MEASURING THE DYNAMIC ENERGY EFFICIENCY OF FPGAS OVER PROCESSORS

by

Muhammad Umair Zafar,

Bachelor of Electrical Engineering(Electronics),

Bahria University, Islamabad, July 2010

A thesis presented to

Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Applied Science

in the Program of

Electrical and Computer Engineering

Toronto, Ontario, Canada, 2016

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public.

Measuring the Dynamic Energy Efficiency of FPGAs over Processors

Muhammad Umair Zafar
Master of Applied Science
Electrical and Computer Engineering
Ryerson University
2016

Abstract

This work investigates the dynamic energy efficiency of the parallel execution model of an FPGA and the sequential execution model of a processor, for latency-insensitive applications. We create the temporal implementations (sequential instructions) of the MCNC benchmarks to be executed on a processor that employs a 4LUT as its functional unit. This processor is ~716 times inefficient for dynamic energy than a 4LUT FPGA, mainly due to the large amount of memory (instruction/data) that is required to encode the 4LUT based instructions. The size of the memory (instruction/data) can be reduced by increasing the data-path width and the logic complexity of the ASIC-based functional units of the processor. Particularly, at 64-bit data-path width and when the (instruction/data) memory sizes are reduced to less than ~9% of their corresponding 4LUT-based instructions, the processor with ASIC-based complex functional unit can achieve higher dynamic energy efficiency than the FPGA for MCNC benchmarks.

Acknowledgements

First of all, I would like to say thanks to my supervisor Dr. Andy Ye, for his support and supervision during this work. I sincerely appreciate the time and efforts he invested in mentoring me.

I would like to say thanks to my friends and lab partners. Also I would like to thank Ryerson University for providing me an opportunity and good environment to perform this research.

I would like to say thanks to my family, without their support this work would have never been possible.

Dedication(s)

To my wife Safoora

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

Microprocessors and Field-Programmable Gate Arrays (FPGAs) are the two main programmable platforms for implementing digital computation. The main computing elements in modern FPGAs are look-up tables (LUTs) and bit-configurable routing resources [1]. In combination, these resources can be effectively used to exploit bit-level data parallelism in order to deliver much higher performances and energy efficiency than today's state-of-the-art processors [2]. Modern microprocessors, however, are increasingly incorporating accelerators, such as GPUs [3], image processing and DSP cores [4], on the same die as CPUs. These accelerators function as advanced functional units and are much more complex in structure than the hard-blocks incorporated on an FPGA.

This difference in complexity is mainly due to the sequential execution nature of a processor and the parallel execution nature of an FPGA. In particular, on a processor, each functional unit is shared over many instructions so only a small number of functional units of each type are needed per processor. On an FPGA, on the other hand, many more hard-blocks of the same type are needed in order to exploit the parallelism. Consequently, FPGAs only can afford to use much simpler accelerators, such as multipliers, as hard-blocks while still maintaining their original parallel execution model [5].

Since the functional units and the hard-blocks are essentially ASICs, which are known to be much more energy efficient than reconfigurable LUTs [6], it is important to understand the impact of this difference in complexity on the energy efficiency of FPGAs and processors. To this end, this work first measures the dynamic energy required to execute MCNC benchmarks on a 4-LUT-based FPGA (spatial implementation). We then serialize the LUT configurations so each configuration is mapped into an instruction that can be executed by a 4-LUT-based sequential processor (temporal implementation). Using this serialized version of the LUT configurations, we measure the effect of increasing functional unit complexity on the dynamic energy efficiency of processors by relating the increase in functional unit complexity to the reduction in instruction and data memory size.

This work is related to [7]. It differs from previous work in two main aspects. First, this work uses a benchmark-based approach over the analytical model from previous work. In particular, the analytical model from [7] only considers homogenous activities, i.e. every single wire is switched at every clock cycle. This over-estimates dynamic energy consumption. In this work, we have provided more accurate results with a simulation based empirical approach of estimating heterogeneous activity (actual switching) on the nets by simulating MCNC benchmark circuits with random inputs.

Secondly, the work in [7] has only compared FPGAs to the simplest possible implementations of a processor (Single 4-LUT), which only use one LUT in its functional unit. It also shows that when using monolithic memories for both instruction and data, such a processor would consume much more dynamic energy than FPGAs [7].

Instructions based on LUT configurations, however, are extremely inefficient and often consume much more memory than it is required by conventional processor instructions – since real

functional units are implemented in either custom ICs or ASICs and can include many hardwired instructions. The goal of this work is to help a designer making a selection between spatial (FPGA) or temporal (CPU) processing platform for implementing a computation based on dynamic energy efficiency. As a result, this work examines the trade-off between data and instruction memory size and the dynamic energy efficiency of processors. We specifically measure the amount of on-chip SRAM based instruction and data memory reduction that is needed from the LUT-based instructions in order for a processor to achieve the same dynamic energy consumption as a LUT-based homogenous FPGA. This result is particularly relevant in the age of dark silicon, where there is a trend for processors to include a variety of application-specific precompiled cores in order to increase their performance and efficiency [8]. Consequently, it is important to quantify the amount of memory reduction that a processor should achieve in order to be more energy efficient than regular LUT-based FPGAs.

1.2 Thesis Organization

The remainder of this thesis is structured as follows: Chapter 2 (Background and Motivation) provides architectural details about some of the platforms available for implementation computations such as FPGAs and CPUs. It also provides the motivation for measuring the effect of functional unit complexity on dynamic energy consumption of processors and FPGAs using an example of 8-bit wide (7:2) compressor tree circuit, Chapter 3 (Experimental Procedure) presents the procedure that is used to measure the dynamic energy consumption of the temporal representation of the MCNC benchmarks on a single 4-LUT based processor and the spatial 4-LUT-based FPGA. Chapter 4 (Experimental Results) presents experimental results and Chapter 5 (Conclusion and Future work) concludes and future direction for further research.

Chapter 2

Background and Motivation

2.1 Field Programmable Gate Arrays

FPGAs provide a hardware programmable platform to implement digital computations. Since their introduction in 1984, they have become a $30 Billion (still growing) industry in 2016. They can provide a huge performance advantage over general purpose sequential processors by providing dedicated hardware such as application-customized data paths. They are re-programmable, and provide a competitive option against standard Cell ASIC development by reducing NRE costs and achieving faster time-to-market [9].

2.1.1 FPGA Architecture

In past few decades FPGA architecture has much evolved from containing a simple network of LUTs (Look-up Tables) and programmable routing circuits to complex network of LUTs and hard-core processing blocks, such as DSPs and BRAMs.

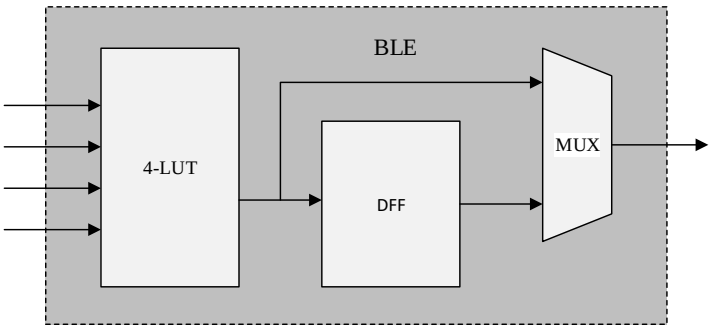

Figure 1. Basic Logic Element

A typical FPGA contains k-input LUTs, DFFs (D-Flip Flops) and bit programmable routing circuits grouped into BLE (Basic Logic Element). Figure 1, shows a basic logic element of FPGA. DFF can used when output for a BLE needs to registered. This is known as 'soft logic' implementation on FPGA. In order to achieve higher densities on chip and reduce their overall delay, these BLEs are grouped together into 'clusters' also known as LB (Logic Blocks) [10]. A study has shown that using 4 BLEs inside a logic block can provide a 5-10% reduction in area for FPGAs [11]. Also [12] shows that 4 input LUT is the most area efficient design mainly because, as the number of inputs for a single LUT grows the complexity of LUT grows exponentially making 4-LUT the most optimal design choice. This is one of the reason we choose 4-LUT for our research further explained in later chapters. Figure 2, shows a group of BLEs packed in a group to form a LB [11].

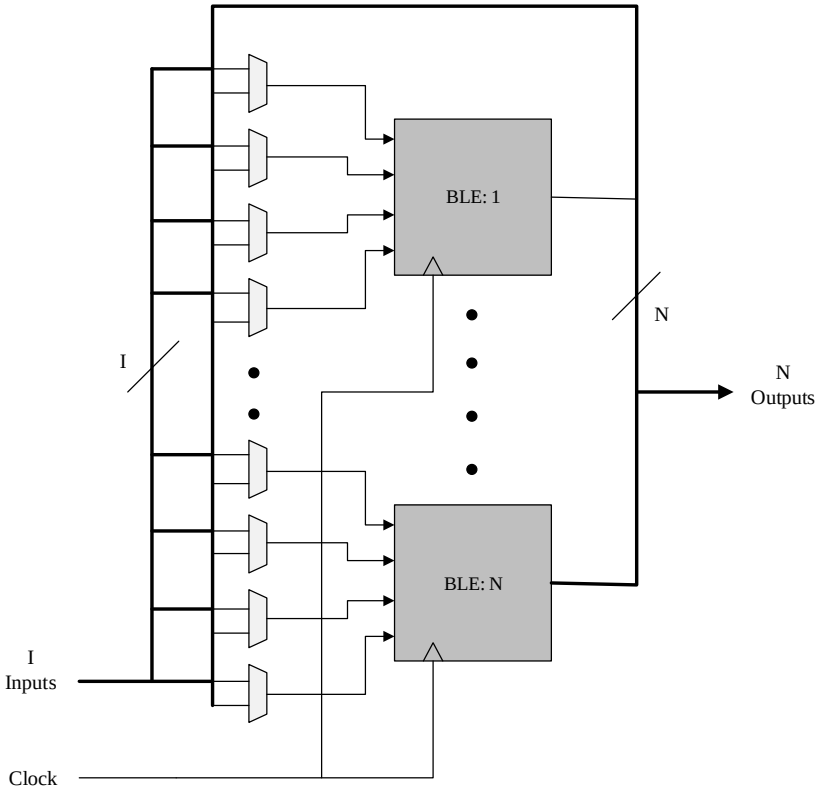

Figure 2. Logic Block

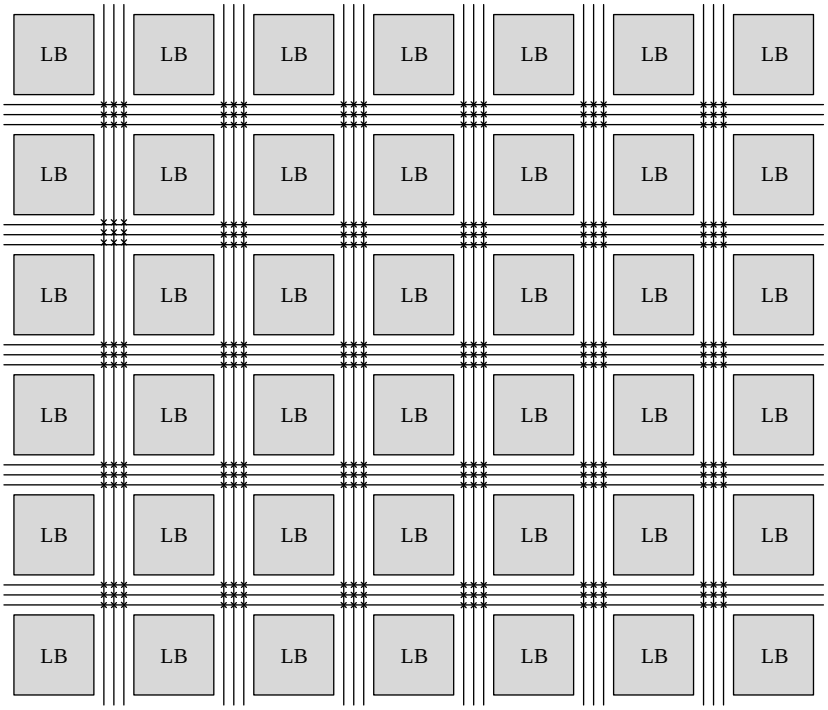

Figure 3. Homogenous FPGA

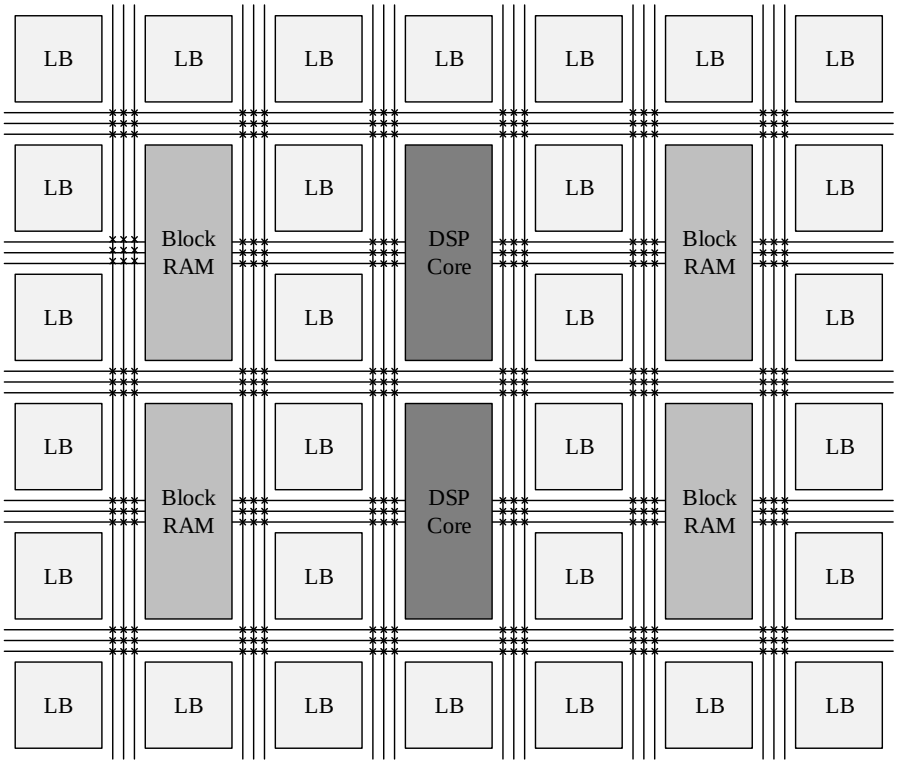

Figure 4. Heterogeneous FPGA

Figure 3, shows a simple homogenous FPGA with LBs and routing network, however most modern FPGAs incorporate the 'hard' blocks such as DSPs or Block RAMs. Even though 'soft logic' can implement any type of digital circuit but having these 'hard' blocks on chip provide more efficiency by trading flexibility. A typical heterogeneous FPGA is shown in Figure 4.

