2011

Cost and performance modeling of the MU-Decoder

Raghavendra Kongari
Louisiana State University and Agricultural and Mechanical College, rkonga1@tigers.lsu.edu

COST AND PERFORMANCE MODELING OF THE MU-DECODER

A Thesis
Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering

in
The Department of Electrical and Computer Engineering

by
Raghavendra Kongari
B.E. in Electronics and Communication Engineering,
Osmania University, India, 2007
May 2011

Acknowledgements

I am indebted to my advisor, Dr. Ramachandran Vaidyanathan, for his exemplary patience, guidance, and support. He taught me the approaches to solve research problems at every obstacle on my way towards completing my thesis. I would also like to extend my gratitude to my committee members, Dr. Bahadir K. Gunturk and Dr. Alexander Skavantzos, for their valuable suggestions and kind support. I also like to thank Dr. Ashok Srivastava for permitting me to use his lab facilities.

I also want to convey my sincere gratitude to Dr. J. Matthew Fannin, and Department of Electrical Engineering, for supporting me financially and making me to concentrate on completion of my thesis. Dr. J. Matthew Fannin has also helped me to build my professional work experience and introduced me to several conference meetings.

I want to thank all my friends who assisted me in my endeavors, with a special thanks to Leeladhar, Bhargav, Upender, Rajkiran, Naga S. Korivi, Pradeep, Dilip, Kalyan, Venu, and Phaneendra for their invaluable support rendered during the progress of this thesis.

I wish to endow my earnest gratitude to my father, Anjaiah Kongari, and my mother, Bhagyamma Kongari, who believed in me and have been through all my rough times. I also want to express special thanks to my entire family for their affection, support and compassion. Finally, I dedicate this thesis to my parents, Anjaiah and Bhagyamma, without whose blessings, this would not have been a reality.

Table of Contents

List of Tables

List of Figures

Abstract

In this thesis we study the implementation details of the MU-Decoders, a recently proposed hardware module that has been theoretically shown to be superior to other methods for generating subsets of large sets. Our study confirms this advantage.

Specifically, we compare the performance of implementations of the LUT-Decoder (the most common configurable decoder) to the MU-Decoder. We show that for while the LUT-Decoder is slightly better than the MU-Decoder for arbitrary (and artifical) inputs, for a large class of inputs called totally ordered subsets, that have practical significance, the MU-Decoder is vastly superior in area than the LUT-Decoder. In terms of delay and power too, the MU-Decoder performs better than the LUT-Decoder.

This work is based on a series of time-optimized and area-optimized implementations of key building blocks of the MU-Decoder that help construct models for both the LUT-Decoder and the MU-Decoder. These models serve to predict the delay, area and power of these decoders at sizes that may not be practical to implement in an academic setting.

As part of the work, multiplexers, one-hot decoders, multicast hard-wiring, and memory modules are implemented and modeled. These are all commonly used elements of digital systems. Therefore this work may also be of independent interest, beyond MU-Decoders.

Chapter 1

Introduction

Technological advances have dramatically reduced the speed of digital systems, particularly computing systems (Moore's Law). However input/output bandwidth has not increased at the same pace. Jordan [9] illustrates this with an example of Intel processors from 1976 to 2006 for which the number of transistors per chip has increased by a factor of 20,000, while the number of pins has increased only by a factor of 30. Thus, pin-limitation is an important issue that could seriously affect the performance of modern systems.

One area where pin-limitation has an important effect is in dynamically reconfigurable systems such as Field Programmable Gate Arrays (FPGAs) [7, 27, 30, 31]. Broadly speaking, an FPGA consists of an array of configurable elements (logic blocks) whose internal and external connections and logic can be configured to suit the problem at hand. Typically, there are hundreds of thousands of such logic blocks in an FPGA, each requiring multiple configuration bits. For example the Xilinx 7-series FPGA (XC7V20002) has 1.9 million configurable logic blocks [31].

Typically one could configure the FPGA partially [7, 27, 28, 30], altering only one part, while leaving the rest unchanged and free to operate. A key part of this partial reconfiguration is selecting the elements to be reconfigured. A focused selection of only those elements requiring reconfiguration entails sending in few configuration bits, however it also increases the number of elements from which this selection is to be made. Consider an FPGA with C configurable logic blocks. Let each block require b bits to configure. Group these configurable blocks into F groups called "frames," each with $\frac{C}{F}$ blocks. A *frame* is the smallest unit that can be separately addressed for reconfiguration. The role of a "configurable decoder" (that we study in this thesis) is to select a subset of these F frames to reconfigure. Here the decoder can be viewed as a subset generator. If c of the C configurable blocks distributed over f of the F frames need to be reconfigured, then $\frac{cF}{C} \leq f \leq c$. The number of bits B needed to configure these f frames is $fb\left(\frac{C}{F}\right)$, so $cb \leq B \leq cb\left(\frac{C}{F}\right)$. To keep B small it is essential to make $\frac{C}{F}$ small, or F large. This implies that the set of frames to chose from is now large and the decoder needs to produce subsets of large sets. Selecting from a large set,

though beneficial computationally, can lose this benefit to the cost of the selecting hardware itself.

As argued in Jordan [9] this problem can be circumvented with a configurable decoder (or subset generator[1]) with low cost and high flexibility. In this thesis we study the implementation of one such decoder called the MU-Decoder and validate the assertion [9, 10] that a MU-Decoder performs better than a LUT-Decoder for structured subsets.

Current solutions for selection of frames that use a one-hot decoder are too slow for dynamic reconfiguration. Using a LUT based decoder (described below) is flexible but too expensive in area. The MU-Decoder has much of the flexibility of the LUT-Decoder but can exploit problem structure to considerably reduce the area cost.

The simplest configurable decoder is a LUT-Decoder, which is simply a $2^m \times n$ look-up table, or a 2^m location memory of word size n. Neither does it have any constraints on the contents of the memory, nor can it exploit them, if constraints are placed. On the other hand a MU-Decoder [10] can exploit constraints of the subsets to be generated and offers, in a sense, more flexibility than the LUT-Decoder for practical situations (where subsets are non-arbitrary). In this thesis we perform a detailed modeling of MU-Decoder implementations and establish this advantage. Previous work on the MU-Decoder [9, 10], were primarily at the theoretical level that established complexity advantages without considering effects of implementation, such as complexity constants, wire area, delay, and parasitic effects in low feature size technologies [16]; we use 45 nm technology for this work. Unlike earlier results [9], our work allows the MU-Decoder and LUT-Decoder to be compared without the assumption that subsets are fixed before the decoder is manufactured.

This work has several contributions. We first identify a few key building blocks of the MU-Decoder. Next, we express each of these building blocks in terms of its "granularity" that affords a level of flexibility in exploring the design space systematically. For each selected design, we synthesize, place and route the design using the Cadence Suite of CAD tools [4] on a virtual chip to estimate its delay, area and power. We use these experimental observations to derive model equations for the MU-Decoder building blocks. These, in turn, lead to models of the LUT-Decoder and the MU-Decoder. These models are used to establish the advantage of the MU-Decoder over the LUT-Decoder for a class of subsets called totally-ordered subsets.

Four key parameters of configurable decoders that are relevant to this thesis are (i) n, the number of elements (or frames) in a set $\mathbb{Z}_n = \{0, 1, \cdots, n-1\}$, whose subsets are being generated, (ii) $\mathcal{S}$, the set of subsets of $\mathbb{Z}_n$ being generated, (iii) $\ell = |\mathcal{S}|$, the number of subsets being generated and (iv) $A^D(n, \mathcal{S}, \ell)$ the area of the configurable decoder "D"; here $D \in \{L, M\}$, for LUT- and MU-Decoder, represents the decoder whose area is being considered. We compare the MU-Decoder and LUT-Decoder for a fixed $n, \mathcal{S}$ and ℓ and use the ratio

[1] It has been shown [9, 10] that a 2^m-to-n decoder is a combinational circuit that generates a set of subsets of $\mathbb{Z}_n = \{0, 1, \cdots, n-1\}$.

$\rho(n, \mathcal{S}, \ell) = \frac{A^L(n,\mathcal{S},\ell)}{A^M(n,\mathcal{S},\ell)}$ as a measure of the performance. If $\rho(n, \mathcal{S}, \ell) > 1$ then the MU-Decoder has lower area, otherwise the LUT-Decoder has lower area. Since the same technology (45 nm) is used for both implementations, the ratio ρ is more meaningful than absolute areas, and extends (in large measure) to future technologies as well.

We show that when $\mathcal{S}$ is an arbitrary (unconstrained) set of subsets then $\rho(n, \mathcal{S}, \ell) < 1$, showing the LUT-Decoder to be better. For example, when $\ell = n$ and $n = 5000$, then $\rho = \frac{1}{1.8}$, whereas when n increases to 10^6, ρ decreases to $\frac{1}{3.4}$. This behavior is less pronounced for larger ℓ. For example when $\ell = n^2$ and $n = 10^6$, we have $\rho = \frac{1}{1.1}$.

While these result samples seem to indicate that the LUT-Decoder may be better than the MU-Decoder, this is not the case in practice where subsets are not as arbitrary as the above results assume. In fact, the subsets generated for use in most algorithms are related to each other. For example in reduction (or other tree based algorithms) the subsets could form a total order (in which all subsets are pairwise comparable). We also applied our model to a MU-Decoder geared to generate totally ordered subsets. For the same data illustrated above we have the following for the totally-ordered subsets. When $\ell = n$ and $n = 5000$, then $\rho = 3.5$, whereas when n increases to 10^6, ρ increases to 33. In simple terms, the last case translates to, for subsets of a 10^6 element set, the MU-Decoder is 33 times lower in area than the LUT-Decoder. As we noted earlier for current FPGAs, a million logic blocks to choose from is not unreasonable. For $\ell = n^2$ and $n = 10^6$, we have $\rho = 110$. Not only is the MU-Decoder more economical in area, but it also is better by a larger factor than the LUT-Decoder was for unconstrained subsets.

Since the MU-Decoder building blocks (one-hot decoder, multiplexer, LUT, and broadcast hardwiring) are commonly used digital logic blocks, the methodology used in this thesis to implement the building blocks along with the building block models may also be of independent interest.

The idea of validating theoretical results (that we undertake in this thesis) is not new (for example [11, 13, 19], illustrating a wide range of models, applications, and levels of abstraction). The building blocks for the MU-Decoder are multiplexers, one-hot decoders, LUTs (SRAM arrays) and hardwiring to broadcast z bits to zn places. These are standard "folklore" elements most of which are taught in the typical first digital logic courses [29]. To our knowledge, there is no separate recent study of these building blocks. Any earlier studies are for older technologies (above 1 μm) that ignore effects in newer technologies (45 nm or below). Most recent simulations and modeling of combinational logic is in the areas of power management and fault-tolerance [11, 18, 19, 21]. We need to study the MU-Decoder blocks separately to ensure that the same technological constraints are applied to all blocks. For the LUT-Decoder (SRAM array that had to be implemented as a sequential circuit) we used a hand made cell layout and the CACTI tool [3] to further adjust our findings to bring them in line with actual implementations that use the standard 6 transistor-based cells [18] rather than latches.

In the Chapter 2, we discuss some preliminary ideas including the MU-Decoder structure.

Chapter 3 details the methodology used to implement and model the MU-Decoder building blocks. One aspect of the modeling process is to theoretically estimate the growth terms in the models for the building blocks (Chapter 4). Chapter 5 derives performance equations for the building blocks and Chapter 6 puts them together and comparatively studies the LUT- and MU-Decoders. Finally in Chapter 7, we summarize our results and identify directions for future research.

Chapter 2

Preliminaries

This chapter introduces the main concepts that needed for the work in subsequent chapters. We start with a set representation of binary sequences in Section 2.1, followed by a brief introduction (Section 2.2) to the "Mapping Unit Decoder or MU-Decoder," its structure and its key elements. Then in Section 2.3, we proceed to explain a top-level design flow of implementation, which provides the basis for work in the Chapter 3.

2.1 Binary Sequences as Sets

Any n-bit binary sequence can be viewed as a subset of $\mathbb{Z}_n = \{0, 1, \cdots, n-1\}$. This view of an n-bit sequence will be useful in describing decoders, in general, and the MU-Decoder in particular. Let each of the n elements of $\mathbb{Z}_n$, represent the position of a bit in an n-bit binary sequence $b_{n-1}\cdots b_1 b_0$, where $b_i \in \{0, 1\}$. Let $B \subseteq \mathbb{Z}_n$. Subset B corresponds to an n-bit binary sequence $b_{n-1}\cdots b_1 b_0$, such that for $0 \leq j < n$, bit $b_j = 1$ iff $j \in B$. For $n = 3$, Table 2.1 shows an example of all $2^n = 8$ possible subsets and their corresponding 3-bit binary sequences.

Table 2.1: Sets corresponding to 3-bit binary sequences

Name	Binary Sequence	Set
B_0	000	{ }
B_1	001	{0}
B_2	010	{1}
B_3	011	{1,0}
B_4	100	{2}
B_5	101	{2,0}
B_6	110	{2,1}
B_7	111	{2,1,0}

2.2 MU Decoder

In general, a *decoder* is a hardware module that maps a small number m of inputs to a larger number n of outputs. In other words, the decoder generates an n-bit output sequence for a given m-bit input sequence, where $m \ll n$. In the notation developed in Section 2.1, an output of an m-to-n decoder is a subset of $\mathbb{Z}_n = \{0, 1, \cdots, n-1\}$. The set of all possible (at most 2^m) outputs of a decoder is the set of subsets $P = \{B_i \subseteq \mathbb{Z}_n : 0 \leq i < 2^m\}$, where B_i is the subset corresponding to input sequence of binary value i. If this mapping between the input and output sequences is fixed, that is if P is fixed, then the decoder is called a *fixed decoder*. If the decoder hardware allows P to be altered then the decoder is said to be a *configurable decoder*; that is, a configurable decoder can have different P's at different times (an example appears later in this section).

One of the most commonly used fixed decoders is the log$n$-to-n *one-hot decoder*. In-fact, the term "decoder" normally refers to a one-hot decoder. In the one-hot decoder, the output sequence has only one active output. Thus for a one-hot decoder, $P = \{\{0\}, \{1\}, \cdots, \{n-1\}\}$ is the set of all one-element subsets of $\mathbb{Z}_n$. Other fixed decoders are possible as well.

An m-to-n look-up-table (LUT)[1] is a memory of 2^m locations, each n-bits wide. This memory can be loaded (configured) with any set of $\min(2^n, 2^m)$ distinct binary sequences. We can consider a LUT as one of the most flexible and simplest examples of an m-to-n configurable decoder, where each location can be configured arbitrarily and independently of others. That is, set P is unconstrained other than to size $\min(2^n, 2^m)$. As an example, suppose we have a $2^2 \times 5$ LUT, that is to be used as a 2-to-5 decoder. Then in one case one could load binary strings $00000, 00001, 00010, 00011$ in the LUT for which $P = \{\phi, \{0\}, \{1\}, \{2, 3\}\}$. Here an input of 00 will produce the empty set ϕ, for example. On the other hand at a different time, one could load strings $00110, 11001, 11111, 01101$ for which $P = \{\{2, 1\}, \{4, 3, 0\}, \{4, 3, 2, 1, 0\}, \{3, 2, 0\}\}$. Now the same input 00 produces subset $\{2, 1\}$.

Jordan and Vaidyanathan [9, 10] have proposed a new configurable decoder called the *Mapping Unit Decoder* (MU-Decoder) that is the main subject of study of this thesis. The basic structure of the MU-Decoder is shown in Figure 2.1.

On whole the MU-Decoder is an m-to-n decoder with y control lines; generally $y \ll m \ll n$. As Figure 2.1 shows, the MU-Decoder structure has two main stages, the first of which is an m-to-z LUT. We now describe the second stage called the *mapping unit*. The mapping unit is a module that multicasts the z-bit (source word Z) input to the n-bit output N. The mapping unit structure consists of a fixed hardwiring, a selector unit and multiplexers connected as shown in the Figure 2.2. The y-bit control lines (or selector word Y) is used to determine the nature of the multicast. The selector unit uses the selector word to control the multiplexers. Each multiplexer can select from z possible inputs, bits multicast by the

[1] The LUT described here is usually called a $2^m \times n$ LUT.

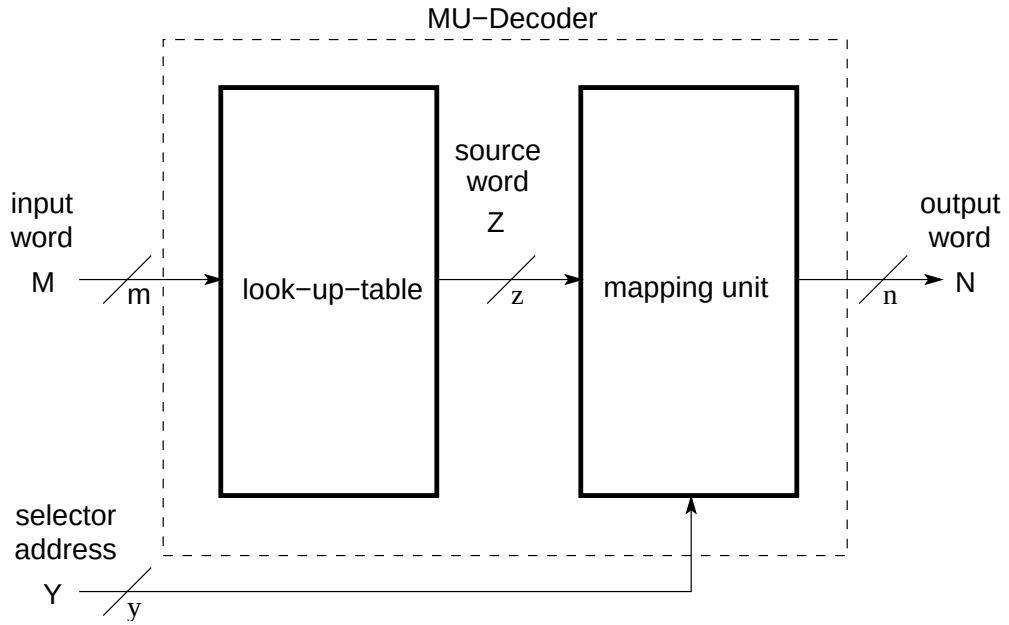


Figure 2.1: A m-to-n MU-Decoder block structure where signals are named (in upper case) corresponding to their widths (in lower case).

fixed hardwiring from a source word to the multiplexers.

The function of the MU-Decoder is described by Jordan and Vaidyanathan [9, 10]. Here we primarily consider its structure from the point of view of distilling its building blocks into key hardware elements and their implementation. As noted earlier, the first stage is simply a LUT. The second stage has multiplexers and the selector unit, which again is a (collection of) LUT(s) [9]. The LUT itself consist of (one or two) one-hot decoder(s) followed by an array of memory cells. Thus the MU-Decoder, and other conventional decoders of interest in this work (one-hot decoder and LUT) can all be studied by examining the following core modules.

- x-to-xn multicast hardwiring
- 2^x-to-1 multiplexer
- 2^x-to-w LUT
  - Array of Static Random Access Memory (SRAM) cells
  - Row and Column one-hot decoders (x-to-2^x)

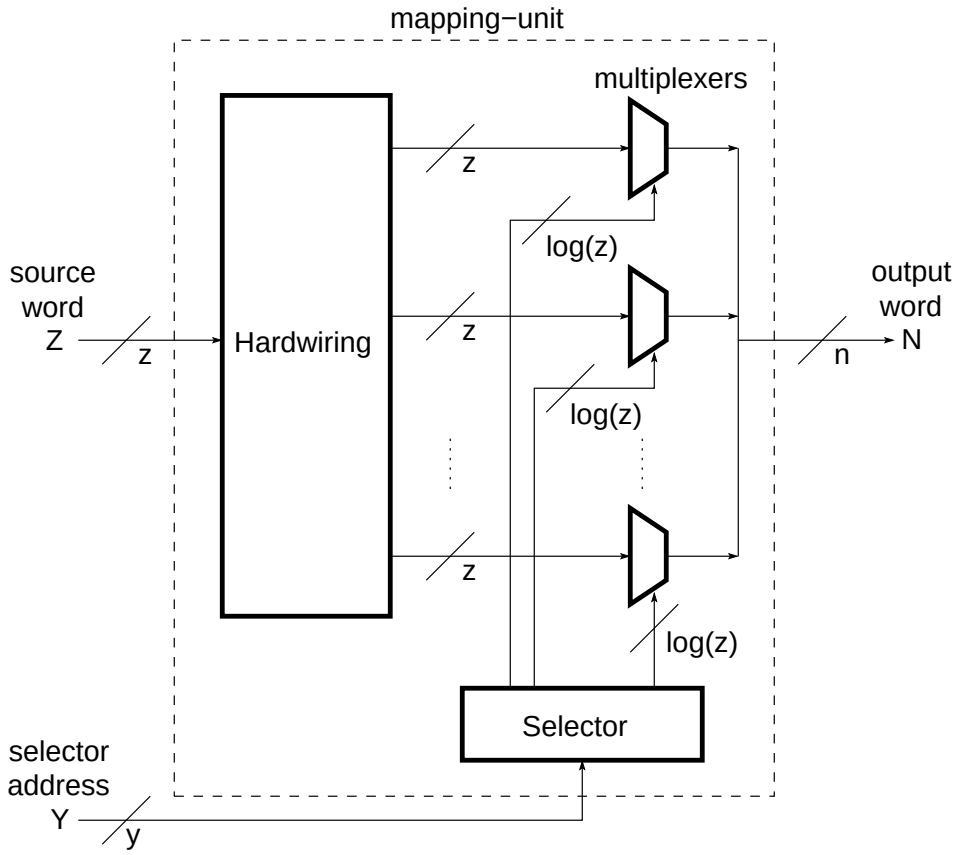


Figure 2.2: Mapping unit structure.

2.3 Design Flow

As noted in Section 2.2, to study and model the MU-Decoder, one needs to understand the implementation of multicast hardwiring, multiplexers, one-hot decoders and LUTs (including one-bit memory cells). This section discusses the Electronic Design Automation (EDA) implementation steps (or design flow) used, all the way from module description to its chip layout. The design flow is broadly classified into three phases, (a) design entry and functional verification, (b) synthesis, and (c) place and route, as shown in Figure 2.3 [5] (with details in Figure 2.4). For these phases we use the Digital Integrated Circuits Bundle of the University Program Software Selection from the Cadence [4] (henceforth called the *Cadence Tools* in this thesis).

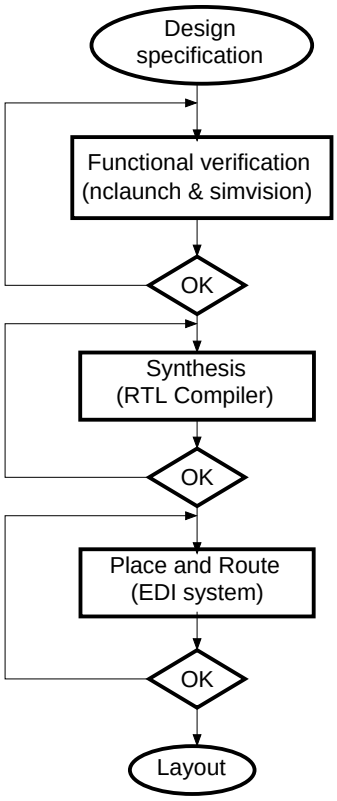


Figure 2.3: Block level digital implementation design flow.

2.3.1 Design Entry and Functional Verification

A design specification for a hardware module relates its inputs to outputs. For example, in a log$n$-to-n one-hot decoder, the design specification is equivalent to stating that an input of binary value $0 \leq i < n$ produces output subset $\{i\}$, where $\{i\}$ denotes an n-bit binary sequence with only bit i being 1. One common method of specifying a design is to use a hardware description language (HDL). In our case Verilog HDL [5, 8] has been used. As an example, the Verilog code for a one-hot decoder is shown in Figure 2.5. We will use this as a running example to illustrate ideas in this section.

To test the functionality of the Verilog description we create a Verilog test-bench [5] that generates (in our case) all possible input patterns and verifies the output. The tools *NC-Launch* [14] (a compiler and elaborator), and SimVision [20] (a waveform viewer), both included in the Cadence Tools, are used for functional verification as shown in Figure 2.6. As a part of the verification the Verilog description is corrected, if needed, and the process is repeated until a correct code results. This phase of the flow ensures a functionally verified Verilog specification of the module which is used in the next synthesis phase.

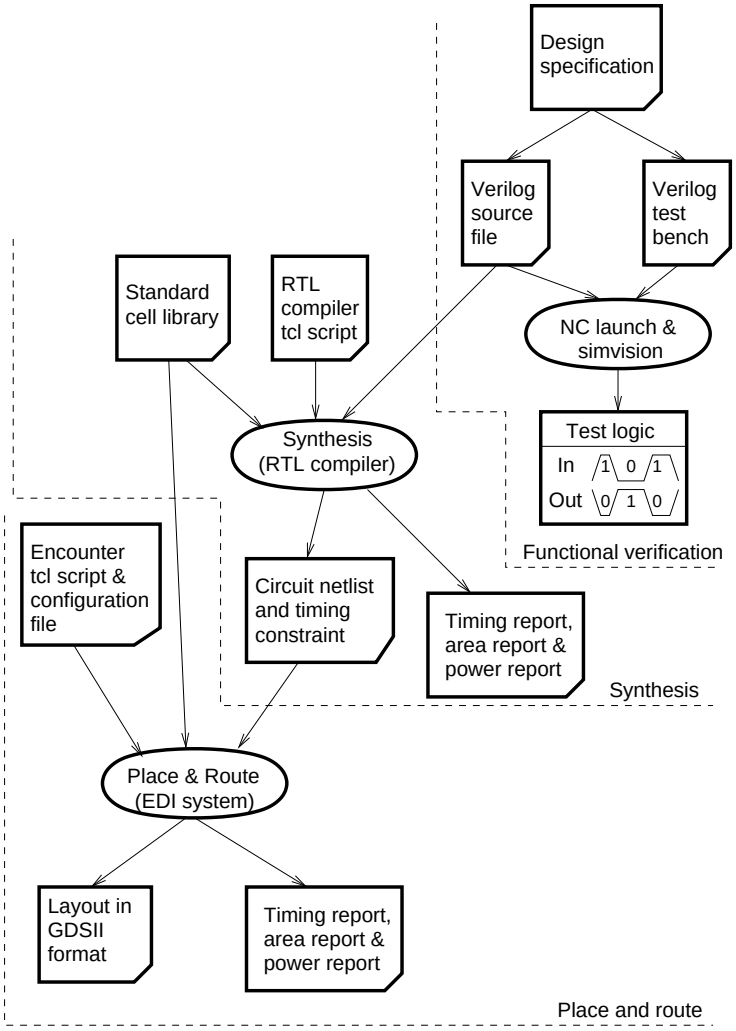


Figure 2.4: Detailed digital implemenatation design flow.

2.3.2 Synthesis

A given hardware module specification can be realized in many different ways. Synthesis [2] selects a suitable realization for the specified constraints and parameters. Synthesis generates a gate-level netlist, which is a schematic of circuit elements (gates for the most part here) and their interconnection. A gate itself can be realized in many ways according to the technology used and constraints such as fan-in, fan-out, speed, area and power. A set of commonly used gates realizations are made available as a standard-cell library. The libraries we used are from North Carolina State University (NCSU) and are called the Cadence Design Kit (cdk) and the Process Design Kit (pdk) [15]. These standard-cell libraries are avialable for free public download. In this work we used 45 nm technology standard-cell library available as

```
module dec_2_to_4_behav(Y, U, En);
    input [1:0] U;
    input En;
    output [3:0] Y;
    reg [3:0] Y;
    always @(U or En)
    begin
        if (En == 1 && U == 0) Y = 4'b0001; else
        if (En == 1 && U == 1) Y = 4'b0010; else
        if (En == 1 && U == 2) Y = 4'b0100; else
        if (En == 1 && U == 3) Y = 4'b1000; else
        if (En == 0) Y = 4'b0000; else Y = 4'bx;
    end
endmodule
```

Figure 2.5: Example Verilog code for the 2-to-4 one-hot decoder.

FreePDK45 [15]. Figure 2.7 shows a broad outline of the synthesis phase.

The Cadence Tools includes a synthesizer called the *RTL Compiler* [24] which performs the synthesis of the hardware module described in Verilog by using modules that are either user defined or from the standard-cell library. The RTL compiler performs the synthesis by executing commands from a script written in the *Tool Command Language* (tcl) script [23] to generate the following: (a) synthesized netlist, (b) estimated time of operation, (c) estimated area, (d) estimated power consumption and (e) timing constraints for all signals in *Synopsis Design Constraint* (sdc) format. An associated graphical user interface (GUI) window can also be used to view the synthesized circuit but cannot be used to perform the synthesis [24] itself. The schematic view of the synthesized circuit of the 2×2^2 one-hot decoder example described in Figure 2.5 is shown in Figure 2.8.

Timing: The tcl script that drives synthesis by the RTL Compiler specifies the technology library to use, Verilog module to synthesize, and timing constraints for the realization. We elaborate on the timing constraints now, as it plays an important part in this work. Let a module input to the RTL Compiler have a timing constraint of T_m; that is, we require the synthesized module (or a part of it, in general) to work with a delay of at most T_m. Suppose the RTL Compiler is unable to synthesize the module within the specified timing constraint, then it estimates a time $T_e > T_m$ for which a synthesis (with timing violation) may be possible. This is specified as a negative slack of $T_m - T_e$. This slack is a hint to the user that perhaps a constraint of T_e may work. However, it is not a guarantee that a constraint of T_e will work.

On the other hand, if the specified constraint T_m is quite loose and a synthesis in time

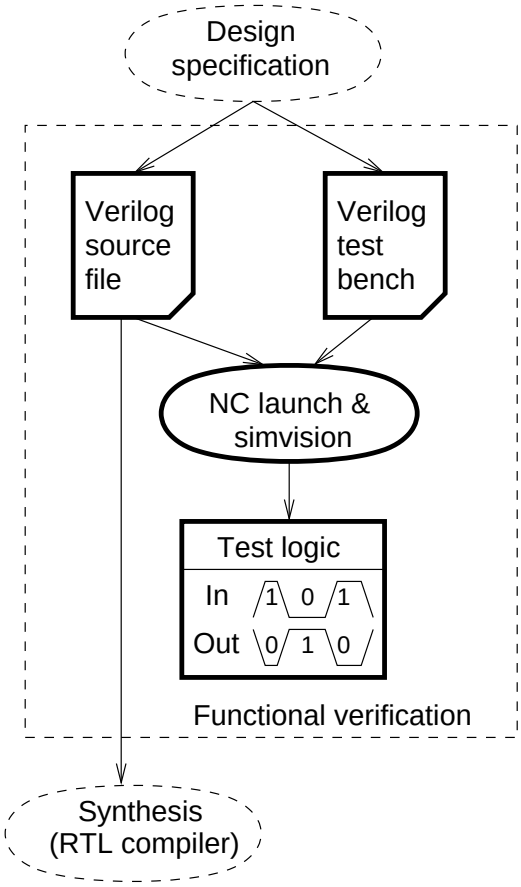


Figure 2.6: Functional verification design flow.

$T_s < T_m$ is possible, then the RTL Compiler generates a synthesized netlist without any timing violation and a positive slack of $T_m - T_s$. Here too, setting T_m to T_s is not a guarantee of a successful synthesis (without timing violation). Again it is only a hint. Hence we perform an iterative process to determine the best possible minimum timing constraint as described in the Section 3.2.1.

2.3.3 Place and Route (PAR)

The process of making a physical circuit from a synthesized netlist is known as *Place and Route* (PAR) [2]. This process invloves placing circuit elements (such as standard-cells) in some physical location on the layout, and routing interconnects between cells as described in the netlist. We use the *Encounter Digital Implementation* (EDI) System [6], again part of the Cadence Tools, for this process. This requires the standard-cell libraries, synthesized netlist and timing constraints in sdc format as shown in Figure 2.9.

Unlike the RTL compiler that cannot work through a GUI [24], the EDI tool can be driven both through a GUI and by tcl scripts [6]. However, to automate the flow for all of our designs, we use the tcl script approach. This approach requires two basic files to perform the place and route process (see Figure 2.9). The first is the *configuration file* that primarily specifies the design netlist, standard-cell library and timing constraints. The second, a tcl script, is needed to specify the commands for the place and route process and floorplan of the circuit layout, including aspect ratio, core utilization factor, and power planning. It also performs other operations such as clock tree synthesis and chip fabrication details that are not useful for this work and are not discussed further.

Like the synthesis phase, the place and route phase also produces data on delay, area and power consumption associated with the circuit implemented. Unlike the synthesis phase that focuses on selecting a good circuit, however, this phase deals with implementing a fixed circuit and accounts for additional details such as parasitic elements due to resistors and capacitors from the interconnects and cells. This phase produces a much more realistic estimate of delay and cost by physically laying cells out and considering the effect of routing in estimating the area. The power consumption is also estimated more accurately.

Unlike the synthesis phase that requires several iterations to arrive an "optimal" circuit for the given constraints, the EDI system requires at most 3 iterations to converge on the design. Results and additional details appear in Section 3.2.2. Figure 2.10 shows the layout of the example schematic of 2×2^2 one-hot decoder in Figure 2.8

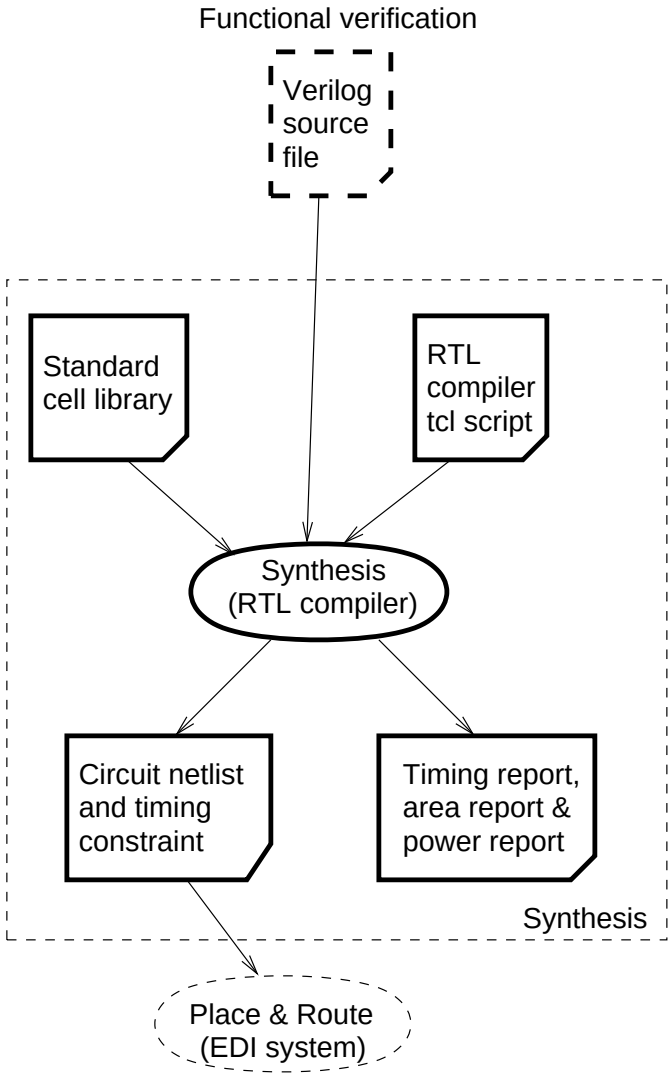


Figure 2.7: Synthesis flow.

