given design. This can be seen in the implementation of a 5-bit adder on our FPGA (see Fig. 4.5 and 4.6).

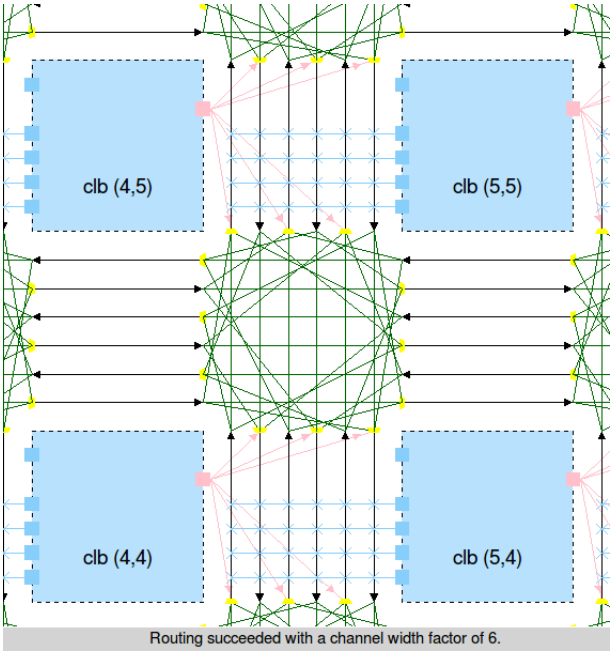


Figure 4.4: CLB pinout: all inputs on left side

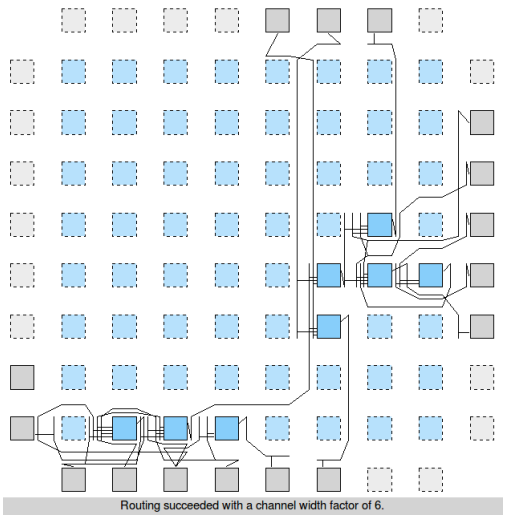


Figure 4.5: Routing of a 5-bit adder needs W=6

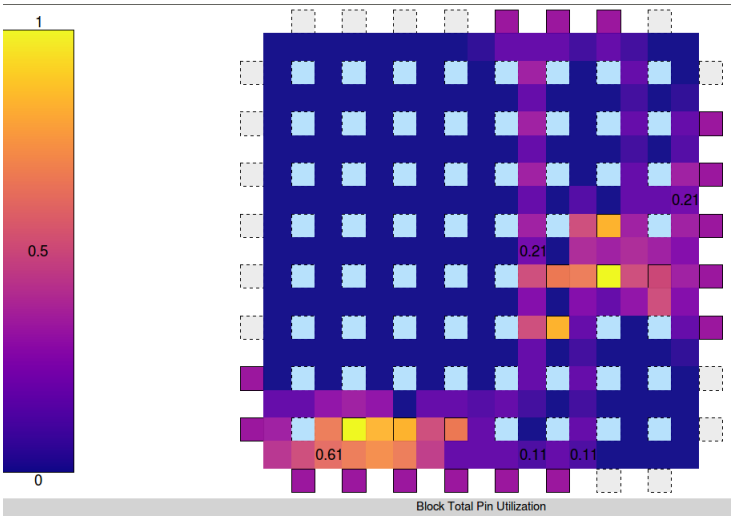


Figure 4.6: Routing congestion : all left IO

- **One input from each side :** In this design (see Fig. 4.7), the input crossbar will select one input from the left 4 wires, 1 input from the right 4 wires, 1 input from the top 4 wires and 1 input from the bottom 4 wires. Thus, all the inputs are not needed to be brough to one side of the CLB. This releives congestion and therefore can route a given logic circuit with a smaller routing width(W). This can be seen in the implementation of a 5-bit adder on our FPGA (see Fig. 4.8 and 4.9).

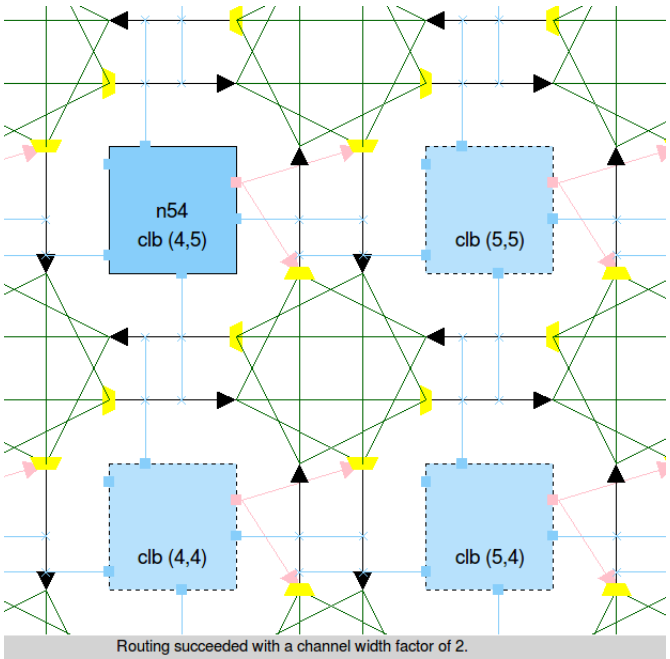


Figure 4.7: CLB pinout: one input from each side

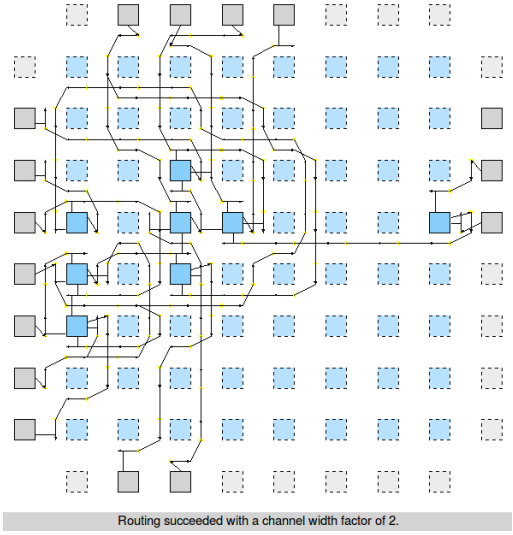


Figure 4.8: Routing of a 5-bit adder needs W=2

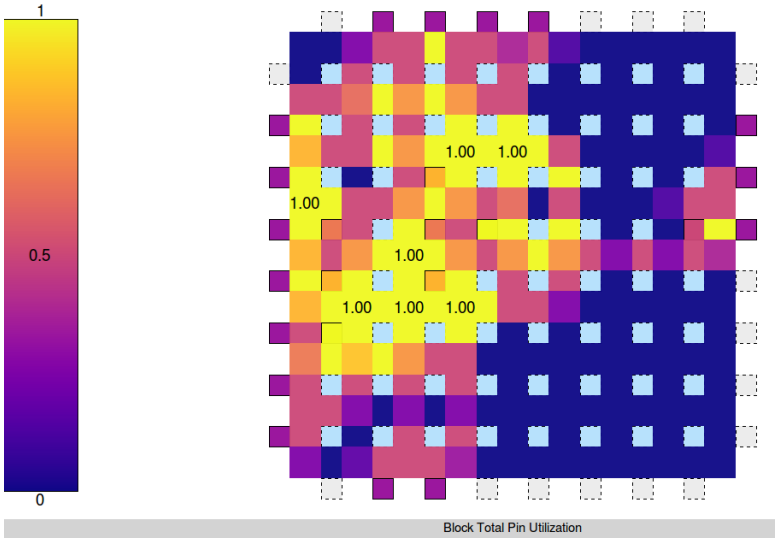


Figure 4.9: Routing congestion: spread IO

This case study leads to the observation that taking one input from each side instead of taking all inputs from one side provides significant reductions in the routing width requirements(a factor of 3x). Thus, the area overheads in making an all sided input crossbar are easily shadowed by the area and delay benefits gained from reduction in routing width(W). Due to this, we use one input from each side for our FPGA design.

4.5 Deciding channel width

To decide upon a channel width, we conduct experiments using vtr with various verilog designs and see how much grid size and routing width is sufficient to route these designs. Two representative verilog codes are detailed below:

- **10-bit adder :** This is a simple 10-bit adder which adds two given inputs to produce the sum. We find that a grid size of 8x8 and channel width of 4 are sufficient to route the design (see Fig. 4.11). We also show the routing congestion for our routed design (see Fig. 4.12).

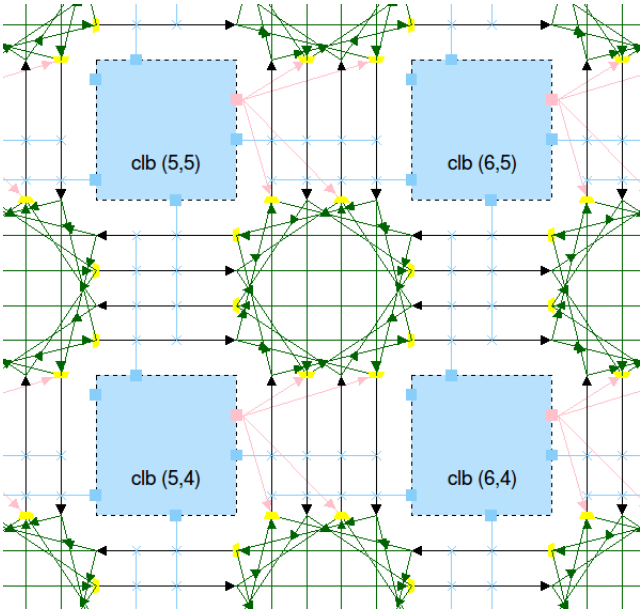


Figure 4.10: CLB pinout and switch matrix closeup

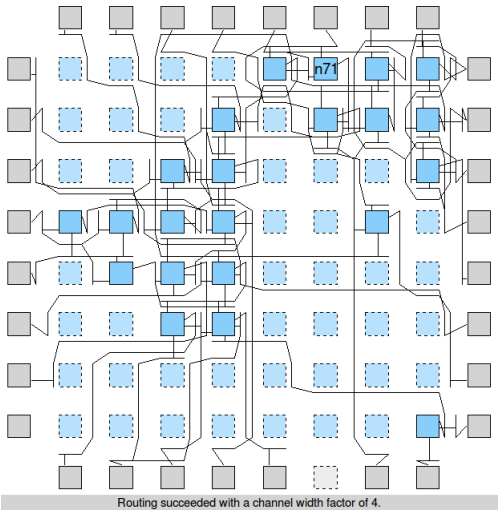


Figure 4.11: Routing of a 10-bit adder needs W=4

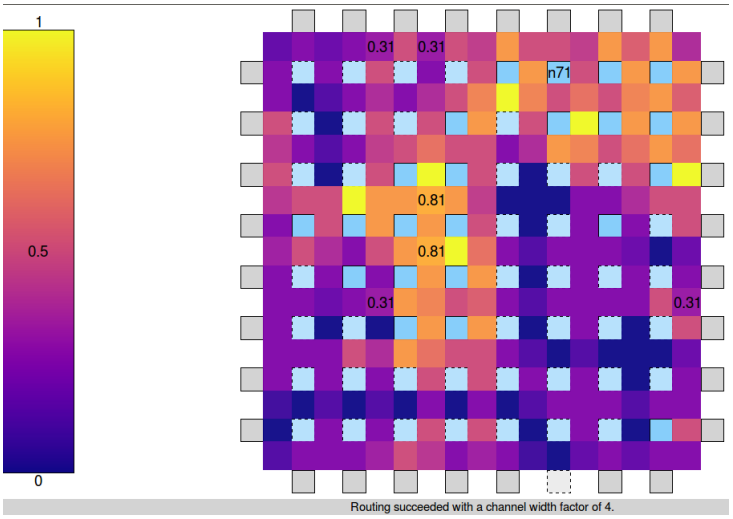


Figure 4.12: Routing congestion

- **A complex memory controller :** We implement the verilog code of a complex memory controller written in Verilog on the FPGA and find out that a grid size of 60x60 is required along with a routing width of 4 (see Fig. 4.13). We also show the routing congestion for our routed design in Fig. 4.14.