2.1.2 CAD for FPGAs

In order to implement a digital circuit on FPGAs, a designer need to design and describe the circuit at a higher abstraction level which is either in HDL code or simply in graphical description. Before this design is ready to be implemented on FPGA, it needs to be converted into a low-level bit stream which can control the routing and logic switches inside the FPGA. This process requires a set of tools, and is known as 'CAD flow' for FPGAs. From design to device, these tools are required to make a lot of different decisions to implement the circuit more efficiently on the chip, while fulfilling all the design constrains such as area, delay and power [13]. Figure 5, shows a flow of these CAD tools [14].

High Level Synthesis:

A recent trend in FPGA CAD flow is the addition of *HLS (High Level Synthesis)*, which provides the designer a highest level of abstraction by expressing the circuit *algorithmically* in more traditional programming languages such as C [15] and OpenCL [16]. Using this algorithmic description of HLS selects the appropriate hardware for implementation.

Logic Synthesis:

Logical synthesis consists of further 3 steps: elaboration, logic optimization and technology mapping. During elaboration, CAD converts the behavioral description of hardware into logical hardware description. Then logic optimization is then performed to improve the quality of resulting

hardware by reducing area, delay or power. After the optimization performed here, mapping generates the netlist of circuit using primitive set of hardware available in FPGA such as LUTs, Flip-Flops and multipliers. A careful CAD implementation at earlier stages can reduce the over power consumption of FPGA, i.e. 7.6% just because of power-aware technology mapping [17].

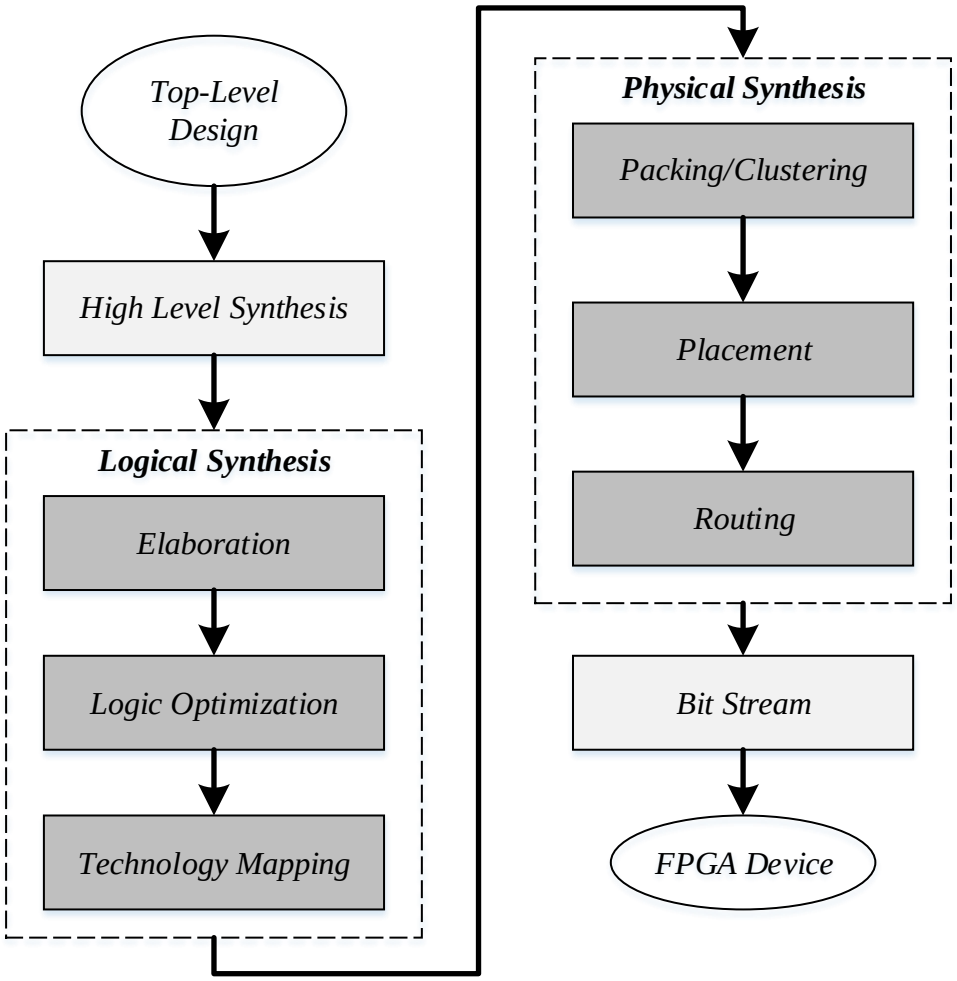

Figure 5. A typical FPGA CAD Flow

Physical Synthesis:

Physical synthesis is performed using various algorithms such as simulated annealing [18] [19] for placement and a variant of Lee's Algorithm [20] for routing [21]. This process consumes a great

amount of compile time for FPGAs CAD flow, and the efficiency (power, speed and area) of final implementation very much depends on the efficiency of CAD tool being used. First step in physical synthesis is *Clustering* or *Packing*, which means grouping together the primitive hardware into the actual blocks of FPGA architecture. A good clustering algorithm also ensures that design is more efficiently routable and thus reducing compile time for later stages such as routing [22] [23].

Next step would *Placement* of these blocks on FPGA, it's one of the very important steps in FPGA implementation, as placement would also determine the accuracy of later steps such as routing. A better placement means less wires to be routed thus reducing the delay and power consumption of the final circuit. A bad placement might actually make a certain design un-routable on FPGA due to a limited number of routing resources available on the chip.

Last step in physical synthesis is *Routing*, which is similar to ASIC design flow, given the placement of different blocks at this point, it is determined how to connect these blocks using the routing resources available on FPGA.

Once the logical and physical synthesis is completed the final design is analyzed to ensure that it meets the design constraints such as delay and area. And finally the bit stream is generated which can be downloaded on the FPGA.

2.2 Processors

Sequential processors have a long and rich history as computational platforms than FPGAs. They are considered one of the most famous platforms for implementing computation. They provide software re-programmability, thus providing much flexibility than other solutions such as ASICs although which comes at a cost of performance. They provide a set of instructions, which are executed sequentially, using these set of instructions a programmer can write a program to

implement any computation. The idea of stored program computers was first implemented in 1940s and used for military applications. There are two types of famous architectures for such computers, which differ in the way they store and access instructions and data; which is either in same or different physical memories.

2.2.1 Von Neumann Machine

Von Neumann architecture [24] stores the data and instructions for Central Processing Unit (CPU) in same physical memory and share the same bus to access them. According to Von Neumann the instructions are stored in a memory and extracted by the processor and then executed in a sequential manner, data stored in the memory is changed while the program is executed. Figure 6, shows a Von Neumann architecture [25].

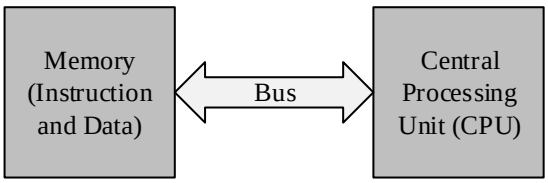

Figure 6. Von Neumann Machine

2.2.2 Harvard Machine

Harvard architecture differs from Von Neumann architectures by storing the instructions and data in separate memories connected with separate busses [26]. Instructions are executed sequentially by the processor, therefore instruction memory can be constructed as sequential access memory which requires less addressing circuitry than random access memories, thus reducing overall power consumption. We have chosen this architecture for our work to implement 4-LUT based processor, which is shown in Figure 10. A general form of Harvard machine is shown in Figure 7.

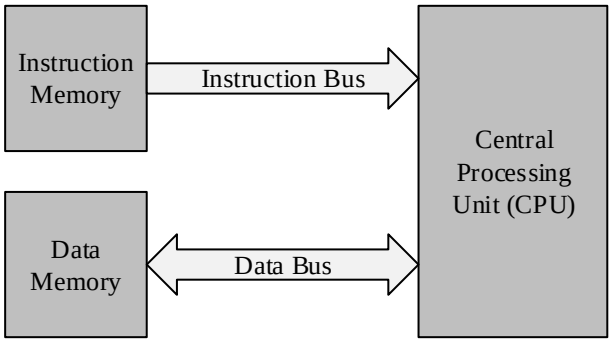

Figure 7. Harvard Architecture

2.2.3 Typical Implementation of Computation on Processors

Implementing computation using processors is much easier and requires less compile time than FPGAs. A typical flow to implement a computation, from a program written in higher level languages such as C and C++ down to hardware is shown in Figure 8 [27].

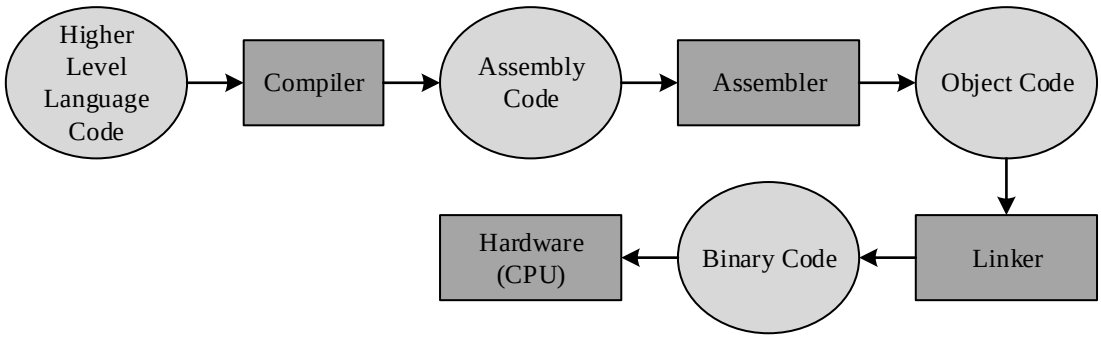

Figure 8. Typical Processor Computation Implementation

Where a *compiler* is a computer program which converts a higher level program, into low level assembly code, which is them read by *assembler* which converts it into *object code* which is the form of input a *linker* would take to generates bits of *0s* and *1s*. As hardware can only understand the instructions in binary code, which is very hard for human programmers to program in, therefore these abstractions provide more ease to program for humans.

2.3 Effects of functional unit complexity on dynamic energy consumption of processors

The dynamic energy consumption of a processor is strongly dependent on the logic complexity of its functional unit. In particular, consider measuring the dynamic energy required to implement an 8-bit wide (7:2) compressor tree [28], as shown in Figure 9, on a sequential processor. As shown, the circuit consists of a series of full adders (FA) acting as (3,2) counters, and one can take the approach proposed in [7] to measure the dynamic energy consumed by the processor by first mapping the circuit into a set of 4-LUTs and then using a LUT-based processor as shown in Figure 10, to execute one LUT configuration at a time.