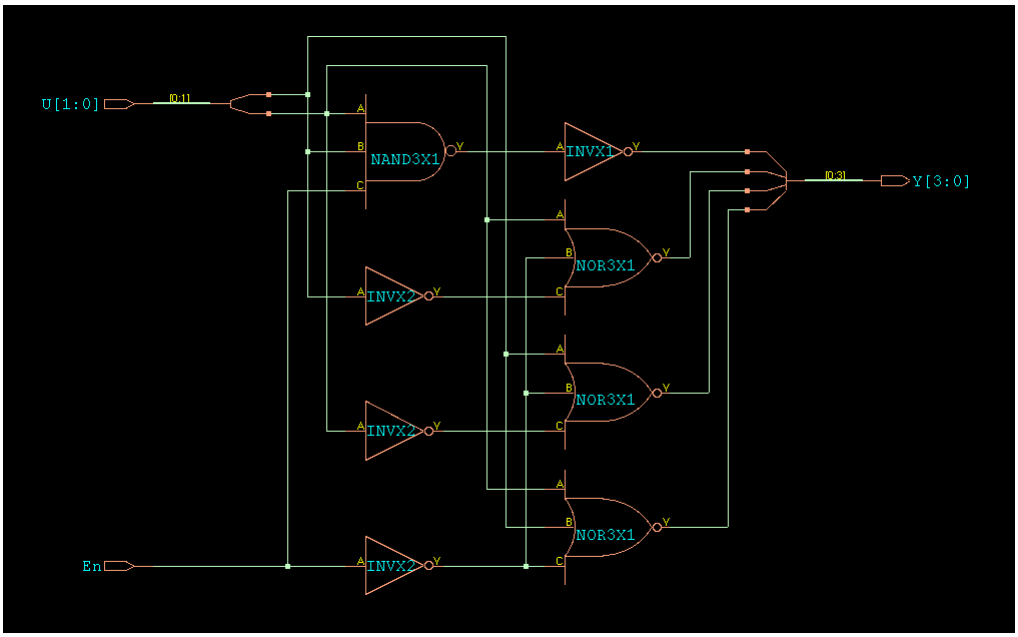


Figure 2.8: Example schematic for the 2-to-4 one-hot decoder.

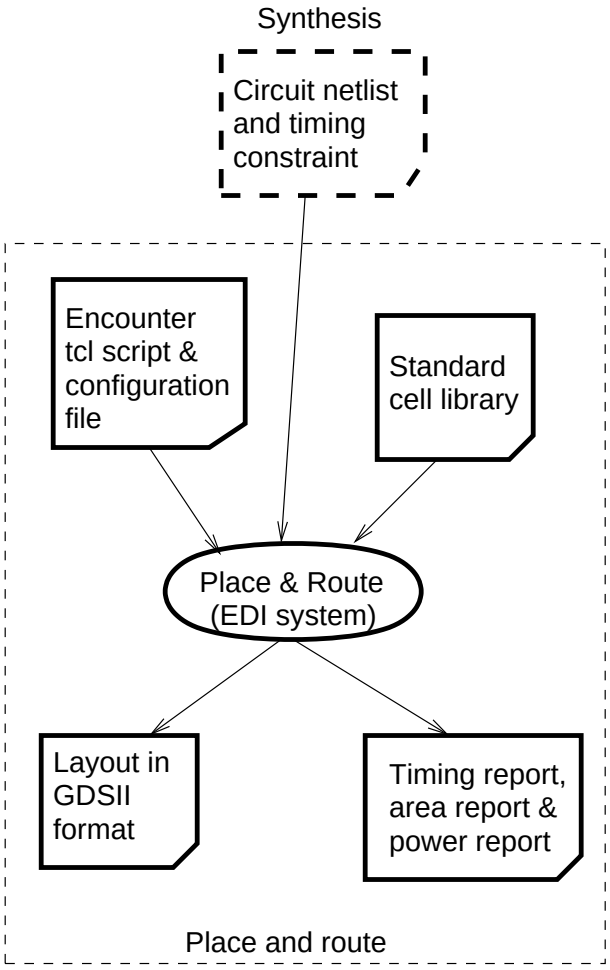


Figure 2.9: PAR design flow.

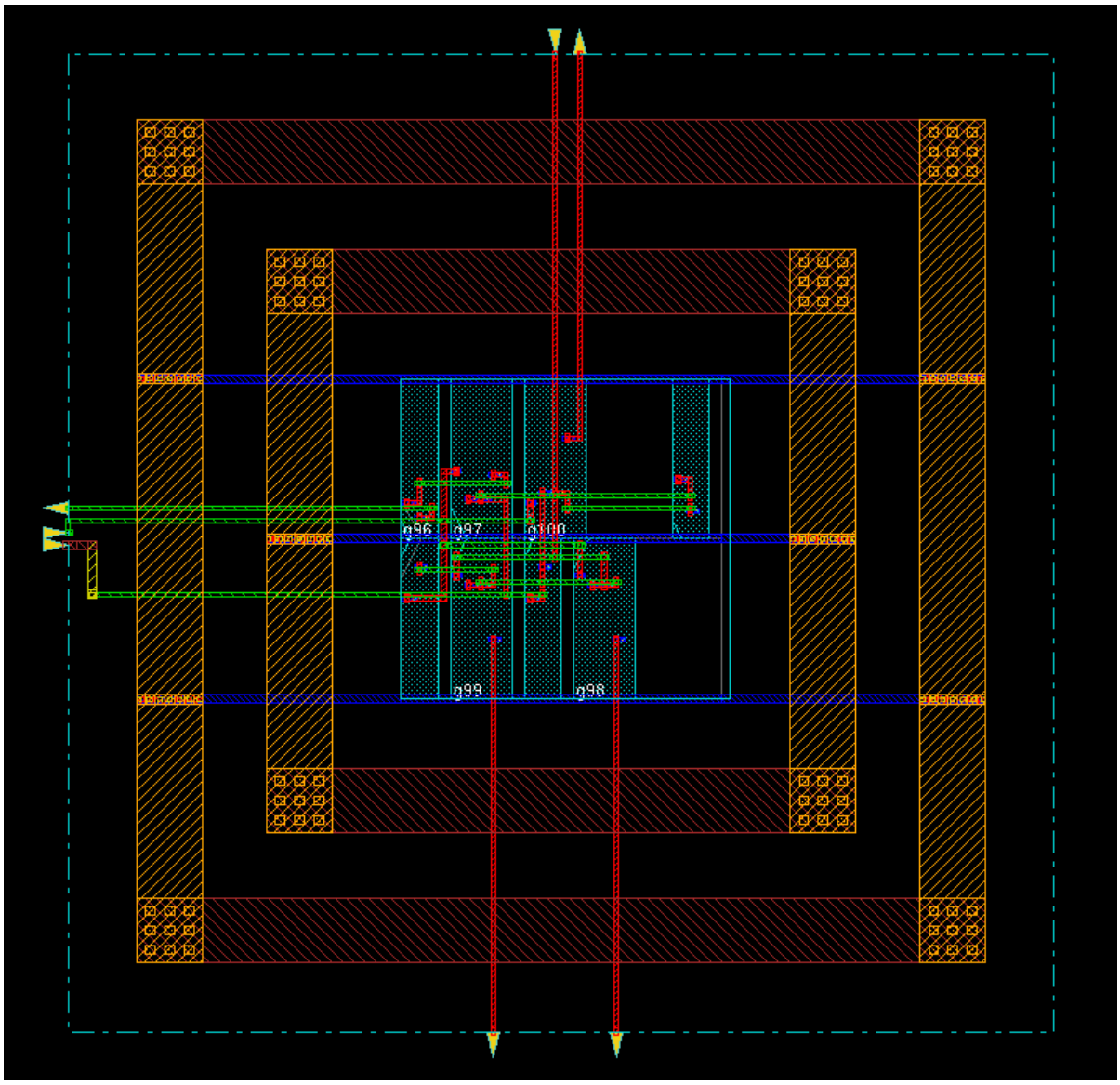


Figure 2.10: Example layout for the 2-to-4 one-hot decoder.

Chapter 3

Methodology

As noted in Section 2.2 in order to model the MU-Decoder we need to study and model four basic modules, one-hot decoder, multiplexer, multicast hardwiring, and LUT. Since there are many different ways to implement each of these modules, we need a systematic approach that captures these implementations. In this chapter we provide the details of the methods and steps used towards this end. This involves (a) identifying a generic way of expressing a module's structure (module representation), (b) examining modules of different *sizes* for each module representation, (c) selecting an approach for implementing a module and (d) modeling the module's cost/performance by equations.

In the next section we provide an overall view of the methodology and subsequent sections (Section 3.2 and 3.3) detail parts of the methodology.

3.1 Granularity-Based Module Representation

Three of the modules (multiplexer, one-hot decoder and LUT) have different methods of implementation with different levels of submodule definition. In particularly, a Verilog specification of a hardware module can be described in many different ways such as behavioral description, structural description, gate-level description, and so on. We need a systematic method of module description that lends itself to automation, without which it is not possible to cover the large design space in reasonable time. We derive a partially structural and a partially behavioral description of the basic modules of the MU-Decoder. We define a term, *granularity*, for each of those modules; that describes the level of structural specification of that module. In this section we briefly describe the way we approach a *granularity-based* representation of a multiplexer, one-hot decoder and LUT. Details of these representation appears in Section 5.1.1 , Section 5.2.1 and Section 5.4.1.

Multiplexer: A 2^x-to-1 multiplexer can either be constructed at the gate-level (resulting from a totally behavioral specification) or, at the other extreme, in terms of two (2^{x-1})-to-1 multiplexers. There are many shades in between. Here, granularity indicates the size of the building blocks used to make the 2^x-to-1 multiplexer. These building blocks can be at the gate level or in terms of (2^{x-1})-to-1 multiplexers or something else in between.

One-Hot Decoder: This is similar to a multiplexer in the sense that a x-to-2^x one-hot decoder can be built out of smaller one-hot decoders. Again the granularity is reflective of the building block size.

LUT: Here granularity has a slightly different meaning. A $2^x \times w$ LUT can be implemented to treat the memory as a $2^x \times w$ array of one-bit cells. This requires an x-to-2^x decoder that selects a row of the array. At the other extreme, the memory can be organized as a $1 \times w2^x$ array that requires no decoder but whose columns, bundled as w-bit words are multiplexed through a 2^x-to-1 w-bit multiplexer. Clearly, there are many (better) options in between. So granularity here is roughly a measure of cell array organization. For this illustration we have focused on the main concept of granularity and ignored some important details that are explained in Section 5.4.1.

The output of this (module representation) part of the methodology is a Verilog template with size and granularity as parameters. It is used by the module implementation part (Section 3.2).

Before we end this section, it should be noted that granularity is a measure of what is left to the RTL compiler (synthesis tool) to do on its own, rather than the structure of the resulting circuit itself. For example, suppose a high granularity constraints a multiplexer to be built out of two smaller multiplexers, then the compiler will follow this directive. However, giving the compiler full freedom to synthesize any way it deems suitable (low granularity) may still yield the same circuit.

3.2 Module Implementation

As indicated earlier (see parts (b) and (c) on page 18) the module in question is implemented for different sizes and granularity-based representation. This stage deals with these implementations. Module implementation has two stages (*i*) Synthesis and (*ii*) Place and Route (PAR).

In general, the number of designs that we study is in the order of thousands and on an average each design has 16 iterations for synthesis stage and around 3 for PAR stage. This makes it virtually impossible to perform the implementations without automation. We

developed shell scripts for automation. The following Sections 3.2.1 and 3.2.2 describe the sythesis and PAR stage including the automation of various phases of the implementation.

3.2.1 Synthesis Methodology

The entire synthesis flow of all required module sizes and granularity, as well as archiving of results is performed by a script written in C shell [17]. Figure 3.1 shows a global view of the script. The numbers on the edges indicates the sequence in which various interactions take place. We now explain the main parts of this script, namely the *Controller*, *Minimum Constraint Generator* and *Archive Manager*.

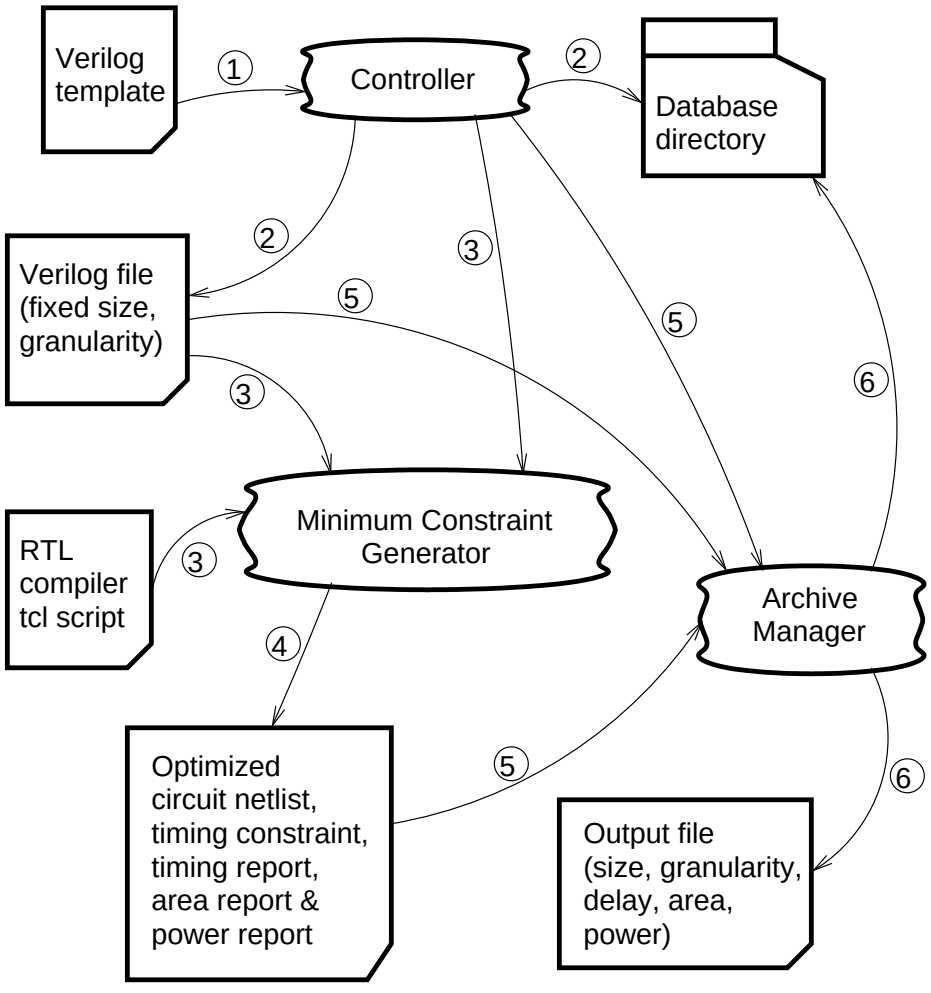


Figure 3.1: Synthesis controlling script

3.2.1.1 Controller

The controller, as the name indicates, orchestrates the flow within the synthesis script. It is responsible for three main tasks: (a) producing a Verilog source file for an appropriate (fixed) module size and granularity, (b) initiating the minimum constraint generator to produce a design with optimized performance and (c) initiating the archive manager to save results in archival files.

Specifically, the controller takes the Verilog source template of the module (from the Module Representation stage of Section 3.1) that has module size and granularity as parameters. It sets these parameter values (in a systematic way) to produce a series of Verilog source files with a good coverage of the design space.

Each time it produces a Verilog source file, it calls the Minimum Constraint Generator with this file. The minimum constraint generator produces a set of optimal designs for the Verilog file (details in Section 3.2.1.2) and returns to the controller. The controller now initiates the Archive Manager to save this information for later use in the PAR stage. The controller now moves to the next design and calls the minimum constraint generator again.

3.2.1.2 Minimum Constraint Generator

As described in Section 2.3.2, the RTL compiler is used to perform the synthesis of a hardware module. It generates a netlist (a circuit of components and connections between them) and an output timing constraint file. The RTL compiler also estimates the speed at which the module may operate, and its area and power consumption for this speed. The reader is referred to Section 2.3.2 for a detailed description of the RTL compiler; we explain the sailent features here, however. Our methodology seeks to model modules where designs are optimized for speed, area or power. Thus the minimum constraint generator runs the RTL compiler three times for each Verilog source file to examine circuits optimized to the three criteria.

For a given timing constraint T_m of the module, let the RTL compiler estimate a time T_e of operation, where $T_e < T_m$. This indicates a positive slack of $T_e - T_m$; that is the RTL compiler estimates that the module can operate faster than the specified constraint. If we apply this estimated time (T_e) as the timing constraint, then the synthesis may not result in a zero slack synthesized netlist. As noted earlier in under "Timing" on page 11, the estimated time of operation is only a hint to the user to determine a zero slack or a small positive slack. Therefore, we perform an iterative binary search between positive and negative slacks until is converges to a zero slack or a small positive slack. The goal of this iterative method is to determine the timing constraint at which the RTL compiler can actually realize a netlist for a zero or a small positive slack. This approach results in a synthesis netlist optimized for time. Figure 3.2 illustrates the flow chart for this part of the script.

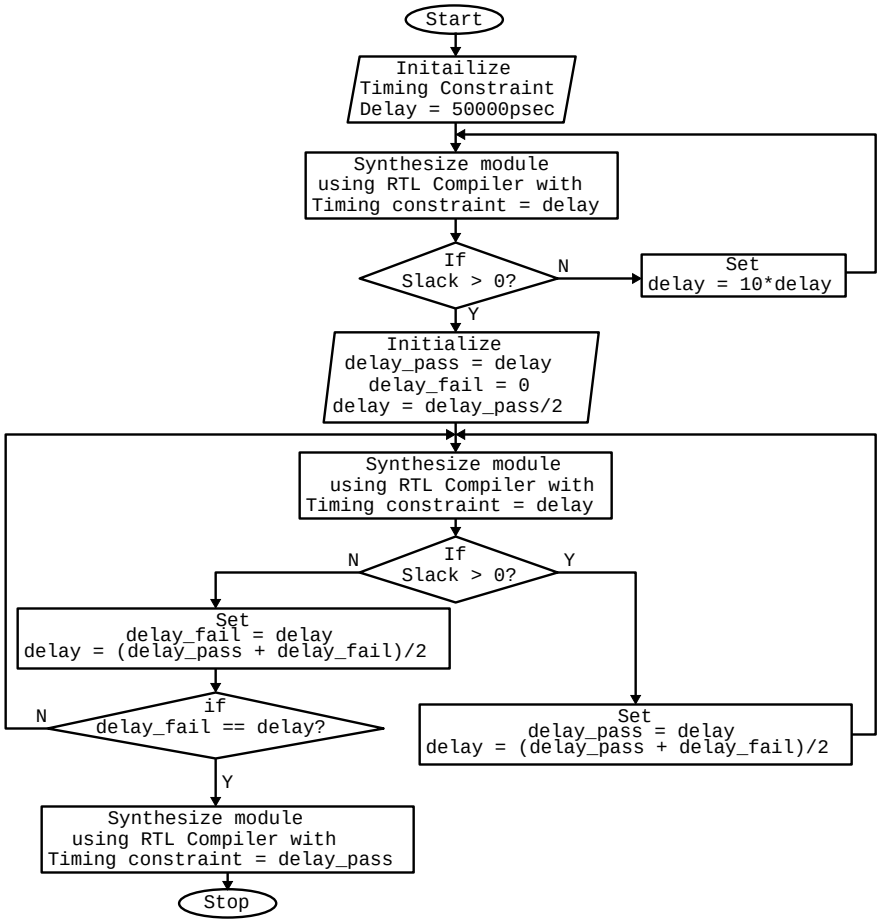


Figure 3.2: Time-optimization flow chart for the MCG.

Note that since the modules that we study are combinational, we use a virtual clock to impose an input timing constraint on the synthesis. Thus, an input timing constraint refers to the clock period of the virtual clock applied on the module. Also note the difference between the input timing constraint (which is a specification of a virtual clock in a tcl script) and output timing constraint (which is a RTL compiler generated timing data file in sdc format for use in the PAR stage).

In the RTL compiler there is no constraint for area optimization [24]. The only constraint that we can change is the timing constraint. The RTL compiler automatically selects the optimal area for a given timing constraint. The priority in deciding/realizing a circuit is given to time and then to area. Thus, a loose (large) timing constraint will result in an optimized area design. That is, a very large timing constraint, say T_∞, produces an optimal area A_∞. The aim here is to generate a timing constraint T_O such that the corresponding area $A_O = A_\infty$ and for time $T < T_O$, the area $A_T > A_O$. Again the minimum constraint generator performs this iteratively using a binary search. Figure 3.3 shows the area optimization flow

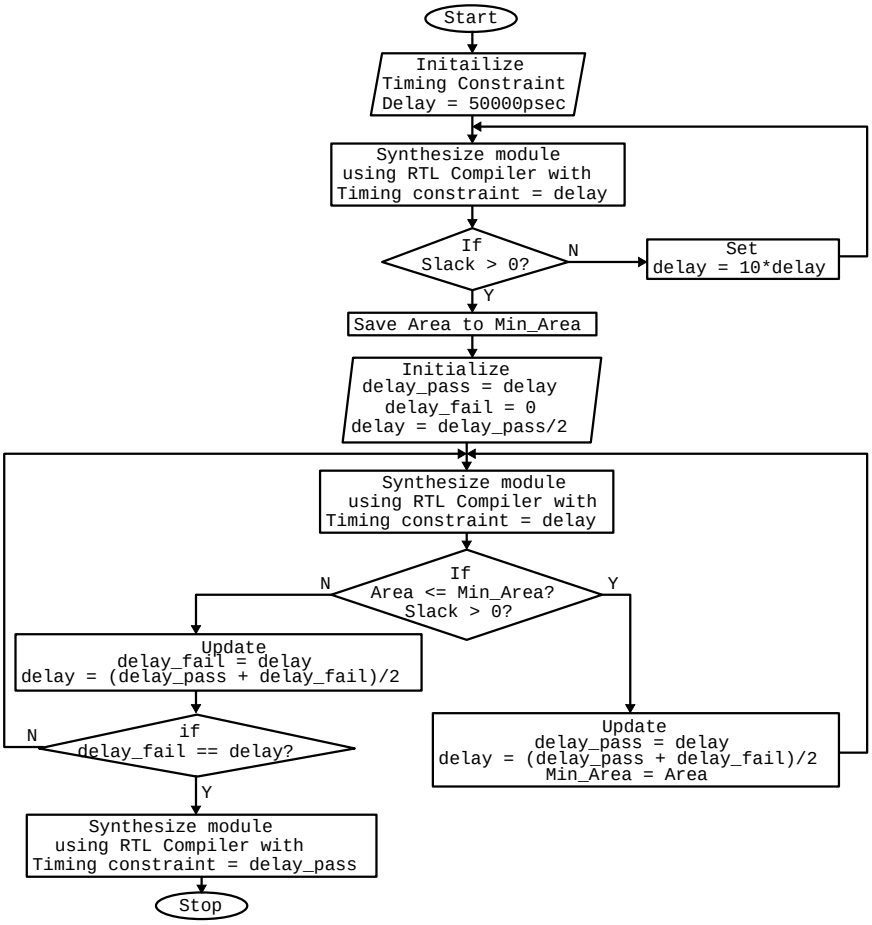


Figure 3.3: Area-optimization flow chart for the MCG.

chart.

It is observed that for some rare cases, the binary search produced an area smaller than A_O but these were very close to A_O.

The RTL compiler has no separate switch to realize a power optimized netlist as well. Power optimized design follows along the same lines as area optimization. Figure 3.4 shows a power optimization flow chart.

3.2.1.3 Archive Manager

This script is responsible for creating a database of optimized design files including Verilog source used (produced by the controller in sequence 2 of Figure 3.1), gate-level netlist, timing constraint in sdc format, area report and power consumption report (sequence 5). It also

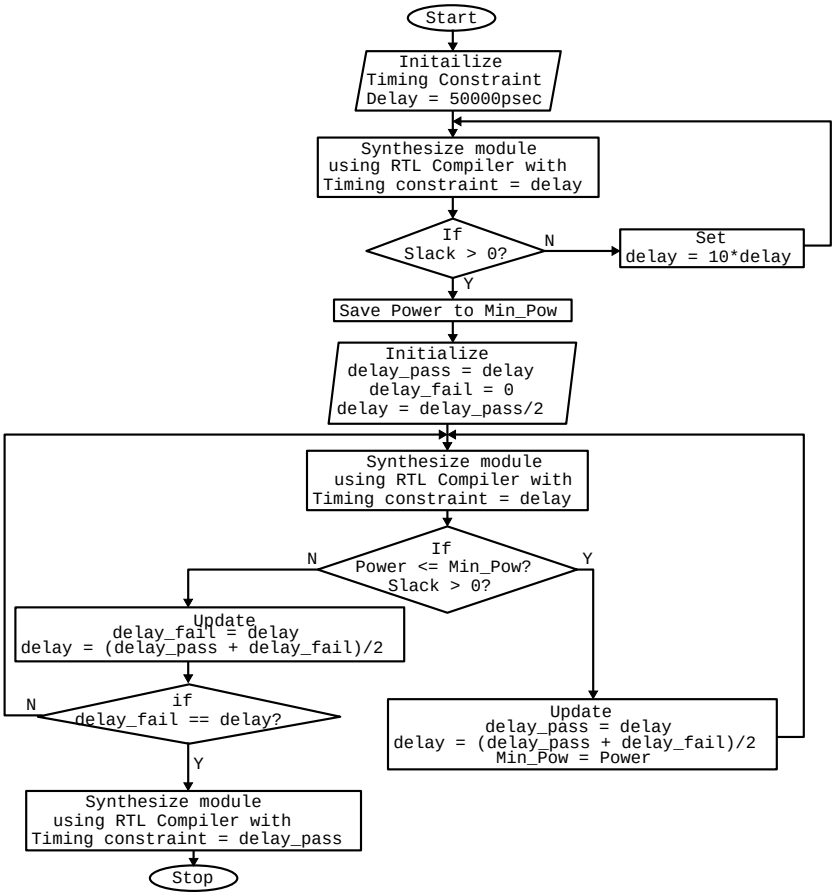


Figure 3.4: Power-optimization flow chart for the MCG.

lists all parameters (module name, size, granularity, timing constraint, area and power) in a spreadsheet compatible file in comma separated value (csv) format. The archive manager also assigns a suitable set of filenames for each implementation to facilitate easy access later.

3.2.2 Place and Route (PAR) Methodology

In this section we explain the Place and Route (PAR) process that places the modules of a design output by the RTL compiler (netlist and output constraint file) on a chip layout and routes connections between these modules. In the process, it considers additional details such as parasitic elements due to resistors and capacitors from the interconnects and cells. Consequently this phase of the implementation accurately measures the model's performance metrics.

As mentioned in Section 2.3.3, we use automated scripts for this phase. The following

section details the PAR phase, including shell scripts.

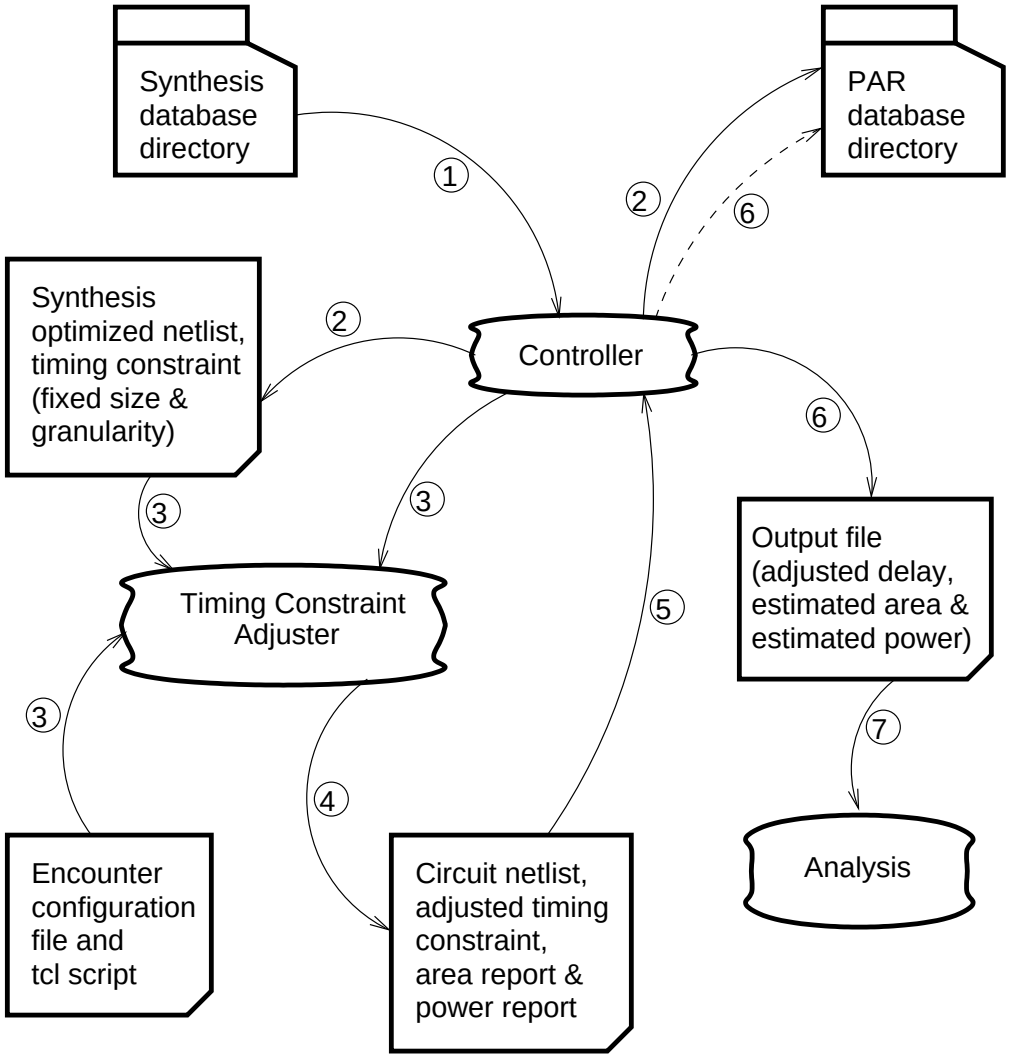


Figure 3.5: Place and Route controlling script

3.2.2.1 Controller

On the whole the controller here and that in the synthesis part have very similar functions (see Figure 3.5). Unlike the synthesis controller, however, the PAR controller copies a design netlist and output timing constraint file and sets up the tcl scripts for the design. It is also responsible for initiating the *Timing Constraint Adjuster* script that zeros in an module speed, based on the estimate from the EDI tool. Here the controller also performs the function of the archive manager of Figure 3.1.

3.2.2.2 Timing Constraint Adjuster

The goal of the timing constraint adjuster is to determine the best speed at which the synthesized netlist can operate within the specified constraints (floorplan and powerplan - see Section 2.3.3). Suppose T_c is the timing constraint produced by the minimum constraint generator (synthesis phase) for a design. The netlist and output timing constraint file of this design are input to the EDI tool. Let T_e be the estimated time given by the EDI tool. The quantity $T_c - T_e$ is the slack. Running the design again with the constraint T_c in the synthesis output constraint file changed to T_e results in a new estimate of speed, and power. Typically this second try results in a zero slack; that is, with the highest speed that the EDI tool can find for the given design and set of constraints. It should be noted that the binary search type of iteration is not needed here (the convergence is rapid in 2, at most 3, tries). Also these iterations are needed only to find the time and power. The area for the first EDI run does not change in subsequent runs.

The output of the PAR stage produces the data used for modeling the module.

3.3 Modeling

For each module, determine a design methodology (expressed, for example, in terms of granularity g (see Section 3.1)) to use to derive the model. Let $B(n)$ refer to the module of size n designed with this methodology. For example, B could be a one-hot decoder, a multiplexer etc. For $B(n)$ there are three possible designs $B_t(n)$, $B_a(n)$, and $B_p(n)$ based on time optimization, area optimization and power optimization. For LUT designs it was not possible to obtain power-optimized designs (see Section 5.4.2). Consequently other power optimized design, although studied, could not be used. Therefore $B_p(n)$ is not considered for the remainder of this thesis.

Each design has three model equations for the three performance metrics (time, area and power). Thus, there are nine functions $\mathcal{T}_t^B(n)$, $\mathcal{A}_t^B(n)$, $\mathcal{P}_t^B(n)$, $\mathcal{T}_a^B(n)$, $\mathcal{A}_a^B(n)$, $\mathcal{P}_a^B(n)$, $\mathcal{T}_p^B(n)$, $\mathcal{A}_p^B(n)$, and $\mathcal{P}_p^B(n)$ that should be discussed. Where the module B is obvious we will omit B from this notation. That is for example, the equation for area in a time optimized design will be written as $\mathcal{A}_t(n)$.

The aim is to obtain each of the functions $\mathcal{T}_t(n)$, $\mathcal{A}_t(n)$, $\mathcal{P}_t(n)$, $\mathcal{T}_a(n)$, $\mathcal{A}_a(n)$, and $\mathcal{P}_a(n)$ for the module in question based on the experimental data. We explain the methodology for $\mathcal{A}_t(n)$. The discussion readily extends to others. The overall methodology consists of four broad steps, not all of which are used with every module. Module-specific details appear in Chapters 5.