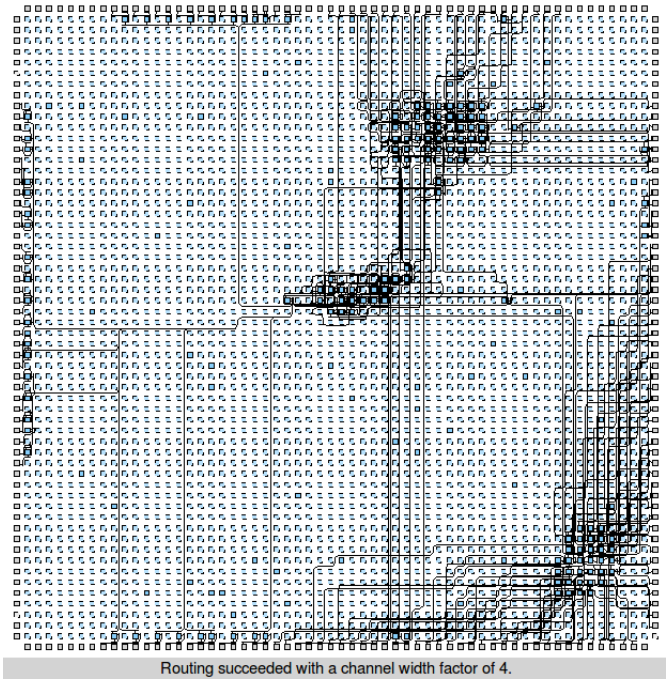


Figure 4.13: Routing the memory controller needs W=4

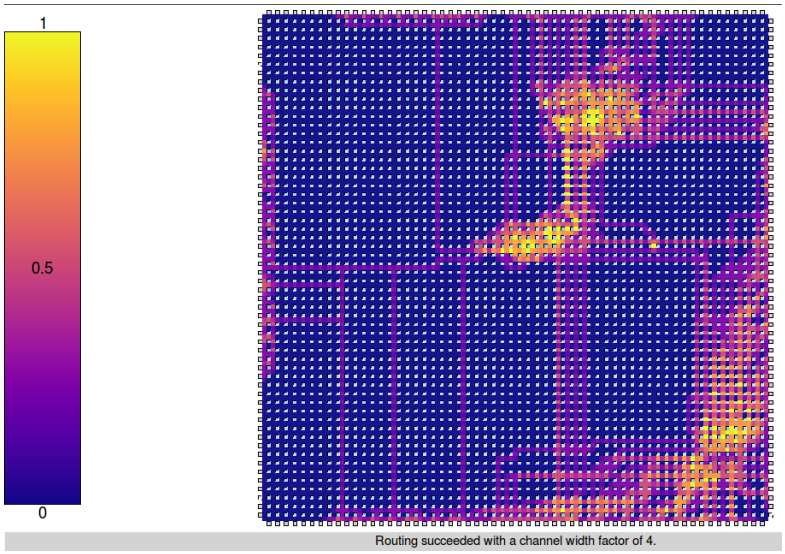


Figure 4.14: Routing congestion

The rest of the experiment results are summarized in Fig. 4.15:

Verilog Design	Grid size		Routing width	
	All left I/O CLB	Spread I/O CLB	All left I/O CLB	Spread I/O CLB
adder.v	6x6	6x6	4	4
multiplier.v	10x10	10x10	8	6
and_latch.v	3x3	3x3	2	2
ch_intrinsics.v	60x60	60x60	6	4
multiclock_separate_and_latch.v	4x4	4x4	4	2
multiclock_output_and_latch.v	4x4	4x4	4	2
multiclock_reader_writer.v	6x6	6x6	6	4

Figure 4.15: Required grid sizes and routing widths for different verilog designs

From this table, we observe that the grid size scales with complex logic circuits but the channel width of 4 is still sufficient for most of the designs we used. This dictates our choice of a channel width factor of 4 for our FPGA design so that we can scale the FPGA if needed without changing the routing.

4.6 Summary

This chapter describes the architecture choices that went into deciding on the structure of the switch matrix. Since we use MUX based switch matrices of width 4, 16 bits are required for configuring each switch matrix. These 16 bits are organized as a 4*4 SRAM array. The switch matrix is simulated for functional correctness in ADE L using spectre. The schematic described in Cadence schematic and the waveforms are presented in Fig. 4.16 and 4.17 respectively.

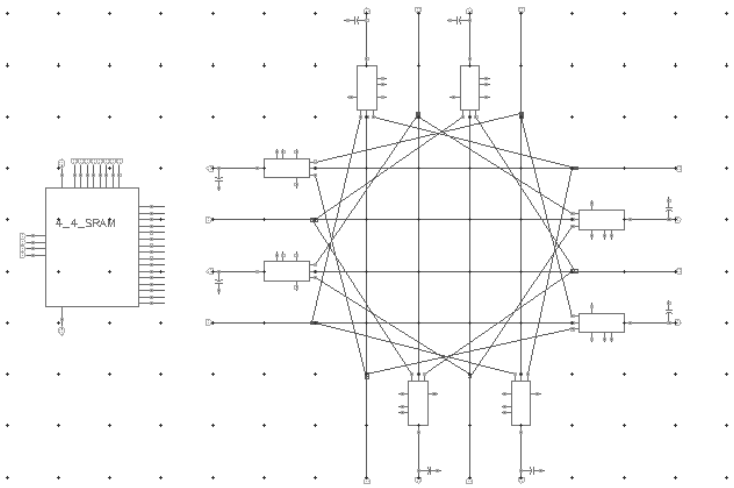


Figure 4.16: Schematic of Switch Matrix

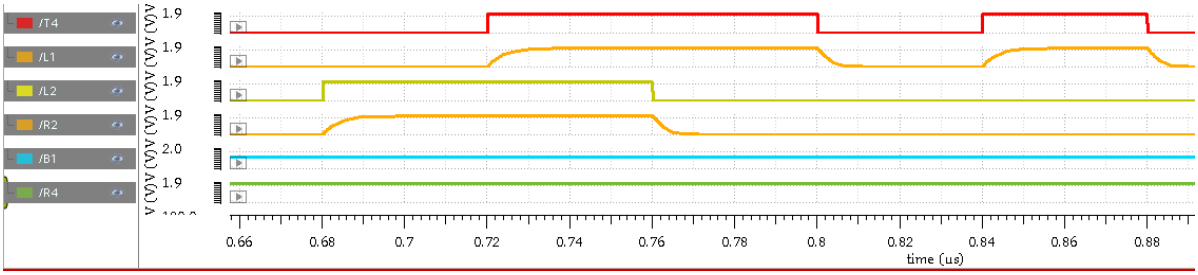


Figure 4.17: Simulation results of Switch Matrix - Center

The final FPGA architecture described in VTR is presented in Fig. 4.18.

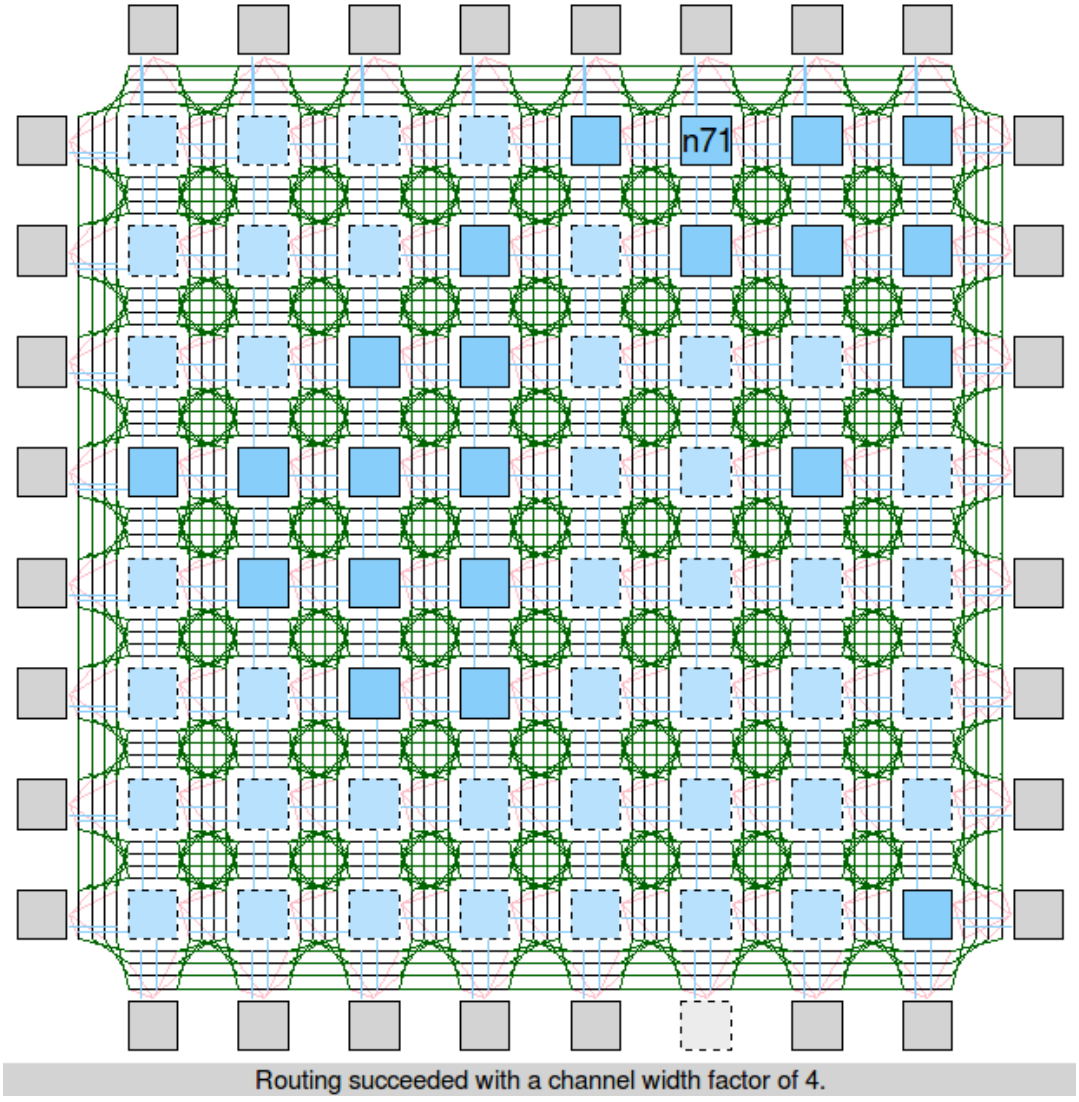


Figure 4.18: 8x8 FPGA structure as seen in VTR

CHAPTER 5

DESIGNING THE I/O BLOCK

5.1 Overview

In the previous chapter, the switch matrix was designed based on experiments run in VTR. This chapter will focus on architecting and designng the I/O Block(Input/Output Block) that will provide I/O to the FPGA fabric.