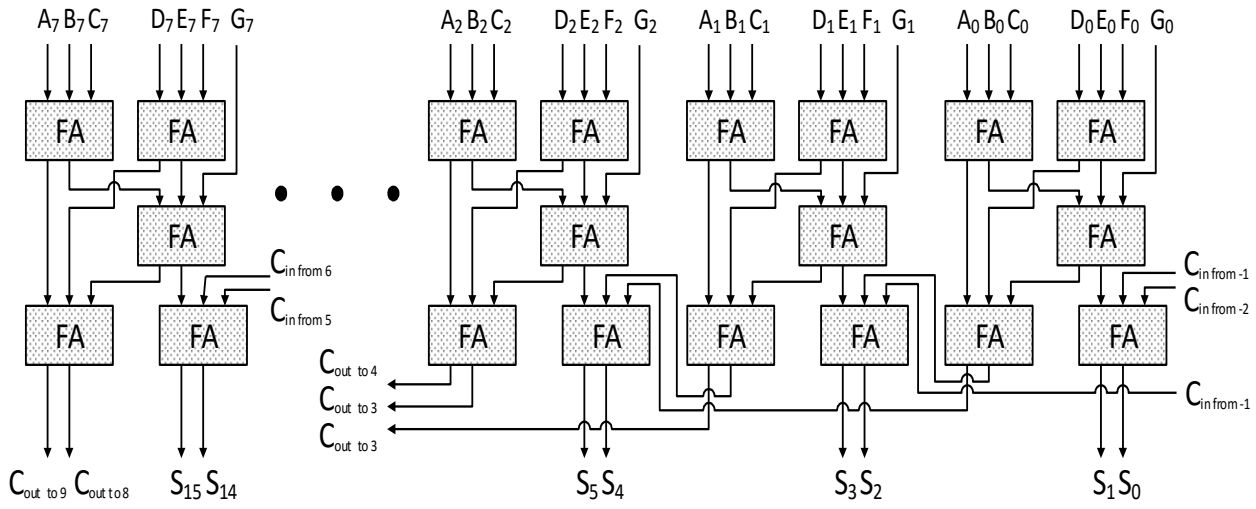

Figure 9. 8bit (7,2) Compressor Tree Circuit.

The processor shown in Figure 10, consists of a functional unit containing a single 4-LUT and connected to two distinct monolithic memory blocks – a sequential access memory for storing instructions and a random access memory for storing data, implemented using 32nm CMOS process [7]. Note that in Table I, "*M*" is the number of words in memory and "*W*" is the width of

each word in bits. As shown in Table I, 80×46 ($M \times W$) bits of instruction memory are required to encode the 4-LUT configurations, where there are 80 instructions (one for each 4-LUT) and each instruction consists of a 16-bit wide 4-LUT configuration bits and 5 6-bit wide addresses to indicate the location of the four (4-LUT inputs) and one (4-LUT output) in data memory. Also, 62×1 ($M \times W$) bits of data memory are needed to store all primary inputs and outputs of the circuit and to provide intermediate storage for internal LUT input and output values.

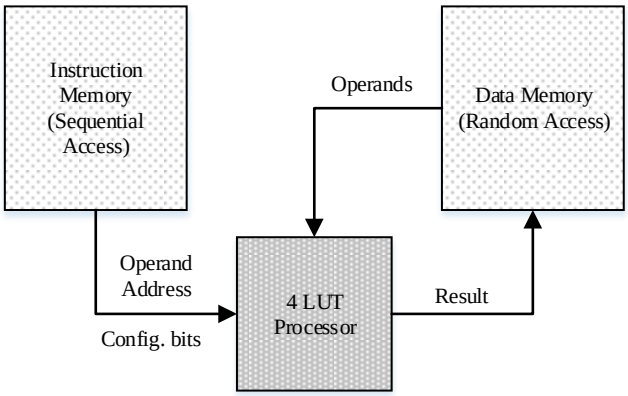

Figure 10. 4LUT Processor System

Once the size of the instruction and data memory is determined for the processor, one can measure the dynamic energy consumed by the processor by summing the total dynamic energy required to read the instructions from the instruction memory, total dynamic energy required to read (operands) and write (results) to the data memory and total dynamic energy required to evaluate each 4-LUT configuration [7].

In particular, the dynamic energy consumed for each read/write access to a memory block is a function of the size of the memory. In particular, for a fixed supply voltage, this energy is proportional to total wiring capacitance that is switched during the memory access and can be

estimated using the following two equations for sequential and random access memory [7], respectively,

$$C_{Random}(W, M) = (log_2(M) + 2(2W + 2))\sqrt{WMA_{bit}}C_u \quad (1)$$

$$C_{sequential}(W, M) = 2(2W + 1)\sqrt{WMA_{bit}}C_u \quad (2)$$

Note that in Equation (1) and Equation (2), "M" is total number of words in the memory block, "W" is the width of each word, "A_{bit}" is the layout area for a one SRAM cell (estimated in [7] as $140f^2$ for a dense SRAM cell, where f is the feature size), and C_u is 6.4×10^{-18} Farads (F), the wiring capacitance per feature size for minimum width wires.

Table I. (7,2) Compressor Tree Dynamic Energy Consumption

		Processor			**FPGA**
		4-LUT	***FA***	***8-bit FA***	
Data Memory	M	62	59	11	11
	W	1	1	8	8
	Capacitance per Access (F) – (1)	8.32×10^{-15}	8.07×10^{-15}	2.8×10^{-14}	2.8×10^{-14}
	No. of Accesses	400	200	25	11
	C_D – Total Capacitance (F)	**3.33×10^{-12}**	**1.61×10^{-12}**	**7.01×10^{-13}**	**3.08×10^{-13}**
Instruction Memory	M	80	40	5	0
	W	46	30	20	0
	Capacitance per Access (F) – (2)	8.54×10^{-13}	3.2×10^{-13}	6.21×10^{-14}	0
	No. of Accesses	80	40	5	0
	C_I – Total Capacitance (F)	**6.84×10^{-11}**	**1.28×10^{-11}**	**3.1×10^{-13}**	**0**
Functional Unit	Capacitance per Access (F)	3.63×10^{-15}	3.19×10^{-14}	2.68×10^{-13}	1.8×10^{-11}
	No. of Access	80	40	5	1
	C_F – Total Capacitance (F)	**2.91×10^{-13}**	**1.28×10^{-12}**	**1.34×10^{-12}**	**1.8×10^{-11}**
Total Capacitance (pF) (C_D+C_I+C_F)		**71.97**	**15.69**	**2.35**	**18.31**
Total Energy (pJ) (V_{dd} = 1.8V)		**116.6**	**25.42**	**3.81**	**29.67**

Table I presents the results obtained from the calculation. We use Equation (1) and Equation (2) to calculate the dynamic energy required to access instruction (sequential access) and data (random access) memory. In addition, we also estimate the 4-LUT evaluation energy using Equation (1) by treating the 4-LUT functional unit as a 16×1 ($M \times W$) bit random access memory. As shown in Table I, overall the 4-LUT processor is required to switch 71.97(pF) of capacitance in order to perform a full set of calculations of the (7:2) compressor tree. Assuming 1.8(V) supply voltage (V_{dd}), this capacitance corresponds to 116.6(pJ) of total dynamic energy, calculated using Equation (3) [29].

$$E_{Dynamic} = 0.5CV_{dd}{}^{2} \qquad (3)$$

It is important to note that this LUT-based functional unit is extremely inefficient since LUTs provide too fine-grain level of configuration. For example, we can replace the 4-LUT based functional unit by a full adder (FA) implemented on ASIC. As shown by column 3 (column – Processor: FA) of Table I, the new processor has significantly reduced instruction memory size as well as significantly reduced the number of accesses to both the instruction and data memory. This leads to a significant reduction in total capacitance switched to access data and instruction memory from 3.33(pF) and 68.4(pF) to 1.61(pF) and 12.8(pF), respectively, and results in the reduction of total dynamic energy from 116.6(pJ) to 25.42(pJ).

Note that, we assume the FA (and the 8-bit FA) functional unit is implemented in ASIC and its energy is estimated using VPR 4.30 place and route result [30] and converted to ASIC energy using the methodology from [6] by assuming ASIC implementations consume 1/14th of the energy of FPGA implementations. It would be an over estimation as a careful ASIC implementation optimized for dynamic energy for such small circuits can be more efficient but for consistency and

to keep the model simple for calculations we use the approach described in [6] throughout this work.

Extending the data-path width per instruction can further reduce the size of the instruction memory and the number of accesses to both the instruction and data memory. As shown by column 5 (column – Processor: 8-bit FA) of Table I, the total dynamic energy consumption is reduced to 3.81(pJ) for a processor with 8-bit wide full adder as its functional unit.

Table I, also lists the total dynamic energy consumed by (7:2) compressor tree placed and routed on a 4-LUT FPGA using VPR 4.30. The total logic and routing capacitance switched for executing one set of compressor tree calculations are then calculated and shown in column 6 (column – FPGA) of Table I. The calculation assumes the primary inputs and outputs of the FPGA are stored in random access memory. The energy required to read configuration bit stream for FPGA was deliberately ignored due to the fact that this energy has very little contribution in total energy and can be amortized on large number of computations FPGAs perform after being configured once. Compared to the processor with 8-bit wide FA (implemented in ASIC) as its functional unit, the FPGA implementation consumes 7.7× more dynamic energy, while the FPGA consumes 3.92× less dynamic energy than the single 4-LUT processor implementation.

Given that processors can gain energy efficiency by using more specialized functional units in order to reduce the number of accesses and size of both instruction and data memory, the energy required to perform the same set of functions on a processor is strongly dependent on the complexity of the functional units that the processor contains. Consequently, for the remainder of this thesis, we measure the amount of instruction and data memory reduction that is required in order for a processor to become more dynamic energy efficient than LUT-based FPGAs.

Chapter 3

Experimental Procedure

3.1 Overview

As in [7], wire-capacitance-based models are used in this work to measure the dynamic energy consumption of processors and FPGAs. In particular, for a processor, we measure the dynamic energy consumed by the processor for accessing its instruction memory, its data memory and its functional units. For an FPGA, we measure both the dynamic energy consumed by the LUT evaluations and the routing network. Finally, ASIC energy is estimated as 1/14th of FPGA energy [6]. The detailed experimental procedure that was used to measure the dynamic energy consumption of the MCNC benchmarks on a processor with a 4-LUT-based functional unit is presented in Section 3.2, and the experimental procedure used to measure the dynamic energy consumption of the same benchmarks on an FPGA is presented in Section 3.3.

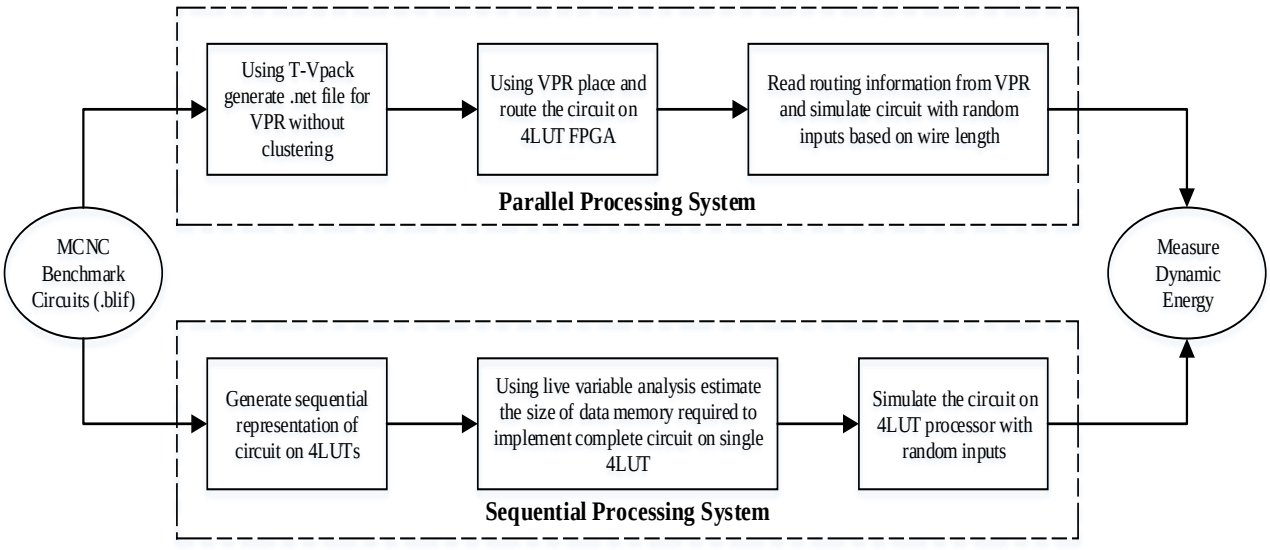

Figure 11. Experimental Procedure (CAD Flow)

3.2 Sequential/Temporal Processing Energy

The CAD flow for estimating the dynamic energy required to execute the MCNC benchmarks on a sequential processing system is shown in Figure 11. In particular, we first map each 4-LUT-based FPGA circuit onto the 4-LUT-based processor as shown in Figure 10. T-Vpack [30] is first used to read each MCNC benchmark circuit in BLIF (Berkeley Logic Interchange Format) [31]. Once read, our tool first creates a directed graph representation of the circuit and then map the directed graph into a sequential temporal representation through breath-first search [32]. In particular, we schedule each LUT into a sequential execution sequence only if the sources of all inputs to the LUT have been scheduled previously, in software terms satisfying the variable dependencies in the program.