- Identification of growth terms
- Regression constraints

- Identification of largest good-fit set
- Post processing

Growth Terms: Recall that discussion here is in the context of the area of a time-optimized design of some module. The function $\mathcal{A}_t(n)$ is calculated theoretically. For example, the area of a n-to-2^n one-hot decoder can be theoretically determined to be $a_1 n 2^n + a_2 2^n + a_3 n + a_4 + a_5 n^3 2^n$, assumes f_i fan-in, f_o fan-out and unit gate area. While wiring area is not significant here, in other cases it may be. Based on this function one may guess that $\mathcal{A}_t(n) = a_1 n 2^n + a_2 2^n + a_3 n + a_4 + a_5 n^3 2^n$. Thus, we have identified functions $n2^n$, 2^n, n and, 1 as candidates for describing $\mathcal{A}_t(n)$. The overall aim is to identify coefficients a_1, a_2, a_3, a_4 and a_5. In general this part identifies the k (say) function candidates needed to describe $\mathcal{A}_t(n)$.

Regression constraints: We now consider two constraints on the coefficients $a_1, a_2, \cdots, a_k$. The first constraint restricts to have positive coefficients. This bound constraint ensures non-negative values for the time, area and power. The second error constraint allows the regression to favor larger values of n, as the aim of this exercise is to use $\mathcal{A}_t(n)$ for large n values that we could not determine by simulation. Quantitatively the constraint is specified in terms of the percentage error to which the regression should restrict its solution. This percentage error is specified as a function such as $1/an$ that decreases with n. So a larger instanse is restricted to have a smaller error. The constant a is selected so that if n_{max} is the largest data point then $a \cong 20/n_{max}$ sets an approximate 5% error bound. Here we use $a = 2$ as in many cases $n_{\max} \cong 15$.

Good-fit set: Implementation results sometime indicate discontinuities that could be modeled piecewise by a set of equations rather than just one. Our interest is primarily on the piece that accurately models large values of n. Restricting the regression to just a few large values could also impact accuracy due to insufficient variation of the data along n to capture its trend. We select the largest set of data for which the regression with the identified growth terms produce acceptably low error.

In general, suppose data is available for $n \in \{a, a+1, \cdots, b\} = S_a$ then the methodology check sets $S_c = \{c, c+1, \cdots, b\}$ for every $a \leq c \leq b - k - 1$, where k is the number of growth terms identified. The value of c that is most suitable is determined by inspection on a case-by-case basis as explained in Chapter 5.

Regression details: We use MATLAB for the regression analysis. The MATLAB [22] constrained linear least-squares function used is

$$\texttt{x} = \texttt{lsqlin}(\texttt{C}, \texttt{d}, \texttt{A}, \texttt{b}, \texttt{Aeq}, \texttt{beq}, \texttt{lb}, \texttt{ub}, \texttt{X0}, \texttt{options})$$

.

The function solves for x in the equation $Cx = d$, subject to constraints specified by A, b, Aeq, beq, lb, and ub. Some or all of these constraints may be omitted.

Constraints lb and ub require the solution to satisfy $lb \leq x \leq ub$. Constraints A and b require $A \leq b$. The constraint Aeq and beq are not used in this work and are not discussed further.

As noted earlier the function solves for $[x]_{k\times 1}$ in the equation

$$[C]_{r\times k}[x]_{k\times 1} = [d]_{r\times 1}$$

Two constraint coefficients to be positive we set $[lb]_{k\times 1} = [\,0\,]_{k\times 1}$ and $[ub]_{k\times 1} = [\,\infty\,]_{k\times 1}$. To favor solutions for larger values of n we set $[A]_{r\times k} = [C]_{r\times k}$ and $[b]_{r\times 1} = [d]_{r\times 1} + [error]_{r\times 1}$. Here we set $[error]_{r\times 1} = \frac{1}{2n}[d]_{r\times 1}$ as discussed earlier.

If r data points are available d is an $r \times 1$ vector representing data points. If k functions $f_j(n)$ for $0 \leq j < k$ have been identified to model $\mathcal{A}_t(n)$, then C is $r \times k$ matrix with each element C_{ij} set to the value of $f_j(n_i)$, where n_i is the size of the module for the i^th data point.

MATLAB does not allow the constraint such as error $e \leq 1/an$ to be specified as a $\geq$ inequality (or use non-linear terms). Therefore the constraint described here only directs the program to reduce positive errors for large n. Negative error are uncostrained. To account for this, we repeat the procedure with $-e \leq 1/an$. Thus, for the negative error constraint on coeffiecient $[\mathtt{A}]_{\mathtt{r\times k}} = -[\mathtt{C}]_{\mathtt{r\times k}}$ and $[\mathtt{b}]_{\mathtt{r\times 1}} = -[\mathtt{d}]_{\mathtt{r\times 1}} + \frac{\mathtt{1}}{\mathtt{2n}}[\mathtt{d}]_{\mathtt{r\times 1}}$. We select among the positive, negative and the one without any constraints favoring higher values of n.

Post Processing: Typically, at this point we would have an equation for $\mathcal{A}_t(n)$. In some cases, where the function shows major discontinuities (such as due to fan-in or fan-out constraints) additional steps may be used to account for them. In the LUT case we also adjust memory-cell parameters, based on independent experiment conducted on individual cells.

Chapter 4

Theoretical Estimation of Time and Area

As noted in Section 3.3 one of the key steps in modeling the building blocks of a MU-Decoder is identifying the growth terms associated with the time, area and power of that building block. That is, for example in order to model the area of a module, we need to theoretically estimate the area, expressed as a function of its size. The modeling then performs a curve-fitting of the module's performance data (time, area and power) by determining the coefficients associated with the growth terms.

Each module's performance metrics (time, area and power) has two contributions (a) due to gates and (b) due to wires (interconnects). These drive the curve characteristics of the quantities that seek to model. Though, in general, the wire part is small, we consider it for an accurate modeling. In this chapter, we derive the order terms in the time and area for the modules required for a MU-Decoder. We consider that the behavior of power to be similar to the that of the area and so no separate terms are identified.

In Section 4.1, we derive the order terms associated with fanning one bit out to ℓ_o places. As an extension of this idea we describe the hardwired multicast from w-bits to $w2^x$ bits in Section 4.2. This is required for the MU-Decoder (see Section 2.2). Though a hardwire module to fan-in ℓ_i bits to 1 bit is not a building block of the MU-Decoder we consider it in Section 4.3, as it plays an important role in deriving order terms for the multiplexer, one-hot decoder and look-up table (LUT). Subsequent Sections 4.4, 4.5, and 4.6 derive the growth terms for one-hot decoder, multiplexer, and LUT respectively.

4.1 Fan-Out

Every gate has a limitation on the number of gates that it can drive. This limitation (that depend on the technology) is called the fan-out of the gate [29] and denoted here by f_o. We used $f_o = 128$ based on experimental study to check when buffers usage starts. If the number of gates that a given gate needs to drive is more than f_o then the circuit may not function as expected. This situation calls for additional buffers to drive the excess gates and ensure a valid/correct circuit operation. That is, to drive $\ell_o > f_o$ outputs, the implementation requires an f_o-ary tree starting from the source gate to ℓ_o sink gates. For example, Figure 4.1 shows an implementation of fanout for $f_o = 3$ and $\ell_o = 27$. Clearly for this example it requires 3 levels of a ternary tree to drive $3^3 = 27$ outputs. The number of gates used is $1+3+3^2 = 13$. Figure 4.1 shows a grid that can be used to estimate the length of the wires. In the example shown, the length of the longest wire (thick lines in Figure 4.1) is given by $0.5+9+0.5+0.5+3+0.5+0.5+1+0.5 = 13+3$ (units) when traversing along the line from top to bottom. Where the length 13 corresponds to horizontal sections of the wire and the length 3 to vertical sections.

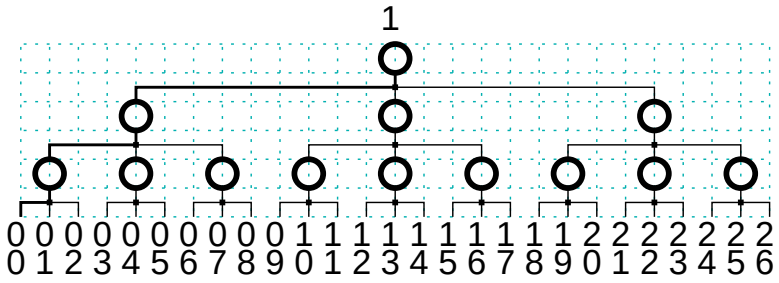


Figure 4.1: Example of a 1-to-27 fan-out ternary tree for $f_o = 3$

In general the number of levels to fan-out to ℓ_o outputs is $\beta_o = \lceil \log_{f_o} \ell_o \rceil$, in other words $f_o^{\beta_o - 1} < \ell \leq f_o^{\beta_o}$. If each buffer has a delay of τ_G, then the gate delay of the fanout circuit is given by $\beta_o \tau_G = \Theta_g(\beta_o) = \Theta_g\left(\frac{\log \ell_o}{\log f_o}\right)$. The wire delay is dependent on the length of the longest wire which is given by $(f_o^{\beta_o - 1} + \cdots + f_o^2 + f_o + 1) + \beta_o = \frac{f_o^{\beta_o}-1}{f_o - 1} + \beta_o$; here β_o is the vertical length. Let τ_W be wire delay per unit length then the wire delay of the fanout circuit is approximately $f_o^{\beta_o - 1}\tau_W \approxeq \left(\frac{l_o}{f_o}\right)\tau_W = \Theta_w\left(\frac{\ell_o}{f_o}\right)$.

The area of the fan-out structure is derived as follows. For clarity we have separated the gate and wire areas using square brackets. Let α_G be the area of a buffer (or gate). Then an (f_o)-ary tree of depth d can be decomposed into a root and f_o subtree each of depth $d-1$.

Thus the area of an ℓ_o fanout f_o-ary tree is,

$$\begin{aligned}
\mathcal{A}(\ell_o) &= [\mathcal{A}(\ell_o/f_o)f_o + \alpha_G] + [\ell_o] \\
&= [(\mathcal{A}(\ell_o/f_o^2)f_o + \alpha_G + \ell_o/f_o])f_o + \alpha_G] + [\ell_o] \\
&= [\mathcal{A}(\ell_o/f_o^2)f_o^2 + \alpha_G(f_o+1)] + [2\ell_o] \\
&= [\mathcal{A}(\ell_o/f_o^3)f_o^3 + \alpha_G(f_o^2+f_o+1)] + [3\ell_o] \\
&\vdots \\
&= [\mathcal{A}(f_o)f_o^{\beta_o-1} + \alpha_G(f_o^{\beta_o-2}+\cdots+f_o^2+f_o+1)] + [(\beta_o-1)\ell_o] \\
&= [(\alpha_G + \ell_o/f_o^{\beta_o-1})f_o^{\beta_o-1} + \alpha_G(f_o^{\beta_o-2}+\cdots+f_o^2+f_o+1)] + [(\beta_o-1)\ell_o] \\
&= [\alpha_G(f_o^{\beta_o-1}+f_o^{\beta_o-2}+\cdots+f_o^2+f_o+1)] + [\beta_o\ell_o] \\
&= [\alpha_G\tfrac{f_o^{\beta_o}-1}{f_o-1}] + [\beta_o\ell_o] \\
&= \Theta_g(\ell_o/f_o) + \Theta_w(\ell_o\tfrac{\log \ell_o}{\log f_o})
\end{aligned}$$

The derived order terms are summarized in the Table 4.1.

Table 4.1: A 1-to-ℓ_o fanout order terms

Quantity	Gate	Wire
Time	$\Theta_g\left(\frac{\log \ell_o}{\log f_o}\right)$	$\Theta_w\left(\frac{\ell_o}{f_o}\right)$
Area	$\Theta_g\left(\frac{\ell_o}{f_o}\right)$	$\Theta_w\left(l_o\frac{\log \ell_o}{\log f_o}\right)$

4.2 Multicast Hardwiring

As noted in Section 2.2, hardwiring in the mapping unit of the MU-Decoder multicasts a z-bit source word to n multiplexers. This multicast hardwiring makes the source word bits available for each multiplexer to select. In this section we estimate the time and area of a w-to-$w2^x$ multicast hardwiring. A general block level structure of w-to-$w2^x$ multicast hardwiring is shown in Figure 4.2.

This multicast requires each of the w-bit data to fan-out to 2^x places. Since the w-bit lines (or vector) needs to go to all 2^x places, the data lines will cross each other. For example, Figure 4.3 shows a 3-to-(3×9) multicast. From the example structure, one can clearly note that the gate delay of the multicast circuit is independent of w the width of each data line. However, the wire length here increases by a factor of w both horizontally and vertically (with respect to the $w = 1$ case). This affect the area as described below.

The length of the longest wire is now $w = 3$ times that of 1-bit fanout. We can also note that the total number of gates has increased by a factor of $w = 3$. In general, for w-to-$w2^x$ multicast hardwiring circuit the gate delay is equal to the gate delay of the 1-to-2^x fan-out.

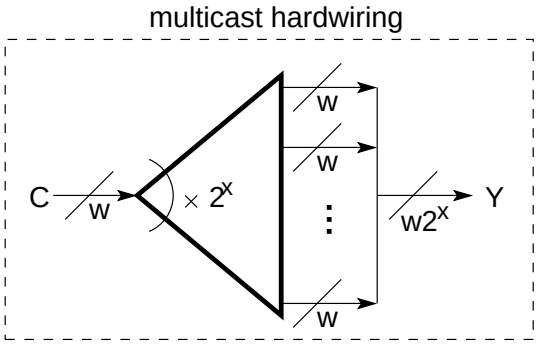


Figure 4.2: Block diagram of a w-to-$w2^x$ multicast hardwiring module.

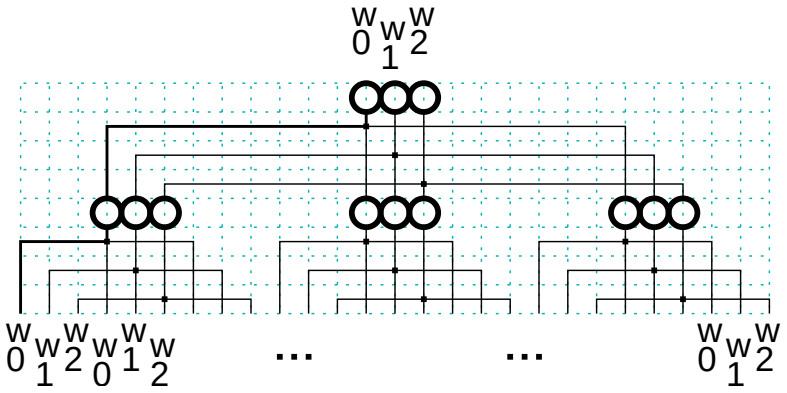


Figure 4.3: Example of a 3-to-(3×9) multicast hardwiring module

That is the gate delay of multicast is $\log_{f_o}(2^x))\tau_G = \frac{x}{\log f_o}\tau_G$. However, the wire length is w times that of the 1-to-2^x fanout; thus the wire delay is $w2^x\tau_W$.

The number of gates is now increased by a factor of w than that of a 1-to-2^x fanout, thus the gate area is given by $\frac{w2^x}{f_o}\alpha_G$. However, the wire length is increased by a factor of w in each of the horizontal and vertical directions, hence the wire area is $w^2(\frac{x2^x}{\log f_o}\alpha_W) = \frac{w^2x2^x}{\log f_o}\alpha_W$. Thus, the time and area order terms for the w-to-$w2^x$ multicast circuit can be summarized as shown in the Table 4.2. Table $tab : FanoutOrder$ is a special case with $w = 1$ and $2^x = \ell_o$.

Table 4.2: A w-to-$w2^x$ multicast hardwiring time and area order terms

Quantity	Gate	Wire
Time	$\Theta_g(\frac{x}{\log f_o})$	$\Theta_w(w2^x)$
Area	$\Theta_g(\frac{w2^x}{f_o})$	$\Theta_w(\frac{w^2x2^x}{\log f_o})$

4.3 Fan-In

A gate has a limitation on the maximum number of inputs that it can have for a given specification. The number of inputs to a gate is technology dependent and we denote this number by f_i in the work. Typically $f_i \cong 3$ for CMOS [29]. To fan-in of $\ell_i > f_i$ inputs, it requires each of f_i-input gates arranged in an f_i-ary tree. An example for $f_i = 2$ and $\ell_i = 16$ is shown in Figure 4.4 which has 4 levels in the binary tree. The number of gates used is $2^3 + 2^2 + 2 + 1 = 15$. The length of the longest wire is $1 + 2 + 2^2 + 2^3 = 15$ horizontal lengths and $2 + 2 + 2 + 2 = 2 \times 4$ vertical lengths.

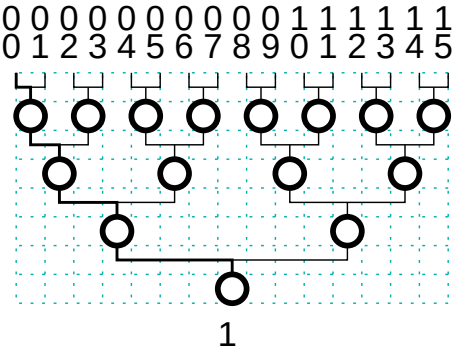


Figure 4.4: Example of a 16-to-1 fan-in binary tree for $f_i = 3$

In general the number of levels to fan-in ℓ_i inputs is $\beta_i = \lceil \log_{f_i} \ell_i \rceil$, in other words $f_i^{\beta_i - 1} < l_i \leq f_i^{\beta_i}$. The fan-in structure is very similar to the fan-out structure except the fact that it uses an f_i-ary tree instead of an f_o-ary tree. Therefore, the gate delay of the fan-in circuit is $\beta_i \tau_G = \frac{\log(l_i)}{\log(f_i)} \tau_G$, and the wire delay is given by $\frac{f_i^{\beta_i} - 1}{f_i - 1} \tau_W$ due to the horizontal length and $\beta_i \tau_W$ due to the vertical length. The net wire delay is approximately $f_i^{\beta_i - 1} \tau_W = \Theta_w \left(\frac{\ell_i}{f_i} \right)$.

Derivation of the area of the fan-in circuit is along the same lines as the fan-out structure. Thus the area for the fan-in is $\left[\alpha_G \frac{f_i^{\beta_i} - 1}{f_i - 1} \right] + [\beta_i \ell_i]$ for the gate and wire term. This is summarized as shown in the Table 4.3.

Table 4.3: A ℓ_i-to-1 fan-in order terms

Quantity	Gate	Wire
Time	$\Theta_g \left(\frac{\log \ell_i}{\log f_i} \right)$	$\Theta_w \left(\frac{\ell_i}{f_i} \right)$
Area	$\Theta_g \left(\frac{\ell_i}{f_i} \right)$	$\Theta_w \left(\ell_i \frac{\log(\ell_i)}{\log(f_i)} \right)$

4.4 One-Hot Decoder

A general structure of an x-to-2^x one-hot decoder is shown in Figure 4.5.

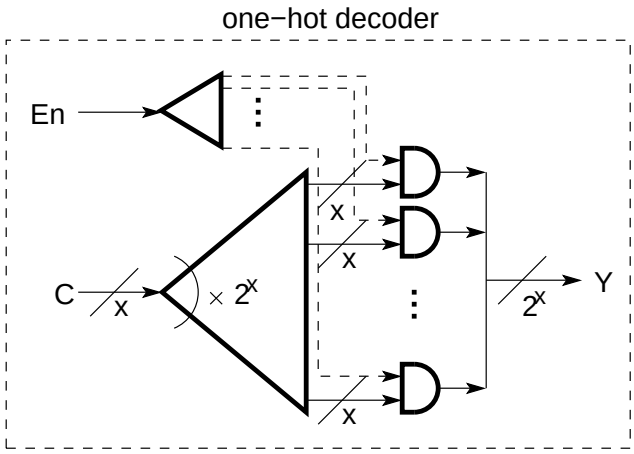


Figure 4.5: Block diagram of a x-to-2^x one-hot decoder.

From the block diagram one can clearly note that the one-hot decoder has an x-to-$x2^x$ hardwiring (from C to the gate inputs), 1-to-2^x fan-out and $(x+1)$-to-1 fan-in to AND gates; there are 2^x such AND gates. Though the 1-to-2^x fan-out is shown in the block diagram (from enable to gate inputs), we will not consider it in the derivation. This fanout runs parallel to the x-to-2^x hardwiring, so its delay will not be a factor. Its area (in order terms) will be consumed by the hardwiring area.

Since the 2^x AND gates are in parallel, the delay of the one-hot decoder is equal to the sum of the delays of the x-to-$x2^x$ multicast hardwiring and a single $(x+1)$-to-1 fan-in (or approximately x-to-1 fan-in). Therefore, the one-hot decoder has delay,

$$\begin{aligned}\mathcal{T} &= \left(\tfrac{x}{\log f_o}\tau_G + x2^x\tau_W\right) + \left(\tfrac{\log(x)}{\log f_i}\tau_G + \tfrac{x}{f_i}\tau_W\right) \\ &= \left[\left(\tfrac{x}{\log f_o} + \tfrac{\log x}{\log f_i}\right)\tau_G\right] + \left[\left(x2^x + \tfrac{x}{f_i}\right)\tau_W\right]\end{aligned}$$

The area of the one-hot decoder is given by the sum of all blocks in its structure, namely the sum of the area of hardwiring and 2^x times the area of x-to-1 fan-in. Therefore the area of the one-hot decoder is,

$$\begin{aligned}\mathcal{A} &= \left(\tfrac{x2^x}{f_o}\alpha_G + \tfrac{x^2x2^x}{\log f_o}\alpha_W\right) + \left(2^x \times \left(\tfrac{x}{f_i}\alpha_G + x\tfrac{\log x}{f_i}\alpha_W\right)\right) \\ &= \left[\left(\tfrac{x2^x}{f_o} + \tfrac{x2^x}{f_i}\right)\alpha_G\right] + \left[\left(\tfrac{x^2x2^x}{\log f_o} + \tfrac{x2^x\log x}{f_i}\right)\alpha_W\right]\end{aligned}$$

The time and area order terms for the x-to-2^x can be summarized as shown in the Table 4.4.

Table 4.4: A x-to-2^x one-hot decoder time and area order terms

Quantity	Gate	Wire
Time	$\Theta_g\left(\frac{x}{\log f_o}+\frac{\log x}{\log f_i}\right)$	$\Theta_w\left(x2^x+\frac{x}{f_i}\right)$
Area	$\Theta_g\left(\frac{x2^x}{f_o}+\frac{x2^x}{f_i}\right)$	$\Theta_w\left(\frac{x^2x2^x}{\log f_o}+\frac{x2^x\log x}{f_i}\right)$

4.5 Multiplexer

A block structure of 2^x-to-1 multiplexer is shown in Figure 4.6.

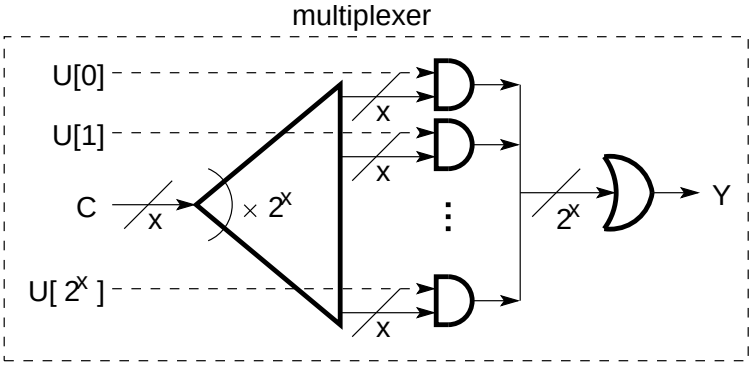


Figure 4.6: Block diagram of a 2^x-to-1 multiplexer.

The building elements of the 2^x-to-1 multiplexer are a x-to-$x2^x$ hardwiring block, an x-to-1 fan-in for 2^x-AND gates and a 2^x-to-1 fan-in OR gate. Notice that the multiplexer's only difference from a one-hot decoder is the final 2^x-to-1 fan-in to an OR gate. Thus the multiplexer time and area can be determined by the addition of the terms of one-hot decoder and the 2^x-to-1 fan-in.

The time for the multiplexer is given by,

$$\begin{aligned}\mathcal{T} &= \left(\left(\frac{x}{\log f_o}+\frac{\log x}{\log f_i}\right)\tau_G+\left(x2^x+\frac{x}{f_i}\right)\tau_W\right)+\left(\left(\log_{f_i}(2^x)\right)\tau_G+\left(\frac{2^x}{f_i}\right)\tau_W\right)\\ &= \left[\left(\frac{x}{\log f_o}+\frac{\log(x)}{\log(f_i)}+\frac{x}{\log(f_i)}\right)\tau_G\right]+\left[\left(x2^x+\frac{x}{f_i}+\frac{2^x}{f_i}\right)\tau_W\right]\end{aligned}$$

The area is given by,

$$\begin{aligned}\mathcal{A} &= \left(\left(\frac{x2^x}{f_o}+\frac{x2^x}{f_i}\right)\alpha_G+\left(\frac{x^2x2^x}{\log f_o}+\frac{x2^x\log(x)}{f_i}\right)\alpha_W\right)+\left(\left(\frac{2^x}{f_i}\right)\alpha_G+\left(2^x\frac{\log(2^x)}{\log f_i}\right)\alpha_W\right)\\ &= \left[\left(\frac{x2^x}{f_o}+\frac{x2^x}{f_i}+\frac{2^x}{f_i}\right)\alpha_G\right]+\left[\left(\frac{x^2x2^x}{\log f_o}+\frac{x2^x\log(x)}{f_i}+\frac{x2^x}{\log f_i}\right)\alpha_W\right]\end{aligned}$$

The order terms for the 2^x-to-1 multiplexer are summarized in the Table 4.5.

Table 4.5: A 2^x-to-1 multiplexer time and area order terms

Quantity	Gate	Wire
Time	$\Theta_g\left(\frac{x}{\log f_o} + \frac{\log x}{\log f_i} + \frac{x}{\log f_i}\right)$	$\Theta_w\left(x2^x + \frac{x}{f_i} + \frac{2^x}{f_i}\right)$
Area	$\Theta_g\left(\frac{x2^x}{f_o} + \frac{x2^x}{f_i} + \frac{2^x}{f_i}\right)$	$\Theta_w\left(\frac{x^2x2^x}{\log f_o} + \frac{x2^x \log x}{\log f_i} + \frac{x2^x}{\log f_i}\right)$

4.6 Look-Up Table (LUT)

As described in Section 2.2, an x-to-w look-up table (LUT) is a memory of 2^x locations, each w-bits wide. A LUT has many different implementations such as using a static RAM (SRAM) or, dynamic RAM (DRAM). In our study we considered SRAM implementation for modeling a LUT. In this implementation, it is general practice to organize one-bit cells in rows and columns to form an array (memory core). Then, row and column one-hot decoders enable (activate) a w-bit data word of SRAM cells for read or write [18].

One such structure of a $2^x \times w$ LUT and its building blocks is shown in the Figure 4.7. In this structure the three main building blocks are row one-hot decoder, column one-hot decoder and memory core.

The memory core in turn has two main logic blocks (a) bit-b_i block and (b) AND-OR block. The structures of these two logic blocks are shown in the Figure 4.8. In this organization of memory cells, 2^g column one-hot decoder output is multicast to each bit-b_i blocks. Since there are w of bit-b_i blocks the column outputs need to drive w gates. Hence a column incurs an overhead due to the 2^g-to-$2^g w$ multicast hardwiring. However each row is controlled by one of the 2^{x-g} row one-hot decoder outputs. This one output needs to drive w bits in that row, so it requires a 1-to-w fanout structure for each row. It is also important to note that the 1-bit SRAM cell has tristate output, hence the 2^{x-g} cell outputs in a column are short circuited. The AND-OR logic block combines the 2^{x-g} short circuited tristate wiring of 2^g cell outputs of the bit-b_i block to produce the data output bit b_i of the LUT.

On the whole, the $2^x \times w$ LUT has $2^x w$ memory-cells, a $(x-g)$-to-2^{x-g} row one-hot decoder, 2^{x-g} fanout (each a 1-to-w fanout), g-to-2^g column decoder, 2^g-to-$2^g w$ multicast hardwiring, w AND-OR logic blocks (each of 2^g-to-1).

In order to estimate the delay of the $2^x \times w$ LUT, we need to consider delay in the two paths from a row and from a column to the 1-bit cells, the delay due to the tristated cell-output and the delay due to the AND-OR logic. That is, the time of the LUT is given by the sum of a 1-bit cell delay, tristate delay and AND-OR logic delay and the maximum of (a) sum of the delay of the row one-hot decoder, 1-to-w fanout and longest horizontal wire, (b) sum of the delay of column one-hot decoder, 2^g-to-$2^g w$ multicast hardwiring and the longest vertical wire.

The delay for the one-hot decoder, fanout, and multicast hardwiring modules are derived in the earlier sections. Now let us consider the tristate output delay. The tristate output delay can be approximated as delay to drive parallel capacitors, which is proportional to the number of parallel capacitors. In the LUT organization, there are 2^{x-g} rows of tristate outputs are shorted, thus the delay is given by $2^{x-g}\tau_W$. We have used the same constant τ_W as to wire delay mainly to indicate that it is small.

From Figure 4.8 (b) the AND-OR logic delay is given by the sum of AND gate delay and the 2^g-to-1 fan-in OR gate delay. Therefore, the AND-OR logic block has a delay of $\left[\left(\frac{g}{\log f_i}+1\right)\tau_G\right]+\left[\frac{2^g}{f_i}\tau_W\right]$.

On the whole, the LUT delay is given by

$$\begin{aligned}\mathcal{T} &= \max(\{\left(\tfrac{x-g}{\log f_o}+\tfrac{\log(x-g)}{\log f_i}\right)\tau_G+\tfrac{(x-g)2^{x-g}}{f_o}\tau_W+\tfrac{\log w}{\log f_o}\tau_G+\tfrac{w}{f_o}\tau_W++w2^g\tau_W\},\\ &\quad \{(\tfrac{g}{\log(f_o)}+\tfrac{\log(g)}{\log(f_i)})\tau_G+\tfrac{g2^g}{f_o}\tau_W+\tfrac{\log(w)}{\log(f_o)}\tau_G+\tfrac{w2^g}{f_o}\tau_W++w2^{x-g}\tau_W\})+\\ &\quad \tau_G+2^{x-g}\tau_W+(1+\tfrac{g}{\log f_i})\tau_G+\tfrac{2^g}{f_i}\tau_W\end{aligned}$$

Now consider the area of the $2^x \times w$ LUT. The LUT area is the sum of all the building blocks. The area of the one-hot decoder, fan-out, multicast hardwiring are derived in the earlier sections. The two building blocks that we need to find the area are memory-cells and the AND-OR logic. Now let us consider the memory-cells organization block. Let us say that each 1-bit cell has $\ell \times b$ size. Then the bit-b_i block has 2^g of such 1-bit cells arranged horizontally (see Figure 4.8 (a)). Thus each each bit-b_i block has $\ell 2^g \times b$ area. In the memory cell arrangement each row has w wires interleaved by (say) q units of gap. Thus the memory cell arrangement, has vertical length $= (b+qw)2^{x-g}$ and the horizontal length $= \ell 2^g w$. Therefore the area of the memory cells is,

$$\begin{aligned}\mathcal{A} &= ((b+qw)2^{x-g})\times(\ell 2^g w)\\ &= (\ell b)w2^x+(\ell q)w^2 2^x\\ &= w2^x\mathcal{A}_g+w^2 2^x\mathcal{A}_w\end{aligned}$$

The area of the AND-OR logic is given by sum of 2^g AND-gates area and 2^g-to-1 fan-in OR gate area. Therefore, the area for the w AND-OR logic blocks is $w2^g\alpha_G+\frac{w2^g}{f_i}\alpha_G+\frac{wg2^g}{\log f_i}\alpha_W$.