5.2 I/O Block architecture

A lot of I/O Block designs have been proposed in the academia and industry. We will try a very simple and minimal I/O block inspired from the XC2064 FPGA by Xilinx[10]. The architectural wireframe of this I/O block is shown in Fig. 5.1.

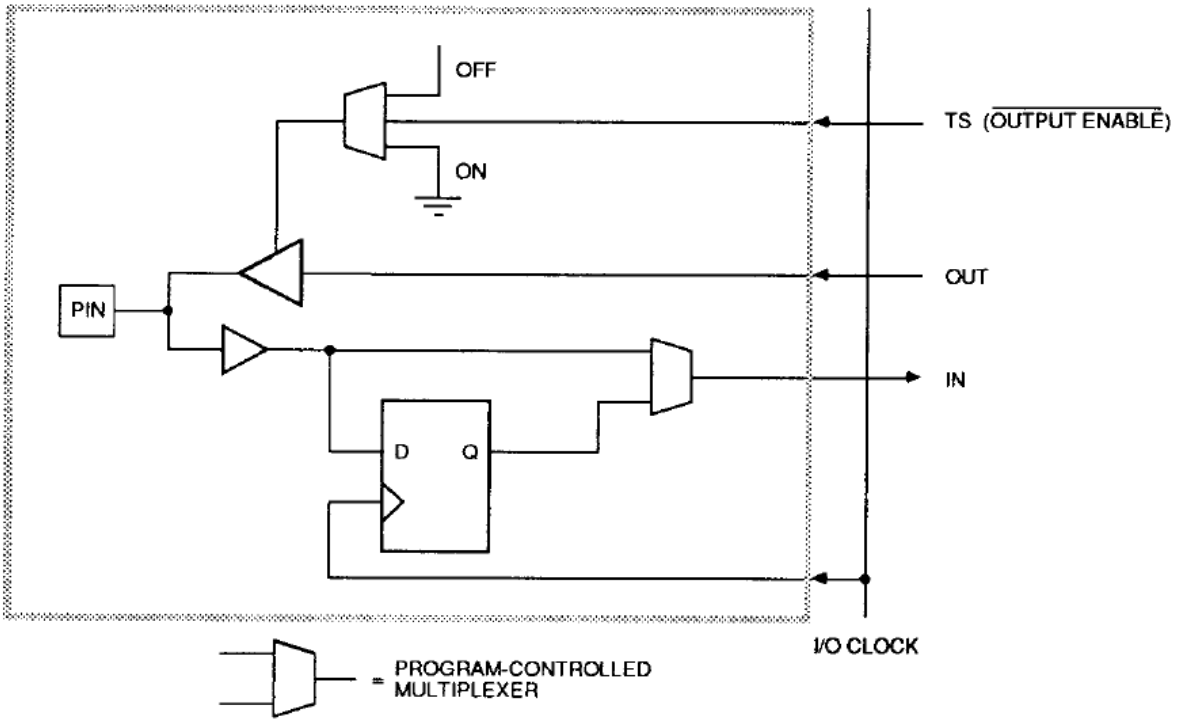


Figure 5.1: Wireframe of the I/O Block

The various components in this structure are explained below:

- **Input buffer :** A simple buffer using two cascaded NOT gates is used at the pin of the FPGA for receiving input from the external world. This buffer acts as a low pass filter for bouncing noise and also fixes the undershoots in HIGH / overshoots in LOW of the incoming signal.
- **PIN :** This is the physical pin exposed to the user to apply inputs or receive outputs.
- **D Flip-Flop :** The D Flip-Flop is used to create a registered version of the input if necessary. If an input signal in a Verilog code only appears in always@(posedge clk), then it can be registered right at its entry and can be used easily at any place required in the FPGA.
- **2X1 Input MUX :** This MUX selects between the direct version or the registered version of the input signal.
- **Tri-state buffer :** This buffer connects one of the neighbouring routing wires to the PIN when the I/O Block is used to send out data to the external world.

- **3X1 tri-state control MUX :** This MUX selects whether the FPGA is sending data out to the external world or not. An output enable signal can also be used from one of the neighbouring routing wires to selectively send output at specific times.

- **4X4 Configuration Memory :** A 16-bit SRAM arranged in a 4X4 6T-SRAM array to hold the configuration bits of the whole I/O Block.

  - 3-bits for selecting the output enable from one of the 4 neighbouring routing wires
  - 3-bits for selecting the output from one of the 4 neighbouring routing wires
  - 3-bits for selecting the wire where external world inputs land from the 4 neighbouring routing wires
  - 1-bit for the 2X1 Input MUX
  - 2-bits for the 3X1 tri-state control MUX
  - 2-bits for the PRE and CLR of the D Flip-Flop to determine initial state
  - 2-bits left unused for future feature updates

5.3 Describing the I/O Block in VTR

We describe our I/O block in VTR XML format and use it in the FPGA architecture used for testing. The XML description of the I/O blocks is shown in Fig. 5.2 for reference.

```
  <!-- Every input pin is driven by 100% of the tracks in a channel, every output pin is driven by 100% of the tracks in a channel -->
  <fc in_type="frac" in_val="1" out_type="frac" out_val="1"/>

  <!-- IOs go on the periphery of the FPGA, for consistency,
      make it physically equivalent on all sides so that only one definition of I/Os is needed.
      If I do not make a physically equivalent definition, then I need to define 4 different I/Os, one for each side of the FPGA
    -->
  <pinlocations pattern="custom">
    <loc side="left">io.outpad io.inpad io.clock</loc>
    <loc side="top">io.outpad io.inpad io.clock</loc>
    <loc side="right">io.outpad io.inpad io.clock</loc>
    <loc side="bottom">io.outpad io.inpad io.clock</loc>
  </pinlocations>

  <!-- Place I/Os on the sides of the FPGA -->
  <power method="ignore"/>
</pb_type>
<!-- Define I/O pads ends -->
```

Figure 5.2: XML description of the I/O Block

5.4 Summary

This chapter describes the architecture of the I/O block used. Each I/O block contains 16 configuration bits to program it correctly. These 16 bits are organized as a 4*4 SRAM array. The schematic of the I/O block described in Cadence Schematic L is shown in Fig. 5.3.

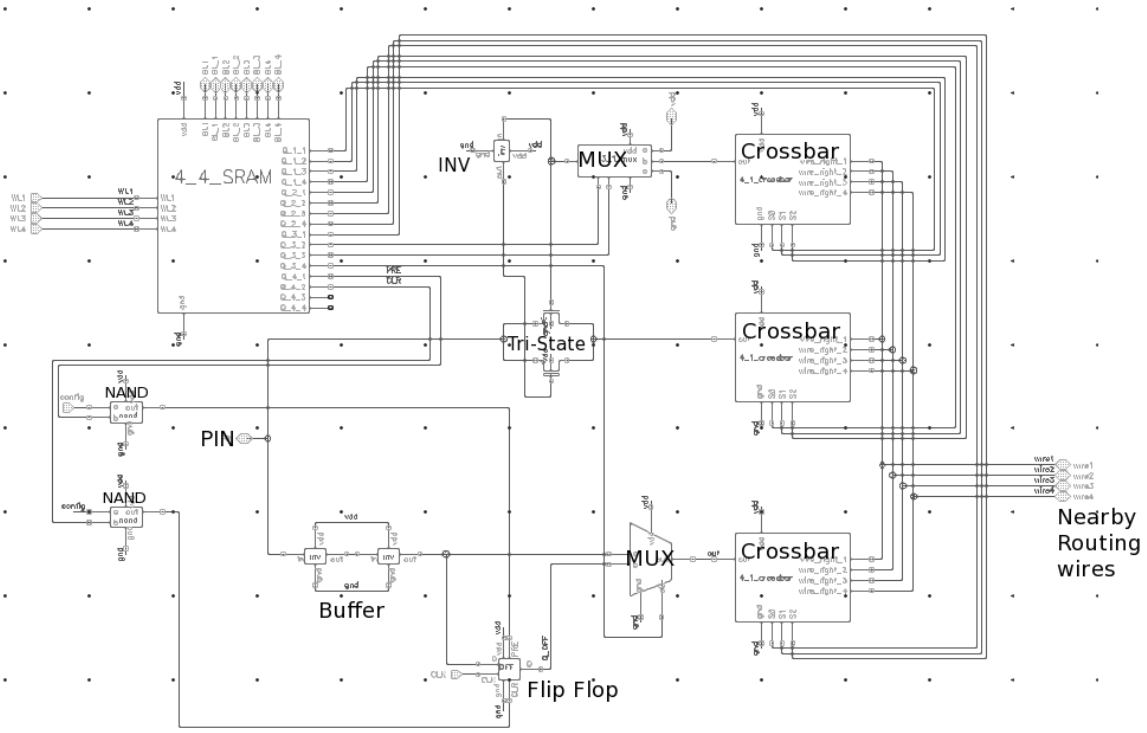


Figure 5.3: I/O Block schematic

CHAPTER 6

PROGRAMMABILITY OF THE FPGA

6.1 Overview

In the previous chapter, the I/O Block was designed in Cadence Schematic L. This chapter will focus on the programmability of the FPGA. We first discuss the configuration bits used across the FPGA, then carefully size the configuration 6-T cell to avoid unintended bit flips. Finally, we design the row and column decoders in Cadence Schematic L.

6.2 Configuration memory architecture

Each CLB requires 32 configuration bits which are organized into two 4X4 SRAM arrays. Each center switch matrix requires 16 configuration bits laid in a 4X4 SRAM array. Each edge switch matrix requires 12 configuration bits laid out in a 4X3 SRAM array. Lastly, each I/O block requires 16 configuration bits laid in a 4X4 SRAM array. All the memory bits have been carefully organised into arrays of column width 4. This allows us to program the FPGA 4 bits at a time. These 4 bits may be supplied parallely or can be internally buffered from a faster single bit stream. We see that increasing this programming parallelism beyond 4 doesn't scale elegantly. Furthermore, programming

more bits at a time increases the current and power consumption of the chip in configuration phase. This is why, we limit us to 4 bits at a time. Also, this only affects the speed with which the FPGA can be programmed and does not affect the FPGA performance at all.

6.3 Sizing a memory cell

The architecture we are using has common wordlines and bitlines. So, we have to take care of the fact that when you are not driving a bitline, its capacitive charge shouldn't change the memory contents even if it's worldline is raised. This situation occurs when you are trying to write neighbouring 4-bits sharing the same wordline. This was taken care of while minimally sizing the 6-T cell (see Fig. 6.1). We used a 5X safety over the bitline capacitance reported by a captab analysis using spectre in ADE L. This is to account for the RC parasitics of the wires in the layout of the FPGA.