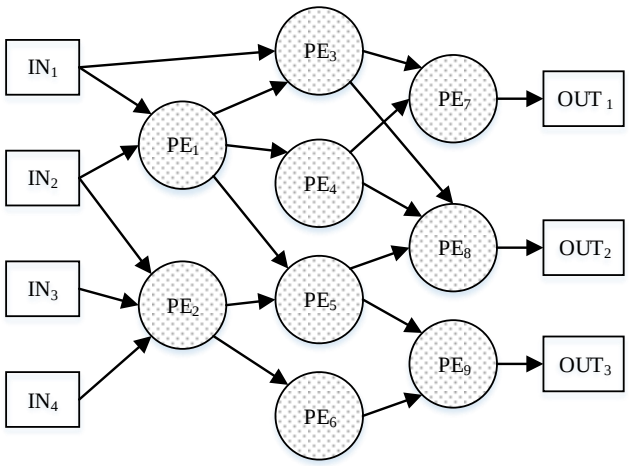

Figure 12. Graph Representation of Circuit

Figure 12, and Figure 13, show an example of such a schedule. In particular, Figure 12, shows the directed graph representation of a circuit. Here the rectangular nodes represent the primary inputs, outputs from registers, primary outputs and inputs to registers. Circular nodes represent single 4-LUT-based processing elements "PE_x", each containing only one 4-LUT configuration. The

scheduler does a breath-first search through the graph and schedules primary inputs and register outputs first. Then each LUT is scheduled only if all nodes that generate the input edges to the LUT have already been scheduled before and the final sequential schedule is shown in Figure 13. Note that in Figure 13, the data memory location associated with OUT_1, OUT_2 and OUT_3 will be updated as part of PE_7, PE_8 and PE_9 execution respectively. Also, any registered outputs for LUTs (DFFs) are also scheduled before all Processing Elements (PEs). For the very first simulation cycle, they are considered to have a random value, which will be updated to the actual LUT result after each later cycle during simulation. Also note that in Figure 13, t_1 to t_9 represents different times during execution. For example, at t_1 the LUT is configured as PE_1 and after evaluation it is reconfigured as PE_2 for second evaluation in t_2, it is then reconfigured as PE_3 for evaluation in t_3, and so on.

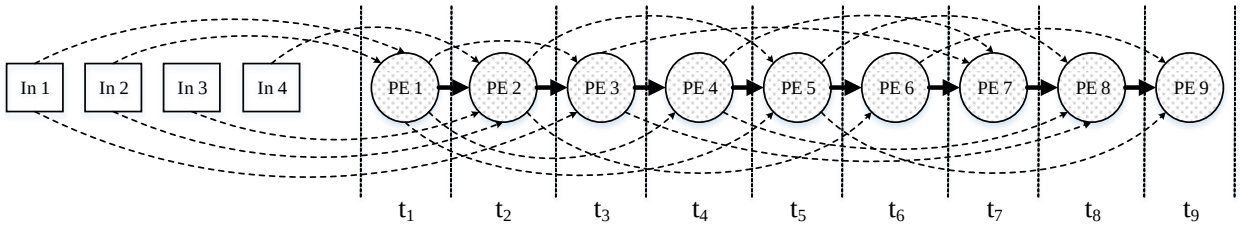

Figure 13. Linked List Representation of Circuit

In order to measure the dynamic energy consumed by the 4-LUT processor system, we perform a cycle by cycle simulation. During each cycle, the processor follows a traditional fetch, decode, execute and store cycle by first reading an instruction from the instruction memory. The instruction contains a 16-bit LUT configuration and five addresses indicating the location of the four inputs to the LUT and the one output of the LUT in data memory. Based on these addresses, the processor

then reads the operands of the instruction from data memory. The 4-LUT instruction is then evaluated and the output written back to data memory.

Consequently, during each execution cycle, the processor would read the instruction memory once and read data memory for the maximum of four times and write to data memory once. Note that the number of read access to the data memory is the instruction dependent. Finally, we calculate the total dynamic energy consumed by the 4-LUT processor in each cycle by summing the dynamic energy consumed, to access the instruction memory, the data memory and the energy required to evaluate the 4-LUT functional unit as shown by Equation (4).

$$E_{Total} = E_{Data\ Memory} + E_{Instruction\ Memory} + E_{Functional\ Unit} \quad (4)$$

Dynamic power consumed by a circuit is directly proportional to the amount of capacitance being charged and discharged during its operation. We assume instruction memory is implemented in sequential access memory, as instructions are executed in strictly sequential manner and sequential access requires less energy than random access due to reduced decoding circuitry. We measure the capacitance consumed by the instruction memory based on Equation (2). Note that in order to estimate the capacitance, we need the size of the instruction memory in terms of the total number of memory words ($M_{Instruction}$) and the width of each memory word in bits ($W_{Instruction}$). "$W_{Instruction}$" and "$M_{Instruction}$" are calculated using Equation (5) and Equation (6), respectively,

$$W_{Instruction} = 2^k + ceil(5log_2(M_{Data})) \quad (5)$$

$$M_{Instruction} = N_{LUTs} \quad (6)$$

where "k" is the size of the LUT, "M_{Data}" is the number of words contained in data memory, "N_{LUTs}" is the total number of LUTs in each circuit and "$ceil$" means ceiling function. "W" and

"M" in Equation (2) is then set to "$W_{Instruction}$" and "$M_{Instruction}$", respectively, to calculate dynamic energy per access for the instruction memory.

Similarly, the capacitance consumed by data memory is measured using Equation (1). Note that LUT output is always a 1 bit wide therefore "W_{Data}" is always equal to 1. "M_{Data}" is computed based on our live variable analysis results, explained in Section 3.2.1.

In particular, since data memory is implemented in random access memory, we can minimize the size of data memory by re-using the memory words containing data that are no longer required by the program (dead variables). Therefore, to measure the accurate size of the required data memory, we used live variable analysis algorithm [33] to eliminate dead variables after the execution of each instruction and measure "M_{Data}" as the largest required memory size during the execution of an entire program. "W" and "M" in Equation (1) are then set to "W_{Data}" and "M_{Data}" respectively to estimate to dynamic energy required for each access to the data memory.

For a single 4-LUT as a functional unit, we estimate the capacitance consumed by the functional unit itself also using Equation (1). Note that during each simulation cycle, each 4-LUT will be accessed twice, containing 1 write and 1 read access. First the 4-LUT is re-configured (1 write access) and then evaluated (1 read access) for each instruction. In particular, for the write access, we assume the write circuitry is implemented using random access memory containing 1 16-bit word by setting "W" to 16 and "M" to 1. For read access, we assume the read circuitry is implemented using the same SRAM cells but organized into 16 1-bit wide words by setting "W" to 1 and "M" to 16.

3.2.1 Live Variable Analysis

In order to measure the correct amount of data memory words "M_{Data}" which is required to perform a single operation on 4-LUT processor, we implemented an algorithm to estimate the minimum amount of data memory required. A naive way of selecting data memory, to execute the complete schedule in Figure 13, is to choose a random memory containing words "M_{Data}" equal to the total number of LUTs "N_{LUTs}" in the circuit. An efficient implementation would be to measure the highest number of intermediate values stored in the data memory at any moment of time, during program execution. The purpose of this algorithm is to accurately measure the "M_{Data}" based on the maximum "alive" variables or intermediate values stored in data memory at any moment of time during program execution.

This algorithm starts by initially going through the schedule in reverse order (starting from the LUTs scheduled at the end and directly connected to OutPads of FPGA), it will update a list of pointers in each node of the linked-list as shown in Figure 13, by providing it a list of pointers to all nodes it is being connected to in the future.

Once each node is aware of its future connections, we can start traversing the linked-list again from the very first node in the schedule and create a list of all connections (live variables) for future nodes. Also at each node we check if that particular node is present in the list, if it is present then it is considered as dead. Figure 14, provides the flow chart for this algorithm. Largest size of this list at any particular time during this traversal would be the minimum amount of data memory required to perform a single operation on 4-LUT processor.

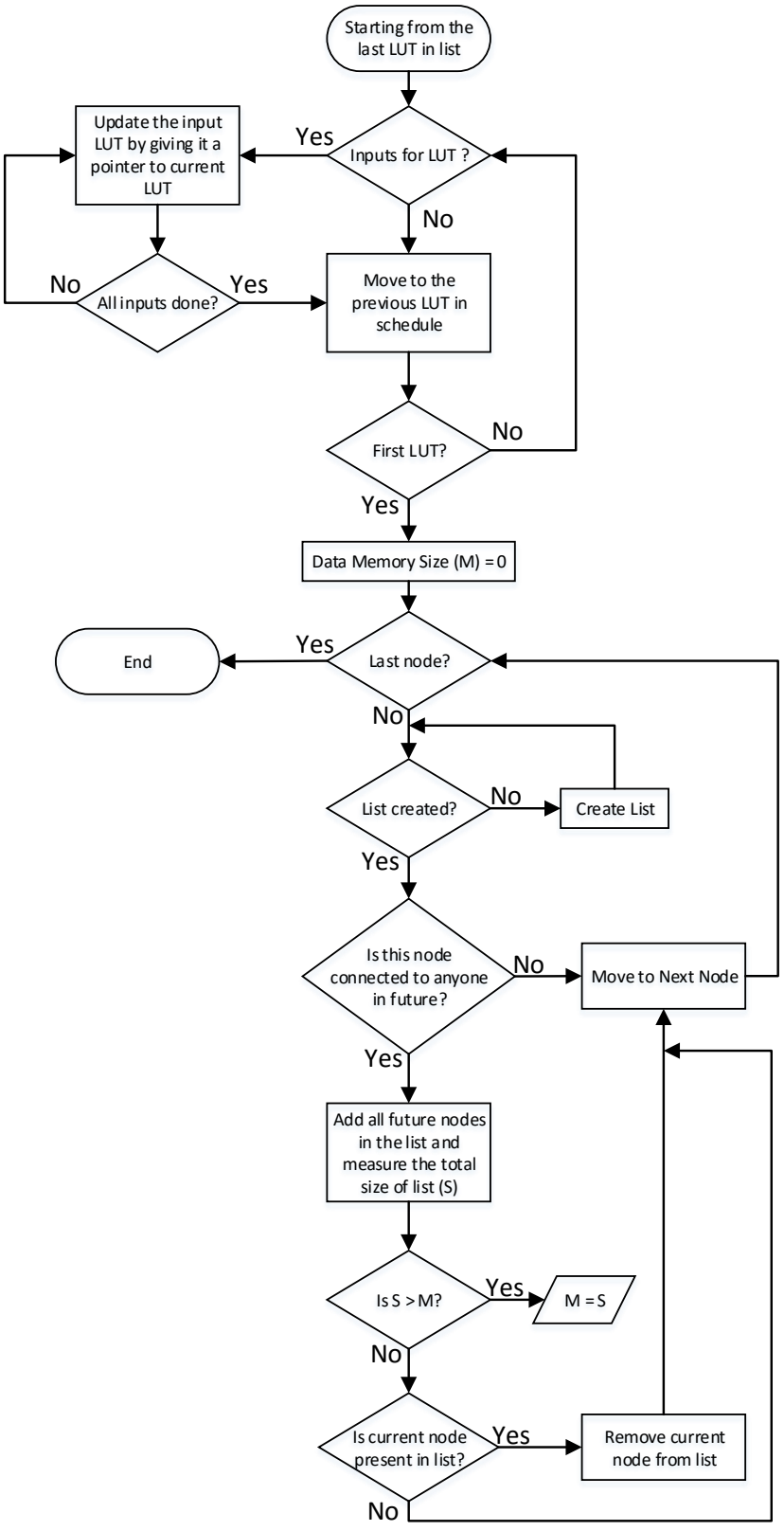

Figure 14. Live Variable Analysis Algorithm

3.3 Parallel/Spatial Processing Energy

Our simulator uses the same schedule to measure the dynamic energy consumption of the 4-LUT-based FPGA that is used to measure the dynamic energy consumption of 4-LUT-based processor. Again a cycle by cycle simulation is performed to measure the actual net activities based on the results of individual LUT evaluations, using random primary inputs. Total dynamic energy consumed by the FPGA is then calculated as the sum of the energy required to switch the routing network and evaluate all LUTs as described in Equation (7).

$$E_{Total(FPGA)} = E_{LUTs\ Evaluations} + E_{Routing\ Network} \qquad (7)$$

To measure LUT energy, we use Equation (1) by assuming each 4-LUT as a random access memory with 16 1-bit wide words. For routing energy, the CAD flow shown in Figure 11, is used. The VPR 4.30 tool chain [30] is used to map the MCNC benchmarks onto an island style FPGA where each logic block contains one 4-LUT and each routing track expands two logic blocks before being interrupted by routing switches. The VPR tools are then used to search for the minimum number of routing tracks that are needed to successfully route each circuit.