Overall the area of the $2^x \times x$ LUT is,

$$\begin{aligned}\mathcal{A} &= w2^x\alpha_G+w^2 2^x\alpha_W+\tfrac{(x-g)2^{x-g}}{f_i}\alpha_G+(\tfrac{(x-g)^3 2^{x-g}}{\log(f_o)}+\tfrac{(x-g)2^{x-g}\log(x-g)}{\log(f_i)})\alpha_W+\\ &\quad (\tfrac{w}{f_o}\alpha_G+\tfrac{w\log(w)}{\log(f_o)}\alpha_W)2^{x-g}+\tfrac{g2^g}{f_i}\alpha_G+(\tfrac{g^3 2^g}{\log(f_o)}+\tfrac{g2^g\log(g)}{\log(f_i)})\alpha_W+\\ &\quad \tfrac{2^g w}{f_o}\alpha_G+\tfrac{(2^g)^2 w\log(w)}{\log(f_o)}\alpha_W+(w2^g+\tfrac{w2^g}{f_i})\alpha_G+\tfrac{wg2^g}{\log f_i}\alpha_W\end{aligned}$$

In general the $g \cong x/2$ is the best granularity, with this case the time and gate order terms are summarized in Table 4.6

Table 4.6: A 2^x-to-1 multiplexer time and area order terms

Quantity	Gate	Wire
Time	$\Theta_g\left(\frac{x}{\log f_o} + \frac{\log x}{\log f_i} + \frac{\log w}{\log f_o}\right)$	$\Theta_w\left(\frac{w2^x}{\log f_o} + \frac{x2^x}{\log f_i} + 2^x + \frac{x}{\log f_i}\right)$
Area	$\Theta_g\left(w2^x + \frac{x2^x}{f_o}\right)$	$\Theta_w\left(w^2 2^x + \frac{2^x w \log w}{\log f_o} + \frac{x^3 2^x}{\log f_o}\right)$

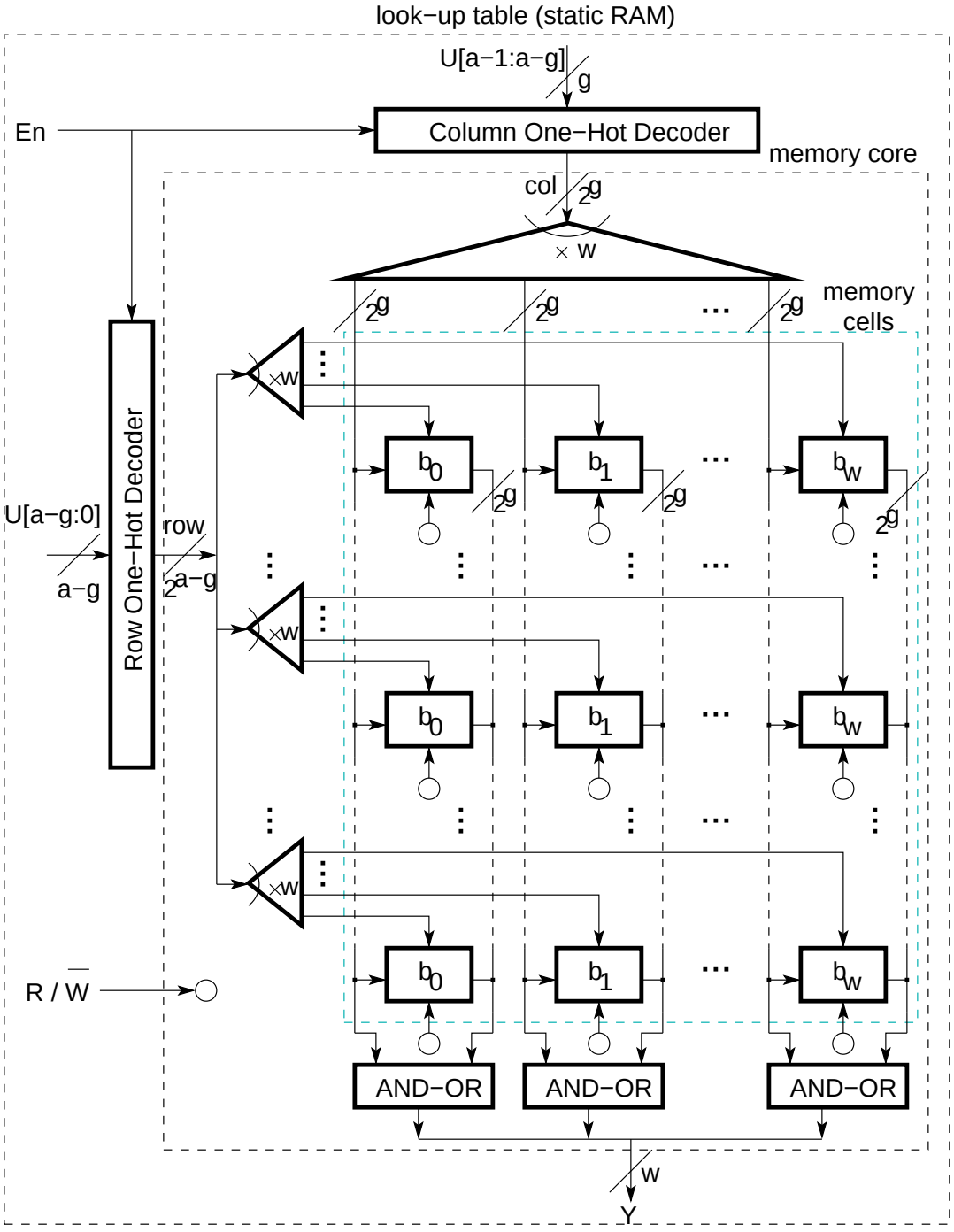
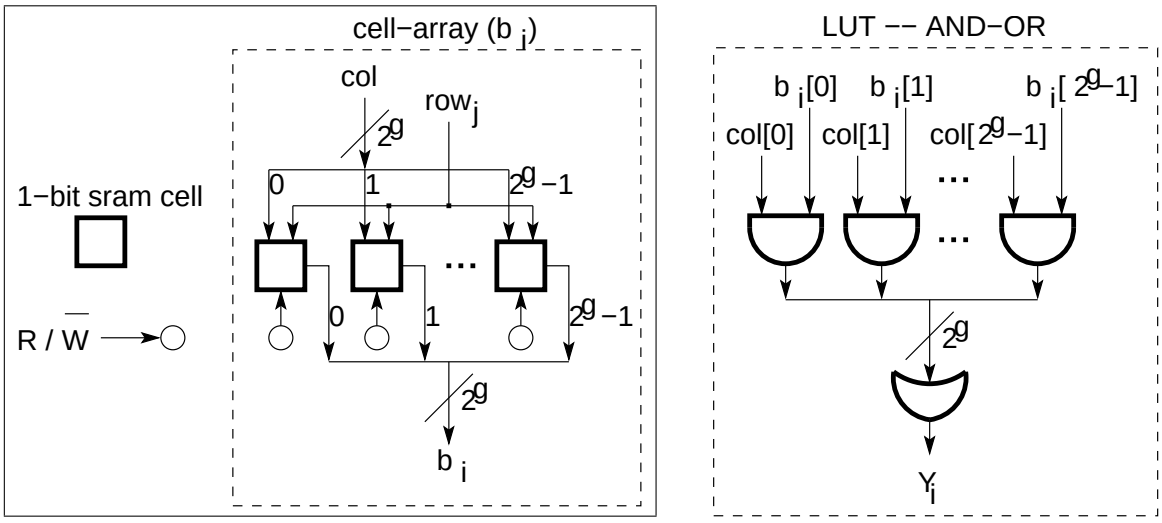


Figure 4.7: Block diagram of a 2^x-to-w LUT.



(a) (b)

Figure 4.8: Structure of a 2^x-to-w LUT (a) Bit b_i block, (b) AND-OR logic block.

Chapter 5

Performance Modeling

In Section 2.2 we have described the structure of a MU-Decoder and identified its basic building blocks as multicast hardwiring, multiplexers, one-hot decoder, and LUT. In this chapter we examine different designs of the building blocks and construct models (equations) for time, area and power for time- and area-optimized designs. These expressions are then used to model the MU-Decoder. All these modules are also frequently used in digital systems, so the study in this chapter may be of independent interest.

The general modeling methodology was explained in Chapter 3. Here we provide additional details that are particular to each of the building blocks of the MU-Decoder. For clarity, each section of this chapter is organized along the same lines as Chapter 3. In each section, we discuss some basic ideas of the module (including definitions and implementations), then we define granularity for that module (except for hardwiring). This is followed by the implementation details and results from the place and route (PAR) phase. Finally we present the modeling details of the module.

In the Chapter 4 we have theoretically estimated growth terms that could constitute to the time and area of each of the building blocks. These terms were classified as gate growth term and the (lower coefficient) wire term. We use all the gate growth terms and the highest order wire growth term to obtain equations that model cost and performance of the module.

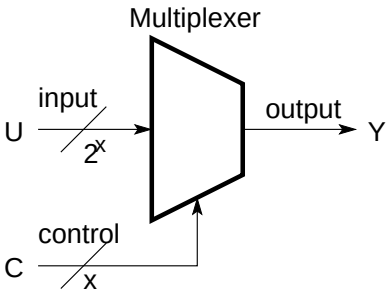
As mentioned in Section 3.2 our experiments are limited by computational resources, and the time taken to implement large sized modules. Broadly speaking, our strategy for modeling a module is to (a) implement all possible sizes and granularities then (b) obtain coefficients to the growth terms based on the implementation data.

In the next few sections we detail the modeling of multiplexers, one-hot decoders, multicast hardwiring and LUTs. Section 5.1 for the multiplexer is more detailed than the rest to provide the reader with a clear view of the methodology. Adding these details (particularly the implementation data)for all building blocks will be unnecessarily cumbersome and will not add any new sight.

5.1 Mulitplexer

As noted in Section 2.2, the multiplexer is one of the basic MU-Decoder modules. Specifically, the mapping unit of the MU-Decoder implements several multicasts. Rolling all these multicasts into the one circuit to use the same n-bit output, requires multiplexers. In this section we examine different multiplexer designs and construct model equations.

A 2^x-to-1 multiplexer is a module that selects one of 2^x inputs to (virtually) connect to a single output. It uses x control bits to determine which of the 2^x inputs to connect to the output. Let the 2^x data bits of the multiplexer be $u_{2^x-1}, u_{2^x-2}, \cdots, u_1, u_0$. Let the control bits be $c_{x-1}, c_{x-2}, \cdots, c_1, c_0$. Let the binary number $c_{x-1}c_{x-2}\cdots c_1c_0$ have value i (where $0 \leq i < 2^x$). Then the multiplexer output is u_i. Figure 5.1 shows the block diagram of a 2^x-to-1 multiplexer and its behavioral Verilog code.




```
module mux_behav(Y, U, C);
    parameter x = 2;
    parameter pow2x = 1<<x;
    input [pow2x -1:0] U;
    input [x-1:0] C;
    output Y;
    wire Y;
    assign Y = U[C];
endmodule
```

Figure 5.1: A 2^x-to-1 multiplexer block diagram (left), its behavioral Verilog code (right).

There are many different ways to implement a multiplexer. These include approaches based on (a) AND-OR logic, (b) transmission gates, (c) NAND or NOR logic, (d) tristate buffers, and (e) granularity. We use the granularity-based approach which is elaborated upon in the next section. Figure 5.2 shows a 2-to-1 multiplexer implementation using the above approaches.

5.1.1 Granularity-Based Multiplexer Representation

A 2^x-to-1 multiplexer can either be constructed at the gate-level or, at the extreme, in terms of two (2^{x-1})-to-1 multiplexers and a 2-to-1 multiplexer. There are many shades in between. Figure 5.3 shows a 2^3-to-1 multiplexer built out of seven 2-to-1 multiplexers. This can be generalized to build a 2^{kg}-to-1 multiplexer out of 2^g-to-1 multiplexers [5]. Figure 5.4 shows one way to build a 2^x-to-1 multiplexer out of multiplexers of size at most 2^g-to-1. Here x need not be an integer multiple of g. We will refer to the implementation in Figure 5.4

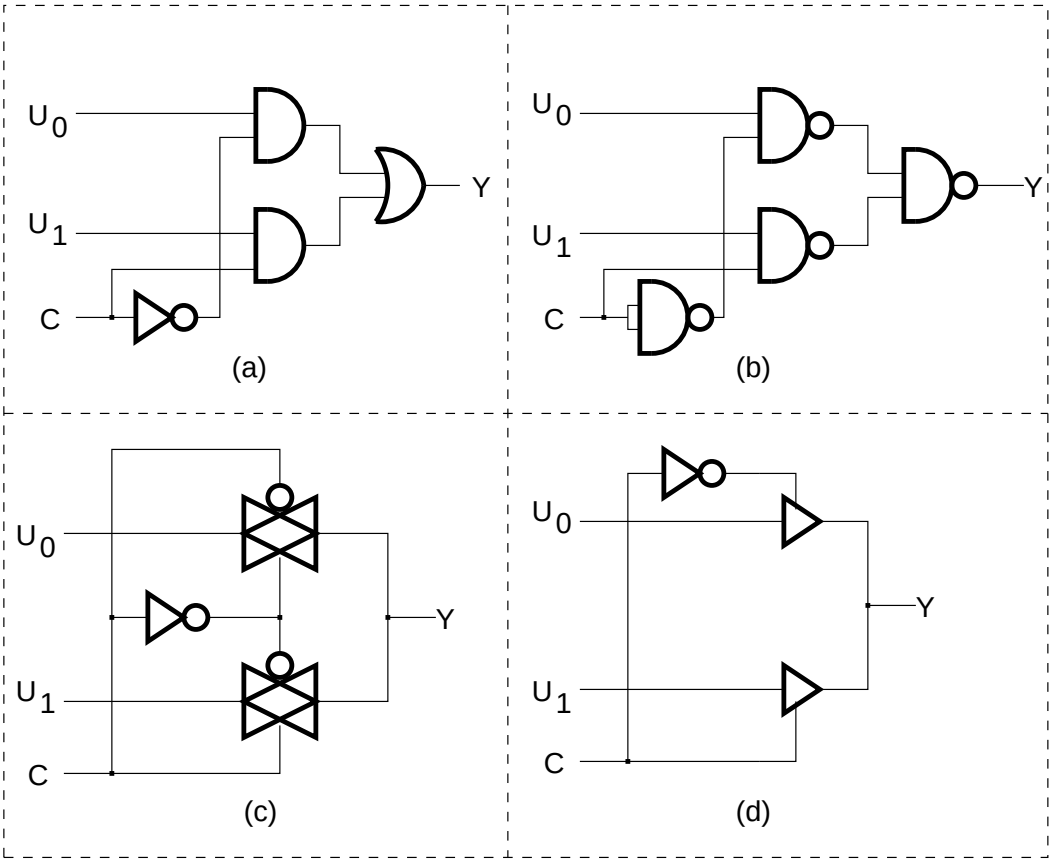


Figure 5.2: Implementations of a 2-to-1 multiplexer, (a) AND-OR logic, (b) NAND logic, (c) transmission gate and (d) tristate buffer.

as a *granularity-g implementation* of a 2^x-to-1 multiplexer. This implementation amounts to a 2^g-ary tree of 2^g-to-1 multiplexers with the root using a $2^{g'}$-to-1 multiplexer, where $g' = x(\text{mod } g)$.

5.1.2 Multiplexer Implementation Results and Analysis

As described Section 3.2, the synthesis controller and PAR controller were used to implement multiplexers of various sizes and granularities. In this section we provide the details of the multiplexer implementation results from the synthesis and PAR stages. This section is organised along the lines of Section 3.2 that describes the general implementation procedure. The reader may benefit from refering to Section 3.2 before proceeding with this section.

A multiplexer has two design parameters, size (x) and granularity (g). The synthesis controller systematically sets these parameters for all combinations such that $x \in \{1, 2, \cdots, 15\}$ and $g \in \{1, 2, \cdots, x\}$. The minimum constraint generator (MCG) is then initiated for a fixed

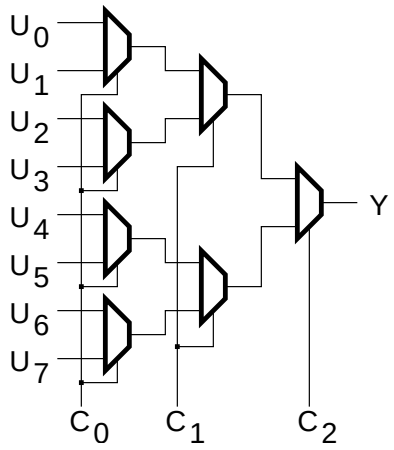


Figure 5.3: Implementation of an 8-to-1 multiplexer using 2-to-1 multiplexers.

x and g. The three optimized designs (time optimized, area optimized and power optimized) are stored in a database directory for further use. Although this work does not consider power optimization, we obtain power optimization data for some modules, the multiplexer is one of those.

The name of the files in which the simulation data is stored is such that it indicates the module name (for this case "mux"), size x, granularity g, file type indicator (synthesis file (s), time report (t), area report (a), power report (p) and timing constraint output file (c)), and the timing constraint value determined by MCG. For example, consider the case where the script (MCG) implements a multiplexer of size $x = 3$ and granularity $g = 2$ and optimizes this design for time. Suppose also that this design has a minimum timing constraint of 250 ps. Then the synthesis related to this design is saved in file "`mux_3_2_s_250.vh`".

Each of the synthesized designs is now processed through the PAR phase as described in the Section 3.2.2. This process adjusts the timing constraint and determines the time, area and power at the layout level. It was observed that the RTL compiler uses a 2^2-to-1 multiplexer circuit to produce a non-inverted output. A 2-to-1 multiplexer requires additional inverters. Therefore, the lowest granularity that our design benefits from is $g = 2$.

Figures 5.5, and 5.6 shows the time, area and power for different design optimizations (time, area and power) and various values of x and g. Each of these figures has three sets of data, corresponding to time, area and power. For example, Figure 5.5 (a) shows data for time optimized design. There are three sets of curves. The first one is the delay of time-optimized designs for various sizes x. The second curve shows the designs' area, however the designs themselves are time-optimized. The third set of curves is the power for time-optimized design. Each of these sets has three curves (solid, dashed and dotted). The solid curve is for a design with granularity $g = x$. The dotted curve is for $g = 2$. The dashed curve represents designs with the best possible (AbsTmin) time-optimization and may include several granularities. For example for the area of time-optimized design

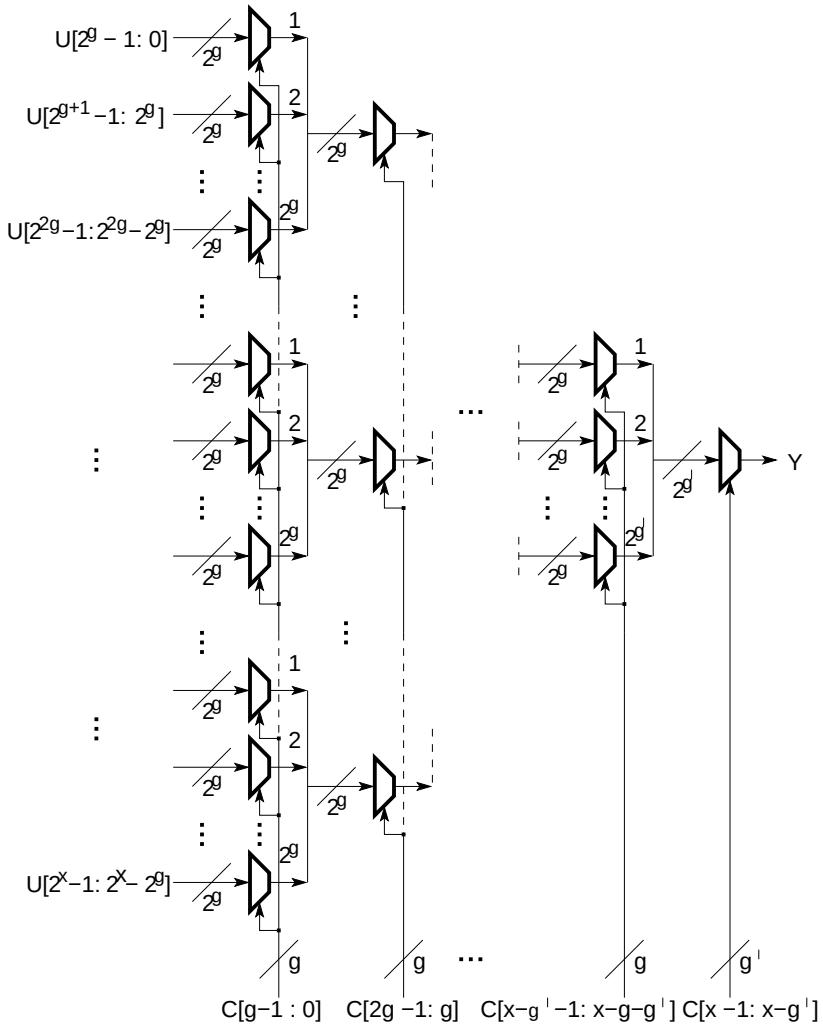


Figure 5.4: Granularity-based multiplexer implementations.

(Figure 5.5 (a)) when $n = 4$ and 7 the $g = 2$ designs have $219\mu m^2$ and $2309\mu m^2$ area, $g = 4$ and 7 (that is $g = x$) have $163.5\mu m^2$ and $1393.8\mu m^2$ area and the AbsTmin curve gives area $163.5\mu m^2$ and $1161\mu m^2$. For these AbsTmin area $g = 1$ and 4. The same ideas apply to area-optimized designs (Figure 5.5 (b)) and power-optimized design (Figure 5.6). The three graphs correspond to $g = 2$ (low granularity), $g = x$ (highest granularity) and the AbsMin. The last curve uses the value of g that gives the best optimized design. This g could be different for different values of x. Although these graph show three curves, it should be noted that data was obtained for x granularities.

For the time optimized designs (Figure 5.5 (a)), clearly AbsTmin time gives the best designs. Although the $g = 2$ case is not as good as AbsTmin, it represents a consistent design methodology. Clearly, $g = x$ is not competitive with $g = 2$ or AbsTmin. Therefore, we will use AbsTmin for modeling time optimized multiplexer designs.

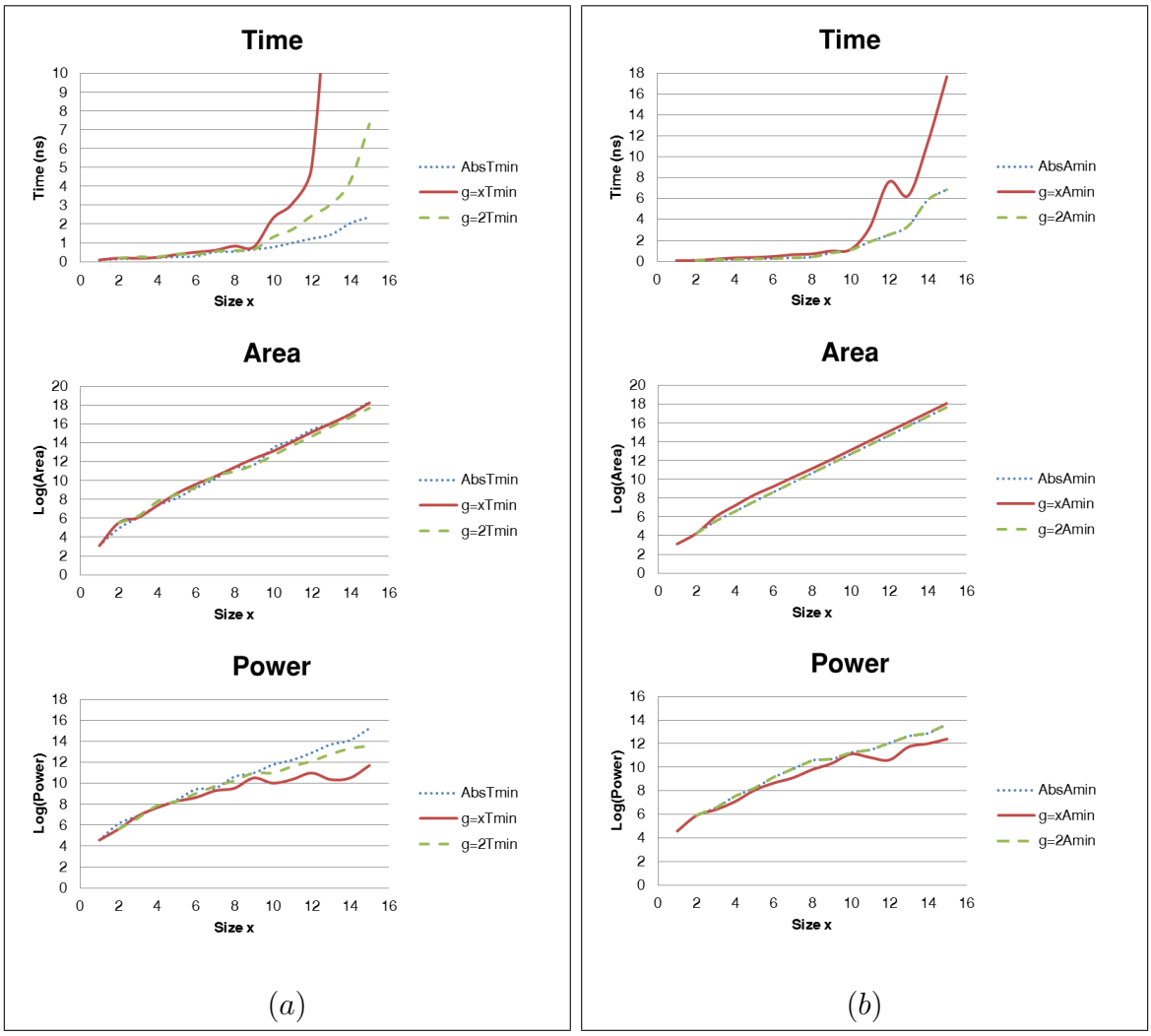


Figure 5.5: Multiplexer (a) time- and (b) area-optimized results

For area optimized designs (Figure 5.5 (b)) there is no significant difference in the area of various approaches, so we consider the curves for time. This clearly shows that AbsAmin and $g = 2$ coincide and they are both better than $g = x$. Therefore we will use AbsAmin for modeling area optimized multiplexer designs.

For power optimization (Figure 5.6) again $g = 2$ and AbsPmin are the best for the same reasons as area optimizations. We present this data but do not use it to model the MU-Decoder.

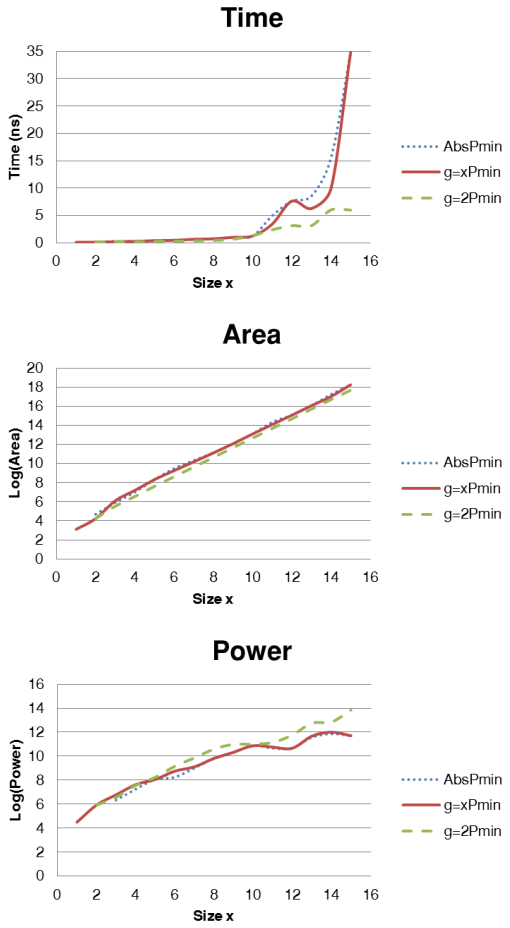


Figure 5.6: Multiplexer power optimized results

5.1.3 Multiplexer Model

A 2^x-to-1 multiplexers time and area growth terms are identified in Section 4.5 and the order terms tabulated in Table 4.5. From these order terms we will pick all gate growth terms and the highest wire growth term.

For the multiplexer time, gate growth terms are x, $\log x$, and 1, and the highest order wire term is $x2^x$. The area growth terms $x2^x$, 2^x, x, 1 (for gate) and $x^3 2^x$ (for wire). The area growth terms are also used for power modeling. The experimental data for time-optimization and area-optimization are now used to determine the coefficients of the growth terms.

The curves in Figure 5.5 represent our best design with respect to time- and area-optimizations. The aim now is to obtain an equation that accurately models these curves, particularly for large values of x. We use linear regression to determine these equations. As

mentioned earlier, our theoretical analysis (see Table 4.5) identifies the growth terms that contribute to the time and area (and hence power) of a multiplexer. What remains is finding the coefficients in these equations.

Recall that for $\mathcal{G} \in \{\mathcal{T}, \mathcal{A}, \mathcal{P}\}$ (for measure of time, area and power), $\omega \in \{t, a\}$ (for time- and area-optimization) and $B \in \{M, 1, H, L\}$ (for multiplexer, one-hot decoder, multicast hardwiring and LUT), the quantity $\mathcal{G}_\omega^B(x)$ denotes the measure $\mathcal{G}$ for building block B of size x optimized with respect to ω. For example consider $\mathcal{T}_t^M(x)$, representing the time for a time-optimized multiplexer of size x.

Table 5.1: Percentage error of time coefficients for time-optimized multiplexers.

	Range of values for x										
x	$(1,15)$	$(2,15)$	$(3,15)$	$(4,15)$	$(5,15)$	$(6,15)$	$(7,15)$	$(8,15)$	$(9,15)$	$(10,15)$	$(11,15)$
4	40.8	40.8	41.0	41.3	–	–	–	–	–	–	–
5	56.3	56.2	56.4	56.8	57.9	–	–	–	–	–	–
6	68.1	68.1	68.2	68.7	69.9	72.3	–	–	–	–	–
7	8.9	8.9	9.0	9.3	10.0	11.6	**14.4**	–	–	–	–
8	21.0	21.0	21.1	21.4	22.3	24.0	**27.1**	28.4	–	–	–
9	12.7	12.6	12.8	13.1	13.8	15.4	**18.2**	19.4	22.7	–	–
10	7.7	7.7	7.8	8.0	8.7	10.2	**12.8**	13.9	16.9	20.8	–
11	5.8	5.8	5.7	5.5	5.0	3.8	**1.7**	0.8	1.7	4.8	9.6
12	10.5	10.5	10.4	10.3	9.8	8.9	**7.2**	6.4	4.4	1.9	2.0
13	7.3	7.3	7.2	7.1	6.8	6.1	**4.8**	4.3	2.8	1.0	1.9
14	14.1	14.1	14.1	14.0	13.9	13.7	**13.2**	13.0	12.5	11.8	10.7
15	7.1	7.1	7.1	7.1	6.9	6.7	**6.2**	6.0	5.4	4.7	3.7

The methodology of Section 3.3 is used to determine the largest possible data set that fits the higher end of x. Table 5.1 shows these sets for $\mathcal{T}_t^M(x)$. Similar tables exist for $\mathcal{A}_t^M(x)$, $\mathcal{P}_t^M(x)$ and $\mathcal{T}_a^M(x)$, $\mathcal{A}_a^M(x)$, $\mathcal{P}_a^M(x)$. Notice in Table 5.1, in general, that as the range of x decreases, so does the error (on the right end of the table). However, a small data set also reduces the reliability of the model as it is based on fewer points. A balance must be struck. Here we selected the $(7, 15)$ range as all its error values (except $x = 8$, that appear to be an outlier) are less than 20%. In general this a determination is done by inspection.

From Table 4.5 we know that $\mathcal{T}_t^M = c_1 x + c_2 \log x + c_3 + c_4 x 2^x$, the last term being the wire term. Along with the data of Table 5.1 we also compute these coefficients c_1–c_4 for the various ranges of x. These are shown in Table 5.2. As noted earlier, the range $x \in [7, 15]$ was selected and this corresponds to the coefficients in bold in Table 5.2. Thus,

$$\mathcal{T}_t^M = [0.0843x + 5.76 \times 10^{-08} \log(x) + 1.35 \times 10^{-08}] + [2.55 \times 10^{-06} x 2^x]$$

We have used the box brackets to delimit the gate terms and the last wire term. The area and power terms are determined similarly.

Table 5.2: Multiplexer time coefficients for each range of x.

	x	$\log(x)$	1	$x2^x$
(1,15)	0.0802	2.60E-11	1.33E-09	2.72E-06
(2,15)	0.0802	3.97E-12	5.37E-12	2.72E-06
(3,15)	0.0803	3.87E-07	8.95E-09	2.72E-06
(4,15)	0.0805	6.12E-11	9.23E-10	2.71E-06
(5,15)	0.0811	9.98E-11	1.63E-10	2.68E-06
(6,15)	0.0822	3.89E-13	4.60E-13	2.64E-06
(7,15)	**0.0843**	**5.76E-08**	**1.35E-08**	**2.55E-06**
(8,15)	0.0852	7.58E-08	2.18E-09	2.51E-06
(9,15)	0.0877	8.58E-09	1.34E-09	2.41E-06
(10,15)	0.0908	3.19E-06	5.95E-06	2.28E-06
(11,15)	0.0956	9.66E-06	5.65E-07	2.09E-06

In summary, the model equations for a multiplexer are

Multiplexer time optimization model equations

$$\mathcal{T}_t^M = [0.0843x + 5.76 \times 10^{-08} \log(x) + 1.35 \times 10^{-08}] + [2.55 \times 10^{-06} x2^x] \quad (5.1)$$

$$\mathcal{A}_t^M = [3.692^x + 36.66x] + [1.90 \times 10^{-03} x^3 2^x] \quad (5.2)$$

$$\mathcal{P}_t^M = [1.05 \times 10^{-05} x2^x + 0.772^x + 260.48x + 0.77] + [8.51 \times 10^{-05} x^3 2^x] \quad (5.3)$$

Multiplexer area optimization model equations

$$\mathcal{T}_a^M = [0.1574x + 6.62 \times 10^{-08} \log(x) + 2.64 \times 10^{-06}] + [1.05 \times 10^{-05} x2^x] \quad (5.4)$$

$$\mathcal{A}_a^M = [2.8 \times 10^{-05} x2^x + 6.27632^x] + [0 \times x^3 2^x] \quad (5.5)$$

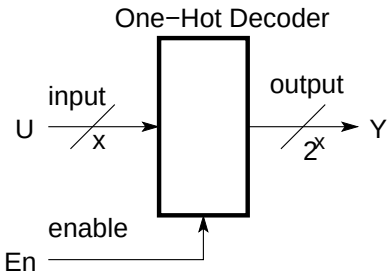
$$\mathcal{P}_a^M = [4.38 \times 10^{-09} x2^x + 0.29392^x + 224x + 0.0034] + [5.1 \times 10^{-13} x^3 2^x] \quad (5.6)$$

5.2 One-Hot Decoder

As noted in Section 2.2, the one-hot decoder is one of the basic modules of the MU-Decoder. Specifically, we use one-hot decoder in building a LUT. It is also one of the conventional decoders. In this section we examine different one-hot decoder designs following the same

general approach and structure used for the multiplexer. We detail only those where it differs from the multiplexer.

A x-to-2^x one-hot decoder is a module that activates (high) one of the 2^x outputs according to the x-bit input. Let the 2^x output bits of the one-hot decoder be $y_{2^x-1}, y_{2^x-2}, \cdots, y_1, y_0$. Let the input bits be $u_{x-1}, u_{x-2}, \cdots, u_1, u_0$. Let the binary number $u_{x-1}u_{x-2}\cdots u_1u_0$ have value i (where $0 \leq i < 2^x$). Then output line y_i is active. Figure 5.7 shows the a x-to-2^x one-hot decoder block diagram and its behavioral Verilog code.




```
module dec_behav(Y, U, En);
    parameter x = 2;
    parameter pow2x = 1<<x;
    input [x - 1:0] U;
    input En;
    output [pow2x-1:0] Y;
    reg [pow2x-1:0] Y;
    reg [pow2x-1:0] Yi;
    always @(U or En)
    begin
        Yi = 1<<U;
        Y = Yi&{pow2x{En}};
    endendmodule
```

Figure 5.7: A x-to-2^x one-hot decoder block diagram (left) and its behavioral Verilog code (right).

The most common ways of implementing a one-hot decoder are based on (a) AND-NOT logic, (b) NAND/NOR logic, and (c) granularity.

5.2.1 Granularity-Based One-Hot Decoder Representation

As in multiplexers, a x-to-2^x one-hot decoder can be built out of smaller one-hot decoders. Figure 5.9 shows a 3-to-2^3 one-hot decoder built out of seven 1-to-2 one-hot decoders. This can be generalized to build a kg-to-2^{kg} one-hot decoder out of g-to-2^g one-hot decoders [29]. Figure 5.10 shows one way to build a x-to-2^x one-hot decoder out of one-hot decoders of size at most g-to-2^g. Here x need not be an integral multiple of g. We will refer to the implantation in Figure 5.10 as a granularity g implementation of a x-to-2^x one-hot decoder. This implementation amounts to a 2^g tree of g-to-2^g one-hot decoder with the root using a h-to-2^h one-hot decoder, where $h = x(\mod g)$.