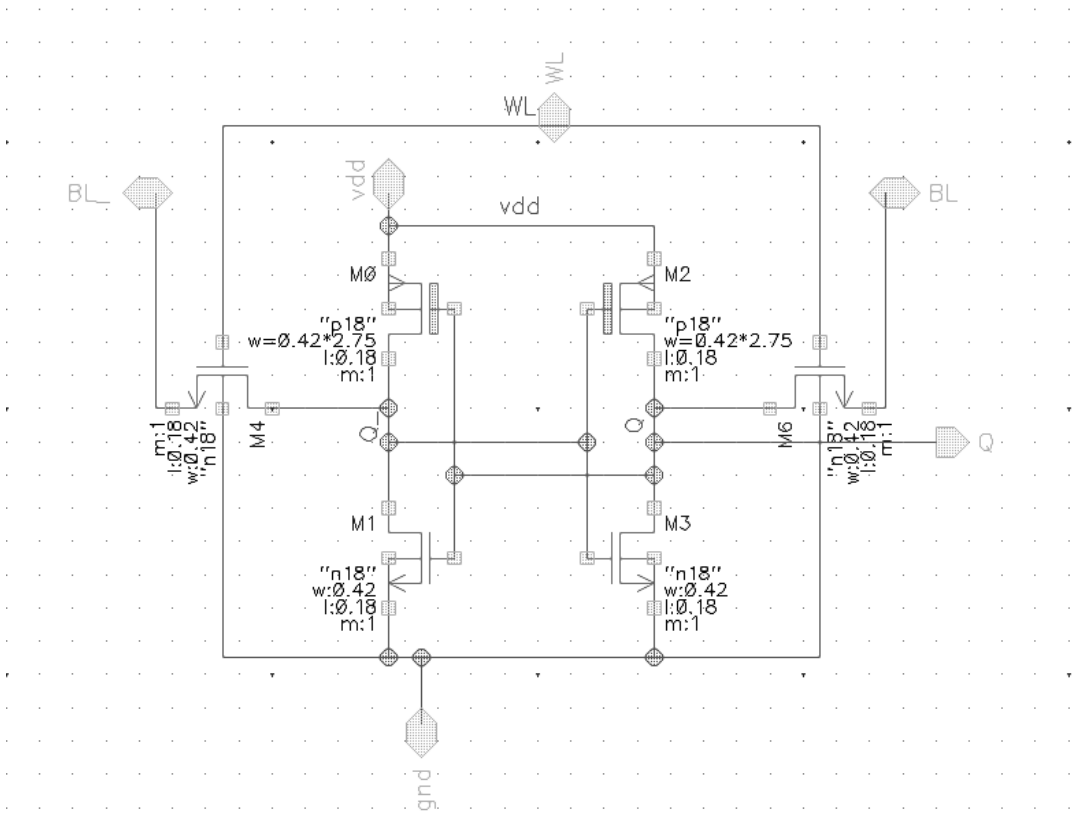


Figure 6.1: 6-T SRAM cell Schematic

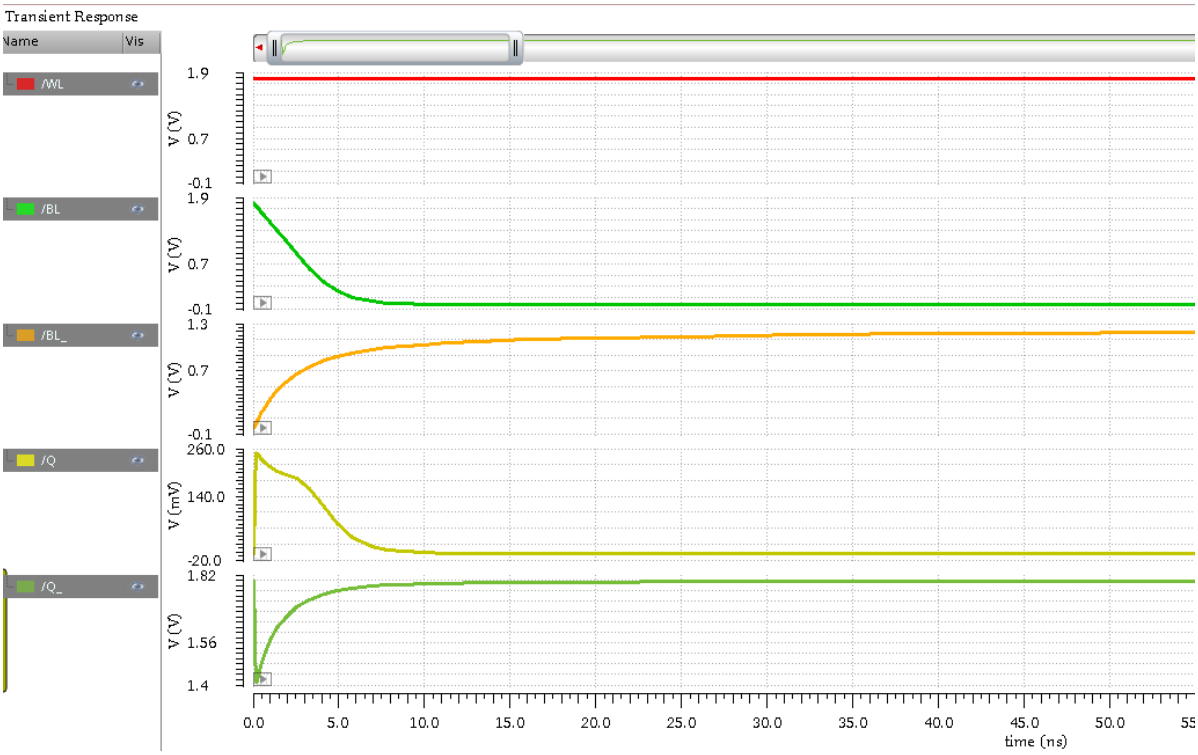


Figure 6.2: 6-T cell parasitic flip recovery

The simulation in Fig. 6.2 shows that when the bitlines are in a state opposite to the memory data at the time when wordline is raised, the cell corrects the stray value on the bitline capacitances with the actual data values instead of getting corrupted.

6.4 Designing the row decoder

We use SKILL language by Candence to automatically generate the schematic from a text file.

6.5 Designing the column decoder

We use SKILL language by Candence to automatically generate the schematic from a text file. The generated schematics of row and column decoders are shown in Fig. 6.3.

(a) row decoder (b) column decoder

Figure 6.3: Schematics of row and column decoder

6.6 Hot-swapping and partial reconfiguration

Since the read circuitory of LUTs uses decoders to read the truth table at any given time, it is independent of the write circuitory of the LUTs. Similar argument can be made for the configuration bits in the CLBs, switch matrices and I/O blocks. Therefore, the

FPGA can be partially reconfigured on the fly. Also, the truth tables can be hot-swapped to change the functionality on the fly. This functionality comes inherently with our FPGA architecture.

6.7 Summary

This chapter describes the programmability of the FPGA. The whole 8X8 FPGA uses 74 wordlines and 108 bitlines. 7 bits are used for row address decoding and 5 bits are used for decoding the 27(108/4) column addresses.

CHAPTER 7

RESULTS & DISCUSSION

7.1 Results

The CLB area and delay were modelled in chapter 3. These models were used to find the optimal LUT size and then the CLB was designed. In chapter 4, different switch matrix designs and parameters were explored to find a suitable switch matrix design. The I/O block was designed in Chapter 5. Chapter 6 discussed the programmability of the FPGA. Here, we present the complete schematic of the FPGA designed in Cadence Schematic L in Fig. 7.1.

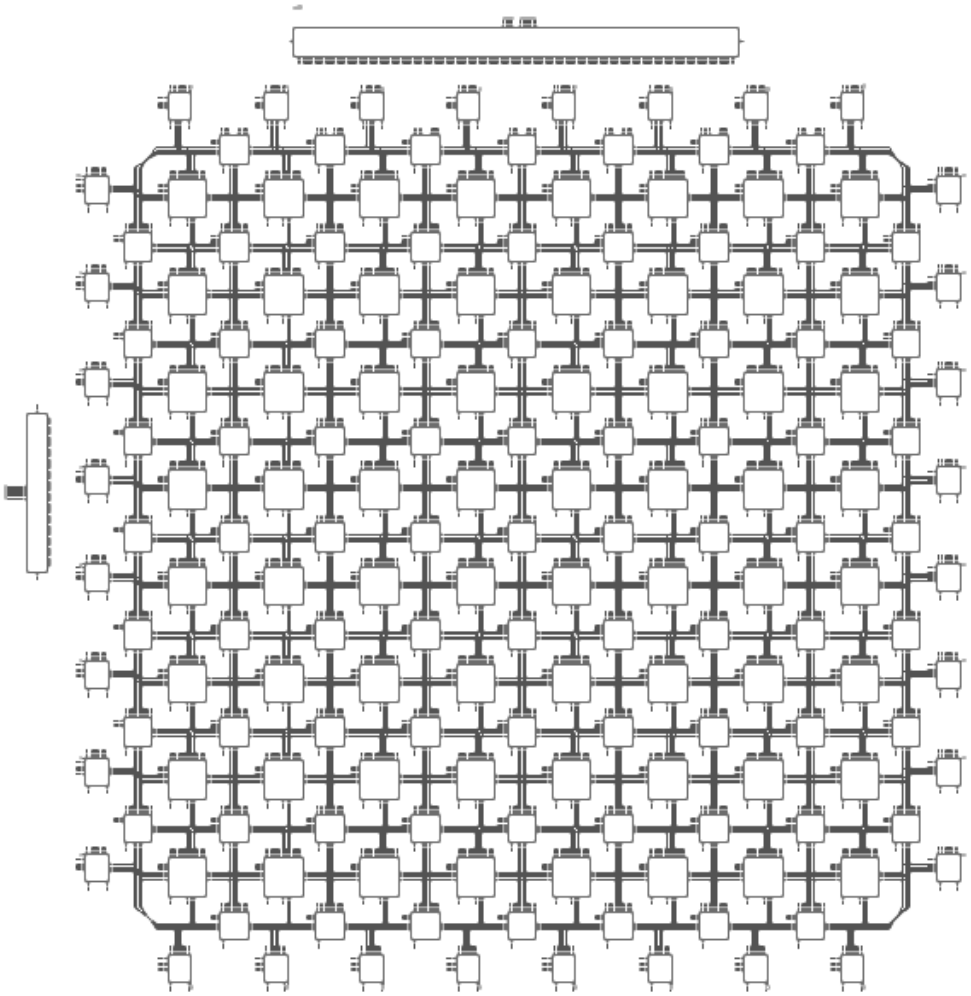


Figure 7.1: Schematic of the FPGA

We went on to test the functionality of the FPGA by implementing a 2-bit OR Gate on this FPGA. The bitstream required to program the FPGA was written by hand in an array format as shown in Fig. 7.2 and was parsed using various scripts to automatically generate the relevant stimulus file to program and test the FPGA.