Assuming each FPGA tile is laid out in a square, the minimum width transistor area [34] per tile (A_{mwt}) reported by the VPR tool is then used to calculate the length of routing tracks that is needed to span one FPGA tile. Minimum width transistor area means the layout area required to implement a smallest transistor in any process. VPR provides a generalized process free area estimation. Minimum width transistor area model for VPR assumes the layout area occupied by the transistor and distance required from its neighboring transistors, such as shown in Figure 15 from [34]. This area model is however improved by [35] by proving that VPR underestimates the actual layout area by a factor of 2.5×, therefore Equation (8) adjusts the actual area by this factor.

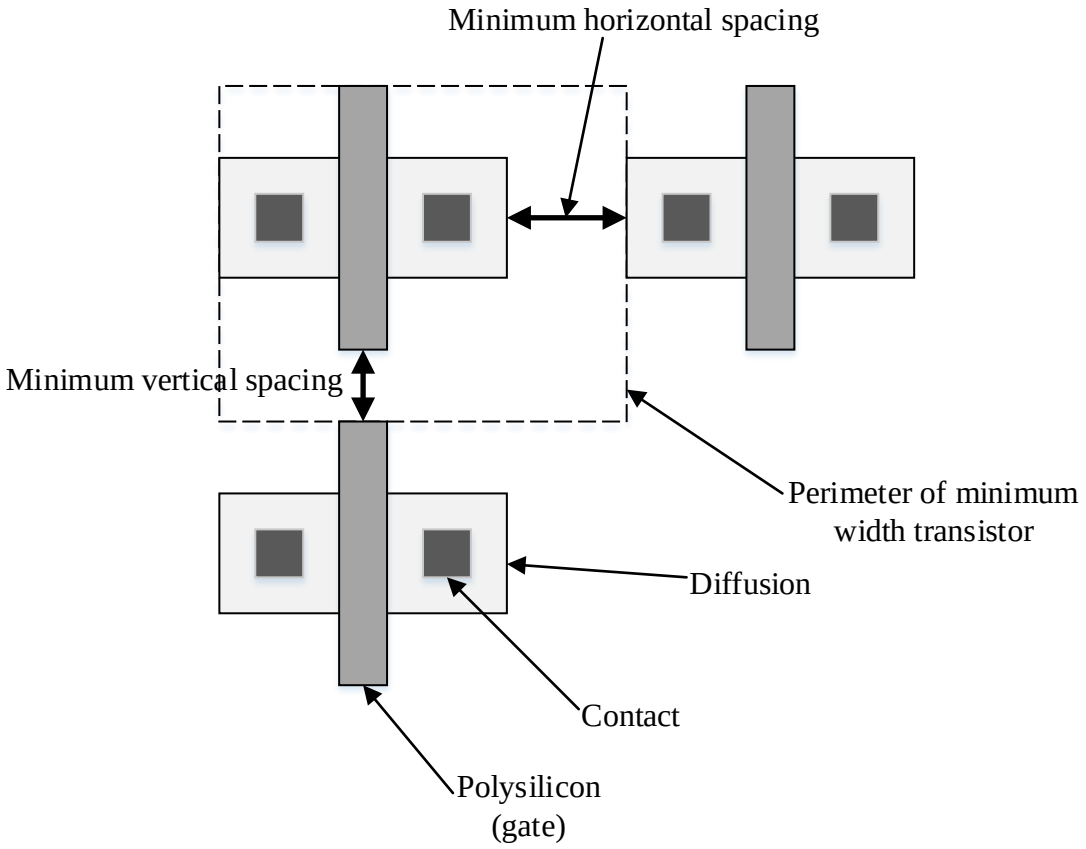

Figure 15. Definition of a minimum width transistor area.

The area is first converted into feature size based area using the lambda(λ)-based layout rules from [29] by assuming each minimum area transistor takes $40f^2$ layout area. The feature size based area is then adjusted by a factor of 2.5× to account for the under-estimation of actual layout area by the minimum width transistor area model [35]. Finally, for each routing connection that corresponds to an edge in our routing schedule (shown as dashed lines in Figure 13,) the Manhattan distance (D_{Man}) that the edge spans on the FPGA is then used to calculate the wiring capacitance of the routing connection "$C_{Routing}$" based on Equation (8).

$$C_{Routing} = D_{Man}\sqrt{2.5A_{mwt}40f^2}C_u \qquad (8)$$

where f is feature size and "C_u" is 6.4×10^{-18} (F) at 32nm process, the wiring capacitance per feature size for minimum width wires [7].

Note that energy required for configuring the LUTs and the routing resources are ignored by our simulation since once configured an FPGA is typically used over many cycles of computations and once amortized over all the cycles the configuration energy becomes negligible.

Chapter 4

Experimental Results

4.1 Overview

We first present the dynamic energy consumption of the simple 4-LUT-based processor and compare the processor's energy consumption to the energy consumption of the 4-LUT-based FPGA for the MCNC benchmarks. The data-path width of the processor is then increased from 1-bit to 256-bits, and its effects on the relative dynamic energy consumption between the processor and the FPGA is then measured. Finally, the effect of reducing the instruction and data memory sizes on the energy efficient of the processor over the FPGA is measured to account for the effect of replacing the 4-LUT based functional unit by more complex ASIC based functional units.

4.2 4-LUT Processor vs 4-LUT FPGA (Dynamic Energy)

Table II provides the results obtained from the experiment performed with a single 4-LUT as the functional unit for the processor. Column 1 lists the name of each benchmark. Column 2, 3, and 4 list the three components of the CPU dynamic energy consumption including the instruction memory, data memory, and functional unit dynamic energy consumption, respectively. Column 5 lists the total dynamic energy consumption of the CPU. The components of FPGA dynamic energy consumption are shown in column 6 and 7 including the routing energy consumption and 4-LUT evaluation energy consumption, respectively. The total FPGA dynamic energy consumption is shown in column 8. Finally, the ratio between the total CPU energy consumption and FPGA energy consumption is shown in column 9. As shown, on average for the MCNC benchmarks, the 4-LUT

processor consumes 717× more dynamic energy to perform the same computation than the 4-LUT FPGA.

Table II. Dynamic Energy Consumption of MCNC Benchmarks 1bit (100% Memory)

MCNC Benchmark	Processor Dynamic Energy (nJ)				FPGA Dynamic Energy (nJ)			Total CPU/ Total FPGA
	Instruction Memory	*Data Memory*	*Functional Unit*	*Total CPU*	*Routing*	*4-LUT Evaluation*	*Total FPGA*	
alu4	224.69	4.15	0.11	228.95	1.37	0.11	1.48	154.58
apex2	307.97	5.70	0.14	313.81	1.85	0.14	1.99	157.36
apex4	169.65	2.95	0.05	172.65	0.62	0.10	0.72	240.26
bigkey	301.90	6.19	0.08	308.17	0.98	0.13	1.11	277.65
clma	3881.01	56.83	0.32	3938.16	3.90	0.63	4.53	869.43
des	240.14	4.63	0.14	244.92	1.91	0.12	2.03	120.89
diffeq	273.66	4.31	0.02	277.99	0.15	0.11	0.27	1038.92
dsip	223.26	5.48	0.07	228.81	0.84	0.10	0.95	242.05
elliptic	1306.20	17.19	0.05	1323.45	0.47	0.27	0.74	1796.47
ex1010	131.33	2.14	0.05	133.52	0.59	0.08	0.67	199.32
ex5p	1436.42	23.80	0.11	1460.33	1.13	0.35	1.47	990.53
frisk	1212.78	15.12	0.04	1227.94	0.27	0.27	0.54	2275.50
misex3	197.59	3.70	0.09	201.37	1.09	0.11	1.19	169.01
pdc	1425.66	25.58	0.15	1451.39	1.75	0.34	2.10	691.75
s298	322.18	5.58	0.04	327.79	0.40	0.15	0.54	605.07
s38417	2719.04	36.48	0.08	2755.61	0.67	0.46	1.13	2436.12
s38584.1	2753.42	39.72	0.09	2793.23	0.76	0.47	1.24	2259.65
seq	277.03	4.85	0.12	282.00	1.54	0.13	1.67	169.03
spla	938.94	16.56	0.12	955.62	1.40	0.28	1.68	569.66
tseng	175.13	2.37	0.01	177.52	0.10	0.08	0.18	970.59
Average:	**925.90**	**14.17**	**0.09**	**940.16**	**1.09**	**0.22**	**1.31**	**717.13**

Table III presents the energy distribution among memories and the functional unit. As show in column 2 of Table III, majority of this energy (98.48%) is consumed in reading the instructions from sequential memory.

Table III. Energy Distribution for Processor

SIMD Bit(s)/Mem Size (%)	**1/100**	**64/100**	**64/9**
Instruction Memory (%)	98.48	36.44	34.27
Data Memory (%)	1.51	63.36	58.78
Functional Unit (%)	0.01	0.20	6.95

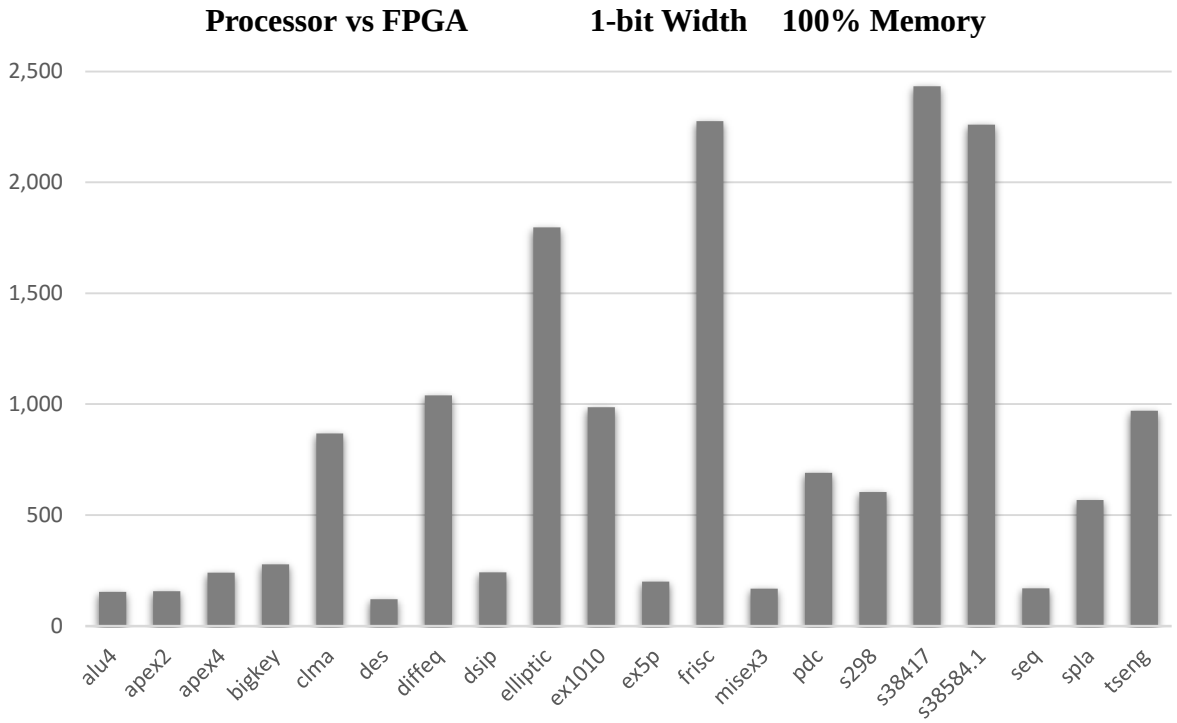

Figure 16. Ratio for MCNC Benchmarks (CPU/FPGA)

Figure 16, shows the difference between the energy consumption of CPU when using single 4-LUT as processor and an island style FPGA for MCNC Benchmarks, placed and routed by VPR.

4.3 Increasing data-path (SIMD)

Processors can increase their dynamic energy efficiency by sharing the same instruction on a set of data that is more than 1-bit wide. This property is known as SIMD, *single instruction multiple data* [7]. Storing a set of bits in the same word in data memory reduces the overall energy per bit required to address the word. FPGAs, however, require a complete replication of same the LUT network for each additional bit. Therefore, using a wider data-path should decrease the energy efficiency gap between the processor and the FPGA.