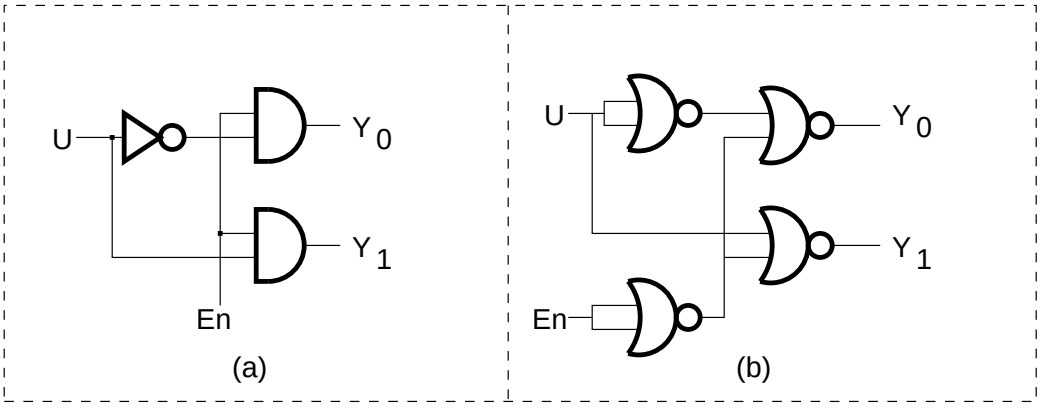


Figure 5.8: Implementations of a 1-to-2 one-hot decoder.

5.2.2 One-Hot Decoder Implementation Results and Model

Figures 5.11 shows the time, area and power for one-hot decoder designs that are time- and area-optimized. As in multiplexer case here too we use the AbsTmin data for time-optimized model and AbsAmin data for the area-optimized one-hot decoder model.

The growth terms identified (see Table 4.4) for the one-hot decoder time are x, $\log x$, 1 (for gate) and $x2^x$ (for wire) and the area growth terms are $x2^x$, 2^x, x, 1 and $x^3 2^x$. In a manner similar to Table 5.1 we determined the best range of x to model $\mathcal{T}_t^1, \mathcal{A}_t^1, \mathcal{P}_t^1, \mathcal{T}_a^1, \mathcal{A}_a^1, \mathcal{P}_a^1$. This results in the following model equations

One-hot decoder time-optimization model equations

$$\mathcal{T}_t^1 = [0.0657x + 1.9 \times 10^{-10} \log(x) + 1.9 \times 10^{-10}] + [1.7 \times 10^{-06} x2^x] \quad (5.7)$$
$$\mathcal{A}_t^1 = [0.1685x2^x + 3.852^x + 9 \times 10^{-18} x] + [0.001x^3 2^x] \quad (5.8)$$
$$\mathcal{P}_t^1 = [9.35 \times 10^{-4} x2^x + 0.65262^x + 859] + [0 \times x^3 2^x] \quad (5.9)$$

One-hot decoder area-optimization model equations

$$\mathcal{T}_a^1 = [0.1647x + 6.54 \times 10^{-10} \log(x) + 3.8 \times 10^{-10}] + [1.96 \times 10^{-6} x2^x] \quad (5.10)$$
$$\mathcal{A}_a^1 = [6.052^x + 1.8x] + [1.82 \times 10^{-06} x^3 2^x] \quad (5.11)$$
$$\mathcal{P}_a^1 = [6.6 \times 10^{-10} x2^x + 5.33 \times 10^{-09} 2^x + 75.8x + 120.2] + [9.4 \times -05x^3 2^x] \quad (5.12)$$

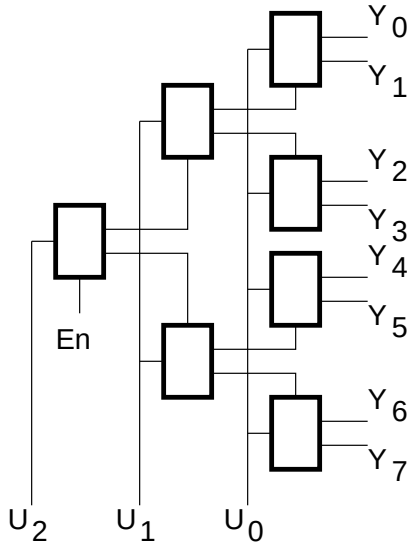


Figure 5.9: Implementation of a 3-to-8 one-hot decoder using 1-to-2 decoders.

5.3 Multicast Hardwiring

The multicast hardwiring building block of a MU-Decoder broadcasts each of the z-bit source words to n output multiplexers. That is, it is a z-to-zn multicast hardwiring. The main idea of the hardwiring unit is to appropriately use buffers to manage fan-outs and route wires to their distribution. Section 4.2 details ideas related to this.

5.3.1 Multicast Hardwiring Implementation Results and Model

To implement a w-to-$w2^x$ hardwiring we need to (i) define a fanout for the input 2^7 was used here and (ii) connect the $w2^x$ outputs to some logic to ensure that the CAD tool uses buffers as needed; we use a AND gate for each of the 2^x output. Subsequently the delay and area of these AND gates were subtracted from the experimentally obtained values to reflect the delay and area for the hardwiring. Figure 5.12 shows the time, area and power for the time-and area-optimized multicast hardwiring designs. We use a different approach for the time compared to the multiplexer and decoder approaches.

Time modeling: Notice from Figure 5.12 that the time graphs are discrete between $x = 7$ and $x = 10$. This is when new buffers are likely added; note that for a given x, the circuit has a 2^x fanout and the assumed limit for fanout was 2^7. We use the growth terms of the multicast hardwiring identified in Table 4.2 to model the time. We first separately model the $x \leq 7$ and $x \geq 10$ cases. Let these yield equations $\mathcal{T}(x, z)$ and $\mathcal{T}'(x, z)$. We use $\mathcal{T}(x, z)$ as

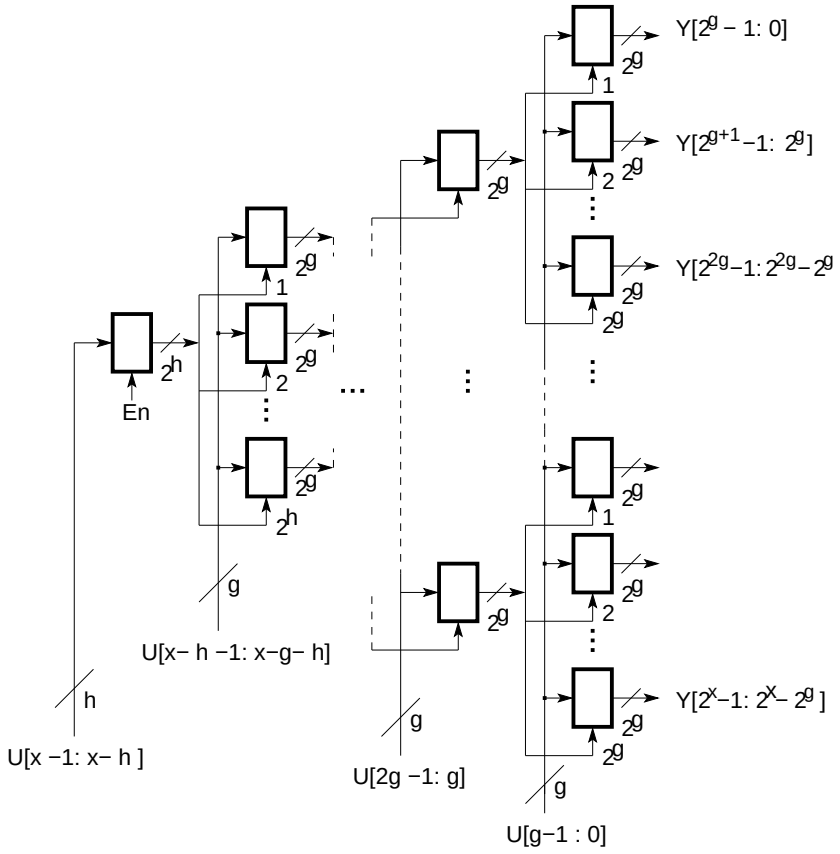


Figure 5.10: Granularity-based one-hot decoder implementations.

the wire delay model by extending $\mathcal{T}$ to $x \in \{8, 9, 10\}$ and $\mathcal{T}'$ to $x \in \{7, 8, 9\}$, we now have $\mathcal{T}$ and $\mathcal{T}'$ for $x \in \{7, 8, 9, 10\}$, for all values of w. The difference $d(x, z) = \mathcal{T}'(x, z) - \mathcal{T}(x, z)$ for $x \in \{7, 8, 9, 10\}$ gives the data to model buffer delay $d(x, z)$ (with same growth terms as $\mathcal{T}$) which turn out to be independent of x, In fact for time optimized design

$$d_t(x, z) = [2.22 \times 10^{-15}x + 0.8311] + [2.22 \times 10^{-15}w2^x] \tag{5.13}$$

and for the area-optimized design

$$d_a(x, z) = [2.22 \times 10^{-15}x + 0.9236] + [2.22 \times 10^{-15}w2^x] \tag{5.14}$$

Notice that $d_t(x, z), d_a(x, z)$ are both nearly independent of w and x. So the gate delay could be written as d_ω for $\omega \in \{t, a\}$. The overall time for a w-to-$w2^x$ multicast hardwiring is given by

$$\mathcal{T}_\omega^H = \mathcal{T}_\omega(x, z) + \left\lceil \frac{x}{7} \right\rceil d_\omega$$

For our data $\mathcal{T}_t(x, z) = [0.0015x + 1.1 \times 10^{-12}] + [2.38 \times 10^{-5}w2^x]$ and $\mathcal{T}_a(x, z) = [0.0015x + 1.1 \times 10^{-12}] + [2.38 \times 10^{-5}w2^x]$. Those along with Equations 5.13 and 5.14 gives the following model equations for the time model of multicast hardwiring

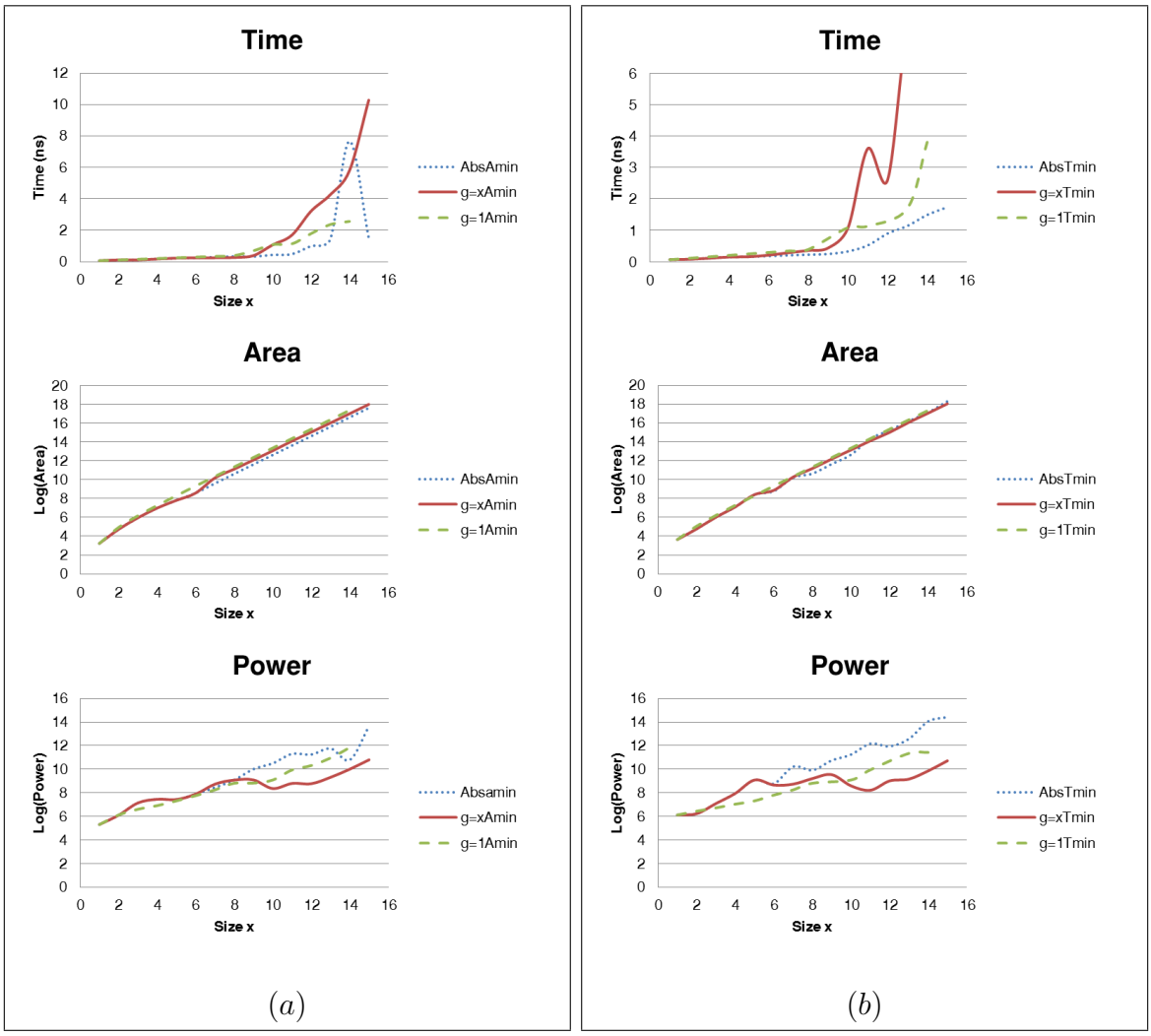


Figure 5.11: One-hot decoder (a) time-and (b) area-optimized results

Time model equations for multicast hardwiring

$$\mathcal{T}_t^H = [0.0015x + 1.1 \times 10^{-12}] + [2.38 \times 10^{-5} w 2^x] + \left\lceil \frac{x}{7} \right\rceil [0.8311] \tag{5.15}$$

$$\mathcal{T}_a^H = [0.0015x + 1.1 \times 10^{-12}] + [2.38 \times 10^{-5} w 2^x] + \left\lceil \frac{x}{7} \right\rceil [0.9236] \tag{5.16}$$

Area and power curves in Figure 5.12 shows the area and power of the time- and area-optimized designs. The growth terms of the area are shown in Table 4.2. Here as buffers are gradually included as required and the jump at $x = 7$ is more gradual. Here we model

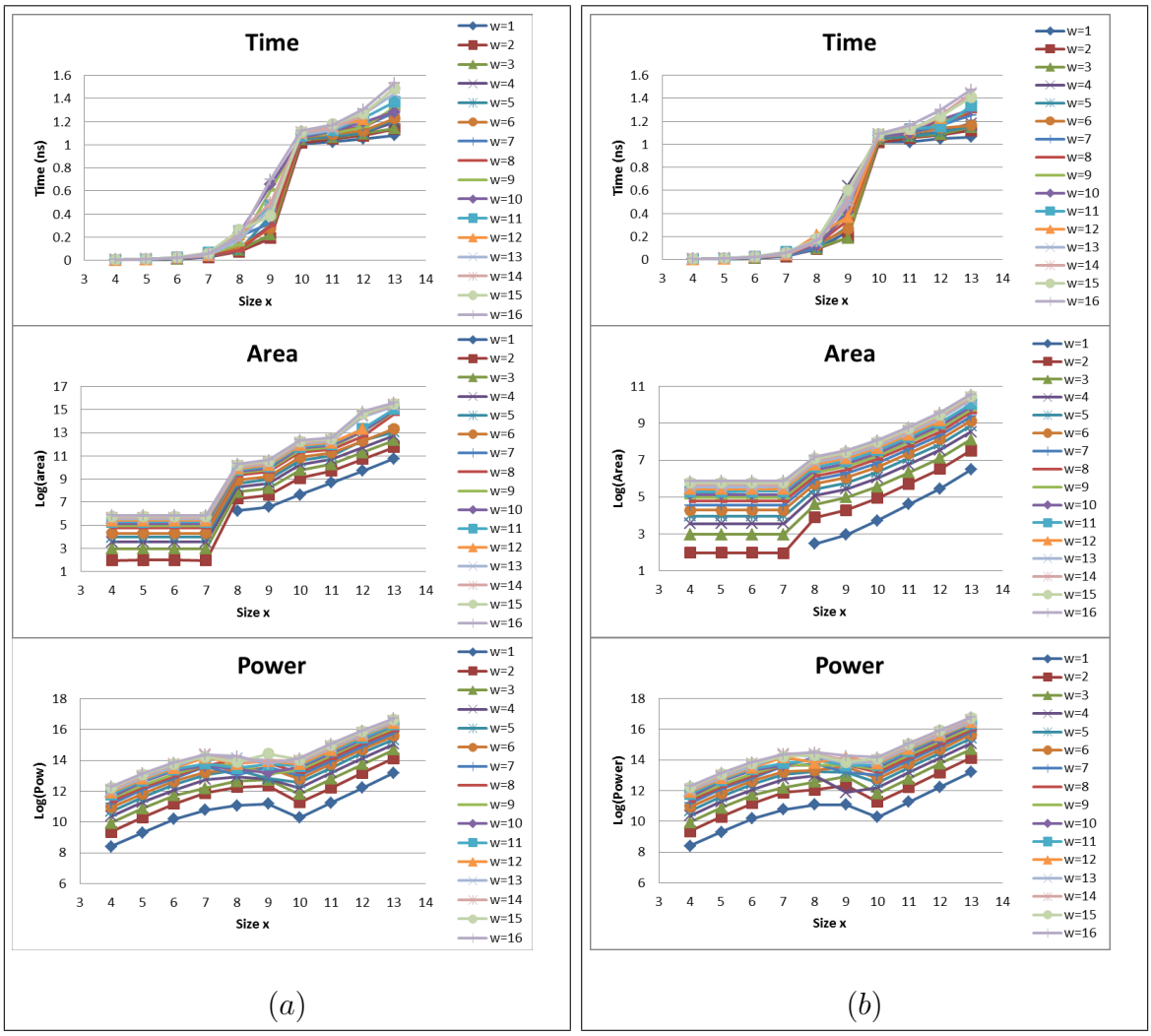


Figure 5.12: Multicast hardwiring time- and area-optimized implementation results

$\mathcal{A}_t^H, \mathcal{A}_a^H, \mathcal{P}_t^H, \mathcal{P}_a^H$ in the usual way for $x > 7$ (as in the multiplexer) to obtain the following model equations.

Area for multicast hardwiring

$$\mathcal{A}_t^H = [0.2036w2^x + 1.96 \times 10^{-05}2^x + 2.06 \times 10^{-04}w + 1.5 \times 10^{-04}] + [9.6 \times 10^{-04}w^2x2^x] \quad (5.17)$$

$$\mathcal{A}_a^H = [0.0106w2^x + 3.25 \times 10^{-06}2^x + 4.584w + 3.18] + [1.9 \times 10^{-06}w^2x2^x] \quad (5.18)$$

Power for multicast hardwiring

$$\mathcal{P}_t^H = [0.7053w2^x + 0.75362^x + 805.5w + 892.8] + [9.24 \times 10^{-11} w^2 x 2^x] \quad (5.19)$$
$$\mathcal{P}_a^H = [0.7232w2^x + 0.6852^x + 849.4w + 885] + [1.81 \times 10^{-08} w^2 x 2^x] \quad (5.20)$$

5.4 Look-Up Table

The look-up table was described in Section 2.2 and 4.6. It's use in this work is twofold. It is an important building block of the MU-Decoder and it is also to be used as a LUT-Decoder against which the MU-Decoder is compared.

The basic building block of a LUT (or SRAM memory) is a single bit storage element. Figure 4.7 shows the standard 6-transistor–(6T) based implementation [18] and its logic diagram. While our cell implementation as a latch is considerably more expensive than the 6T configuration, we account for this in the model as explained in Section 5.4.2.

5.4.1 Granularity-Based LUT Representation

A 2^x-to-w LUT can be implemented using a $2^x \times w$ cell array with a $x - to - 2^x$ decoder for the address. One could also arrange the $w2^x$ cells as a $2^{x-g} \times w2^g$ array with a $(x-g)$-to-2^{x-g} row one-hot decoder and g-to-2^g one-hot column decoder where outputs are broadcast to w columns (Figure 4.7). We use g as a measure of the granularity of the LUT.

5.4.2 Implementation Results and Model

Figure 5.13 shows the LUT time, area and power for the time- and area-optimized designs. These when modeled using the growth terms of Table 4.6 yield the following equations for time

Time model equations for LUT

$$\mathcal{T}_t^L = [0.3171x + 0 \times \log_2 x + 0.0984 \log_2 w + 0.4958] + [5.6 \times 10^{-05} w2^x] \quad (5.21)$$
$$\mathcal{T}_a^L = [1.11x + 3.86 \times 10^{-15} \log_2 x + 0.2976 \log_2 w + 2.25 \times 10^{-15}] + [3.86 \times 10^{-05} w2^x] \quad (5.22)$$

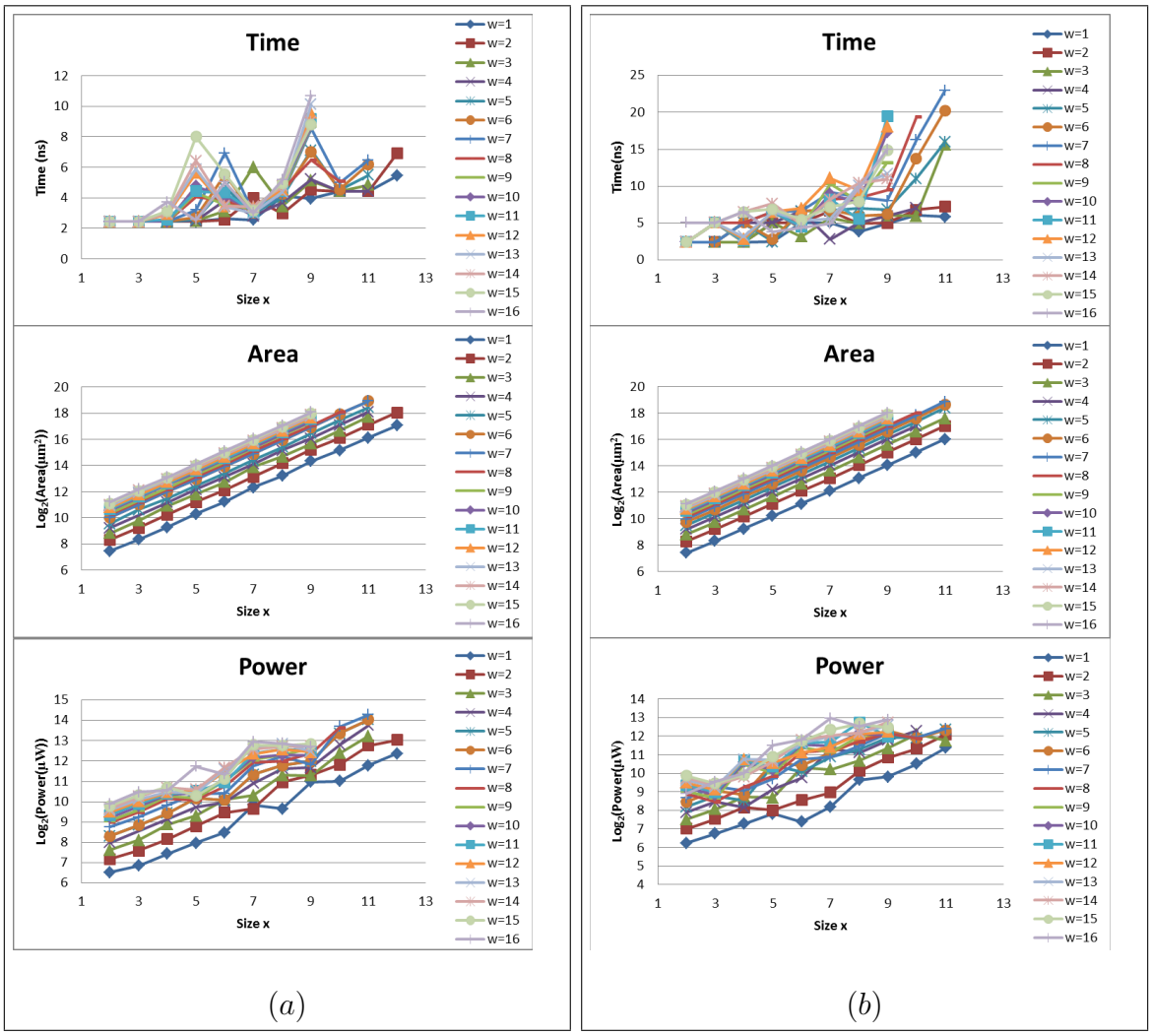


Figure 5.13: Multicast hardwiring (a) time- and (b) area-optimized implementation results

For area we have the following preliminary equations

$$\mathcal{A}_t^L = [34.38w2^x + 0.03472^x + 0 \times w + 0] + [0 \times w^2 2^x] \tag{5.23}$$

$$\mathcal{A}_a^L = [32.79w2^x + 0.36912^x + 0 \times w + 0] + [0 \times w^2 2^x] \tag{5.24}$$

Notice that the $w^2 2^x$, w and 1 terms suggested by the theoretical model have 0 coefficients. That is, the area (and power as shown later) are directly proportional to the cell area $w2^x$. The coefficient $w2^x$ must be adjusted, however, as our latch-based implementation is much larger than the 6T implementation.

We used a hand made design with 500 nm technology for a 6T implementation of a single cell. We compared this with a 500 nm latch implementation of a cell and the hand made design had an area that was 23 times smaller. In translating this to the 45 nm technology that we use for the LUT and to factor in a margin of advantage for the LUT-Decoder due to newer technologies [16] we used a conservative factor of 50 to reduce the LUT area. Thus, the model equations for the LUT area are

Area model equations for LUT

$$\mathcal{A}_t^L = \left[\frac{1}{50}(34.38w2^x) + 0.03472^x\right] + [0] = [0.6876w2^x + 0.03472^x] + [0] \tag{5.25}$$

$$\mathcal{A}_a^L = \left[\frac{1}{50}(32.79w2^x) + 0.36912^x\right] + [0] = [0.6557w2^x + 0.36912^x] + [0] \tag{5.26}$$

Notice that $\mathcal{A}_t^L \cong \mathcal{A}_a^L$ that is there is no difference in area due to time and area optimizations.

This factor of 50 was further verified using CACTI tool [3] that has an option for RAMs, for $w = 8$ and 16.

We did not find the power for the hand made design. A comparison using CACTI showed a difference by a factor of about 2, mostly in favor of our design. Therefore we use the power model as it is without an adjustment factor.

power model equations for LUT

$$\mathcal{P}_t^L = [1.18w2^x + 0.22362^x + 55.6w + 1005] + [9.5487 \times 10^{-08}w^2 2^x] \tag{5.27}$$

$$\mathcal{P}_a^L = [0w2^x + 0.652^x + 62.4w + 0] + [0.02w^2 2^x] \tag{5.28}$$

Chapter 6

The MU-Decoder

Recall the MU-Decoder introduced in Section 2.2. In this chapter, we use the models for the multiplexers, multicast hardwiring and look-up table (LUT) developed in Chapter 5 to construct a model for the MU-Decoder.

Recall that an m-to-n decoder's output can be viewed as a subset of $\mathbb{Z}_n = \{0, 1, 2, \cdots, n-1\}$ (Section 2.2). We will use this idea to compare the performance of the MU-Decoder to the LUT-Decoder; a m-to-n LUT-Decoder is simply a $2^m \times n$ LUT. We show that the LUT-Decoder performs better than the MU-Decoder when the set of subsets to be generated is totally arbitrary (this entails extremely conservative assumptions on the nature of the subsets used). Most problems in practice display some structure, so the subsets used by them are not arbitrary and unrelated. Therefore, the above "better performance" of the LUT-Decoder may be misleading. To establish this assertion, we examine a class of subsets called "totally ordered subsets" that have practical applications in problem such as reductions. For totally ordered subsets the MU-Decoder now outperforms the LUT-Decoder by a much larger margin.

In the next section we revisit the structure of the MU-Decoder and cast it in terms of the three building blocks, multiplexers, multicast hardwiring, and LUTs. We also derive equations that model the delay, area and power of the MU-Decoder, in terms of the corresponding quantities derived in Sections 5.1–5.4. In Section 6.2 we compare the LUT-Decoder and the MU-Decoder based on the number of subsets they can generate.

6.1 The MU-Decoder Structure and Model

Recall the structure of the MU-Decoder in Figure 2.1 (page 7). This figure shows the selector as a single monolithic collection of LUTs. Here we separate this collection and pair each output multiplexer with one small LUT from this collection. The resulting structure is shown

in Figure 6.1. For easier reference, the 2^m-to-z LUT will be called LUT1 and the collection of LUTs in the selector unit as LUT2 (each LUT2(i) is a 2^y-to-log z LUT). Individual LUTs in this collection may be called LUT2(i) with some appropriate value of $0 \leq i < n$. The z-to-zn multicast hardwiring unit will be called HW1, whereas the y-to-yn multicast hardwiring unit in the selector unit will be called as HW2. As in the case of LUT2(i), each individual z-to-1 multiplexer will be called MUX(i). In fact, MUX(i) and LUT2(i), will be paired to form output i. Notice that while the area of the MU-Decoder is the sum of the areas of its building blocks, the time is due to longer of the two paths (from inputs M and Y) to the output N.

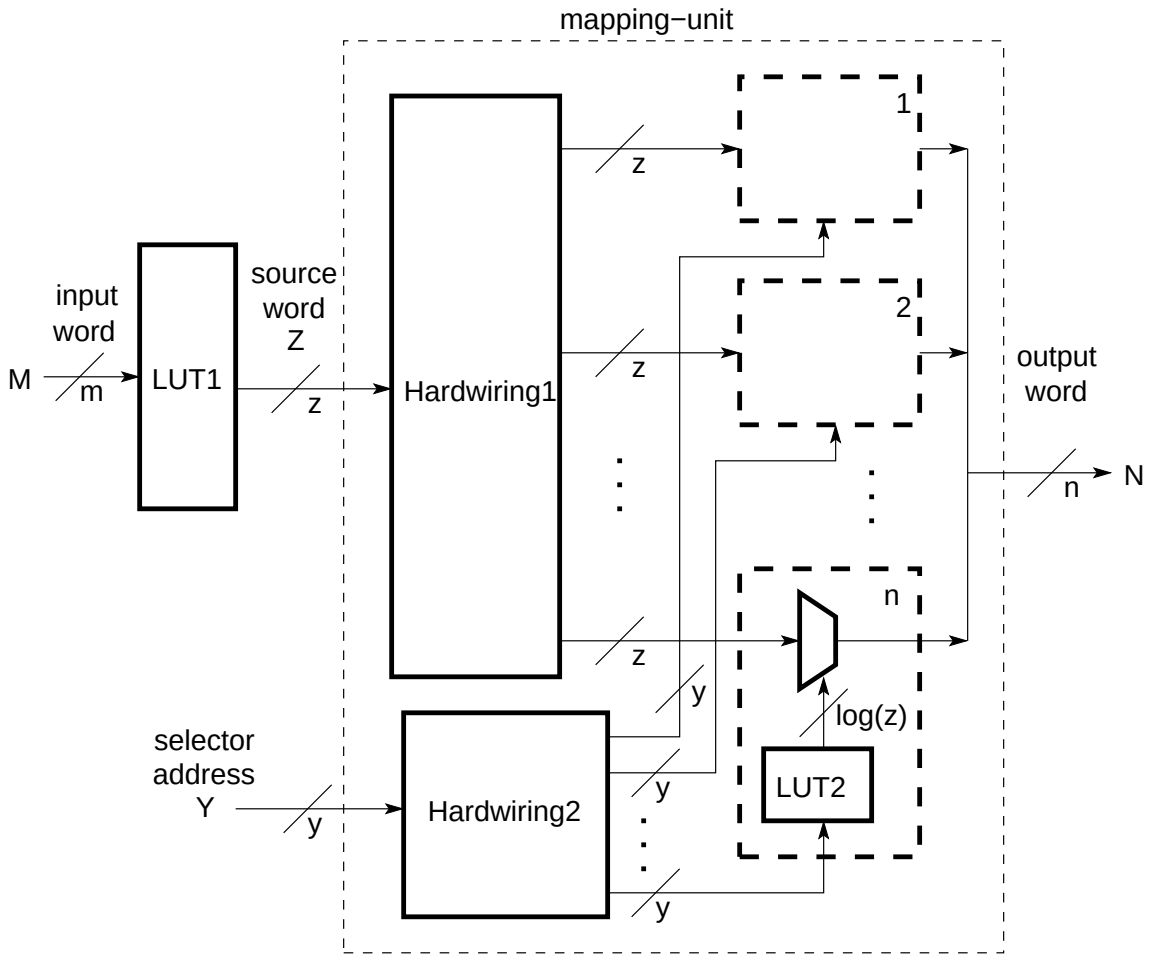


Figure 6.1: Structure of MU-Decoder $MU(m, y, z, n)$.

Thus the area of the MU-Decoder is

$$A = A_{L1} + A_{H1} + A_{H2} + (A_{L2} + A_M)n$$

and the time is

$$T = \max(t_{L1} + t_{H1}, t_{L2} + t_{H2}) + t_M$$

and the power is modeled similar to the area, therefore power is

$$P = P_{L1} + P_{H1} + P_{H2} + (P_{L2} + P_M)n$$

Notice that in Figure 6.1 different modules could be optimized for different performance measures. For example, if the path from M to N is longer (that is, with higher delay) than that from Y to N, then each LUT2(i) and HW2 can be area-optimized, without significant change in delay of the MU-Decoder. This sort of optimization is possible within a path as well. For example if the HW1 takes less time than LUT1, then LUT1 could be time-optimized while HW1 is area-optimized.

The theoretical analysis of the LUT- and MU-Decoder [10] has shown that their delays are all $O(\log n)$. It is in the area that large differences can appear. We will therefore consider reduction in areas of the decoder as the primary objective. Subject to this objective, we will use time-optimized designs.

It was observed in Section 5.4 that the area of time-optimized and area-optimized LUT implementation are nearly the same. Therefore, we use area-optimized LUTs in all cases (including for the LUT-Decoder). Figure 6.2 shows a typical contribution of various modules of the MU-Decoder. It can be seen that HW1 is the main non-LUT contributor of the MU-Decoder area. We will use area-optimized designs for the HW1. All other non-LUT components (namely multiplexers and HW2) will be time-optimized.