I/O Block left-1 and top-1 are used for inputs and I/O block top-2 is used for output. After programming the FPGA with the bitstream, the waveforms obtained are shown in Fig. 7.3. The delay path for this design is identified as below:

- **Output fall time :** $36ns$

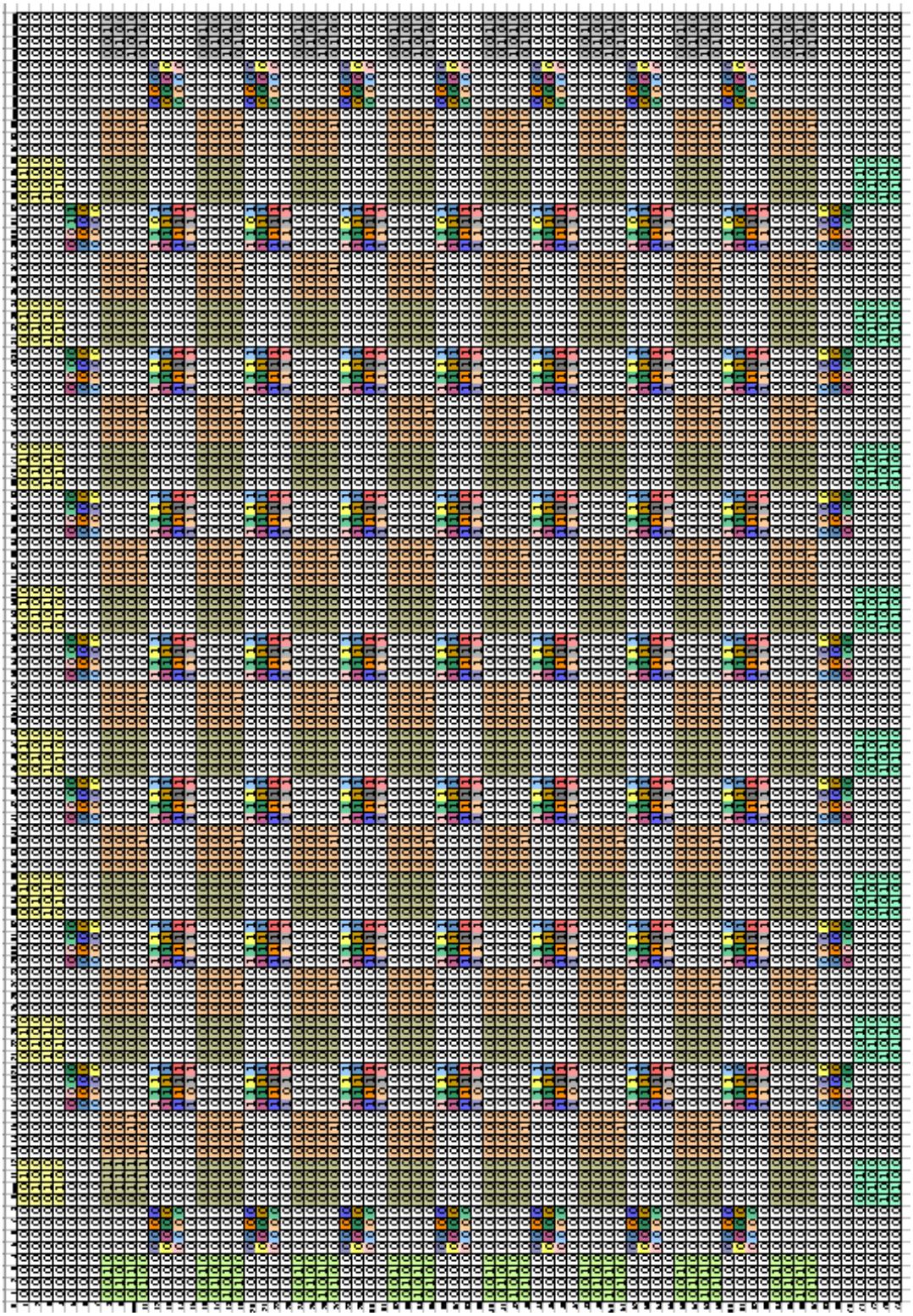


Figure 7.2: Bitstream for 2-input OR gate

- **I/O Block top-1 :** The total delay in this block is $7.43ns$ which breaks down as follows:
  - Input buffer : $0.35ns$
  - 2X1 MUX : $0.14ns$
  - 4X1 Crossbar : $6.94ns$
- **CLB 1 :** The total delay in this block is $24.44ns$ which breaks down as follows:
  - 4X4 Crossbar : $0.19ns$
  - 2X1 MUX : $1.25ns$
  - 2X1 MUX : $2.95ns$
  - 4X1 Crossbar : $20.05ns$
- **Switch Matrix :** The total delay in this block is $3.94ns$
- **I/O Block top-2 :** The total delay in this block is $0.19ns$ which breaks down as follows:
  - 4X1 Crossbar : $0.17ns$
  - Tx-Gate : $0.02ns$

- **Output rise time :** $50ns$

The operating frequency for this design comes out to be $\approx 20Mhz$ based on schematic level simulations. We might expect a degradation of around 2 after layout and thus say that the design will work at $\approx 10Mhz$ for safety.

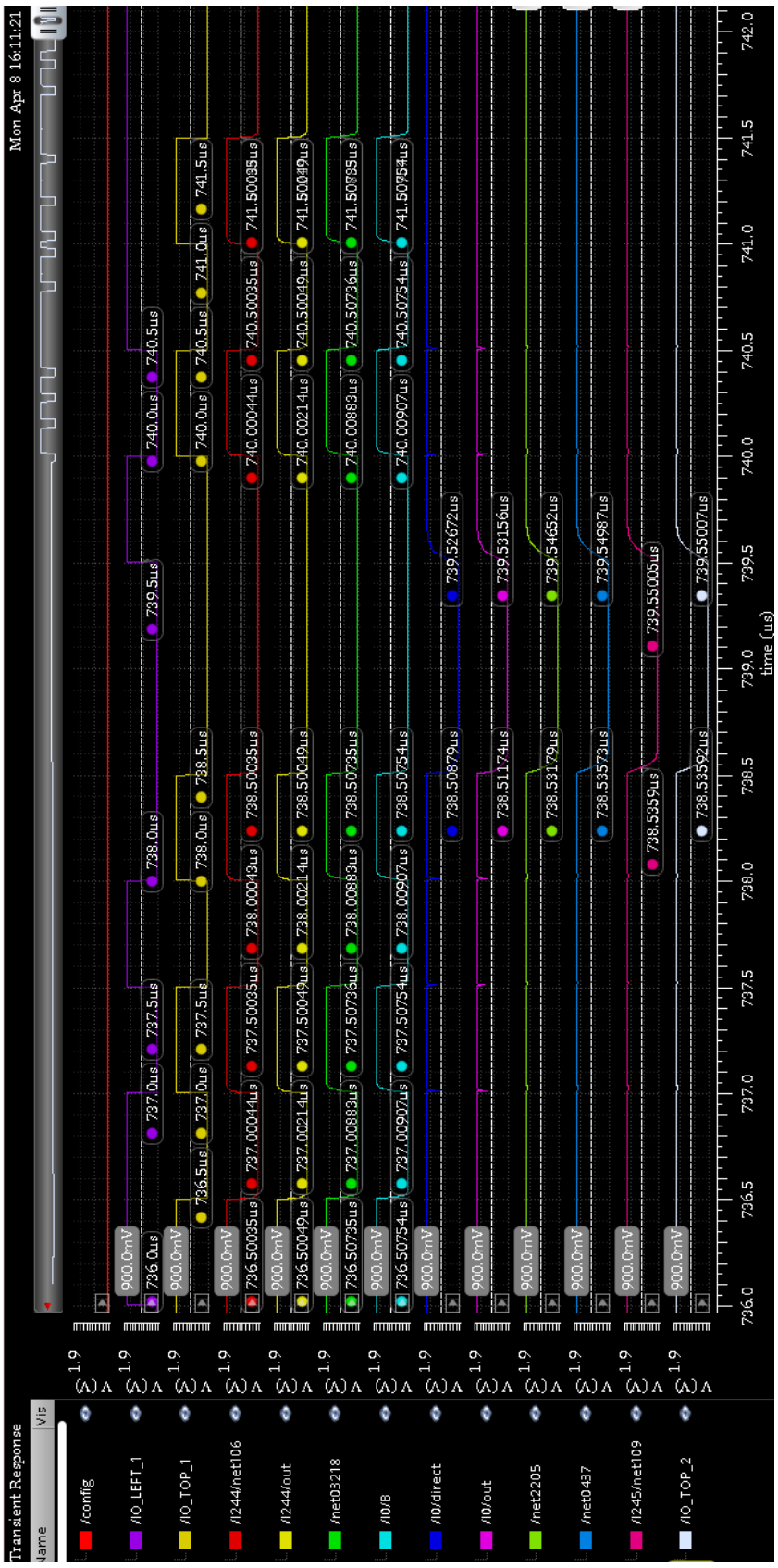


Figure 7.3: Output waveforms for an OR Gate

7.2 Discussion

We develop models and approximations to help architect an Island-Style FPGA. The models/experiments help to decide critical design parameters like LUT size, routing width etc. The FPGA schematic is then designed from CLB, switch matrix, I/O block and decoder schematics which themselves are made up of smaller schematics. The FPGA functionality is then tested to see the delay path and programming is verified. Although, we have used Cadence Schematic L to proove the design decisions involved, this flow is not suitable for bigger FPGAs. The simulation of a small OR Gate implementation took hours and thus it is not recommended to go beyond that. The designer, after doing a preliminary analysis and testing, can shift to a completely automated digital flow from Verilog to layout. Part of the reason we didn't chose that flow is that SCL 180nm PDK is known to have incomplete file structure for a smooth digital flow. Moreover, ours is a small and repetitive design and thus, the layout can even be done by hand.

CHAPTER 8

CONCLUSION

This chapter gives a brief of the research work carried out and the conclusions drawn from the same. The future work is also discussed in this chapter.

8.1 Conclusion

FPGAs are very important tools for prototyping digital circuits, accelerating parallel designs and implementing minimal logic at very low cost. Minimal FPGAs are quite promising to be used in products where very little logic is required. There aren't many open source FPGA design flows. We try to develop the same. We first begin with developing models of area and delays for a CLB. These models help to decide the LUT size. Then we perform experiments using different orientations of input pins on a CLB and decide on a spread design. After this, we experiment with different designs in VTR on our FPGA and decide the routing width. Then the programmability decisions are made and the schematics are designed in Schematic L. The 6-T memory cell is sized carefully to avoid bit-flips when nearby cells are being written. Smaller schematics are tested individually for functionality and then the FPGA schematic is designed. The functionality of the FPGA is then tested by implementing a simple circuit and the delay breakdown is identified.

8.2 Scope for Future Research

The future research in this area can be in the following directions:

- Developing the layout of a CLB, Switch Matrix and I/O Block
- Laying out the FPGA from its constituents usking SKILL and performing post-layout simulations.
- Designing a framework to automate the layout of an NxN FPGA
- Delving into the digital flow for optimizing sizing of transistors at a finer granularity and then developing models to be used for a generic NxN FPGA

BIBLIOGRAPHY

[1] D.-K. Electronics, "Portfolio of lattice's miniature family of fpgas."
Available at https://www.digikey.in/en/product-highlight/l/lattice/fpga-portfolio.

[2] J. Rose, "Fpga architecture research."
Available at http://www.eecg.toronto.edu/~jayar/research/architecture.html.

[3] H. Gao, Y. Yang, X. Ma, and G. Dong, "Analysis of the effect of lut size on fpga area and delay using theoretical derivations," in *Sixth international symposium on quality electronic design (isqed'05)*, pp. 370–374, March 2005.

[4] "Verilog to routing."
Available at https://verilogtorouting.org/.

[5] V. Betz and J. Rose, "Circuit design, transistor sizing and wire layout of fpga interconnect," in *Proceedings of the IEEE 1999 Custom Integrated Circuits Conference (Cat. No.99CH36327)*, pp. 171–174, May 1999.

[6] B. S. Landman and R. L. Russo, "On a pin versus block relationship for partitions of logic graphs," *IEEE Transactions on Computers*, vol. C-20, pp. 1469–1479, Dec 1971.

[7] P. K. Chan, M. D. F. Schlag, and J. Y. Zien, "On routability prediction for field-programmable gate arrays," in *30th ACM/IEEE Design Automation Conference*, pp. 326–330, June 1993.

[8] M. I. Masud, "Fpga routing structures: A novel switch block and depopulated interconnect matrix architectures."
Available at http://www.ece.ubc.ca/~stevew/papers/pdf/imran_masc.pdf.

[9] D. Tavana, W. Yee, S. Young, and B. Fawcett, "Logic block and routing considerations for a new sram-based fpga architecture," in *Proceedings of the IEEE 1995 Custom Integrated Circuits Conference*, pp. 511–514, May 1995.

[10] Xilinx, "Datasheet of xilinx 2064."
Available at `https://www.datasheetspdf.com/pdf/625540/Xilinx/XC2064/1`.