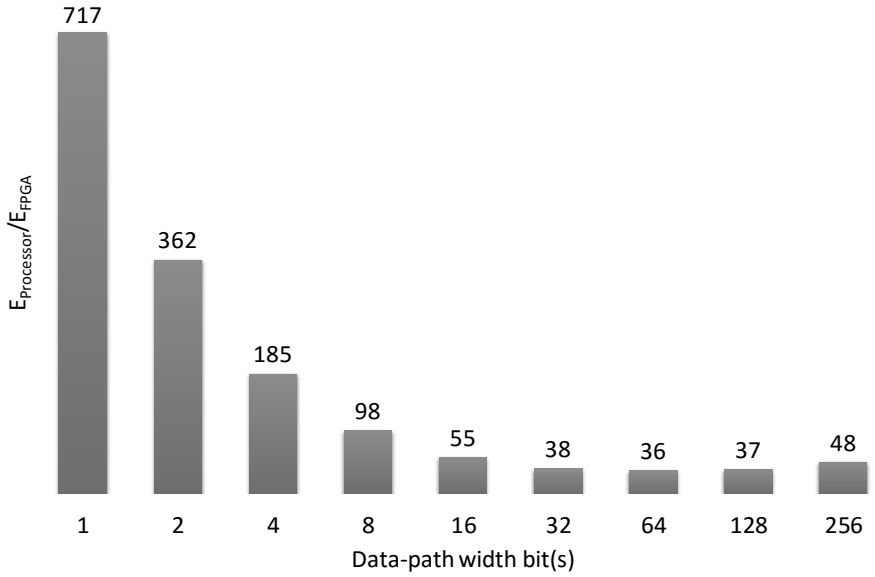

Figure 17. Dynamic Energy Ratio Vs Increasing Bit Width

Figure 17, shows the ratio between processor dynamic energy consumption ($E_{Processor}$) and FPGA dynamic energy consumption (E_{FPGA}) as a function of data path width. As shown, the ratio of the dynamic energy consumed by the processor and the FPGA decreases as the data-path width is increased. The ratio reaches a minimum at 64 bits, where the processor consumes 36× more energy than the FPGA. After 64-bit width, this ratio starts to increase due to the non-linear nature of the random access memory model as a function of bit width [7].

Note that at 1-bit wide data-path, the processor contains a single 4-LUT as its functional unit. However, for multiple bit wide data-paths, the processor contains one additional 4-LUT for each additional bit. This increases the per access dynamic energy consumed by the functional unit. Data memory energy also increases as its width increases. However, there is no change in dynamic energy consumption by the instruction memory. This behavior is summarized by column 3 of Table III, where, at 64-bit wide data-path width, the data memory consumes the most dynamic energy (63.36%), instruction memory consumes (36.44%) of the total dynamic energy and the functional unit still consumes a very small (0.2%) but increasing amount of the total dynamic energy due to the reduction in the overall all dynamic energy consumption of the processor.

Table IV, Table V, Table VI, Table VII, Table VIII, Table IX, Table X and Table XI provides the energies in (n)Joules for all 20 MCNC Benchmarks, at 2, 4, 8, 16, 32, 64, 128 and 256 bits wide data paths, respectively. In each table; column 1, lists the MCNC Benchmark function; column 2, provides the energy consumed by instruction memory for processor; column 3, provides the energy consumed by data memory; column 4, provides the energy consumed by the array of 4-LUT(s) in functional units; column 5, lists the total energy consumed by CPU, while column 6 and 7 lists the FPGA energy and ratio of $E_{Processor}$ and E_{FPGA}. While last row provides the arithmetic mean for all 20 MCNC Benchmarks, except the last cell for 'Total CPU/ Total FPGA' provides the ratio between 'Arithmetic Mean of CPU' by 'Arithmetic Mean of FPGA'.

Table IV. Dynamic Energy Consumption of MCNC Benchmarks 2bit (100% Memory)

MCNC Benchmark	Processor Dynamic Energy (nJ)				FPGA Dynamic Energy (nJ)	Total CPU/ Total FPGA
	Instruction Memory	*Data Memory*	*Functional Unit*	*Total CPU*		
alu4	56.17	1.80	0.05	58.03	0.74	78.39
apex2	76.99	2.47	0.07	79.53	1.00	79.88
apex4	42.41	1.28	0.03	43.72	0.36	121.68
bigkey	75.48	2.67	0.04	78.19	0.56	140.82
clma	970.25	24.15	0.16	994.56	2.27	438.68
des	60.04	2.01	0.07	62.12	1.01	61.35
diffeq	68.42	1.87	0.01	70.29	0.13	525.57
dsip	55.82	2.36	0.03	58.21	0.47	123.17
elliptic	326.55	7.36	0.03	333.93	0.37	905.68
ex1010	359.11	10.17	0.05	369.32	0.74	500.12
ex5p	32.83	0.94	0.02	33.80	0.33	101.32
frisk	303.20	6.49	0.02	309.70	0.27	1147.69
misex3	49.40	1.61	0.04	51.05	0.60	85.58
pdc	356.41	10.93	0.07	367.42	1.05	350.70
s298	80.54	2.42	0.02	82.98	0.27	305.65
s38417	679.76	15.53	0.04	695.33	0.56	1236.22
s38584.1	688.35	16.87	0.04	705.27	0.62	1140.21
seq	69.26	2.11	0.06	71.42	0.84	85.51
spla	234.73	7.11	0.06	241.90	0.84	287.80
tseng	43.78	1.03	0.01	44.82	0.09	490.35
Average:	231.48	6.06	0.05	237.58	0.66	362.41

Table V. Dynamic Energy Consumption of MCNC Benchmarks 4bit (100% Memory)

MCNC Benchmark	Processor Dynamic Energy (nJ)				FPGA Dynamic Energy (nJ)	Total CPU/ Total FPGA
	Instruction Memory	*Data Memory*	*Functional Unit*	*Total CPU*		
alu4	56.17	3.50	0.11	59.78	1.48	40.35
apex2	76.99	4.77	0.14	81.91	2.00	41.04
apex4	42.41	2.50	0.05	44.96	0.72	62.53
bigkey	75.48	5.14	0.08	80.70	1.11	72.51
clma	970.25	45.64	0.32	1016.21	4.53	224.28
des	60.04	3.90	0.14	64.08	2.03	31.62
diffeq	68.42	3.61	0.02	72.04	0.27	269.21
dsip	55.82	4.55	0.07	60.43	0.94	64.01
elliptic	326.55	14.02	0.05	340.62	0.74	462.12
ex1010	359.11	19.33	0.11	378.54	1.48	256.22
ex5p	32.83	1.84	0.05	34.72	0.67	51.96
frisk	303.20	12.41	0.04	315.64	0.54	584.98
misex3	49.40	3.12	0.09	52.60	1.19	44.16
pdc	356.41	20.77	0.15	377.34	2.10	179.52
s298	80.54	4.68	0.04	85.26	0.54	156.98
s38417	679.76	29.44	0.08	709.28	1.13	627.73
s38584.1	688.35	31.85	0.09	720.29	1.24	582.35
seq	69.26	4.08	0.12	73.46	1.67	43.93
spla	234.73	13.59	0.12	248.45	1.68	147.68
tseng	43.78	2.01	0.01	45.81	0.18	250.52
Average:	231.48	11.54	0.09	243.11	1.31	185.29

Table VI. Dynamic Energy Consumption of MCNC Benchmarks 8bit (100% Memory)

MCNC Benchmark	Processor Dynamic Energy (nJ)				FPGA Dynamic Energy (nJ)	Total CPU/ Total FPGA
	Instruction Memory	*Data Memory*	*Functional Unit*	*Total CPU*		
alu4	56.17	7.63	0.21	64.01	2.96	21.61
apex2	76.99	10.37	0.28	87.65	3.98	22.01
apex4	42.41	5.47	0.10	47.99	1.44	33.38
bigkey	75.48	11.12	0.16	86.76	2.22	39.03
clma	970.25	97.01	0.65	1067.91	9.05	117.98
des	60.04	8.49	0.29	68.81	4.05	16.99
diffeq	68.42	7.85	0.04	76.31	0.53	142.82
dsip	55.82	9.86	0.13	65.81	1.89	34.85
elliptic	326.55	30.04	0.11	356.70	1.47	242.05
ex1010	359.11	41.35	0.21	400.66	2.96	135.58
ex5p	32.83	4.06	0.10	36.99	1.34	27.58
frisk	303.20	26.70	0.08	329.97	1.08	305.75
misex3	49.40	6.81	0.17	56.38	2.38	23.64
pdc	356.41	44.43	0.30	401.14	4.20	95.52
s298	80.54	10.19	0.08	90.81	1.08	83.76
s38417	679.76	62.74	0.16	742.66	2.25	329.73
s38584.1	688.35	67.65	0.18	756.18	2.47	305.55
seq	69.26	8.89	0.24	78.39	3.34	23.47
spla	234.73	29.25	0.24	264.22	3.37	78.48
tseng	43.78	4.41	0.03	48.22	0.37	131.92
Average:	231.48	24.72	0.19	256.38	2.62	97.76

Table VII. Dynamic Energy Consumption of MCNC Benchmarks 16bit (100% Memory)

MCNC Benchmark	Processor Dynamic Energy (nJ)				FPGA Dynamic Energy (nJ)	Total CPU/ Total FPGA
	Instruction Memory	*Data Memory*	*Functional Unit*	*Total CPU*		
alu4	56.17	18.37	0.42	74.97	5.92	12.66
apex2	76.99	24.91	0.57	102.48	7.98	12.84
apex4	42.41	13.22	0.21	55.84	2.87	19.44
bigkey	75.48	26.60	0.32	102.40	4.45	22.99
clma	970.25	229.04	1.29	1200.59	18.09	66.36
des	60.04	20.42	0.58	81.04	8.09	10.01
diffeq	68.42	18.88	0.08	87.37	1.07	81.57
dsip	55.82	23.62	0.27	79.71	3.78	21.09
elliptic	326.55	71.39	0.21	398.15	2.95	135.12
ex1010	359.11	98.10	0.42	457.63	5.91	77.41
ex5p	32.83	9.85	0.19	42.88	2.67	16.04
frisk	303.20	63.63	0.15	366.98	2.16	170.02
misex3	49.40	16.40	0.34	66.14	4.76	13.88
pdc	356.41	105.39	0.60	462.40	8.38	55.16
s298	80.54	24.50	0.15	105.20	2.16	48.60
s38417	679.76	148.49	0.32	828.58	4.54	182.32
s38584.1	688.35	159.56	0.35	848.27	4.95	171.48
seq	69.26	21.39	0.48	91.13	6.68	13.64
spla	234.73	69.71	0.48	304.93	6.73	45.32
tseng	43.78	10.65	0.05	54.48	0.73	74.47
Average:	231.48	58.71	0.37	290.56	5.24	55.40

Table VIII. Dynamic Energy Consumption of MCNC Benchmarks 32bit (100% Memory)

MCNC Benchmark	Processor Dynamic Energy (nJ)				FPGA Dynamic Energy (nJ)	Total CPU/ Total FPGA
	Instruction Memory	*Data Memory*	*Functional Unit*	*Total CPU*		
alu4	56.17	47.44	0.85	104.46	11.85	8.82
apex2	76.99	64.21	1.14	142.34	15.94	8.93
apex4	42.41	34.21	0.41	77.04	5.75	13.39
bigkey	75.48	68.49	0.63	144.60	8.89	16.27
clma	970.25	583.71	2.01	1555.97	28.12	55.34
des	60.04	52.72	1.16	113.91	16.21	7.03
diffeq	68.42	48.68	0.15	117.25	2.14	54.77
dsip	55.82	60.79	0.54	117.15	7.57	15.48
elliptic	326.55	182.72	0.42	509.69	5.90	86.43
ex1010	359.11	250.81	0.84	610.76	11.83	51.63
ex5p	32.83	25.57	0.38	58.78	5.35	10.99
frisk	303.20	163.16	0.31	466.67	4.32	108.11
misex3	49.40	42.37	0.68	92.45	9.55	9.68
pdc	356.41	269.42	1.20	627.03	16.75	37.44
s298	80.54	63.20	0.31	144.06	4.34	33.20
s38417	679.76	378.78	0.64	1059.19	9.02	117.46
s38584.1	688.35	406.44	0.71	1095.50	9.89	110.75
seq	69.26	55.20	0.95	125.41	13.35	9.40
spla	234.73	178.76	0.96	414.45	13.46	30.79
tseng	43.78	27.55	0.10	71.43	1.46	48.84
Average:	231.48	150.21	0.72	382.41	10.08	37.93

Table IX. Dynamic Energy Consumption of MCNC Benchmarks 64bit (100% Memory)

MCNC Benchmark	Processor Dynamic Energy (nJ)				FPGA Dynamic Energy (nJ)	Total CPU/ Total FPGA
	Instruction Memory	*Data Memory*	*Functional Unit*	*Total CPU*		
alu4	56.17	127.76	1.69	185.63	23.71	7.83
apex2	76.99	172.78	2.28	252.05	31.90	7.90
apex4	42.41	92.24	0.82	135.47	11.49	11.79
bigkey	75.48	184.19	1.27	260.94	17.78	14.68
clma	970.25	1560.35	4.60	2535.20	64.41	39.36
des	60.04	141.95	2.32	204.30	32.42	6.30
diffeq	68.42	131.01	0.31	199.73	4.28	46.72
dsip	55.82	163.42	1.08	220.32	15.13	14.56
elliptic	326.55	489.64	0.26	816.46	3.70	220.71
ex1010	359.11	671.70	1.11	1031.92	15.57	66.26
ex5p	32.83	69.05	0.76	102.65	10.70	9.59
frisk	303.20	437.73	0.04	740.97	0.54	1370.15
misex3	49.40	114.11	1.36	164.87	19.07	8.64
pdc	356.41	721.51	1.82	1079.74	25.43	42.45
s298	80.54	170.14	0.62	251.31	8.67	28.97
s38417	679.76	1013.33	0.71	1693.80	9.92	170.69
s38584.1	688.35	1086.28	0.84	1775.47	11.70	151.79
seq	69.26	148.64	1.91	219.81	26.73	8.22
spla	234.73	479.57	1.34	715.65	18.80	38.07
tseng	43.78	74.29	0.21	118.28	2.93	40.40
Average:	231.48	402.49	1.27	635.23	17.74	35.80