Because of variations in technology and other local parameters, the results are more meaningful when presented relative to each other, rather than as absolute values. We present our data for $5000 \leq n \leq 10^{11}$. However, $n = 10^{11}$ is of purely academic interest. Current FPGAs (for example XILINX Virtex 7series [31]) use around two million logic blocks. Here $n \cong 2 \times 10^6$. Therefore we present two sets of results, one for $5000 \leq n \leq 10^6$ and the other for $5000 \leq n \leq 10^{11}$.

6.2 Basis for Comparing LUT- and MU-Decoders

The two configurable decoders that are compared are (a) LUT-Decoder and (b) MU-Decoder. We will use the subsets that each decoder can generate as a basis for the comparison; that is, we will compare decoders that are guaranteed to generate the same set of subsets. To make these ideas specific let LUT(m,n) denote a $2^m \times n$ LUT-Decoder, and let $MU(m, y, z, n)$ denote a MU-Decoder as described in Figure 6.1. Suppose we are given a set $\mathcal{S} = \{S_0, S_1, \cdots, S_{\ell-1}\}$ of subsets of $\mathbb{Z}_n$. Let LUT(m,n) and $MU(m, y, z, n)$ be the "smallest" LUT- and MU-Decoder that can generate S; while the idea of smallest is clear for a LUT-Decoder, it need further clarification for a MU-Decoder (see next two sections). We now consider two classes of subsets of $\mathbb{Z}_n$.

6.2.1 MU-Decoder Configuration for Arbitrary Subsets

Let $\mathcal{S} = \{S_0, S_1, \cdots, S_{\ell-1}\}$ be a set of arbitrary (unrelated) subsets of $\mathbb{Z}_n$. The following results have been established in Jordan and Vaidyanathan [9, 10].

Theorem 6.2.1 [9, 10] For any $1 \leq m < 2^n - 1$, a LUT(m, n) can produce 2^m arbitrarily selected subsets of $\mathbb{Z}_n$. The gate cost of the LUT is $O(2^m(m+n))$. ■

Notice that to produce 2^{m_1} subsets of $\mathbb{Z}_n$ a $2^{m_1} \times n$ LUT-Decoder is required. We consider three values of $m_1 \in \{\log n - \log\log n, \log n \text{ and } 2\log n\}$ that produce $\frac{n}{\log n}, n$ and n^2 subsets of $\mathbb{Z}_n$. These values represent a reasonable range of operation and will largely be representative of other values that we could consider.

Theorem 6.2.2 [9, 10] For any $1 \leq m < 2^n - 1$, $z \leq n$ and $y \geq 0$, a $MU(m, y, z, n)$ can produce at least $\min(2^n, 2^y \lfloor \log_2 z \rfloor)$ arbitrary subsets of $\mathbb{Z}_n$. The MU-Decoder has a gate cost of $O(2^m(m+z) + n\log z(z + 2^y))$. ■

This theorem, although accurate, is reflective of a worst case scenario in which each source word of the MU-Decoder with all 2^y values of Y produces only one subset of $\mathcal{S}$; That is although it could produce as many as 2^y subsets, only one which belongs to $\mathcal{S}$ is considered of interest. We show in Section 6.2.2 that the $MU - Decoder$ can do much better.

We now use the results of Theorems 6.2.1 and 6.2.2 to determine appropriate values for m_2, y and z for a given set of 2^{m_1} arbitrary subsets of $\mathbb{Z}_n$.

From Theorems 6.2.1 and 6.2.2, in order to make the subsets equal for both decoders,

$$2^{m_1} = \min\{2^{m_2}, 2^y \lfloor \log_2 z \rfloor\} \tag{6.1}$$

Since $m_1 = m_2$, we will select $2^y \lfloor \log_2 z \rfloor$ to be no less than 2^{m_1}. In fact we should set

$$2^y \lfloor \log_2 z \rfloor = 2^{m_1} \tag{6.2}$$

From Theorem 6.2.2, the gate cost of the MU-Decoder is $O(2^{m_2}(m_2 + z) + n\log z(z + 2^y))$. To make the last term optimal we will set $z \cong 2^y$. This implies that $y \cong \log_2 z$. Then $2^{m_2} = 2^{m_1} = 2^y \lfloor \log_2 z \rfloor \cong y2^y = z\log z$.

Observe that if $a = \frac{b}{\log b}$ then $a\log a = \Theta(b)$. Here we have a requirement that $z\log z = 2^{m_1}$, so we select $z = \left\lceil \frac{2^{m_1}}{m_1} \right\rceil$. Since $2^{m_1} = 2^y \lfloor \log z \rfloor$, we have $y = m_1 - \lfloor \log \lfloor \log_2 z \rfloor \rfloor$. The floor on the outer log ensures a slightly larger value of y to ensure the condition of Equation 6.1. Finally to ensure that $z < n$ we select a constant $c = 16$ and require $z \leq \frac{n}{c}$. In summary,

for arbitrary subset,

$$m_1 \in \{\log n - \log\log n, \log n, 2\log n\}$$

$$m_2 = m_1$$

$$z = \left\lceil \min\left(\frac{n}{c}, \frac{2^{m_1}}{m_1}\right)\right\rceil$$

$$y = m_1 - \lfloor \log \lfloor \log z \rfloor \rfloor$$

Figure 6.2 shows the contribution of the MU-Decoder building blocks for $m_1 = \log n - \log\log n$, $\log n, 2\log n$ and $5000 \leq n \leq 10^{11}$. Clearly, HW1 is the main contribution to the MU-Decoder area, particularly as n becomes larger. Figure 6.3, and 6.4 show the ratio of the areas of the MU-Decoder to that of the LUT-Decoder, plotted against n. This ratio is small for a while (around 2) and then reises rapidly. This rapid rise is due to the high cost of HW1 (see Figure 6.2). The choice of values for m_1 simply dictates when contribution of HW1 predominates. In general, the larger the value of m_1, the longer the MU-Decoder area is closer to that of the LUT-Decoder.

At this point one may ask what the use of the MU-Decoder is, when its area is always greater than that of the LUT-Decoder. To see this, it may be instructive to see how a MU-Decoder produces subsets; details appear in Jordan and Vaidyanathan [10].

A $MU(m, y, z, n)$ produces 2^m different source words each z-bit long. The y-bit selector address generates 2^y different ways to multicast a source word. The multicasts have been selected in such a way that each multicast can to generate at least $\lfloor \log z \rfloor$ arbitrary subsets. In the process it uses up at most $\lfloor \log z \rfloor$ of the 2^m source words available to it. Thus, each source word (in the worst case) is used with exactly one multicast to produce exactly one subset of interest. Since there are 2^m source words, 2^x interesting subsets are generated.

What is missed in this worst case analysis is that (a) a multicast may generate more than $\lfloor \log z \rfloor$ subsets and (b) each source word may generate an interesting subset with more than one multicast. This greatly improves the utility of the MU-Decoder.

In the next section we consider "Totally-Ordered subsets" whose structure the MU-Decoder exploits, and which the LUT-Decoder fails to use.

6.2.2 MU-Decoder Configuration for Total-Order Subsets

The following definitions and results are from Jordan and Vaidyanathan [10].

Definition 6.2.1 [10] A set $\mathcal{S} = \{S_i \subset \mathbb{Z}_n : 0 \leq i < k\}$ is said to be *totally-ordered*, iff $S_i \subset S_{i+1}$ for all $0 \leq i < k-1$.

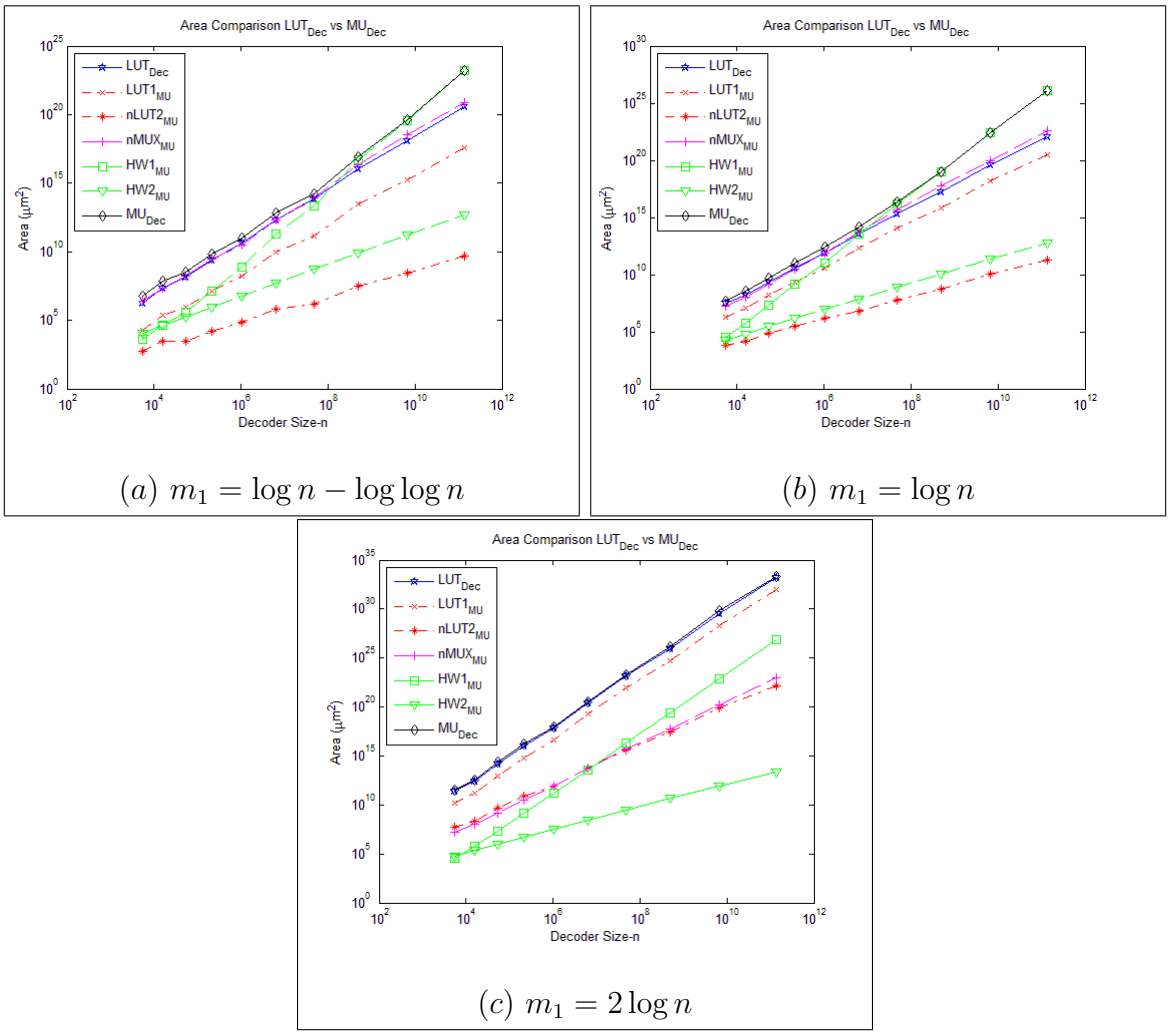


Figure 6.2: Contribution of building blocks for arbitrary subsets.

A totally-ordered subset may be encountered in several computations including fan-in converge-cast or broadcast/multicast.

Definition 6.2.2 [10] Two sets of subsets $\mathcal{S}_1$, and $\mathcal{S}_2$ are isomorphic iff $|\mathcal{S}_1| = |\mathcal{S}_2|$ and for each $S_i \in \mathcal{S}_1$ there is a unique $S_j \in \mathcal{S}_2$ such that $|S_i| = |S_j|$

For $0 \leq i < \Gamma$, let $\mathcal{S}_i = \{S_{i,0}, S_{i,1}, \cdots, S_{i,\ell-1}\}$ be a set of subsets of $\mathbb{Z}_n$. For any i, i' let $\mathcal{S}_i$ and $\mathcal{S}_{i'}$ be isomorphic. In fact let $|S_{i,j}| = |S_{i',j}|$. If each $\mathcal{S}_i$ is totally-ordered, then set $\mathcal{S} = \bigcup_{i=0}^{\Gamma-1} \mathcal{S}_i$ is a set of Γ isomorphic set of subsets, each a totally-ordered set of ℓ subsets of $\mathbb{Z}_n$. We call such an $\mathcal{S}$ as a (Γ, ℓ) *totally ordered* set. In this notation a single totally-ordered

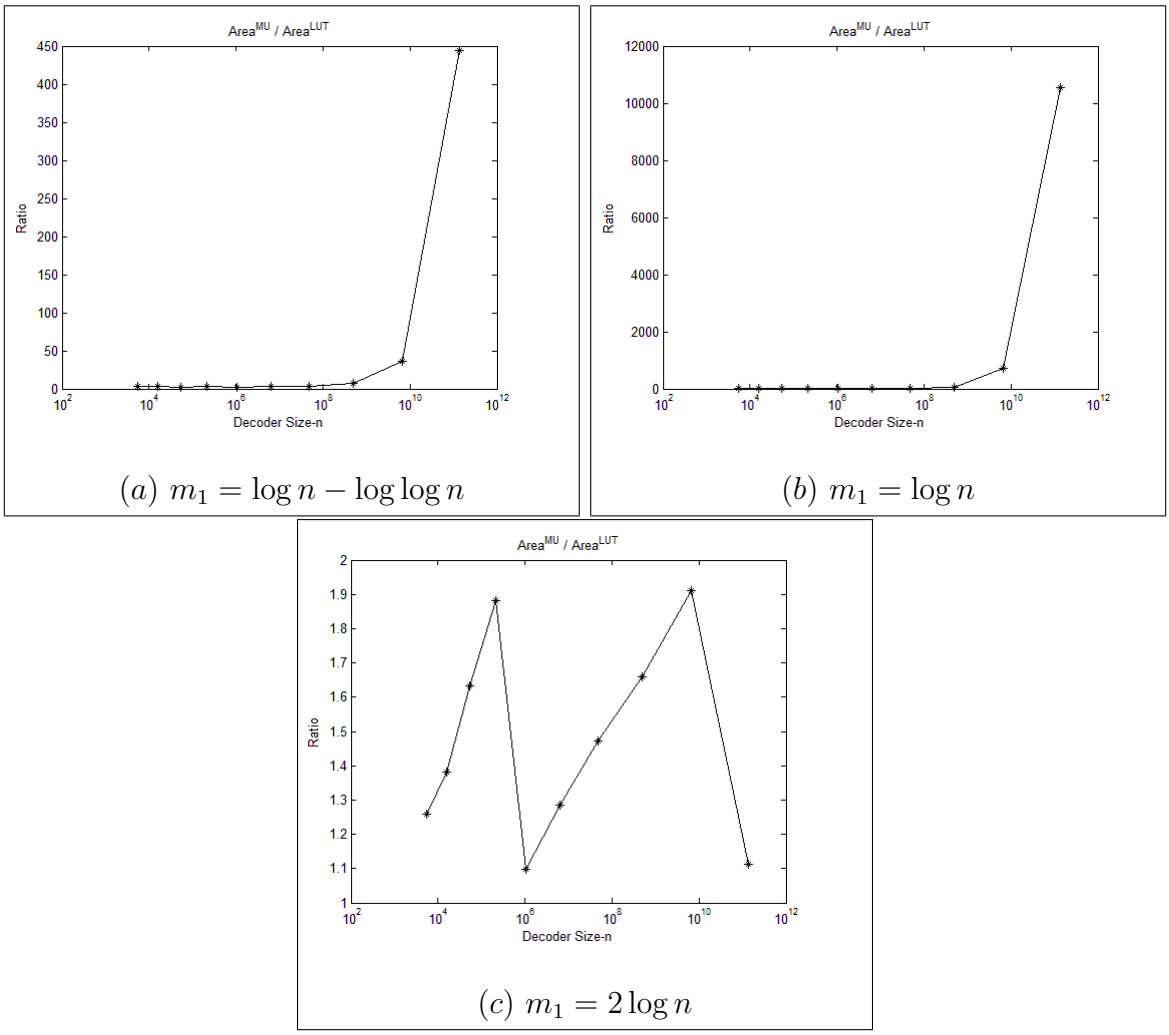


Figure 6.3: Ratio of MU- to LUT-Decoder areas for arbitrary subsets ($5000 \leq n \leq 10^{11}$).

set is a $(1, \ell)$ totally ordered set and an arbitrary set of subsets is a $(\Gamma, 1)$ totally ordered set.

The following theorem is a small extension to the results of Jordan and Vaidyanathan [10].

Theorem 6.2.3 For any $1 < m \leq 2^n - 1$, $z \leq n$, $y \geq 0$, $\Gamma \geq 1, and \ell \geq 2$, a MU-Decoder, $MU(m, y, z, n)$ can produce a (Γ, ℓ) totally ordered set of $\mathbb{Z}_n$ if $\Gamma\ell = \min\{2^m\Gamma, 2^y(z-1)\}$. ■

For our analysis we assume $\Gamma = 2^\gamma$ and $\ell = 2^\lambda$. There are 2^γ sets of isomorphic subsets, each containing 2^λ totally-ordered subsets. To generate these $2^{\gamma+\lambda}$ subsets on a $LUT(m, n)$

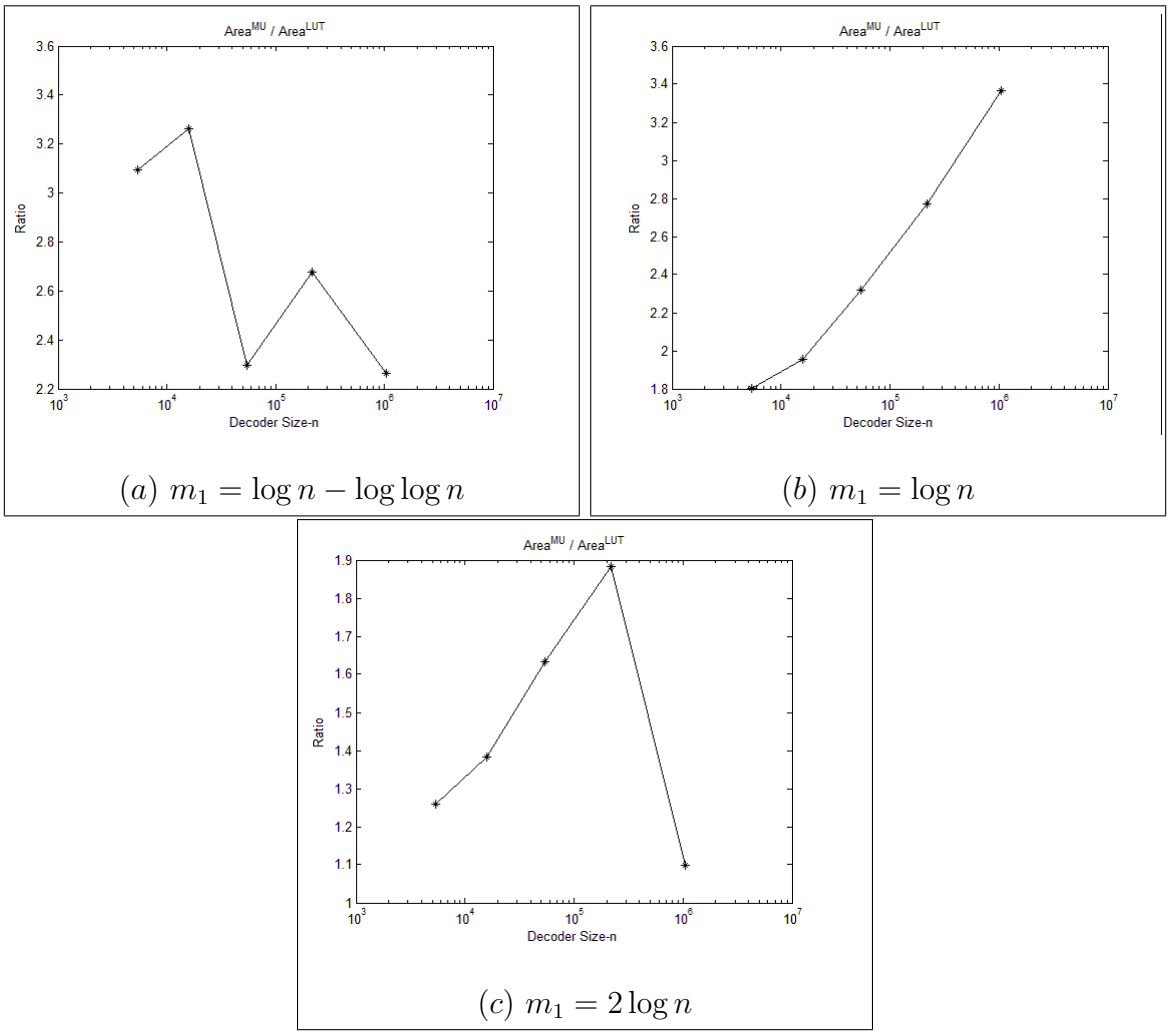


Figure 6.4: Ratio of MU- to LUT-Decoder areas for arbitrary subsets ($5000 \leq n \leq 10^6$).

we require $m_1 = \gamma + \lambda$. That is a LUT simply uses the fact that there are $2^{\gamma+\lambda}$ subsets of interest and fails to exploit the structure of these subsets.

On a MU-Decoder, $MU(m, y, z, n)$, the number of subsets $2^{\gamma+\lambda}$ satisfies (by Theorem 6.2.3)

$$2^{m_1} = 2^{\gamma+\lambda} = \min\{2^m 2^\gamma, (z-1)2^y\} = \min\{2^{m_1}, (z-1)2^y\}$$

The idea is to set $2^{m_2+\gamma} = 2^{\gamma+\lambda}$; that is $m_2 = \lambda$. Since $m_1 = \gamma + \lambda$, we have $m_2 = m_1 - \gamma$. We then select z and y, so that $(z-1)2^y \geq 2^{m_2+\gamma} = 2^{m_1}$. As before, let $z = 2^y$ (to balance the gate cost of the MU-Decoder). Then $(z-1)2^y \cong z^2 \geq 2^{m_1}$. We select $z = \left\lceil 2^{\frac{m_1}{2}} \right\rceil$ and $y = \lceil \log z \rceil$.

With this selection $z < n$ results in $m_1 < 2\log n$. For the $m_1 \in \{\log n - \log\log n, \log n\}$ cases we can select $z = \left\lceil 2^{\frac{m_1}{2}} \right\rceil$. However when $m_1 = 2\log n$, selecting $z = \left\lceil 2^{\frac{m_1}{2}} \right\rceil$ will set $z = n$ (or $\frac{n}{c}$ if it is upper bounded as in the arbitrary case). This will not let the MU-Decoder fully exploit the structure of totally-ordered sets. Here we take a different approach for the $m_1 = 2\log n$.

Select $y = \left\lceil \frac{m_1\epsilon}{1+\epsilon} \right\rceil$ for some constant $\epsilon > 1$ and $z = 1 + \left\lceil 2^{\frac{y}{\epsilon}} \right\rceil$. Now $(z-1)2^y = \left\lceil 2^{\frac{y}{\epsilon}} \right\rceil 2^y \geq 2^{y(1+\frac{1}{\epsilon})}$. Since $y = \left\lceil \frac{m_1\epsilon}{1+\epsilon} \right\rceil$, we have $(z-1)2^y \geq 2^{y(1+\frac{1}{\epsilon})} = 2^{\left\lceil \frac{m_1\epsilon}{1+\epsilon} \right\rceil \left(\frac{1+\epsilon}{\epsilon}\right)} \geq 2^{m_1}$ as required to satisfy equation. For our implementation $\epsilon = 2.5$ worked the best.

In summary,

for totally ordered subset,

$$m_1 \in \{\log n - \log\log n, \log n, 2\log n\}$$

$$m_2 = m_1 - \gamma$$

$$z = \begin{cases} \left\lceil 2^{\frac{m_1}{2}} \right\rceil, & \text{if} \quad m_1 < 2\log n \\ 2^{\frac{y}{\epsilon}}, & \text{if} \quad m_1 = 2\log n \end{cases}$$

$$y = \begin{cases} \lceil \log z \rceil, & \text{if} \quad m_1 < 2\log n \\ \left\lceil \frac{m_1\epsilon}{1+\epsilon} \right\rceil, & \text{if} \quad m_1 = 2\log n \end{cases}$$

We consider three values for γ, 0, $\left\lceil \frac{y}{2} \right\rceil$, and y, representing a single total-ordered set at one extreme and 2^y totally ordered sets are the other extreme.

In all these cases $m_2 = m_1 - \gamma$ is reduced compared to the arbitrary subset case. The quantity $z \cong \sqrt{2^{m_1}}$ is again smaller than the $z \cong \frac{2^{m_1}}{m_1}$ in the arbitrary case. Consequently $y = \log z$ is also reduced. As a results all building blocks are much smaller now.

Figures 6.5 show the contributions of the building blocks for $\gamma = 0$. Clearly multiplexer is the predominant contributor for the range of values of n that are of interest. Figures 6.6 and 6.7 show similar behavior for $\gamma = \frac{y}{2}$ and y. Even for the $m_1 \in \{\log n, 2\log n\}$ cases, the MU-Decoder is superior (see Figure 6.8). Here the cost HW1 slowly creeps up but is still small enough to not make a big difference.

Figures 6.8, – 6.13 show the ratio of the areas of the LUT-Decoder to that of the MU-Decoder. Clearly there is a marked improvement in the MU-Decoder's performance. For the $n = 10^6$, $m_1 = \log n$ case and $\gamma = 0$ (the most conservative case) the MU-Decoder has 33 times less area than the LUT-Decoder. At the other extreme for $n = 10^6, m_1 = 2\log n$ and $\gamma = y$, MU-Decoder area is 164 times better.

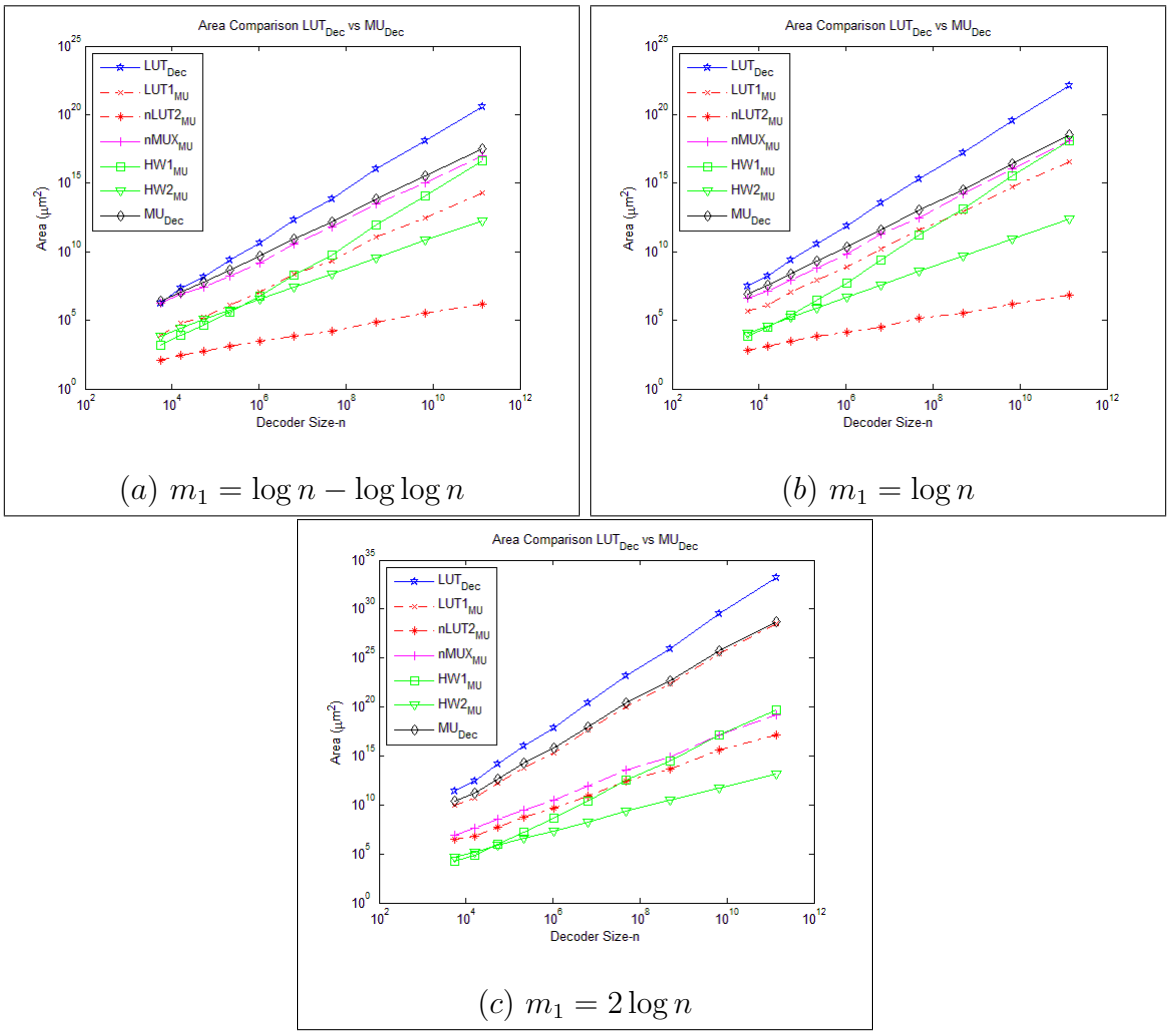
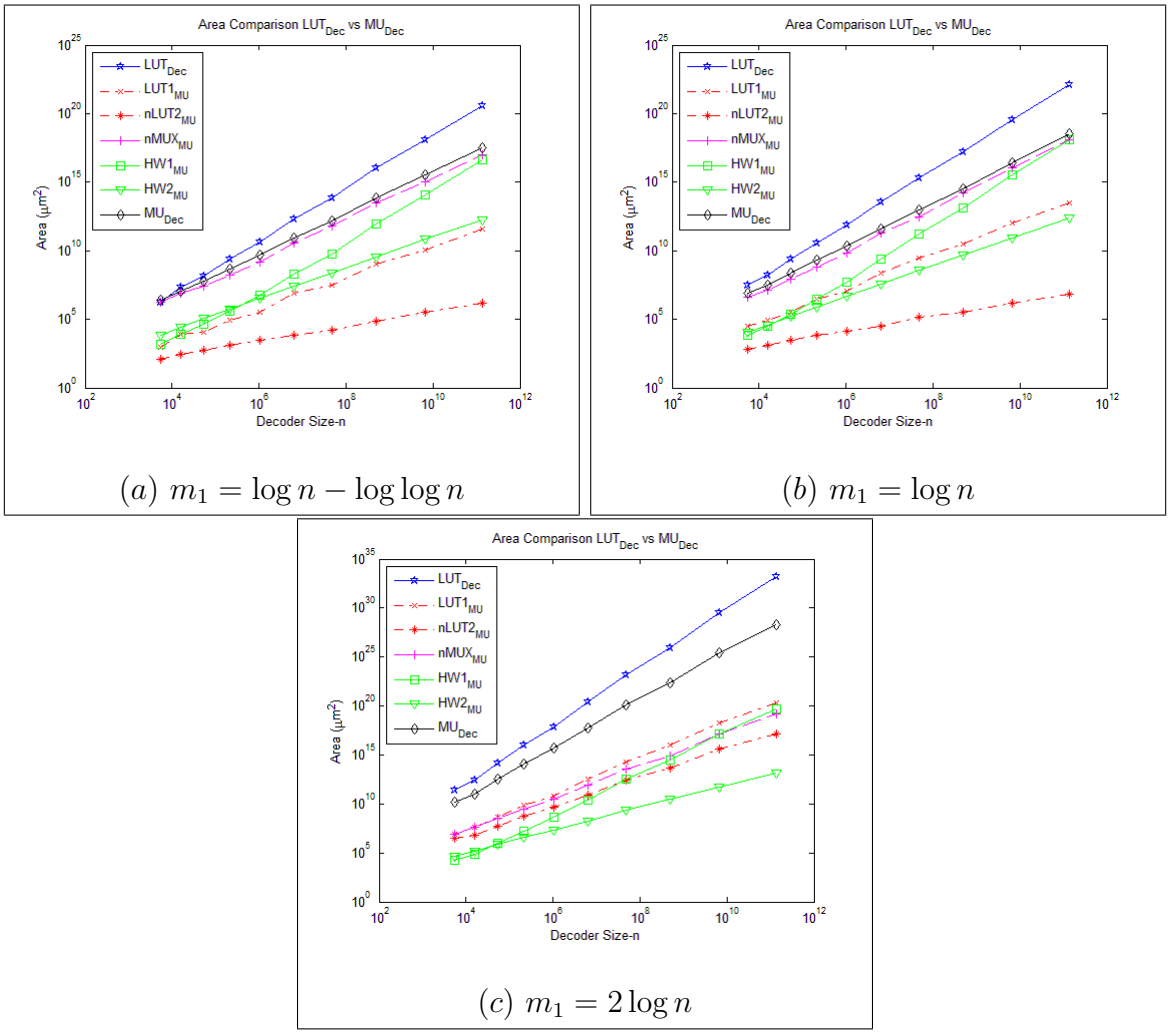


Figure 6.5: Contribution of building blocks for totally ordered subsets ($\gamma = 0$).