APPENDIX A

A.I Architecture XML file of our FPGA

```
<architecture>

  <!--
       ODIN II specific config begins
       Describes the types of user-specified netlist blocks (in blif,
          ↪ this corresponds to
       ".model [type_of_block]") that this architecture supports.

       Note: Basic LUTs, I/Os, and flip-flops are not included here as
          ↪ there are
       already special structures in blif (.names, .input, .output, and .
          ↪ latch)
       that describe them.
  -->
  <models>
  </models>
  <!-- ODIN II specific config ends -->

  <!-- Physical descriptions begin -->
  <layout>
     <auto_layout aspect_ratio="1.000000">
        <!--Perimeter of 'io' blocks with 'EMPTY' blocks at corners-->
        <perimeter type="io" priority="100"/>
        <corners type="EMPTY" priority="101"/>
        <!--Fill with 'clb'-->
        <fill type="clb" priority="10"/>
```

```
        </auto_layout>
</layout>
        <device>
                <sizing R_minW_nmos="6065.520020" R_minW_pmos
                    ↪ ="18138.500000"/>
                <area grid_logic_tile_area="7238.080078"/>
                <chan_width_distr>
                <x distr="uniform" peak="1.000000"/>
                        <y distr="uniform" peak="1.000000"/>
                </chan_width_distr>
                <switch_block type="wilton" fs="3"/>
        <connection_block input_switch_name="ipin_cblock"/>
        </device>
        <switchlist>
                <switch type="mux" name="0" R="0.000000" Cin="0.000000e
                    ↪ +00" Cout="0.000000e+00" Tdel="7.958000e-11"
                    ↪ mux_trans_size="2.074780" buf_size="19.261999"/>
        <!--switch ipin_cblock resistance set to yeild for 4x minimum
            ↪ drive strength buffer-->
        <switch type="mux" name="ipin_cblock" R="1516.380005" Cout="0."
            ↪ Cin="0.000000e+00" Tdel="7.362000e-11" mux_trans_size
            ↪ ="1.240240" buf_size="auto"/>
        </switchlist>
        <segmentlist>
                <segment freq="1.000000" length="1" type="unidir" Rmetal
                    ↪ ="0.000000" Cmetal="0.000000e+00">
                    <mux name="0"/>
                    <sb type="pattern">1 1</sb>
                    <cb type="pattern">1</cb>
                </segment>
        </segmentlist>

  <complexblocklist>

    <!-- Define I/O pads begin -->
```

```
<!-- Capacity is a unique property of I/Os, it is the maximum number
    ↪  of I/Os that can be placed at the same (X,Y) location on the
    ↪ FPGA -->
<pb_type name="io" capacity="1">
  <input name="outpad" num_pins="1"/>
  <output name="inpad" num_pins="1"/>
  <clock name="clock" num_pins="1"/>

  <!-- IOs can operate as either inputs or outputs.
       Delays below come from Ian Kuon. They are small, so they
          ↪ should be interpreted as
       the delays to and from registers in the I/O (and generally I/
          ↪ Os are registered
       today and that is when you timing analyze them.
       -->
  <mode name="inpad">
    <pb_type name="inpad" blif_model=".input" num_pb="1">
      <output name="inpad" num_pins="1"/>
    </pb_type>
    <interconnect>
      <direct name="inpad" input="inpad.inpad" output="io.inpad">
        <delay_constant max="4.791000e-11" in_port="inpad.inpad"
           ↪ out_port="io.inpad"/>
      </direct>
    </interconnect>

  </mode>
  <mode name="outpad">
    <pb_type name="outpad" blif_model=".output" num_pb="1">
      <input name="outpad" num_pins="1"/>
    </pb_type>
    <interconnect>
      <direct name="outpad" input="io.outpad" output="outpad.outpad">
        <delay_constant max="1.557000e-11" in_port="io.outpad"
           ↪ out_port="outpad.outpad"/>
```

```
      </direct>
    </interconnect>
  </mode>

  <!-- Every input pin is driven by 100% of the tracks in a channel,
     ↪ every output pin is driven by 100% of the tracks in a channel
     ↪  -->
  <fc in_type="frac" in_val="1" out_type="frac" out_val="1"/>

  <!-- IOs go on the periphery of the FPGA, for consistency,
     make it physically equivalent on all sides so that only one
        ↪ definition of I/Os is needed.
     If I do not make a physically equivalent definition, then I
        ↪ need to define 4 different I/Os, one for each side of the
        ↪  FPGA
    -->
  <pinlocations pattern="custom">
    <loc side="left">io.outpad io.inpad io.clock</loc>
    <loc side="top">io.outpad io.inpad io.clock</loc>
    <loc side="right">io.outpad io.inpad io.clock</loc>
    <loc side="bottom">io.outpad io.inpad io.clock</loc>
  </pinlocations>

  <!-- Place I/Os on the sides of the FPGA -->
  <power method="ignore"/>
</pb_type>
<!-- Define I/O pads ends -->

<!-- Define general purpose logic block (CLB) begin -->
<pb_type name="clb">
  <input name="I" num_pins="4" equivalent="full"/>
  <output name="O" num_pins="1" equivalent="instance"/>
  <clock name="clk" num_pins="1"/>

  <!-- Describe basic logic element. -->
```

```
<!-- Define 6-LUT mode -->
    <pb_type name="ble4" num_pb="1">
      <input name="in" num_pins="4"/>
      <output name="out" num_pins="1"/>
      <clock name="clk" num_pins="1"/>

      <!-- Define LUT -->
      <pb_type name="lut4" blif_model=".names" num_pb="1" class="
         ↪ lut">
        <input name="in" num_pins="4" port_class="lut_in"/>
        <output name="out" num_pins="1" port_class="lut_out"/>
        <!-- LUT timing using delay matrix -->
        <delay_matrix type="max" in_port="lut4.in" out_port="lut4.
           ↪ out">
          2.063000e-10
          2.063000e-10
          2.063000e-10
          2.063000e-10
        </delay_matrix>
      </pb_type>

      <!-- Define flip-flop -->
      <pb_type name="ff" blif_model=".latch" num_pb="1" class="
         ↪ flipflop">
        <input name="D" num_pins="1" port_class="D"/>
        <output name="Q" num_pins="1" port_class="Q"/>
        <clock name="clk" num_pins="1" port_class="clock"/>
                  <!-- setup time included in LUT delay -->
        <T_setup value="0.000000e-10" port="ff.D" clock="clk"/>
        <T_clock_to_Q max="8.406000e-11" port="ff.Q" clock="clk"/>
      </pb_type>

      <interconnect>
        <direct name="direct1" input="ble4.in" output="lut4[0:0].in
           ↪ "/>
```

```
        <direct name="direct2" input="lut4.out" output="ff.D">
          <!-- Advanced user option that tells CAD tool to find LUT+
             ↪ FF pairs in netlist -->
          <pack_pattern name="ble6" in_port="lut4.out" out_port="ff.
             ↪ D"/>
        </direct>
        <direct name="direct3" input="ble4.clk" output="ff.clk"/>
        <mux name="mux1" input="ff.Q lut4.out" output="ble4.out">
        </mux>
      </interconnect>
    </pb_type>
<interconnect>
  <!-- We use a full crossbar to get logical equivalence at inputs
     ↪ of CLB -->
  <complete name="crossbar" input="clb.I ble4[0:0].out" output="
     ↪ ble4[0:0].in">
    <delay_constant max="5.043000e-11" in_port="clb.I" out_port="
       ↪ ble4[0:0].in"/>
    <delay_constant max="5.031000e-11" in_port="ble4[0:0].out"
       ↪ out_port="ble4[0:0].in"/>
  </complete>
  <complete name="clks" input="clb.clk" output="ble4[0:0].clk">
  </complete>
  <direct name="clbouts1" input="ble4[0:0].out" output="clb.O"/>
</interconnect>

<!-- Every input pin is driven by 100% of the tracks in a channel,
   ↪ every output pin is driven by 100% of the tracks in a channel
   ↪  -->
<fc in_type="frac" in_val="1" out_type="frac" out_val="1"/>

<pinlocations pattern="custom">
    <loc side="left">clb.I[0] clb.clk</loc>
    <loc side="top">clb.I[1]</loc>
    <loc side="right">clb.I[2] clb.O</loc>
```

```
        <loc side="bottom">clb.I[3]</loc>
      </pinlocations>

    <!-- Place this general purpose logic block in any unspecified
       ↪ column -->
    </pb_type>
  <!-- Define general purpose logic block (CLB) ends -->

 </complexblocklist>
</architecture>
```

A.II Bitstream to binary stimulus - script

```
import sys
import csv
sys.stdout = open('memory_data_compact.txt', 'w')
data = list(csv.reader(open('memory_mapping.csv')))

sys.stdout.write('BL1 BL1 0 000 ')

for j in range(1,17):
      for k in range(1,10):
            sys.stdout.write(str(data[j][4*k-3]))
print(' 00000000000000')

sys.stdout.write('BL2 BL2 0 000 ')
for j in range(1,17):
      for k in range(1,10):
            sys.stdout.write(str(data[j][4*k-2]))
print(' 00000000000000')

sys.stdout.write('BL3 BL3 0 000 ')
for j in range(1,17):
```

```
        for k in range(1,10):
                sys.stdout.write(str(data[j][4*k-1]))
print(' 00000000000000')

sys.stdout.write('BL4 BL4 0 000 ')
for j in range(1,17):
        for k in range(1,10):
                sys.stdout.write(str(data[j][4*k]))
print(' 00000000000000')
```

A.III binary sequences to .scs - script

```
import sys
import fileinput
orig_stdout = sys.stdout
sys.stdout = open('stimulus.scs', 'w')
TIM=147
print('simulator lang=spectre')
print('parameters VDD = 1.8')
print('//tr is (transition time)/2')
print('parameters tr = 1n')
print('//f is the fall time')
print('//parameters f = 100p')
print('//K is the signal period for programming')
print('parameters K = 5u')
print('//L is the signal period for running')
print('parameters L = 0.5u')
print('\n')
print('vVdd (vdd 0) vsource dc=VDD')
print('vGnd (gnd 0) vsource dc=0')
print('\n')
```

```
lines = [line.rstrip('\n') for line in open('memory_data_compact.txt','r
    ↪ ')]

while '' in lines:
    lines.remove('')

for line in lines:
        words=line.split()
        print('v'+words[0]+' ('+words[1]+ ' '+words[2]+') vsource type=
            ↪ pwl wave=\[')
        sent=words[3:]
        sen=''.join(sent)
        sen=sen.replace(" ", "")
        bits=list(sen)
        for i in range(len(bits)):
                ostr=""
                ostr+="+ ("
                if i==0:
                        ostr+=" 0*K ) "
                else:
                        if i<TIM:
                                ostr+=str(i).rjust(2)+'*K+tr) '
                        else:
                                ostr+=str(TIM).rjust(2)+'*K+'+str(i-TIM).
                                    ↪ rjust(2)+'*L+tr) '
                if bits[i]=='0':
                        if i<TIM:
                                ostr+='0 ('+str(i+1).rjust(2)+'*K-tr) 0'
                        else:
                                ostr+='0 ('+str(TIM).rjust(2)+'*K+'+str(i-
                                    ↪ TIM+1).rjust(2)+'*L-tr) 0'
                if bits[i]=='1':
                        if i<TIM:
                                ostr+='VDD ('+str(i+1).rjust(2)+'*K-tr) VDD'
                                    ↪
```

```
                else:
                        ostr+='VDD ('+str(TIM).rjust(2)+'*K+'+str(i-
                            ↪ TIM+1).rjust(2)+'*L-tr) VDD'
        print(ostr)
print('+ ]')

if words[0][0]=='B' and words[0][1]=='L':
        print('vBL_'+words[0][2:]+' (BL_'+words[1][2:]+ ' '+words
            ↪ [2]+') vsource type=pwl wave=\[')
        sent=words[3:]
        sen=''.join(sent)
        sen=sen.replace(" ", "")
        bits=list(sen)
        for i in range(len(bits)):
                ostr=""
                ostr+="+ ("
                if i==0:
                        ostr+=" 0*K ) "
                else:
                        if i<TIM:
                                ostr+=str(i).rjust(2)+'*K+tr) '
                        else:
                                ostr+=str(TIM).rjust(2)+'*K+'+str(i-
                                    ↪ TIM).rjust(2)+'*L+tr) '
                if bits[i]=='0':
                        if i<TIM:
                                ostr+='VDD ('+str(i+1).rjust(2)+'*K-
                                    ↪ tr) VDD'
                        else:
                                ostr+='VDD ('+str(TIM).rjust(2)+'*K
                                    ↪ +'+str(i-TIM+1).rjust(2)+'*L-
                                    ↪ tr) VDD'
                if bits[i]=='1':
                        if i<TIM:
```

```
                                    ostr+='0 ('+str(i+1).rjust(2)+'*K-tr
                                        ↪ ) 0'
                                else:
                                    ostr+='0 ('+str(TIM).rjust(2)+'*K+'+
                                        ↪ str(i-TIM+1).rjust(2)+'*L-tr)
                                        ↪ 0'
                            print(ostr)
                    print('+ ]')

sys.stdout.close()
sys.stdout=orig_stdout
# Read in the file
with open('stimulus.scs', 'r') as file :
  filedata = file.read()

# Replace the target string
filedata = filedata.replace('BL', 'BitLine')

# Write the file out again
with open('stimulus.scs', 'w') as file:
  file.write(filedata)
```