Table X. ***Dynamic Energy Consumption of MCNC Benchmarks 128bit (100% Memory)***

MCNC Benchmark	Processor Dynamic Energy (nJ)				FPGA Dynamic Energy (nJ)	Total CPU/ Total FPGA
	Instruction Memory	*Data Memory*	*Functional Unit*	*Total CPU*		
alu4	56.17	352.36	3.39	411.93	47.45	8.68
apex2	76.99	476.17	3.98	557.15	55.76	9.99
apex4	42.41	254.51	1.64	298.57	22.99	12.99
bigkey	75.48	506.92	1.96	584.36	27.44	21.30
clma	970.25	4285.04	9.19	5264.49	128.68	40.91
des	60.04	391.39	4.63	456.06	64.84	7.03
diffeq	68.42	361.19	0.61	430.21	8.56	50.28
dsip	55.82	450.18	2.16	508.15	30.27	16.79
elliptic	326.55	1346.49	1.11	1674.14	15.48	108.15
ex1010	359.11	1846.56	2.80	2208.46	39.17	56.39
ex5p	32.83	190.72	1.53	225.08	21.41	10.52
frisk	303.20	1204.50	0.66	1508.35	9.18	164.37
misex3	49.40	314.69	2.73	366.81	38.16	9.61
pdc	356.41	1983.57	4.22	2344.20	59.10	39.67
s298	80.54	468.97	0.66	550.17	9.23	59.62
s38417	679.76	2785.19	1.43	3466.38	19.99	173.39
s38584.1	688.35	2982.02	1.67	3672.04	23.35	157.23
seq	69.26	409.73	3.23	482.22	45.22	10.66
spla	234.73	1319.52	3.26	1557.51	45.68	34.10
tseng	43.78	204.96	0.42	249.16	5.85	42.58
Average:	231.48	1106.73	2.56	1340.77	35.89	37.36

Table XI. Dynamic Energy Consumption of MCNC Benchmarks 256bit (100% Memory)

MCNC Benchmark	Processor Dynamic Energy (nJ)				FPGA Dynamic Energy (nJ)	Total CPU/ Total FPGA
	Instruction Memory	*Data Memory*	*Functional Unit*	*Total CPU*		
alu4	56.17	983.68	6.19	1046.05	86.69	12.07
apex2	76.99	1328.93	8.52	1414.44	119.25	11.86
apex4	42.41	710.86	2.71	755.97	37.88	19.96
bigkey	75.48	1412.49	4.50	1492.47	63.01	23.69
clma	970.25	11937.63	17.77	12925.66	248.82	51.95
des	60.04	1092.54	8.68	1161.25	121.47	9.56
diffeq	68.42	1008.34	0.65	1077.41	9.04	119.16
dsip	55.82	1256.27	3.75	1315.83	52.48	25.08
elliptic	326.55	3754.02	2.21	4082.78	30.95	131.91
ex1010	359.11	5147.49	5.03	5511.63	70.49	78.19
ex5p	32.83	532.92	2.48	568.23	34.70	16.38
frisk	303.20	3359.29	1.31	3663.79	18.35	199.67
misex3	49.40	878.71	4.88	932.99	68.28	13.66
pdc	356.41	5529.24	7.86	5893.51	109.98	53.59
s298	80.54	1309.36	1.90	1391.80	26.61	52.31
s38417	679.76	7760.94	2.86	8443.56	40.06	210.79
s38584.1	688.35	8309.68	3.35	9001.39	46.84	192.18
seq	69.26	1143.84	7.04	1220.13	98.50	12.39
spla	234.73	3680.12	6.52	3921.37	91.29	42.95
tseng	43.78	572.42	0.26	616.46	3.61	170.95
Average:	231.48	3085.44	4.92	3321.84	68.91	48.20

4.4 Compressing Memories at 64-bit SIMD

As discussed in Section 2.3, the energy consumed by a processor can be reduced by increasing the complexity of the functional unit that it contains. This increase in turn reduces both the amount of data and instruction memory that are required to execute the same set of operations. To measure this effect of increasing functional unit complexity on the overall dynamic energy gap between the processor and the FPGA, we measure the overall dynamic energy consumption as a function of reduced memory size. In our measurement, we assume both the instruction memory and data memory are reduced by the same amount and a reduction in memory size leads to a proportional reduction in memory access.

Note that as the complexity of a functional unit increases, the energy per access would increase. The number of accesses, however, will be proportionally reduced. Consequently, in this work we assume the overall total dynamic energy consumed by the functional unit ($E_{Functional\_Unit}$ in Equation (4)) would remain the same as the complexity of the functional unit is increased.

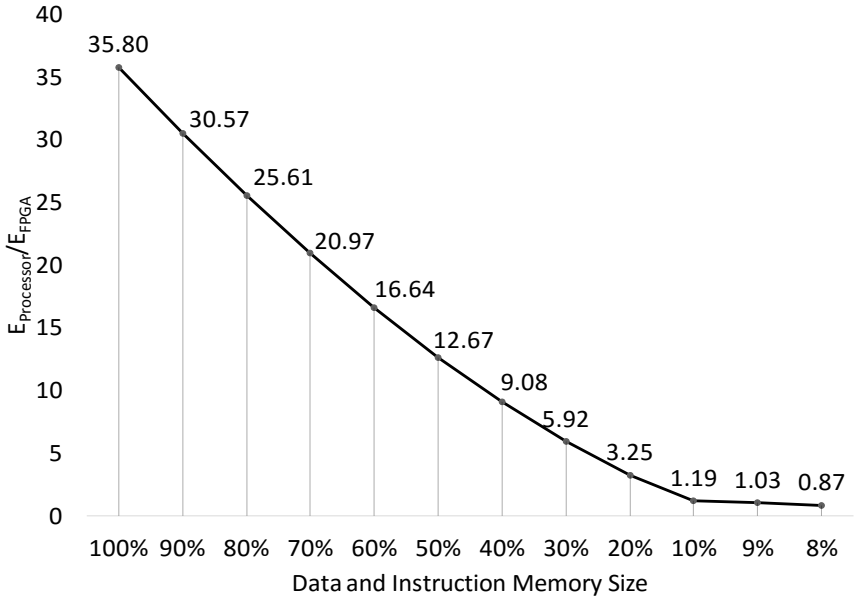

Figure 18. Average Energy Ratio for MCNC Benchmarks at 64bit(s)

Table XII. Dynamic Energy at 64bit(s) 9% Memory Size

MCNC Benchmark	CPU Dynamic Energy (nJ)				FPGA Dynamic Energy (nJ)			Total CPU/ Total FPGA
	Instruction Memory	*Data Memory*	*Functional Unit*	*Total CPU*	*Routing*	*LUT Evaluation*	*Total FPGA*	
alu4	6.04	13.61	6.76	26.42	87.36	7.34	94.70	0.28
apex2	8.31	18.33	9.11	35.75	118.43	9.06	127.49	0.28
ex4	4.57	9.79	3.29	17.65	39.96	6.09	46.05	0.38
bigkey	8.13	19.62	5.08	32.84	62.93	8.23	71.16	0.46
clma	104.77	166.25	18.40	289.41	249.50	8.06	257.56	1.12
des	6.48	15.12	9.26	30.86	122.01	7.67	129.69	0.24
diffeq	7.38	13.96	1.22	22.55	9.90	7.21	17.10	1.32
dsip	6.02	17.37	4.32	27.71	53.82	6.61	60.43	0.46
elliptic	35.26	52.07	1.06	88.39	29.77	15.00	14.77	5.98
ex1010	3.53	7.28	3.06	13.87	37.67	5.13	42.81	0.32
ex5p	38.74	71.51	4.44	114.69	72.31	10.19	62.12	1.85
frisk	32.72	46.63	0.15	79.51	17.46	15.30	2.16	36.83
misex3	5.32	12.15	5.45	22.92	69.58	6.74	76.32	0.30
pdc	38.46	76.82	7.28	122.56	112.24	10.30	101.94	1.20
s298	8.68	18.09	2.48	29.25	25.40	9.31	34.71	0.84
s38417	73.37	107.92	2.88	184.17	43.25	2.97	40.29	4.57
s38584.1	74.32	115.70	3.34	193.36	48.78	2.07	46.71	4.14
seq	7.47	15.80	7.63	30.91	98.44	8.44	106.89	0.29
spla	25.35	51.09	5.36	81.80	89.66	14.57	75.09	1.09
tseng	4.73	7.85	0.83	13.41	6.64	5.05	11.69	1.15
Average:	**24.98**	**42.85**	**5.07**	**72.90**	**69.76**	**8.27**	**70.98**	**1.03**

Figure 18 shows the ratio between the dynamic energy consumption of the processor ($E_{processor}$) and the dynamic energy consumption of the FPGA (E_{FPGA}) as the instruction and data memory sizes are reduced. As shown, when the memory size (and access) is reduced to around 9% of the memory required for a processor that employs 64-bit wide 4-LUT-based functional unit, the processor consumes the same amount of dynamic energy as the 4-LUT-based FPGA.

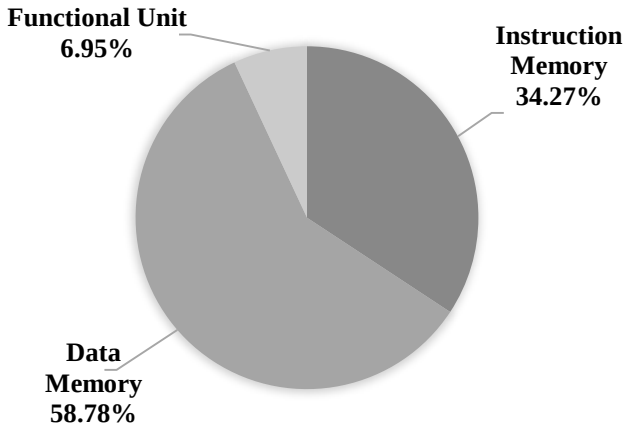

Figure 19. Energy Distribution at 64bit (9% Memory)

The detailed dynamic energy consumption at the 9% memory sizes is shown in Table XII for each benchmark circuit. As shown the energy consumption ratio between the processor and FPGA also varies as the benchmark set is varied. In particular, for the largest 10 benchmarks, on average the processor consumes 1.21x the energy of the 4-LUT-based FPGA at 9% memory size. For the smallest 10 benchmarks, on the other hand, the processor consumes only 0.36x the energy of the 4-LUT-based FPGA at the same memory size – i.e. the processor is already 3× times more dynamic energy efficient than the FPGA for the smaller benchmarks.

Finally, Figure 19 provides the energy distribution among the functional unit and the instruction and data memory. As shown, as the size of the data and instruction memory is reduced, the functional unit consumes an increased proportion of the total dynamic energy. At 9% memory size,

6.95% of the total dynamic energy is consumed by the functional unit while the instruction and data memory consume 34.27% and 58.78% of the total dynamic energy respectively. Figure 20 provides the energies consumed by both CPU and FPGA at 9% memory for 64-bit wide data path.