6.2.3 Delay and Power

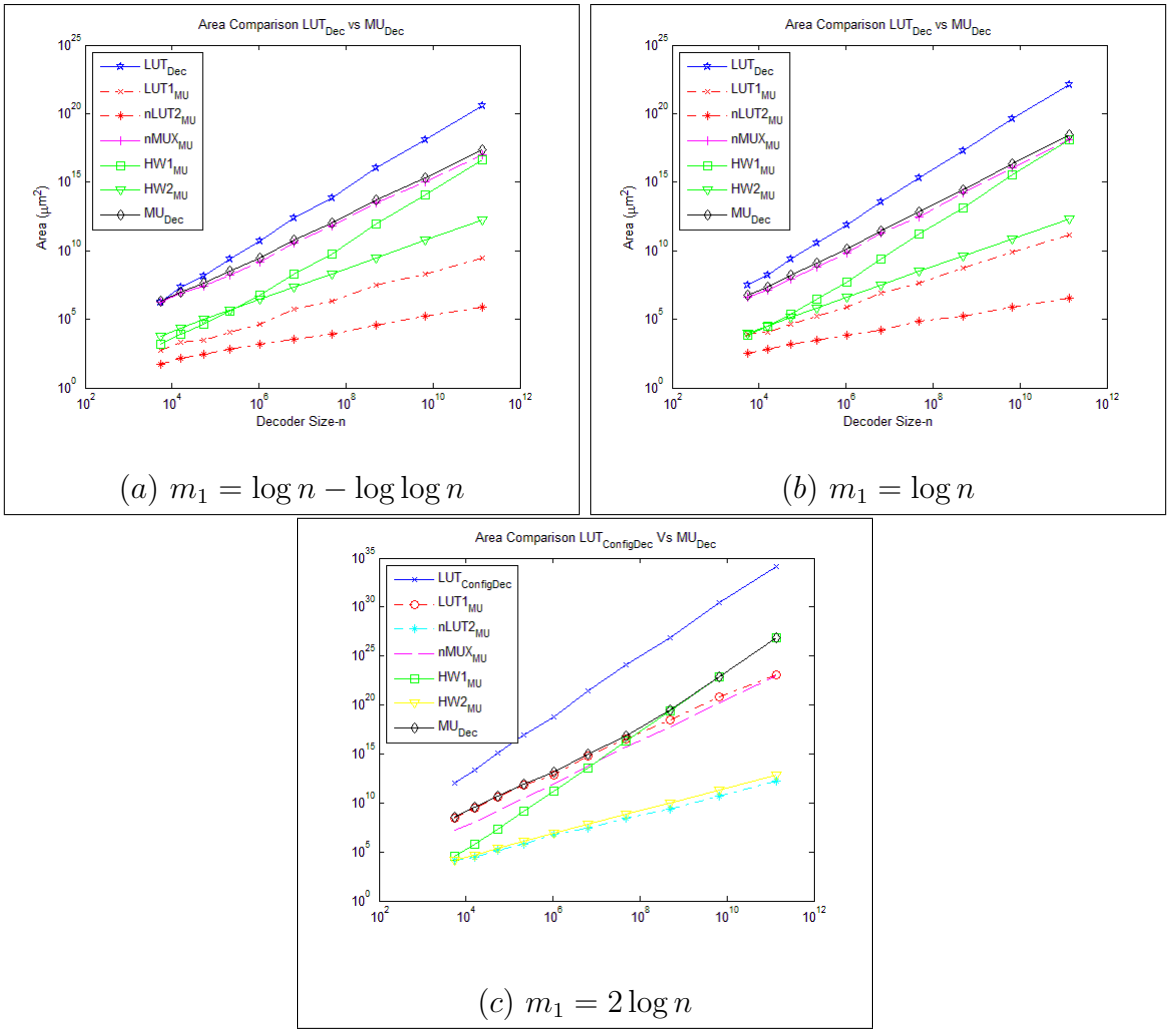
To complete the comparison, we also plot relative delays and power of the two decoders for the various cases. In all cases, the MU-Decoder seems faster. However, this result is less reliable as we could not accurately assertion the delay of memory cells. The estimate of area, however, of Section 6.2 are accurate. Figures 6.14–6.17 show the ratio of delays of the LUT- and MU-Decoders. Similar graphs for the power are shown in Figure 6.18–6.21. Overall MU-Decoder requires less power than the LU-Decoder even for the arbitrary case, as the area bottleneck (hardwiring) contributes very little to the power budget.



$(a)\ m_1 = \log n - \log\log n$ $(b)\ m_1 = \log n$

$(c)\ m_1 = 2\log n$

Figure 6.6: Contribution of building blocks for totally ordered subsets ($\gamma = \frac{y}{2}$).



$(a)\ m_1 = \log n - \log\log n$ $(b)\ m_1 = \log n$

$(c)\ m_1 = 2\log n$

Figure 6.7: Contribution of building blocks for totally ordered subsets ($\gamma = y$).

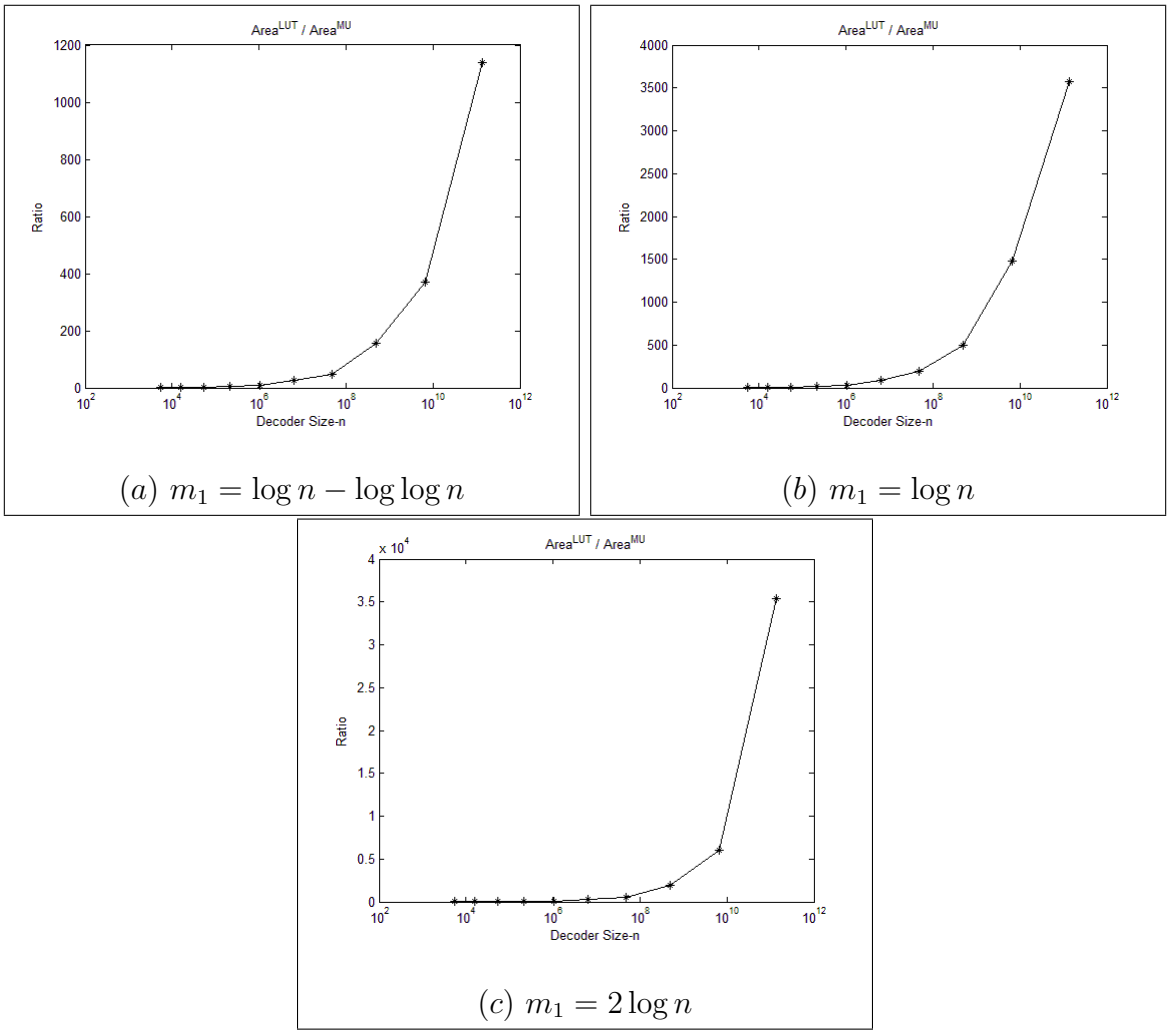


Figure 6.8: Ratio of LUT- to MU-Decoder areas for ($\gamma = 0$) totally ordered subsets ($5000 \leq n \leq 10^{11}$).

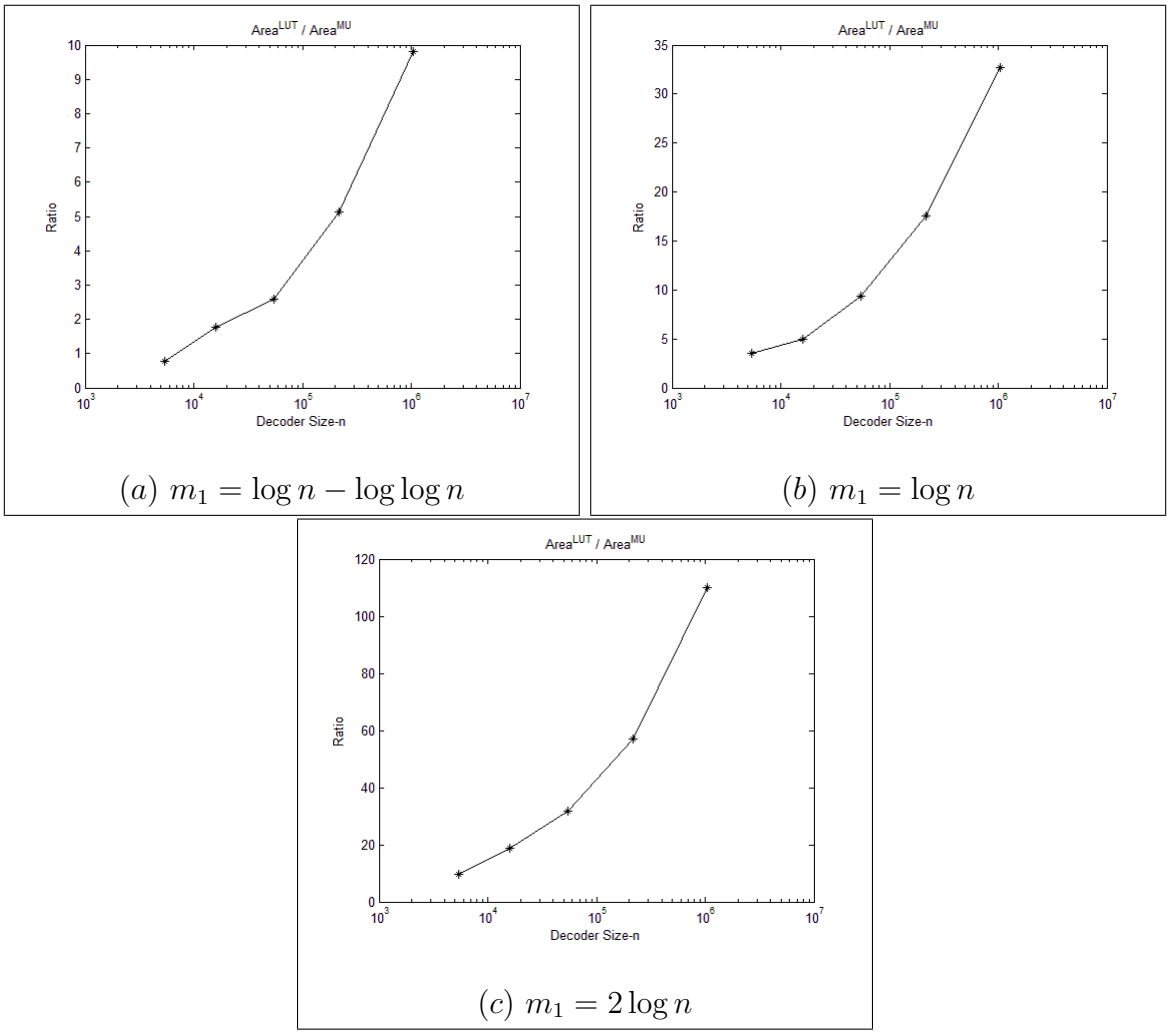


Figure 6.9: Ratio of LUT- to MU-Decoder areas for ($\gamma = 0$) totally ordered subsets ($5000 \leq n \leq 10^6$).

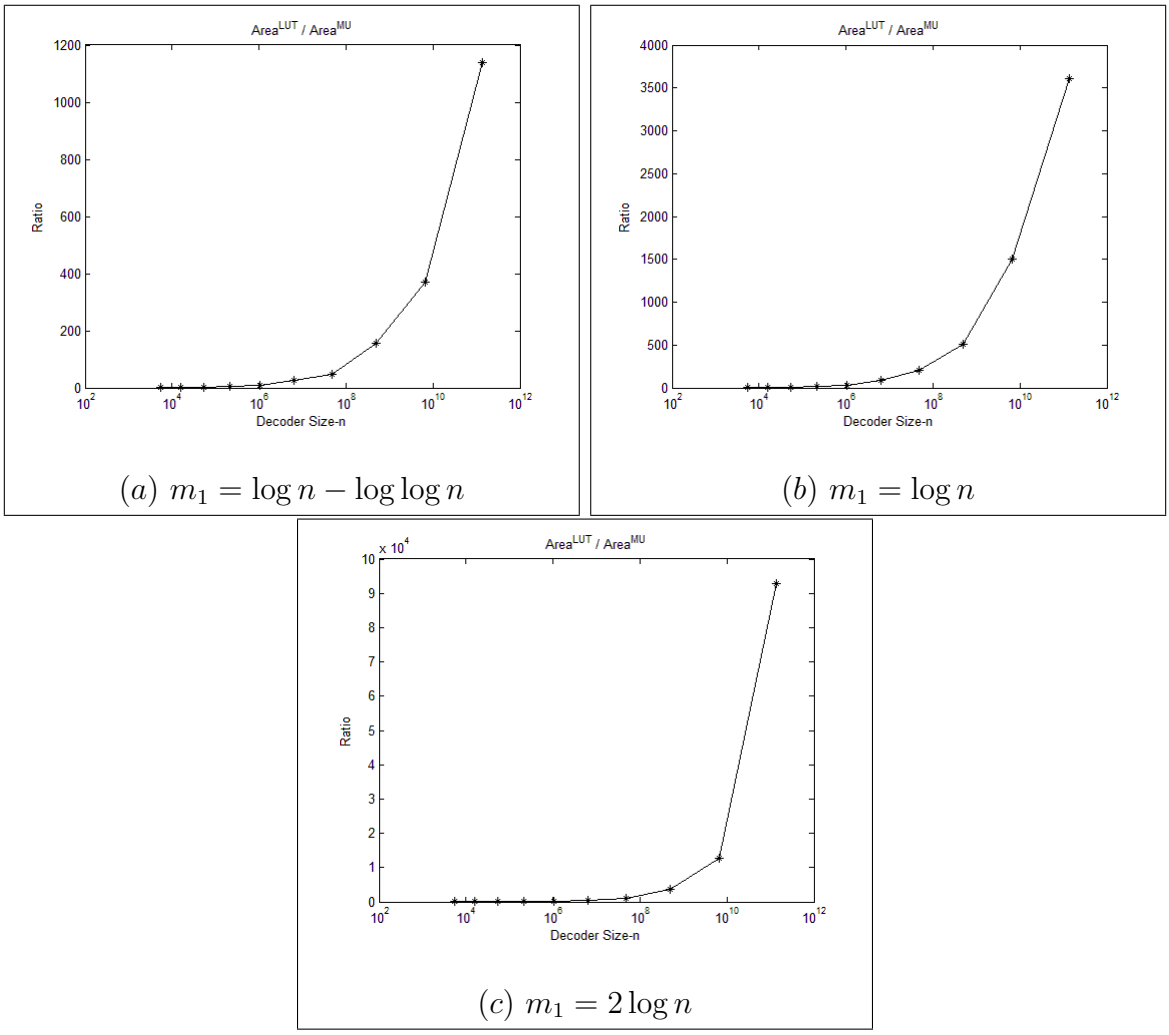


Figure 6.10: Ratio of LUT- to MU-Decoder areas ($\gamma = \frac{y}{2}$) totally ordered subsets ($5000 \leq n \leq 10^{11}$).

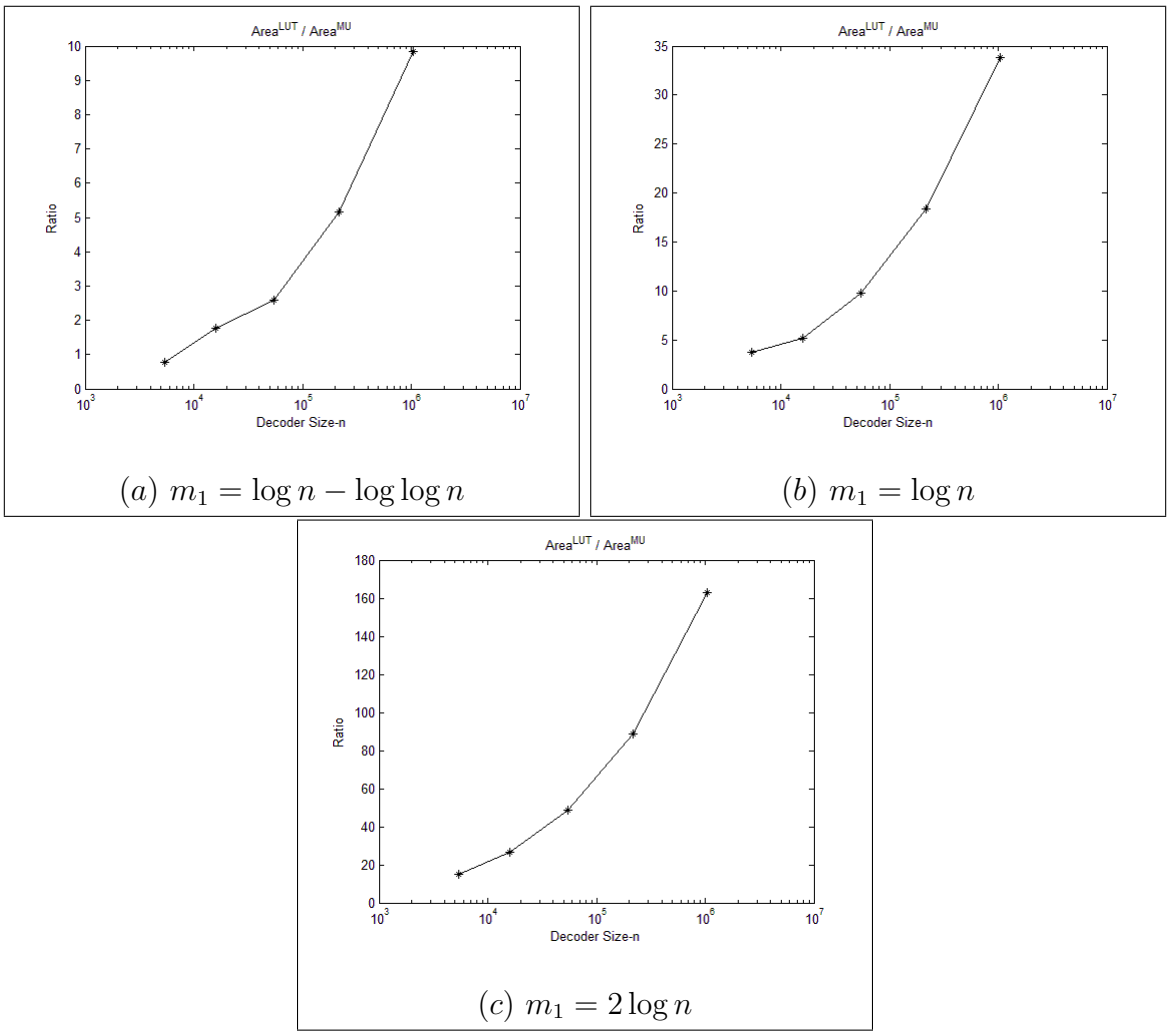
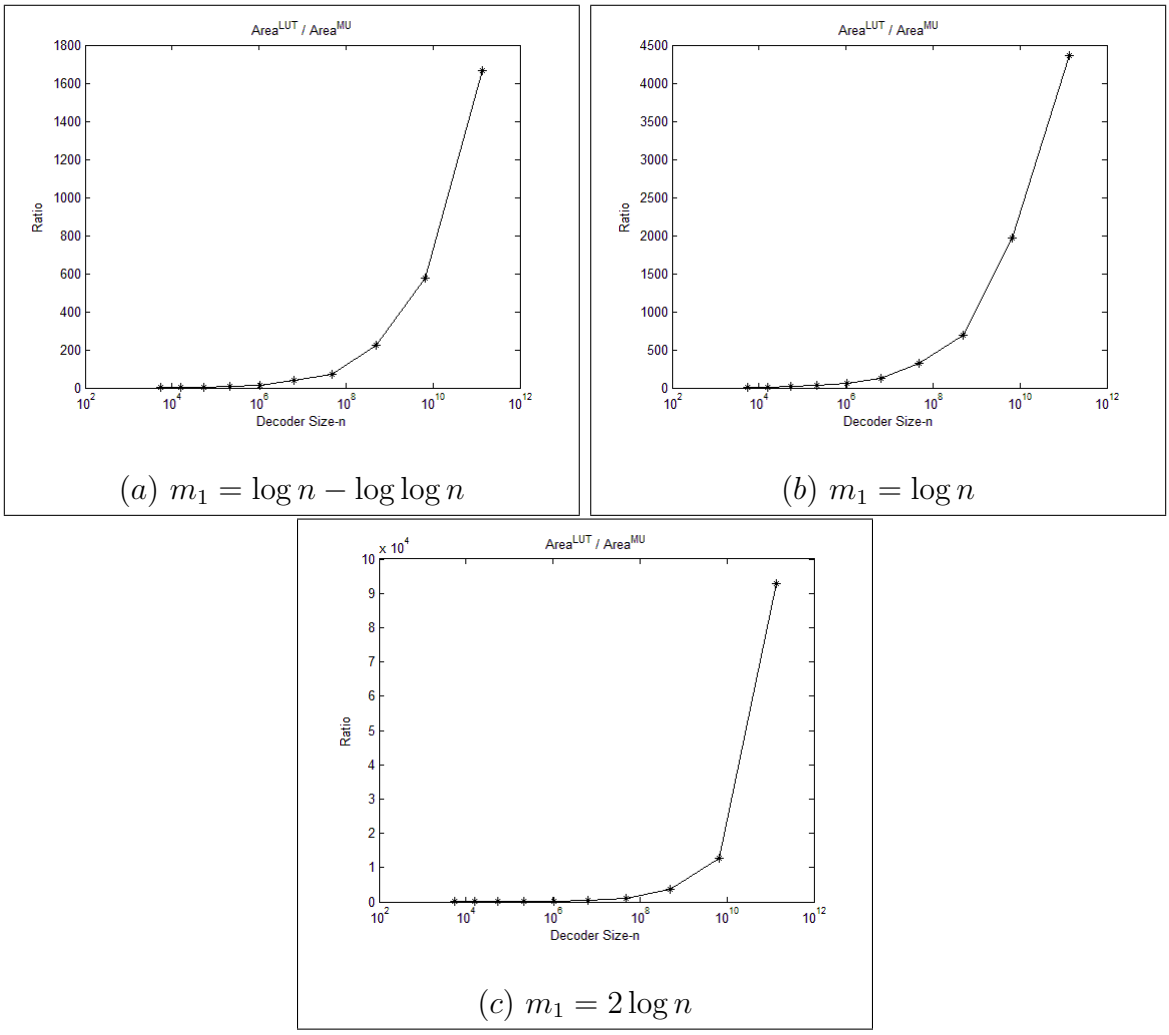


Figure 6.11: Ratio of LUT- to MU-Decoder areas for ($\gamma = \frac{y}{2}$) totally ordered subsets ($5000 \leq n \leq 10^6$).



(a) $m_1 = \log n - \log\log n$

(b) $m_1 = \log n$

(c) $m_1 = 2\log n$

Figure 6.12: Ratio of LUT- to MU-Decoder areas ($\gamma = y$) totally ordered subsets ($5000 \leq n \leq 10^{11}$).

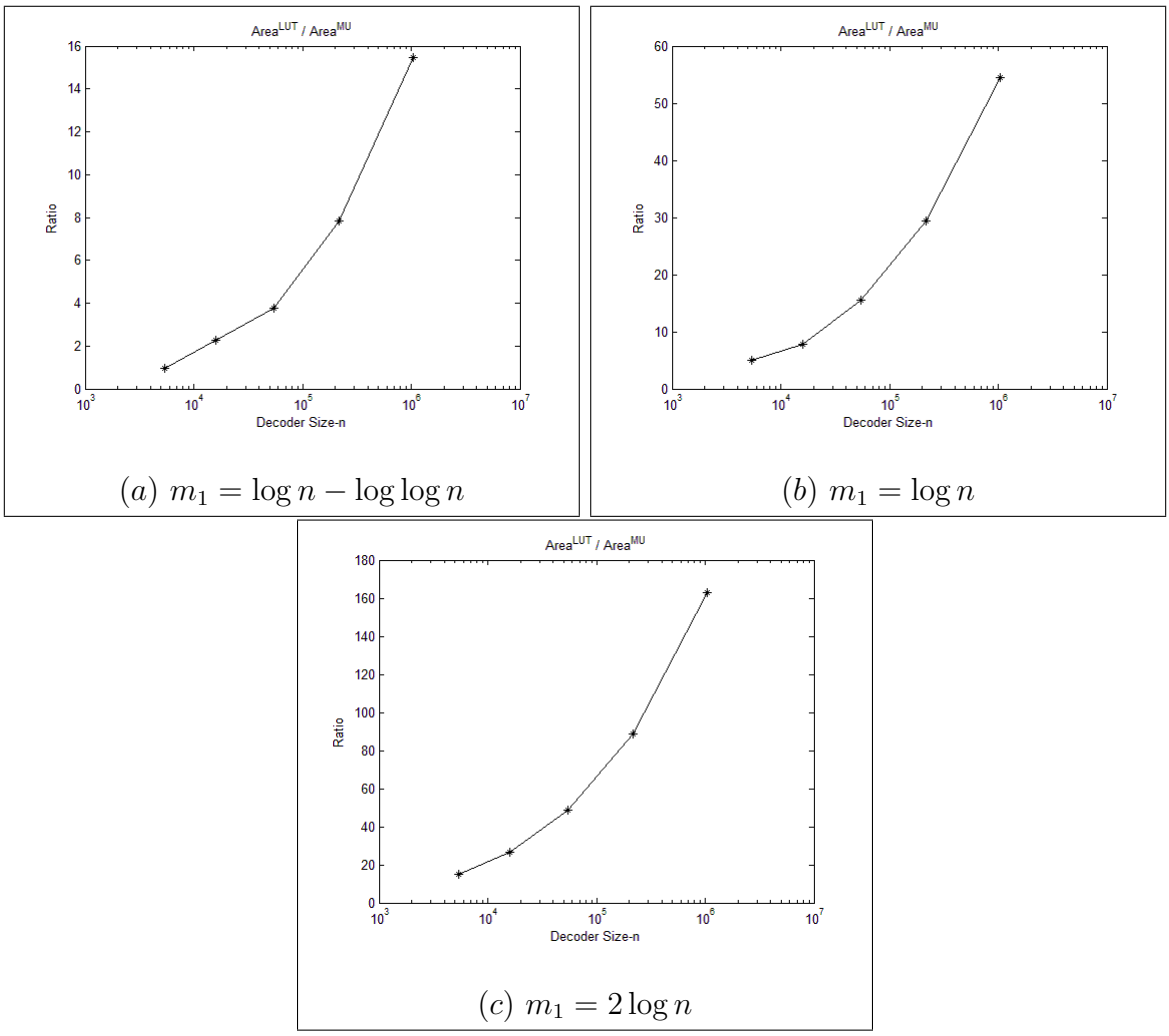


Figure 6.13: Ratio of LUT- to MU-Decoder areas for ($\gamma = y$) totally ordered subsets ($5000 \leq n \leq 10^6$).

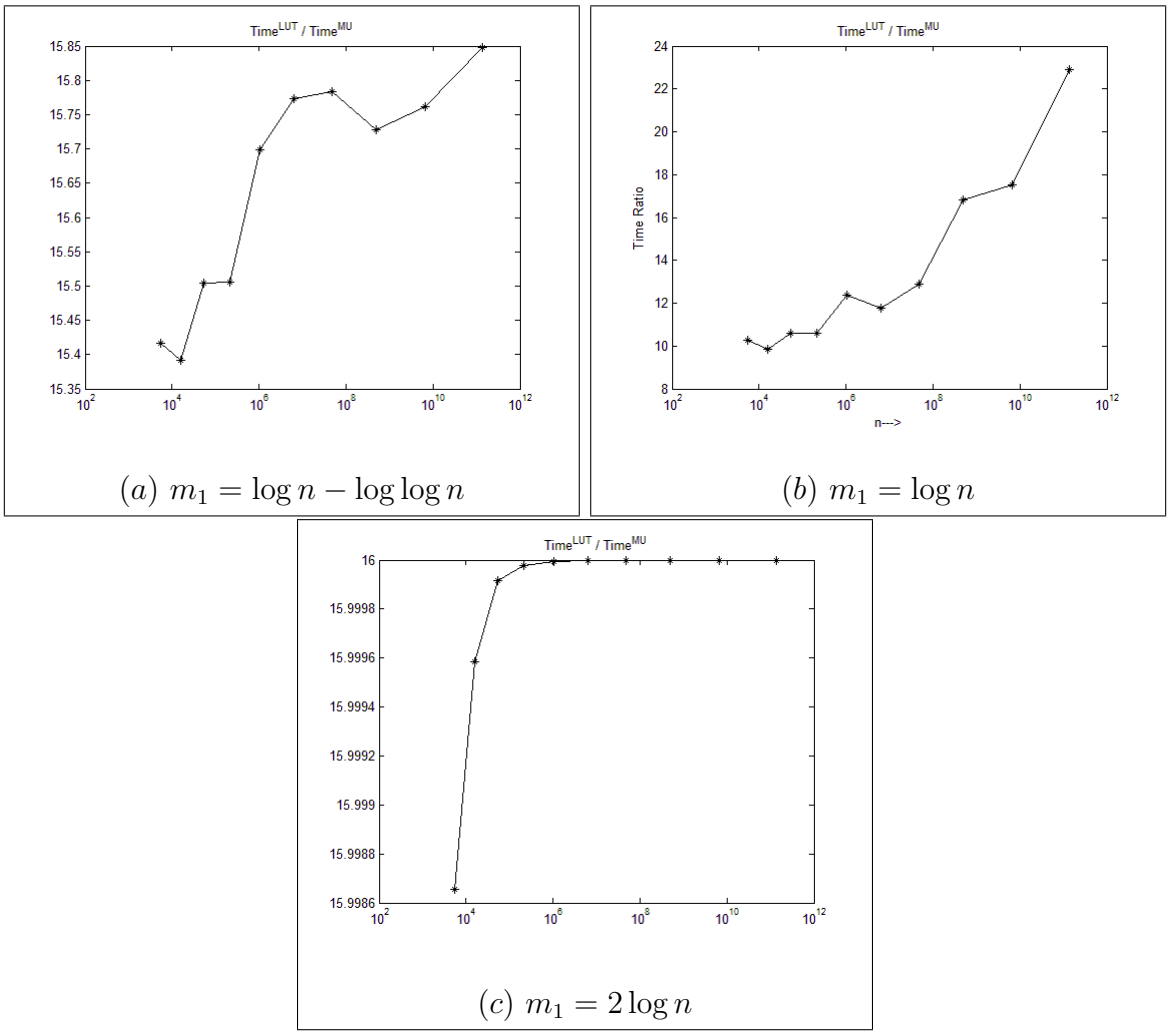


Figure 6.14: Ratio of LUT- to MU-Decoder time for arbitrary subsets ($5000 \leq n < 10^{11}$).