A.IV .scs for D Flip Flop

```
simulator lang=spectre
parameters VDD = 1.8
//tr is (transition time)/2
parameters tr = 50p
//f is the fall time
//parameters f = 100p
//K is the signal period
parameters K = 40n
```

```
vVdd (vdd 0) vsource dc=VDD
vGnd (gnd 0) vsource dc=0


vPRE (PRE 0) vsource type=pwl wave=\[
+ ( 0*K ) VDD ( 1*K-tr) VDD
+ ( 1*K+tr) VDD ( 2*K-tr) VDD
+ ( 2*K+tr) VDD ( 3*K-tr) VDD
+ ( 3*K+tr) VDD ( 4*K-tr) VDD
+ ( 4*K+tr) VDD ( 5*K-tr) VDD
+ ( 5*K+tr) VDD ( 6*K-tr) VDD
+ ( 6*K+tr) VDD ( 7*K-tr) VDD
+ ( 7*K+tr) VDD ( 8*K-tr) VDD
+ ( 8*K+tr) VDD ( 9*K-tr) VDD
+ ( 9*K+tr) VDD (10*K-tr) VDD
+ (10*K+tr) VDD (11*K-tr) VDD
+ (11*K+tr) 0 (12*K-tr) 0
+ (12*K+tr) 0 (13*K-tr) 0
+ (13*K+tr) 0 (14*K-tr) 0
+ (14*K+tr) 0 (15*K-tr) 0
+ (15*K+tr) VDD (16*K-tr) VDD
+ (16*K+tr) VDD (17*K-tr) VDD
+ (17*K+tr) VDD (18*K-tr) VDD
+ (18*K+tr) VDD (19*K-tr) VDD
+ (19*K+tr) VDD (20*K-tr) VDD
+ (20*K+tr) VDD (21*K-tr) VDD
+ (21*K+tr) VDD (22*K-tr) VDD
+ (22*K+tr) VDD (23*K-tr) VDD
+ (23*K+tr) VDD (24*K-tr) VDD
+ (24*K+tr) VDD (25*K-tr) VDD
+ (25*K+tr) VDD (26*K-tr) VDD
+ (26*K+tr) VDD (27*K-tr) VDD
+ (27*K+tr) VDD (28*K-tr) VDD
+ (28*K+tr) VDD (29*K-tr) VDD
```

```
+ (29*K+tr) VDD (30*K-tr) VDD
+ (30*K+tr) VDD (31*K-tr) VDD
+ (31*K+tr) VDD (32*K-tr) VDD
+ (32*K+tr) VDD (33*K-tr) VDD
+ ]
vCLR (CLR 0) vsource type=pwl wave=\[
+ ( 0*K ) 0 ( 1*K-tr) 0
+ ( 1*K+tr) 0 ( 2*K-tr) 0
+ ( 2*K+tr) 0 ( 3*K-tr) 0
+ ( 3*K+tr) 0 ( 4*K-tr) 0
+ ( 4*K+tr) 0 ( 5*K-tr) 0
+ ( 5*K+tr) 0 ( 6*K-tr) 0
+ ( 6*K+tr) 0 ( 7*K-tr) 0
+ ( 7*K+tr) 0 ( 8*K-tr) 0
+ ( 8*K+tr) 0 ( 9*K-tr) 0
+ ( 9*K+tr) 0 (10*K-tr) 0
+ (10*K+tr) 0 (11*K-tr) 0
+ (11*K+tr) VDD (12*K-tr) VDD
+ (12*K+tr) VDD (13*K-tr) VDD
+ (13*K+tr) VDD (14*K-tr) VDD
+ (14*K+tr) VDD (15*K-tr) VDD
+ (15*K+tr) VDD (16*K-tr) VDD
+ (16*K+tr) VDD (17*K-tr) VDD
+ (17*K+tr) VDD (18*K-tr) VDD
+ (18*K+tr) VDD (19*K-tr) VDD
+ (19*K+tr) VDD (20*K-tr) VDD
+ (20*K+tr) VDD (21*K-tr) VDD
+ (21*K+tr) VDD (22*K-tr) VDD
+ (22*K+tr) VDD (23*K-tr) VDD
+ (23*K+tr) VDD (24*K-tr) VDD
+ (24*K+tr) VDD (25*K-tr) VDD
+ (25*K+tr) VDD (26*K-tr) VDD
+ (26*K+tr) VDD (27*K-tr) VDD
+ (27*K+tr) VDD (28*K-tr) VDD
+ (28*K+tr) VDD (29*K-tr) VDD
```

```
+ (29*K+tr) VDD (30*K-tr) VDD
+ (30*K+tr) VDD (31*K-tr) VDD
+ (31*K+tr) VDD (32*K-tr) VDD
+ (32*K+tr) VDD (33*K-tr) VDD
+ ]
vCLK (CLK 0) vsource type=pwl wave=\[
+ ( 0*K ) 0 ( 1*K-tr) 0
+ ( 1*K+tr) 0 ( 2*K-tr) 0
+ ( 2*K+tr) VDD ( 3*K-tr) VDD
+ ( 3*K+tr) VDD ( 4*K-tr) VDD
+ ( 4*K+tr) 0 ( 5*K-tr) 0
+ ( 5*K+tr) 0 ( 6*K-tr) 0
+ ( 6*K+tr) VDD ( 7*K-tr) VDD
+ ( 7*K+tr) VDD ( 8*K-tr) VDD
+ ( 8*K+tr) 0 ( 9*K-tr) 0
+ ( 9*K+tr) 0 (10*K-tr) 0
+ (10*K+tr) VDD (11*K-tr) VDD
+ (11*K+tr) VDD (12*K-tr) VDD
+ (12*K+tr) 0 (13*K-tr) 0
+ (13*K+tr) 0 (14*K-tr) 0
+ (14*K+tr) VDD (15*K-tr) VDD
+ (15*K+tr) VDD (16*K-tr) VDD
+ (16*K+tr) 0 (17*K-tr) 0
+ (17*K+tr) 0 (18*K-tr) 0
+ (18*K+tr) VDD (19*K-tr) VDD
+ (19*K+tr) VDD (20*K-tr) VDD
+ (20*K+tr) 0 (21*K-tr) 0
+ (21*K+tr) 0 (22*K-tr) 0
+ (22*K+tr) VDD (23*K-tr) VDD
+ (23*K+tr) VDD (24*K-tr) VDD
+ (24*K+tr) 0 (25*K-tr) 0
+ (25*K+tr) 0 (26*K-tr) 0
+ (26*K+tr) VDD (27*K-tr) VDD
+ (27*K+tr) VDD (28*K-tr) VDD
+ (28*K+tr) 0 (29*K-tr) 0
```

```
+ (29*K+tr) 0 (30*K-tr) 0
+ (30*K+tr) VDD (31*K-tr) VDD
+ (31*K+tr) VDD (32*K-tr) VDD
+ (32*K+tr) 0 (33*K-tr) 0
+ ]
vD (D 0) vsource type=pwl wave=\[
+ ( 0*K ) VDD ( 1*K-tr) VDD
+ ( 1*K+tr) VDD ( 2*K-tr) VDD
+ ( 2*K+tr) VDD ( 3*K-tr) VDD
+ ( 3*K+tr) VDD ( 4*K-tr) VDD
+ ( 4*K+tr) VDD ( 5*K-tr) VDD
+ ( 5*K+tr) VDD ( 6*K-tr) VDD
+ ( 6*K+tr) VDD ( 7*K-tr) VDD
+ ( 7*K+tr) VDD ( 8*K-tr) VDD
+ ( 8*K+tr) VDD ( 9*K-tr) VDD
+ ( 9*K+tr) VDD (10*K-tr) VDD
+ (10*K+tr) VDD (11*K-tr) VDD
+ (11*K+tr) VDD (12*K-tr) VDD
+ (12*K+tr) VDD (13*K-tr) VDD
+ (13*K+tr) VDD (14*K-tr) VDD
+ (14*K+tr) VDD (15*K-tr) VDD
+ (15*K+tr) VDD (16*K-tr) VDD
+ (16*K+tr) VDD (17*K-tr) VDD
+ (17*K+tr) VDD (18*K-tr) VDD
+ (18*K+tr) VDD (19*K-tr) VDD
+ (19*K+tr) VDD (20*K-tr) VDD
+ (20*K+tr) VDD (21*K-tr) VDD
+ (21*K+tr) VDD (22*K-tr) VDD
+ (22*K+tr) VDD (23*K-tr) VDD
+ (23*K+tr) VDD (24*K-tr) VDD
+ (24*K+tr) VDD (25*K-tr) VDD
+ (25*K+tr) VDD (26*K-tr) VDD
+ (26*K+tr) VDD (27*K-tr) VDD
+ (27*K+tr) VDD (28*K-tr) VDD
+ (28*K+tr) VDD (29*K-tr) VDD
```

```
+ (29*K+tr) VDD (30*K-tr) VDD
+ (30*K+tr) VDD (31*K-tr) VDD
+ (31*K+tr) VDD (32*K-tr) VDD
+ (32*K+tr) VDD (33*K-tr) VDD
+ ]
```

A.V Decoder SKILL code

```
load("./skill/Schematic.il")
procedure(DecoderSchematic(libName cellName WIDTH OUTPUTS)
  ;; Define the local variables
  prog((cvid x y pin pinName k h temp a b out inst)
    ;; Open the cell and its schematic view
    cvid = dbOpenCellViewByType(libName cellName "schematic" "schematic"
       ↪  "a")
        ;; Clean the schematic view
    ece432DeleteObjectsSchematic(cvid)
        ;; The coordinate of the origin point for the schematic
    x=0
    y=0
    ;; Create top level pins
        ;; Create input pins
        for(u 0 WIDTH-1
               a=sprintf(nil "A%L" u)
               pin=ece432SchematicCreatePin(cvid a "input" x:y "R0")
        y=y+0.5
    );; end for(u
        ;; Create output pins
        for(q 0 OUTPUTS-1
               out=sprintf(nil "WL%L" q)
               pin=ece432SchematicCreatePin(cvid out "output" x:y "R0")
        y=y+0.5
```

```
);; end for(q
    ;; Create inout pins
foreach(pinName list("vdd" "gnd")
  pin=ece432SchematicCreatePin(cvid pinName "inputOutput" x:y "R0")
  y=y+0.5
)
    ;; The coordinate of the origin point for the schematic
x=1
y=0
    ;; Create the decoder using tx-gates
    for(k 0 OUTPUTS-1
      ;; Define the input and output net name of the first inverter
      temp=k
      if(mod(temp 2)==1 then
            add0="A0"
      else
            add0="A0_"
  )
  temp=temp/2
      if(mod(temp 2)==1 then
            add1="A1"
      else
            add1="A1_"
  )
  temp=temp/2
      if(mod(temp 2)==1 then
            add2="A2"
      else
            add2="A2_"
  )
  temp=temp/2
      if(mod(temp 2)==1 then
            add3="A3"
      else
            add3="A3_"
```

```
)
temp=temp/2
    if(mod(temp 2)==1 then
          add4="A4"
    else
          add4="A4_"
)
temp=temp/2
    if(mod(temp 2)==1 then
          add5="A5"
    else
          add5="A5_"
)
temp=temp/2
    if(mod(temp 2)==1 then
          add6="A6"
    else
          add6="A6_"
)
    a=sprintf(nil "%s%s%s%s" add0 add1 add2 add3)
    b=sprintf(nil "%s%s%s" add4 add5 add6)
    out=sprintf(nil "WL%L" k)
    ;; Instantiate the kth tx-gate with ece432SchematicCreateInst()
       ↪ procedure,
    ;; which is defined in the file 'ece432Schematic.il'.
    ;; This procedure needs 10 paramters
    inst = ece432SchematicCreateInst(
                ;; Cell view id
                    cvid
                    ;; Name of the library containing the inverter
                       ↪ cell
                    libName
                ;; The tx-gate cell name
             "and_2"
                    ;; Cell view type, always be "symbol" here
```

```
        "symbol"
        ;; Instance name
    sprintf(nil "AND_%L" k)
        ;; The following list defines the connections of
           ↪ the left side pins
        ;; in the symbol view; For our inverter, 'in' is
           ↪ the only pin on the left.
        ;; The inverter pin 'in' is connected with the
           ↪ net 'inName'.
        ;; If there are multiple pins on the left side,
           ↪ you can use the below way to define.
        ;; list(list("in1" in1net)
        ;; list("in2" in2net))
        ;; If there is no pin on the left side, this
           ↪ parameter should be 'nil'.
        list(list("a" a)
                    list("b" b))
        ;; The following list defines the connections of
           ↪ the right side pins
        ;; in the symbol view; For our inverter, 'out'
           ↪ is the only pin on the right.
        list(list("out" out))
        ;; The following list defines the connections of
           ↪ the top side pins
        ;; in the symbol view; For our inverter, 'VDD'
           ↪ is the only pin on the top.
        list(list("vdd" "vdd"))
        ;; The following list defines the connections of
           ↪ the bottom side pins
        ;; in the symbol view; For our inverter, 'VSS'
           ↪ is the only pin on the bottom.
    list(list("gnd" "gnd"))
        ;; Location of the instance
x:y
```

```
                    ;; Rotation of the instance, such as "R0", "R90
                       ↪ ", "R180", "R270", "MX", "MY", ...
                    "R0"
                  );; end inst =
      y=y-2
    );; end for(k
    ;; The coordinate of the origin point for the schematic
x=-5
y=0
    ;; Create the decoder using tx-gates
    for(k 0 15
      ;; Define the input and output net name of the first inverter
      temp=k
      if(mod(temp 2)==1 then
            add0="A0"
      else
            add0="A0_"
  )
  temp=temp/2
      if(mod(temp 2)==1 then
            add1="A1"
      else
            add1="A1_"
  )
  temp=temp/2
      if(mod(temp 2)==1 then
            add2="A2"
      else
            add2="A2_"
  )
  temp=temp/2
      if(mod(temp 2)==1 then
            add3="A3"
      else
            add3="A3_"
```

```
)
    a=sprintf(nil "%s" add0)
    b=sprintf(nil "%s" add1)
    c=sprintf(nil "%s" add2)
    d=sprintf(nil "%s" add3)
    out=sprintf(nil "%s%s%s%s" add0 add1 add2 add3)
    ;; Instantiate the kth tx-gate with ece432SchematicCreateInst()
       ↪ procedure,
    ;; which is defined in the file 'ece432Schematic.il'.
    ;; This procedure needs 10 paramters
    inst = ece432SchematicCreateInst(
                ;; Cell view id
                    cvid
                    ;; Name of the library containing the inverter
                       ↪ cell
                    libName
                ;; The tx-gate cell name
            "and_4"
                    ;; Cell view type, always be "symbol" here
                    "symbol"
                    ;; Instance name
                sprintf(nil "AND4_%L" k)
                    ;; The following list defines the connections of
                       ↪ the left side pins
                    ;; in the symbol view; For our inverter, 'in' is
                       ↪ the only pin on the left.
                    ;; The inverter pin 'in' is connected with the
                       ↪ net 'inName'.
                    ;; If there are multiple pins on the left side,
                       ↪ you can use the below way to define.
                    ;; list(list("in1" in1net)
                    ;; list("in2" in2net))
                    ;; If there is no pin on the left side, this
                       ↪ parameter should be 'nil'.
                    list(list("a" a)
```

```
                                    list("b" b)
                                    list("c" c)
                                    list("d" d))
                        ;; The following list defines the connections of
                           ↪  the right side pins
                        ;; in the symbol view; For our inverter, 'out'
                           ↪ is the only pin on the right.
                        list(list("out" out))
                        ;; The following list defines the connections of
                           ↪  the top side pins
                        ;; in the symbol view; For our inverter, 'VDD'
                           ↪ is the only pin on the top.
                        list(list("vdd" "vdd"))
                        ;; The following list defines the connections of
                           ↪  the bottom side pins
                        ;; in the symbol view; For our inverter, 'VSS'
                           ↪ is the only pin on the bottom.
                    list(list("gnd" "gnd"))
                        ;; Location of the instance
                x:y
                        ;; Rotation of the instance, such as "R0", "R90
                           ↪ ", "R180", "R270", "MX", "MY", ...
                        "R0"
                      );; end inst =
      y=y-2
    );; end for(k
    ;; The coordinate of the origin point for the schematic
x=-8
y=0
    ;; Create the decoder using tx-gates
    for(k 0 7
      ;; Define the input and output net name of the first inverter
      temp=k
      if(mod(temp 2)==1 then
            add0="A4"
```

```
    else
          add0="A4_"
)
temp=temp/2
    if(mod(temp 2)==1 then
          add1="A5"
    else
          add1="A5_"
)
temp=temp/2
    if(mod(temp 2)==1 then
          add2="A6"
    else
          add2="A6_"
)
    a=sprintf(nil "%s" add0)
    b=sprintf(nil "%s" add1)
    c=sprintf(nil "%s" add2)
    out=sprintf(nil "%s%s%s" add0 add1 add2)
    ;; Instantiate the kth tx-gate with ece432SchematicCreateInst()
       ↪ procedure,
    ;; which is defined in the file 'ece432Schematic.il'.
    ;; This procedure needs 10 paramters
    inst = ece432SchematicCreateInst(
                ;; Cell view id
                    cvid
                    ;; Name of the library containing the inverter
                       ↪ cell
                    libName
                ;; The tx-gate cell name
              "and_3"
                    ;; Cell view type, always be "symbol" here
                    "symbol"
                    ;; Instance name
                sprintf(nil "AND3_%L" k)
```

```
        ;; The following list defines the connections of
           ↪  the left side pins
        ;; in the symbol view; For our inverter, 'in' is
           ↪  the only pin on the left.
        ;; The inverter pin 'in' is connected with the
           ↪ net 'inName'.
        ;; If there are multiple pins on the left side,
           ↪ you can use the below way to define.
        ;; list(list("in1" in1net)
        ;; list("in2" in2net))
        ;; If there is no pin on the left side, this
           ↪ parameter should be 'nil'.
        list(list("a" a)
                    list("b" b)
                    list("c" c))
        ;; The following list defines the connections of
           ↪  the right side pins
        ;; in the symbol view; For our inverter, 'out'
           ↪ is the only pin on the right.
        list(list("out" out))
        ;; The following list defines the connections of
           ↪  the top side pins
        ;; in the symbol view; For our inverter, 'VDD'
           ↪ is the only pin on the top.
        list(list("vdd" "vdd"))
        ;; The following list defines the connections of
           ↪  the bottom side pins
        ;; in the symbol view; For our inverter, 'VSS'
           ↪ is the only pin on the bottom.
    list(list("gnd" "gnd"))
        ;; Location of the instance
x:y
        ;; Rotation of the instance, such as "R0", "R90
           ↪ ", "R180", "R270", "MX", "MY", ...
        "R0"
```

```
                    );; end inst =
        y=y-2
      );; end for(k
;; ******************************Address inverters
   ↪ ***************************
      ;; The coordinate of the origin point for the schematic
   x=-12
   y=0
      ;; Create the decoder using tx-gates
      for(k 0 6
        in=sprintf(nil "A%L" k)
        out=sprintf(nil "A%L_" k)
        ;; Instantiate the kth tx-gate with ece432SchematicCreateInst()
           ↪ procedure,
        ;; which is defined in the file 'ece432Schematic.il'.
        ;; This procedure needs 10 paramters
        inst = ece432SchematicCreateInst(
                    ;; Cell view id
                       cvid
                       ;; Name of the library containing the inverter
                          ↪ cell
                       libName
                    ;; The tx-gate cell name
                 "inv"
                       ;; Cell view type, always be "symbol" here
                       "symbol"
                       ;; Instance name
                    sprintf(nil "INVERTER_%L" k)
                       ;; The following list defines the connections of
                          ↪ the left side pins
                       ;; in the symbol view; For our inverter, 'in' is
                          ↪ the only pin on the left.
                       ;; The inverter pin 'in' is connected with the
                          ↪ net 'inName'.
```

```
                    ;; If there are multiple pins on the left side,
                       ↪ you can use the below way to define.
                    ;; list(list("in1" in1net)
                    ;; list("in2" in2net))
                    ;; If there is no pin on the left side, this
                       ↪ parameter should be 'nil'.
                    list(list("in" in))
                    ;; The following list defines the connections of
                       ↪  the right side pins
                    ;; in the symbol view; For our inverter, 'out'
                       ↪ is the only pin on the right.
                    list(list("out" out))
                    ;; The following list defines the connections of
                       ↪  the top side pins
                    ;; in the symbol view; For our inverter, 'VDD'
                       ↪ is the only pin on the top.
                    list(list("vdd" "vdd"))
                    ;; The following list defines the connections of
                       ↪  the bottom side pins
                    ;; in the symbol view; For our inverter, 'VSS'
                       ↪ is the only pin on the bottom.
                list(list("gnd" "gnd"))
                    ;; Location of the instance
            x:y
                    ;; Rotation of the instance, such as "R0", "R90
                       ↪ ", "R180", "R270", "MX", "MY", ...
                    "R0"
                  );; end inst =
      y=y-2
    );; end for(k
    ;; Check, save and close the cell view
    schCheck(cvid) ;; check the schematic connectivity
dbSave(cvid)
dbClose(cvid)
return(t)
```

```
  );; end prog
);; end procedure
/*====================================================================
 'let' is the main entrance of the skill program.
====================================================================*/
let((WIDTH libName cellName) ;; N, libName and cellName are variables
  WIDTH = 7
  OUTPUTS = 74
  ;; The name of the library we will put the new cell in
  libName="sumitfpga"
  sprintf(cellName "RowDecoder")
  DecoderSchematic(libName cellName WIDTH OUTPUTS)
  printf("=== Cell %L Schematic has been created! ===\n", cellName)
) ;;end let
```