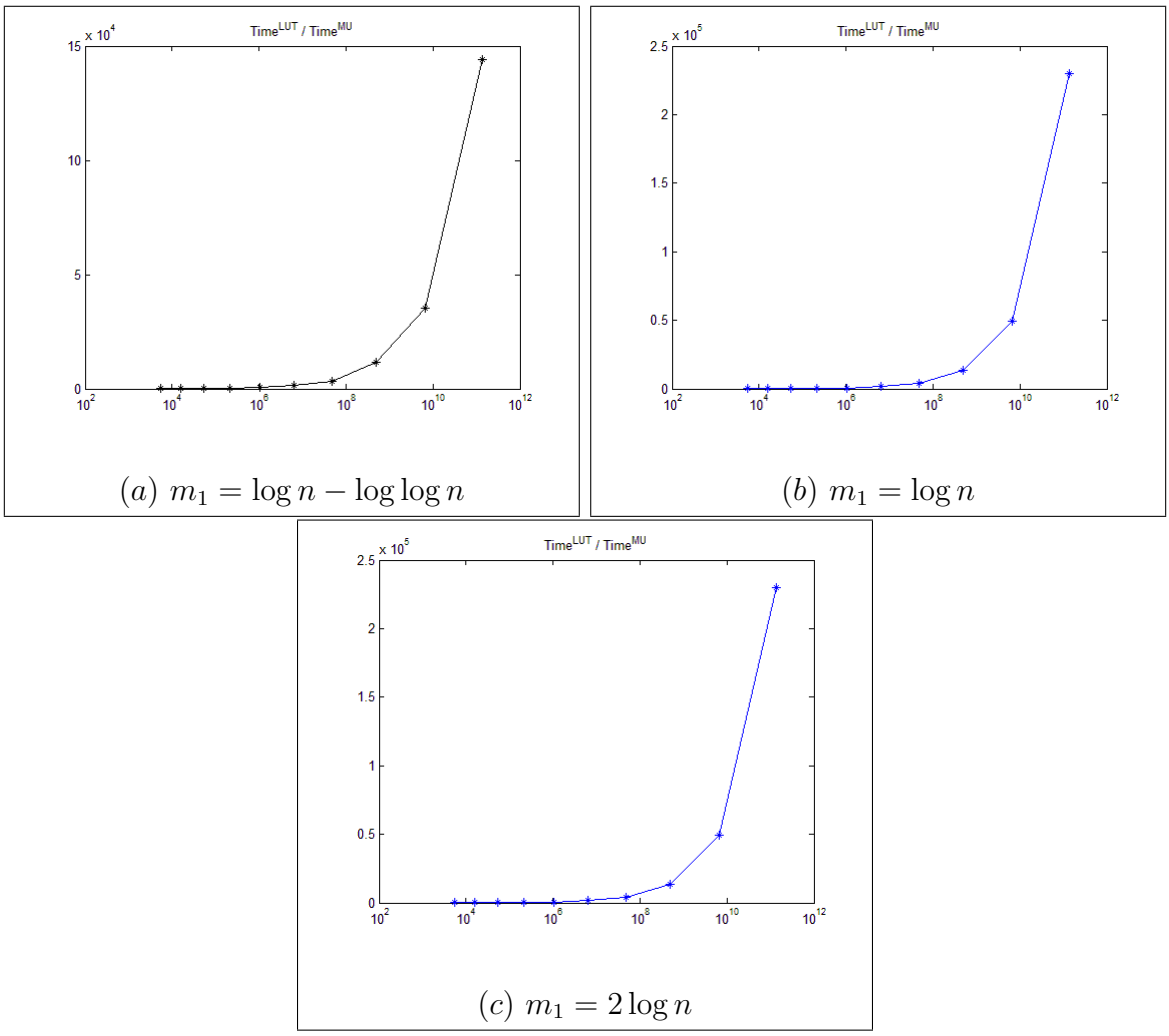
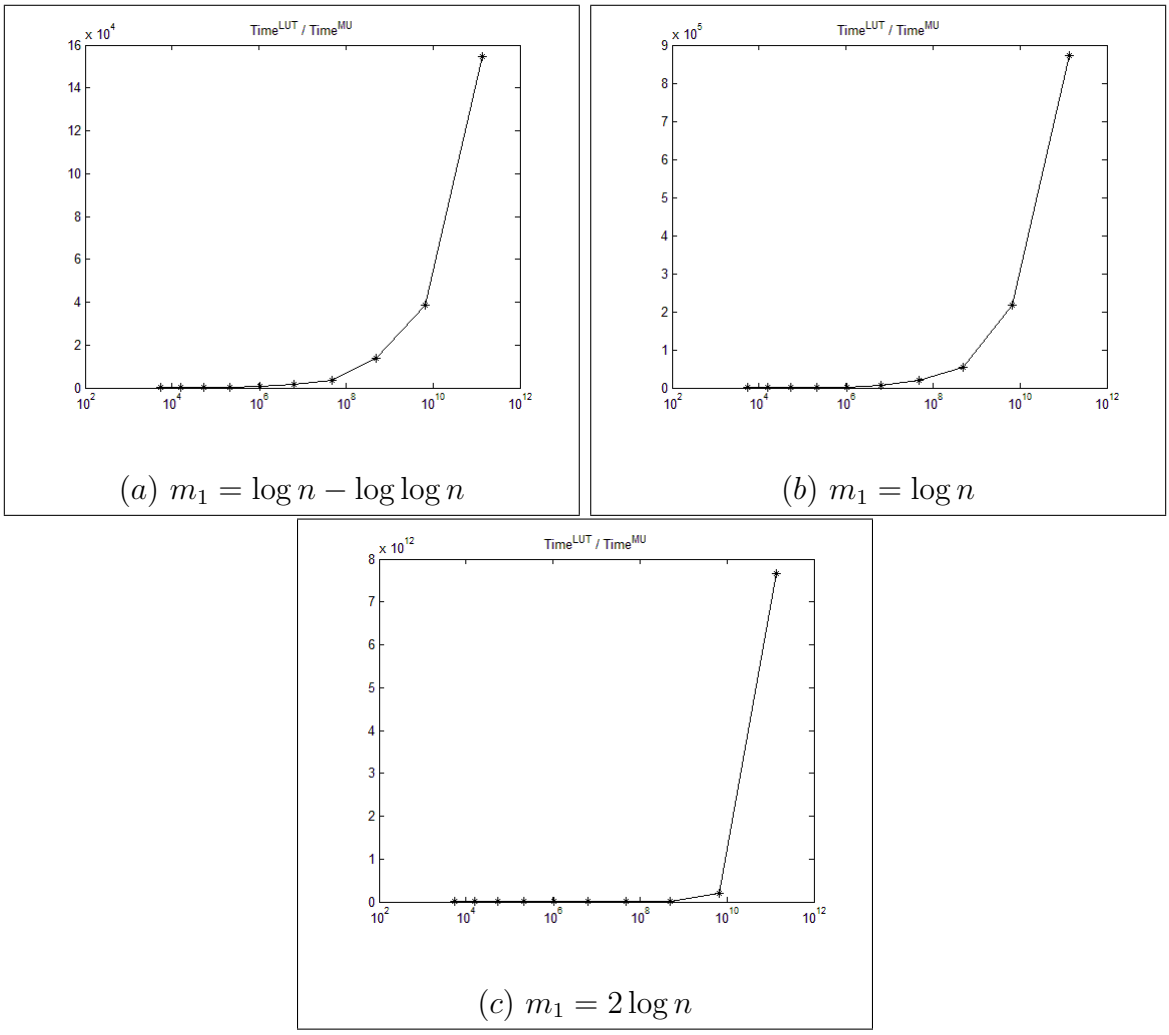


Figure 6.15: Ratio of LUT- to MU-Decoder time for ($\gamma = 0$) totally ordered subsets ($5000 \leq n < 10^{11}$).

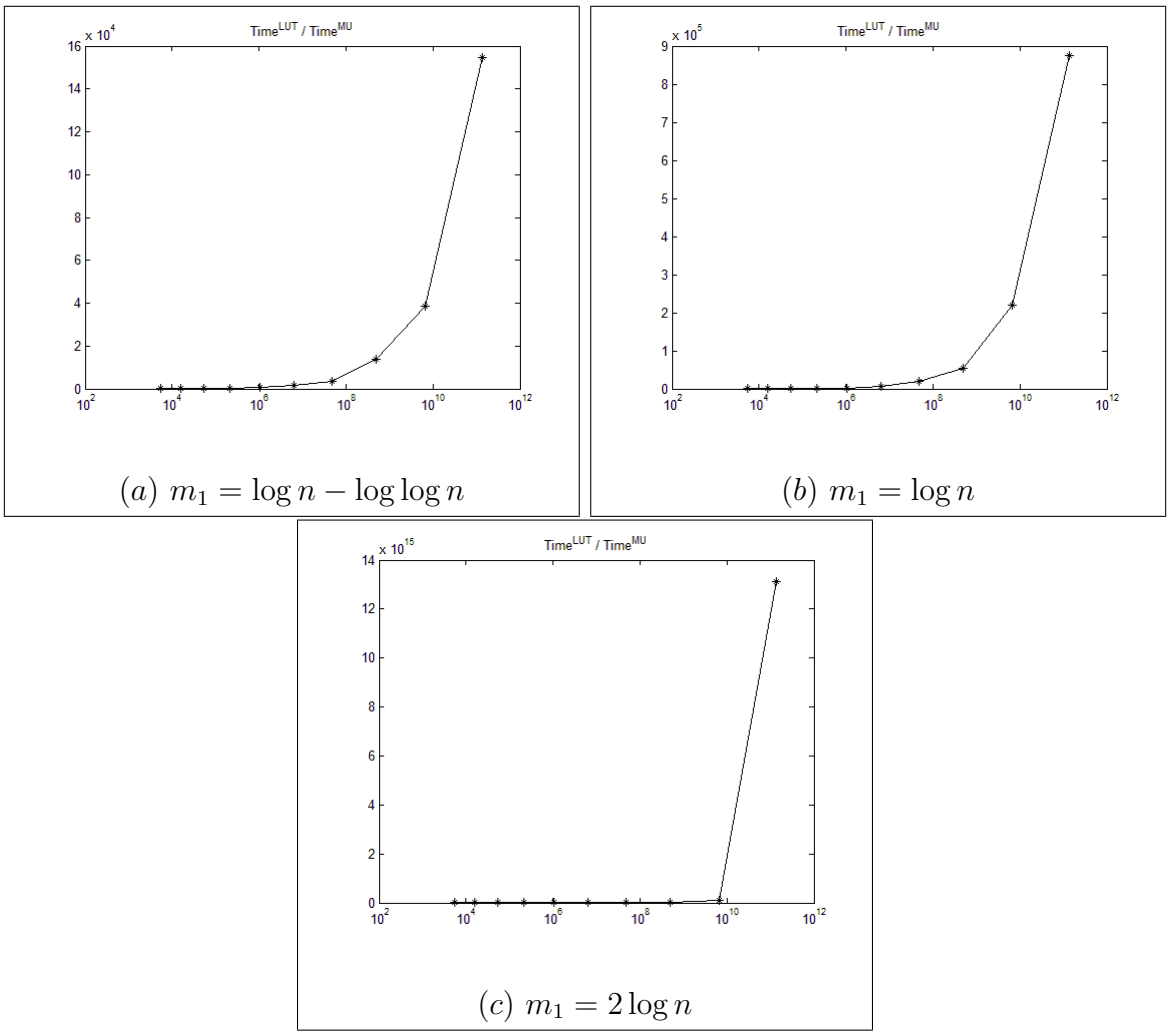


(a) $m_1 = \log n - \log\log n$

(b) $m_1 = \log n$

(c) $m_1 = 2\log n$

Figure 6.16: Ratio of LUT- to MU-Decoder time for ($\gamma = \frac{y}{2}$) totally ordered subsets ($5000 \leq n < 10^{11}$).



(a) $m_1 = \log n - \log\log n$ (b) $m_1 = \log n$ (c) $m_1 = 2\log n$

Figure 6.17: Ratio of LUT- to MU-Decoder time for $(\gamma = y)$ totally ordered subsets $(5000 \leq n < 10^{11})$

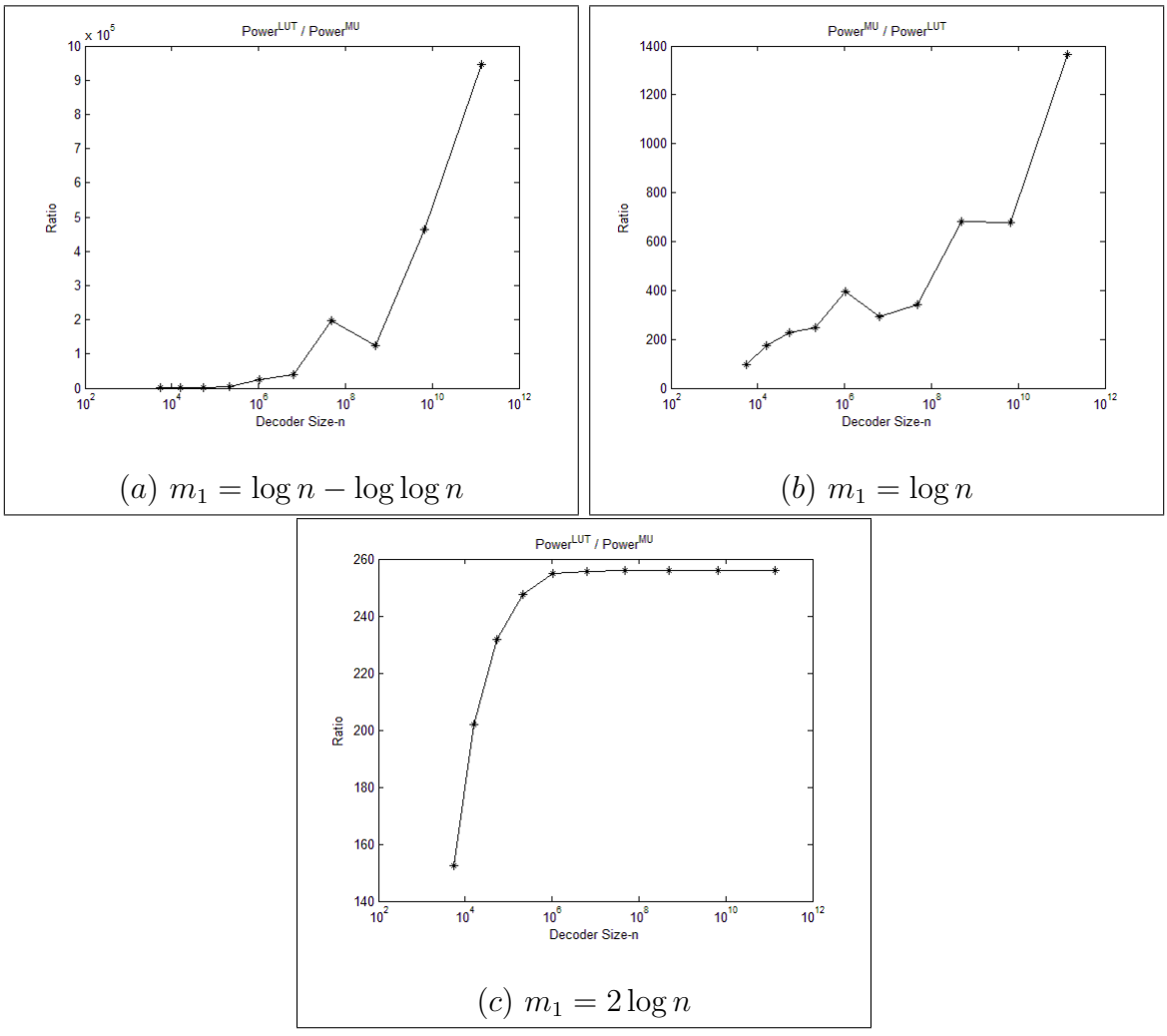


Figure 6.18: Ratio of LUT- to MU-Decoder power for arbitrary subsets ($5000 \leq n < 10^{11}$).

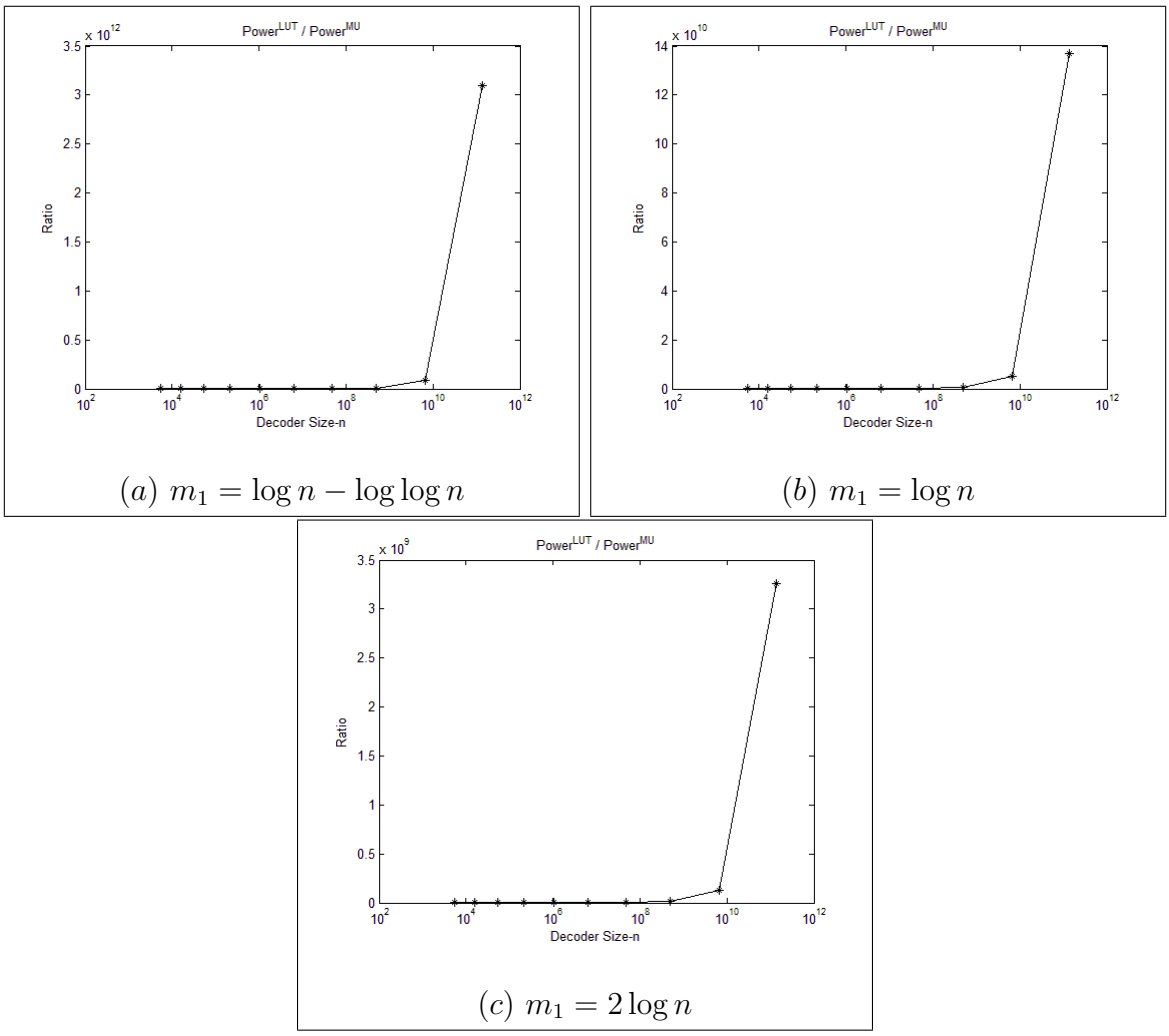


Figure 6.19: Ratio of LUT- to MU-Decoder power for ($\gamma = 0$) totally ordered subsets ($5000 \leq n < 10^{11}$).

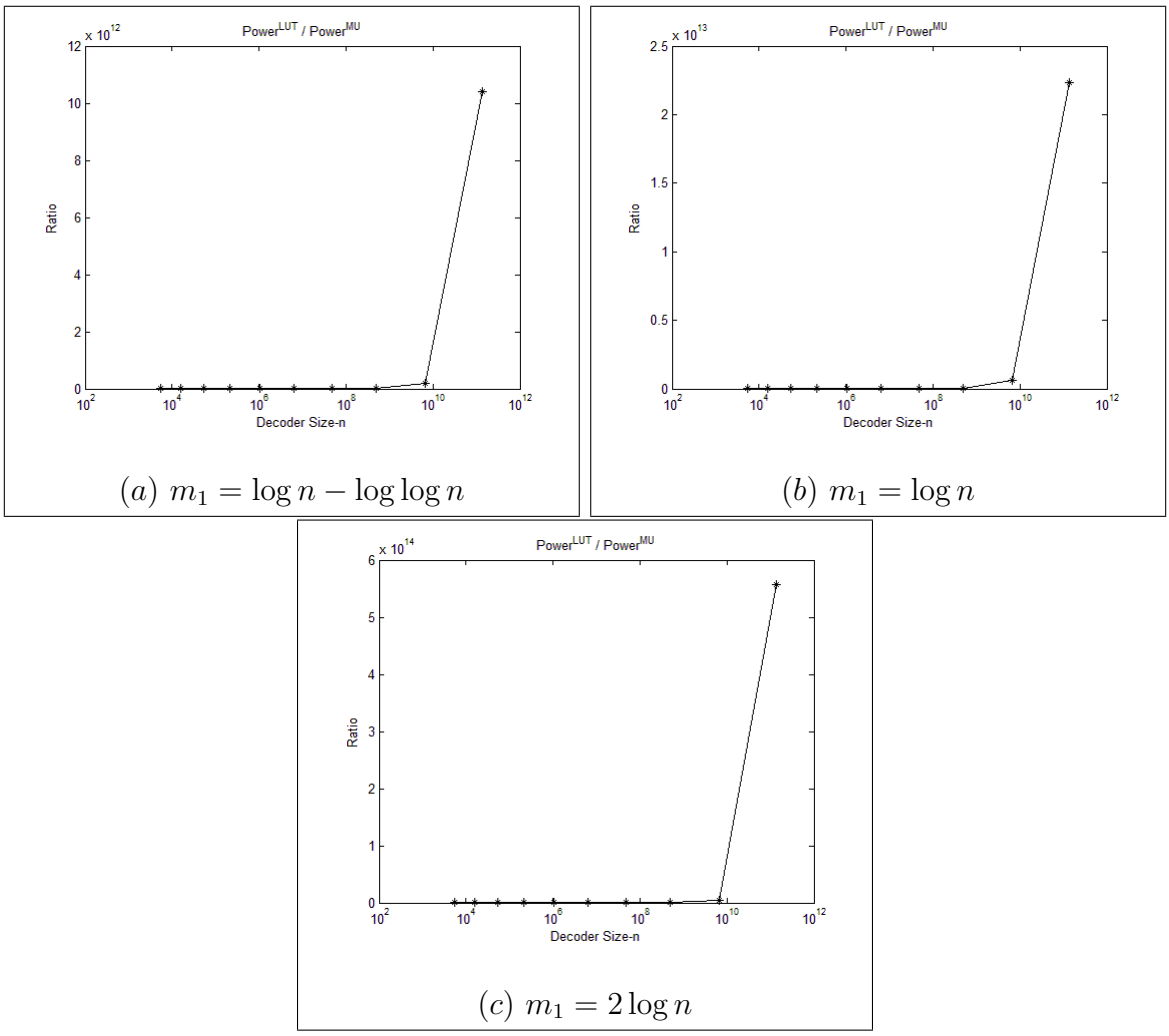


Figure 6.20: Ratio of LUT- to MU-Decoder power for ($\gamma = \frac{y}{2}$) totally ordered subsets ($5000 \leq n < 10^{11}$).

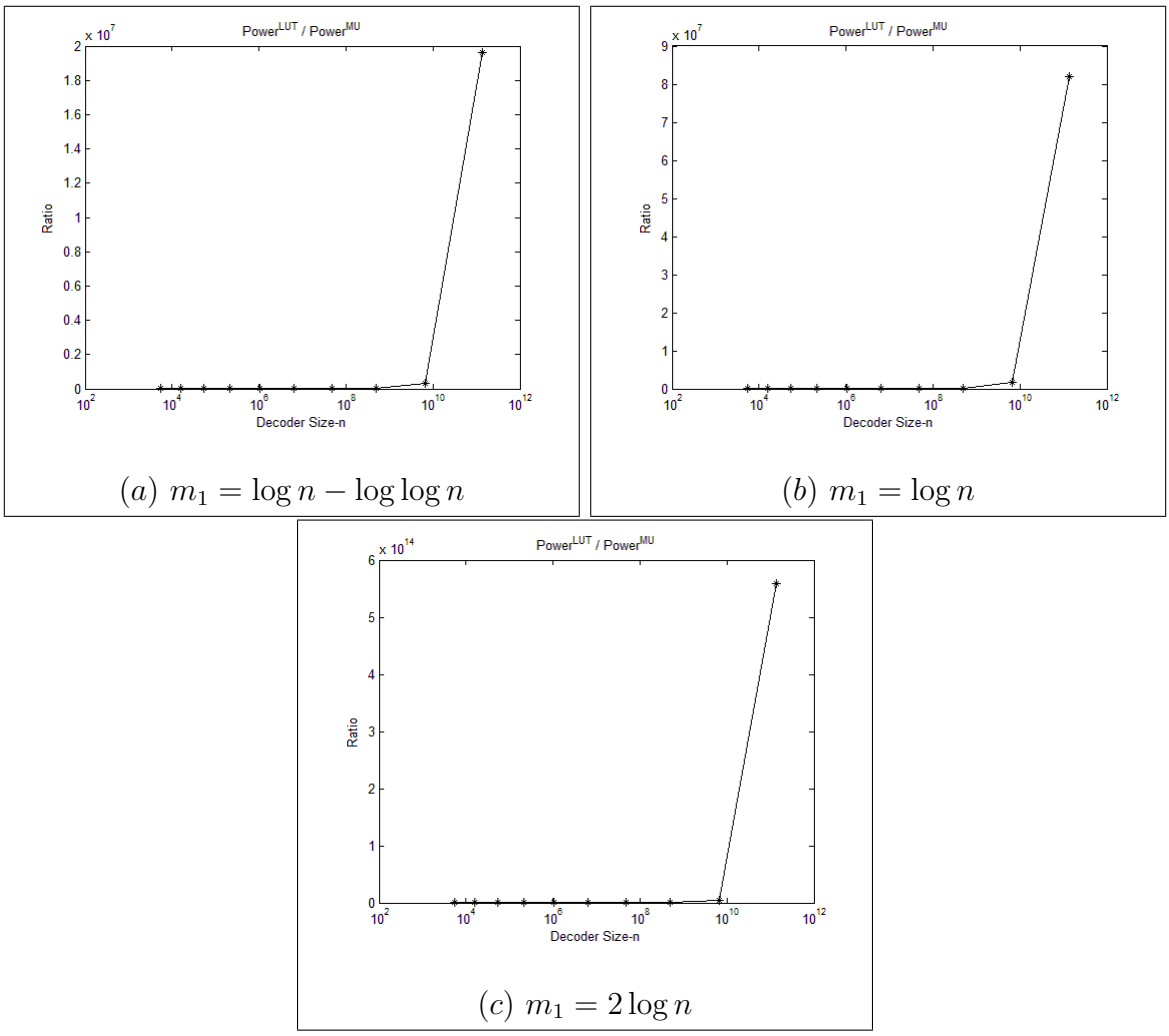


Figure 6.21: Ratio of LUT- to MU-Decoder power for ($\gamma = y$) totally ordered subsets ($5000 \leq n < 10^{11}$).

Chapter 7

Concluding Remarks

In this thesis, we have studied implementations of the MU-Decoder proposed by Jordan and Vaidyanathan [9, 10] and compared it to the standard LUT-(based) Decoder. We have shown that while the LUT-Decoder may appear more versatile than the MU-Decoder at first sight, these advantages are based on the worst case that rely on the existence of certain (rare, in our opinion) inputs that tend to mask the potential of the MU-Decoder. For most practical algorithms that exhibit some structure, the MU-Decoder performs much better than the LUT-Decoder.

Quantitatively, for the worst case the MU-Decoder with n output bits requires 3.4 more area than the LUT-Decoder for $n = 10^6$, this ratio is 1.8 for $n = 5000$. However if we add structure to the decoder output (in the form of totally-ordered subsets) we show that the MU-Decoder requires less than 3.5 times area for $n = 5000$ and less than 33 times area for $n = 10^6$. The time and power performance of the MU-Decoder is also better than those of the LUT-Decoder.

The above results were possible through models that we derived for the performance of LUT- and MU-Decoders. To obtain these models, we identified building blocks of the MU-Decoder (one of which is also a LUT). Then we implemented (by synthesis, place and route) various designs of the building blocks, theoretically identified growth terms for their delay and area, and used linear interpolation to curve-fit the implementation data. This gave us equations for the delay, area and power for time-optimized and area-optimized designs for each of the building blocks which, in turn, allowed us to extrapolate these results to sizes that were not possible to implement with the available computational resources. It also allowed us to predict the performance of the LUT- and MU-decoders.

In the process we developed a design methodology. We also identified MU-Decoder parameters that allow for a fair comparison of the LUT- and MU-Decoders. These and the building block models may also be of independent interest.

The work in this thesis has opened up several additional directions for future research.

An immediate validation of our MU-Decoder model could be through the implementation of MU-Decoders of reasonable size. We did not follow up on this direction due to limits on computational resources; most basic building blocks themselves ran for weeks before a reasonably large module could be implemented. One avenue for alleviating this problem is to use the Cadence tools with super-threading [24].

We determined power-optimized designs for all building blocks, except the memory cell used in a LUT. This was because the memory cell was implemented as a latch, where power consumption can be reduced almost arbitrarily by slowing the circuit operation. Unlike the combinational logic, the operating speed of sequential logic is defined by the clock or gate (enable) which need not be as fast as that allowed by the gate and latch timing parameters (set-up and hold time). An interesting addition to this work could be to obtain power-optimized LUT designs that would also enable us to model power-optimized MU-Decoder.

Jordan and Vaidyanathan [9, 10] identified totally ordered subsets as one of the structured outputs that MU-Decoder are suited to. Jordan [9] also identified the class of ASCEND/DESCEND subsets (representing hypercube and butterfly-like topologies) [1, 26] that could also benefit from the use of the MU-Decoder. A study of this, and other subset classes, would be of interest.

One of the most expensive parts of the MU-Decoder for unconstrained subsets is the multicast hardwiring. Its cost could be reduced somewhat as indicated in Vaidyanathan and Jordan [25]. A totally different implementation of the hardwiring is required here. Technological improvements such as 3-D packaging [12] and optical interconnects may also help.

The version of the MU-Decoder used in this thesis is called the *universal MU-Decoder* which is versatile, but is also the most expensive. Jordan [9] has considered a very restrictive hardwiring with which the MU-Decoder (given an arbitrary but fixed subset that is specified pre-manufacture) outperforms the LUT by a $\Omega\left(\frac{\log n}{\log \log n}\right)$ factor. Are there other restriction on hardwiring that the MU-Decoder can benefit from, for example, subsets in which element pair are always together or never together?

Bibliography

[1] A. Ali and R. Vaidyanathan, "Exact Bounds on Running ASCEND/DESCEND and FAN-IN Algorithms on Synchronous Multiple Bus Networks," IEEE Transactions on Parallel and Distributed Systems, vol. 7, no. 8, pp. 783–790, August 1996.

[2] E. Brunvand, "*Digital VLSI Chip Design with Cadence and Synopsis CAD Tools*," Addison-Wesley, Pearson Edition, Boston, USA, 2010.

[3] CACTI, `http://quid.hpl.hp.com:9081/cacti/sram.y` [Last retrieved: April 22, 2011].

[4] Cadence University Program Software Selection, 2010.

[5] M. D. Ciletti, "*Advanced Digital Design with the Verilog HDL*," Prentice Hall, Pearson Education, Inc. NJ, USA, 2002.

[6] Encounter Digital Implementation System User Guide, Product Version 9.1.1, 2010.

[7] M. B. Gokhale and P. S. Graham, "*Reconfigurable Computing-Accelerating Computing with Field Programmable Gate Arrays,*" Springer Verlag, First edition, The Netherlands, 2005.

[8] IEEE SA–1364–1995–IEEE Standard Hardware Description Language Based on the Verilog(R) Hardware Description Language, `http://standards.ieee.org/findstds/standard/1364-1995.html` [Last retrieved: April 22, 2011].

[9] M. C. Jordan, "A Configurable Decoder for Pin-Limited Applications," Master's Thesis, Department of Electrical and Computer Engineering, Louisiana State University, 2006.

[10] M. C. Jordan, R. Vaidyanathan, "MU-Decoders: A class of fast and efficient configurable decoders," Proceedings of IEEE International Symposium on Parallel and Distributed Processing, workshops and PhD Forum, Atlanta, GA, 19-23 April 2010.

[11] A. B. Kahng, B. Lin, K. Samadi, "Improved on-chip router analytical power and area modeling," Proceedings of 2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC), Taipei, Taiwan, 18-21 January 2010.

[12] M. Karnezos, "3D packaging: Where all Technologies Come Together," Proceedings of IEEE/CPMT/SEMI 29th International Symposium on Electronics Manufacturing Technology, pp. 64-67, 2004.

[13] S. Krishnaswamy, S. M. Plaza, I. L. Markov, J. P. Hayes, "Enhancing Design Robustness with Reliability-aware Resynthesis and Logic Simulation," Proceedings of IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2007, pp. 149-154, 2007.

[14] NC-Launch User Guide, Product Version 8.2, 2008.

[15] Free PDK45:Contents-NCSU EDA Wiki, `http://www.eda.ncsu.edu/wiki/FreePDK45:Contents` [Last retrieved: April 22, 2011].

[16] B. D. Olson, O. A. Amusan, S. Dasgupta, L. W. Massengill, A. F. Witulski, B. L. Alles, K. M. Warren, D. R. Ball, "Analysis of Parasitic PNP Bipolar Transistor Mitigation Using Well Contacts in 130 nm and 90 nm CMOS Technology," Proceedings of IEEE Transactions on Nuclear Science, pp. 894-897, August 2007.

[17] E. Quigley, "*UNIX Shells by Example*," Prentice Hall PTR, New Jersey, 1997.

[18] K. Roy, S. C. Prasad, "*Low Power VLSI Circuit Design,*" John Willey & Sons, New York, 2000.

[19] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, L. Alvisi, "Modeling the effect of Technology Trends on the Soft Error rate of Combinational Logic," Proceedings of the International Conference on Dependable Systems and Networks, 2002.

[20] SimVision User Guide, Product Version 8.2, 2008.

[21] J. Singh, D. K. Pradhan, S. Hollis, S. P. Mohanty, J. Mathew, "Single Ended 6T SRAM with Isolated Read-Port for Low-Power Embedded Systems," Proceedings of Design, Automation and Test in Europe Conference and Exhibition, Nice, France, 20-24 April 2009.

[22] The MathWorks, Optimization ToolBox, `http://www.mathworks.com/help/toolbox/optim/ug/lsqlin.html` [Last retrieved: April 22, 2011].

[23] Tcl Developer Site, `http://www.tcl.tk/` [Last retrieved: April 22, 2011].

[24] Using Encounter RTL Complier, Product Version 10.1, 2010.

[25] R. Vaidyanathan and M. C. Jordan, "*Configurable Decoder with Applications in FPGAs*," United States Patent Application 20100180098, July 15, 2010.

[26] R. Vaidyanathan and A. Padmanabhan, "Bus-Based Networks for Fan-In and Uniform Hypercube Algorithms," Parallel Computing, vol. 21, pp. 1807-1821, 1995.

[27] R. Vaidyanathan and J. L. Trahan, "*Dynamic Reconfiguration on the R-Mesh,*" Handbook of Parallel Computing: Models, Algorithms and Applications, S. Rajasekaran and J. Reif, eds., CRC Press, 2007.

[28] R. Vaidyanathan and J. L. Trahan, "*Dynamic Reconfiguration: Architectures and Algorithms,*" Kluwer Academic/Plenum Publishers, January 2004.

[29] J. F. Wakerly, "*Digital Design Principles and Practices,*" Prentice Hall, Third edition updated, Upper Saddle River, New Jersey, 2001.

[30] M. J. Wirthlin and B. L. Hutchings, "DISC: The Dynamic Instruction Set Computer," Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing, J. Schewel, ed., Proceedings of SPIE, vol. 2607, 1995, pp. 92-103.

[31] XILINX, 7 Series FPGAs Overview, Advance Product Specification, DS180 (v1.6) March 28, 2011.

Vita

Raghavendra Kongari was born in February of 1986, in Hyderabad, Andhra Pradesh, India. He attended primary education in Pochampally Public School and received his secondary education in Sri Santhosh Vidyaniketan High School in Bhoodhan Pochampally. After high school, Kongari received admission to the Department of Electronics and Communication Engineering, at the Chaitanya Bharathi Institute of Technology, one of the prestigious institute in Andhra Pradesh, India. In June 2007, he received his Bachelor of Engineering in Electronics and Communication Engineering degree from Osmania University, Hyderabad, India. After his graduation, he joined Medha Servo Drives pvt. Ltd., India, where he served as design engineer in Control Electronics Department. After working for one year he came to the United States of America to pursue master's degree. He then joined the Department of Electrical Engineering in Louisiana State University, Baton Rouge, in August 2008. He worked as teaching assistant in the Department of Electrical Engineering from August 2008 to May 2011. And also worked under Dr. J. Matthew Fannin as graduate research assistant from August 2009 to May 2011. He will attain his Master of Science in Electrical Engineering degree in the spring of 2011.