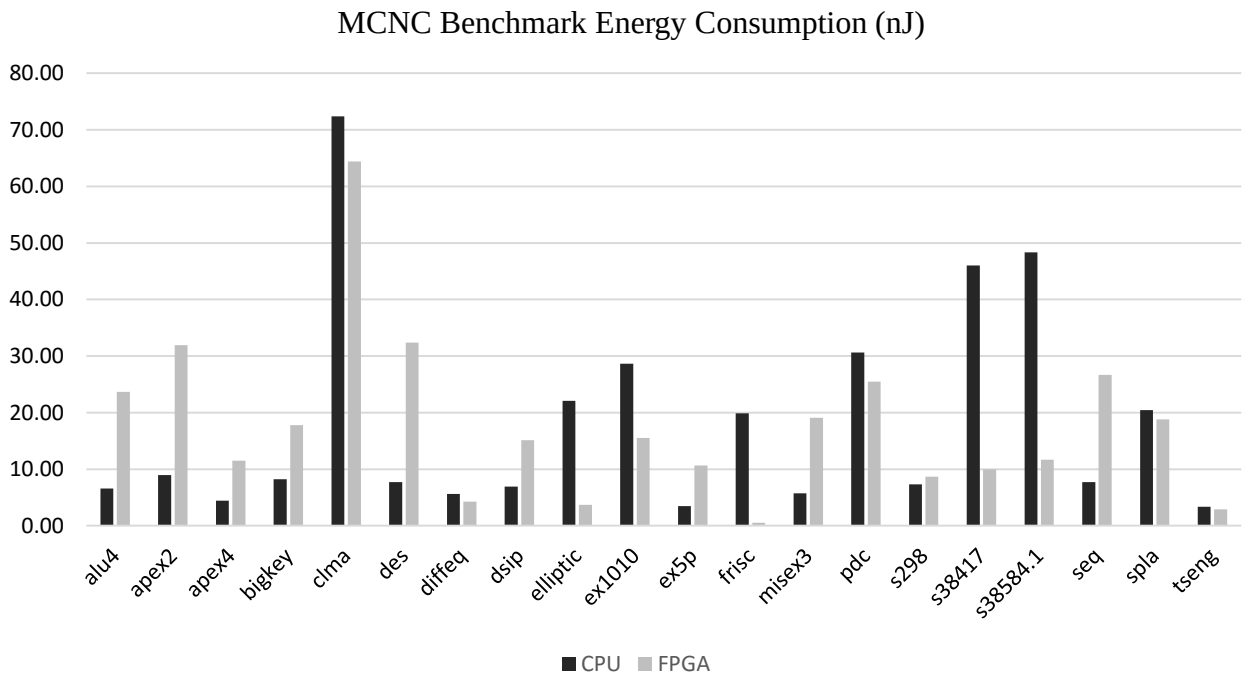

Figure 20. Dynamic Energy Consumption at 64bit(s) 9% Memory

Chapter 5

Conclusion and Future work

5.1 Conclusion

In this work, we first measured the dynamic energy difference between the 4-LUT FPGA and a 4-LUT-based sequential processor, by implementing the MCNC benchmarks on both processing platforms. A single 4-LUT processor was found to be much more inefficient in consuming dynamic energy than a 4-LUT FPGA. We optimized the processor for dynamic energy by increasing the number of 4-LUTs in its functional units and then performed the same measurements for our benchmarks. We showed that the average dynamic energy ratio between the processor and the FPGA is at a minimum when the processor contains 64 4-LUTs as its functional unit. Adding more 4-LUTs increases this ratio.

We then measured dynamic energy gap between the processor and the CPU as the complexity of the functional unit is increased. We found that for the MCNC benchmarks, on average, if a processor with ASIC-based functional unit can perform the same computation while consuming less than 9% of the instruction and data as is required by the same processor with 64-bit wide 4-LUT-based functional unit, it can be more efficient in dynamic energy than a 4-LUT-based FPGA. Our results show that the dynamic energy consumption of processors and FPGAs is not only a function of their sequential and parallel execution models but also a function of the complexity of the functional units/accelerators that they contain.

5.2 Future work

Future work, can be performed by adding static energy consumption estimations for the system. Also memory models can be improved by adding gate capacitances as well which are ignored here to keep the model simple and only wire capacitances are considered. Our simulator can be grouped as a part of the CAD tool chain for implementing future designs on complex System-On-Chip (SoC), containing both sequential and parallel processing capabilities on same Integrated Circuit (IC). After some further exploration this tool can make decisions at the compile-time to distribute computations between the parallel and sequential processing elements available, based on the dynamic energy requirements.

Since it is much easier to increase the complexity of a functional unit in the sequential execution model of a processor than the parallel execution model employed by an FPGA, it is highly important for the FPGA research community to continue to increase the complexity of the hard blocks that FPGAs employ while maintaining the utilization of these blocks under the parallel execution model in order for FPGAs to maintain their dynamic energy advantage over the processors.

Appendix A

Source Code (.c and .h Files)

Source code for simulation is provided in this appendix, we have reused some code from T-Vpack to read the MCNC Benchmarks in BLIF format in file '*read_blif.c*' and some utility functions in file '*util.c*'. Some part of '*main.c*' is also reused from T-Vpack, rest of the code in files '*graph.c*', '*simulation.c*' and '*read_route.c*' is completely new code. This section provides the code for all these file, which includes the reused and new code both in the order we used for our simulation.

Header files and makefiles to compile this code on Linux operating system are provided after the c-files. In the end of this appendix, I have provided the BLIF files used for the example in chapter 2.

Following is the list of all files:

c-files:

1. main.c
2. read_blif.c
3. util.c
4. graph.c
5. simulation.c
6. read_route.c

h-files:

1. globals.h
2. graph.h
3. read_blif.h
4. read_route.h
5. simulation.h
6. util.h
7. vpack.h

blif files:

1. 8_bit_compressor_tree
2. 8_Full_Adders
3. Full_Adder

c-files:

main.c:

```
#include <stdio.h>
#include <string.h>
#include "graph.h"
#include "util.h"
#include "vpack.h"
#include "globals.h"
#include "read_blif.h"
#include "simulation.h"
#include "read_route.h"

#define BIG_INT 32000

int num_nets, num_blocks;
int num_p_inputs, num_p_outputs, num_luts, num_latches;
struct s_net *net;
struct s_block *block;
struct s_node *graph;
int graph_size;

int num_input_pins (int iblk) {
/* Returns the number of used input pins on this block. */

 int conn_inps;
 switch (block[iblk].type) {
 case LUT:
    conn_inps = block[iblk].num_nets - 1;   /* -1 for output. */
    break;
 case LUT_AND_LATCH:
    conn_inps = block[iblk].num_nets - 2;  /* -2 for output + clock */
    break;
 case LATCH:
    conn_inps = block[iblk].num_nets - 2;  /* -2 for output + clock */
    break;
 default:
/* This routine should only be used for logic blocks. */
    printf("Error in num_input_pins:  Unexpected block type %d"
           "\n for block %d.  Aborting.\n", block[iblk].type, iblk);
```

```
    exit(1);
    break;
 }
 return (conn_inps);
}
int main (int argc, char *argv[]) {
 char title[] = "\nDynamic Energy Consumption: 4-LUT FPGA vs CPU(Functional
Unit: 4-LUT) version 1.0\
 \nt-vpack [blif] [route input] [route output] [sim_cycles]
[area_clb(lambda)]\
 [datapath_width(bits)] [inst_mem_%(1.0, 0.9, 0.8...0)] [data_mem_%(1.0, 0.9,
0.8...0)]\n";

 char *blif_file = argv[1];         //Input Blif File
 char *route_input_file = argv[2];  //Input Route Information from VPR
 char *route_output_file = argv[3];
 int sim_cycles = atoi(argv[4]);    //Number of Simulation Cycles to be
performed
 float area_clb = atof(argv[5]);    //CLB area in Lambda from VPR
 int data_bits = atoi(argv[6]);           //Datapath width
 float inst_comp = atof(argv[7]);
 float data_comp = atof(argv[8]);

 printf("%s",title);
 read_blif(blif_file, 4);
 echo_input(blif_file, 4, "input.echo");
 printf("\nAfter packing to LUT+FF Logic Blocks:\n");
 printf("LUT+FF Logic Blocks: %d.  Total Nets: %d.\n", num_blocks -
       num_p_outputs - num_p_inputs, num_nets);
FILE *data_file;
data_file = fopen("data.out","a");
fprintf(data_file,"SimCycles:%d
%s\tL:%d\tFF:%d\t",sim_cycles,blif_file,num_luts,num_latches);
fclose(data_file);
graph_size = (num_blocks-num_p_outputs)+1;
graph = generate_graph(block,net,num_blocks,num_p_outputs,graph_size);
struct s_node* schedule = generate_schedule(graph,block,net);
update_net_lengths(schedule, route_input_file, route_output_file);
int data_mem = measure_data_mem(schedule,num_p_inputs+num_luts+num_latches);
simulate(schedule,sim_cycles,num_luts,num_latches,area_clb,data_mem,data_bits
,inst_comp,data_comp);
printf("\nComplete.\n\n");
 return (0);
}
```

read_blif.c:

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "util.h"
#include "vpack.h"
#include "globals.h"
#include "read_blif.h"
/* This source file will read in a FLAT blif netlist consisting     *
* of .inputs, .outputs, .names and .latch commands.  It currently   *
* does not handle hierarchical blif files.  Hierarchical            *
* blif files can be flattened via the read_blif and write_blif      *
* commands of sis.  LUT circuits should only have .names commands;  *
* there should be no gates.  This parser performs limited error     *
* checking concerning the consistency of the netlist it obtains.    *
* .inputs and .outputs statements must be given; this parser does   *
* not infer primary inputs and outputs from non-driven and fanout   *
* free nodes.  This parser can be extended to do this if necessary, *
* or the sis read_blif and write_blif commands can be used to put a *
* netlist into the standard format.                                 *
* V. Betz, August 25, 1994.                                         *
* Added more error checking, March 30, 1995, V. Betz                */
static int *num_driver, *temp_num_pins;
/* # of .input, .output, .model and .end lines */
static int ilines, olines, model_lines, endlines;
static struct hash_nets **hash;
static char *model;
static FILE *blif;
static int add_net (char *ptr, int type, int bnum, int doall);
static void get_tok(char *buffer, int pass, int doall, int *done, int
lut_size);
static void init_parse(int doall);
static void check_net (int lut_size);
static void free_parse (void);
static void io_line (int in_or_out, int doall);
static void add_lut (int doall, int lut_size);
static void add_latch (int doall, int lut_size);
static void dum_parse (char *buf);
static int hash_value (char *name);
static void add_truth_table(char*,char*);
static void fill_table(int);
```

```
static void decode_address(char*,int*);
static int address_valid(char*);
void read_blif (char *blif_file, int lut_size) {
 char buffer[BUFSIZE];
 int pass, done, doall;
 blif = my_fopen (blif_file, "r", 0);
 for (doall=0;doall<=1;doall++) {
    init_parse(doall);
/* Three passes to ensure inputs are first blocks, outputs second and    *
 * LUTs and latches third.  Just makes the output netlist more readable. */
    for (pass=1;pass<=3;pass++) {
       linenum = 0;   /* Reset line number. */
       done = 0;
       while((my_fgets(buffer,BUFSIZE,blif) != NULL) && !done) {
          get_tok(buffer, pass, doall, &done, lut_size);
       }
       rewind (blif);  /* Start at beginning of file again */
    }
 }
 fclose(blif);
 check_net(lut_size);
 free_parse();
}
static void init_parse(int doall) {
/* Allocates and initializes the data structures needed for the parse. */
 int i, len;
 struct hash_nets *h_ptr;
 if (!doall) {  /* Initialization before first (counting) pass */
    num_nets = 0;
    hash = (struct hash_nets **) my_calloc(sizeof(struct hash_nets *),
            HASHSIZE);
 }
/* Allocate memory for second (load) pass */
 else {
    net = (struct s_net *) my_malloc(num_nets*sizeof(struct s_net));
    block = (struct s_block *) my_malloc(num_blocks*
        sizeof(struct s_block));
    num_driver = (int *) my_malloc(num_nets * sizeof(int));
    temp_num_pins = (int *) my_malloc(num_nets*sizeof(int));

    for (i=0;i<num_nets;i++) {
       num_driver[i] = 0;
```

```
       net[i].num_pins = 0;
    }
    for (i=0;i<HASHSIZE;i++) {
       h_ptr = hash[i];
       while (h_ptr != NULL) {
          net[h_ptr->index].pins = (int *) my_malloc(h_ptr-
>count*sizeof(int));
/* For avoiding assigning values beyond end of pins array. */
          temp_num_pins[h_ptr->index] = h_ptr->count;
          len = strlen (h_ptr->name);
          net[h_ptr->index].name = (char *) my_malloc ((len + 1)*
sizeof(char));
          strcpy (net[h_ptr->index].name, h_ptr->name);
          h_ptr = h_ptr->next;
       }
    }
/*    printf("i\ttemp_num_pins\n\n");
    for (i=0;i<num_nets;i++) {
       printf("%d\t%d\n",i,temp_num_pins[i]);
    }  */
 }
/* Initializations for both passes. */
 ilines = 0;
 olines = 0;
 model_lines = 0;
 endlines = 0;
 num_p_inputs = 0;
 num_p_outputs = 0;
 num_luts = 0;
 num_latches = 0;
 num_blocks = 0;
}
static void get_tok (char *buffer, int pass, int doall, int *done,
       int lut_size) {
/* Figures out which, if any token is at the start of this line and *
 * takes the appropriate action.                                    */

#define TOKENS " \t\n"
 char *ptr;
  ptr = my_strtok(buffer,TOKENS,blif,buffer);
 if (ptr == NULL) return;
  else if (strcmp(ptr,".names") == 0) {
    if (pass == 3) {
```

```
      add_lut(doall, lut_size);
      if (doall)
            fill_table(0);
   }
   else {
      dum_parse(buffer);
   }
   return;
}
else if (strcmp(ptr,".latch") == 0) {
   if (pass == 3) {
      add_latch (doall, lut_size);
   }
   else {
      dum_parse(buffer);
   }
   return;
}
else if (strcmp(ptr,".model") == 0) {
   ptr = my_strtok(NULL,TOKENS,blif,buffer);
   if (doall && pass == 3) { /* Only bother on main second pass. */
      if (ptr != NULL) {
         model = (char *) my_malloc ((strlen(ptr)+1) * sizeof(char));
         strcpy(model,ptr);
      }
      else {
         model = (char *) my_malloc (sizeof(char));
         model[0] = '\0';
      }
      model_lines++;              /* For error checking only */
   }
   return;
}
else if (strcmp(ptr,".inputs") == 0) {
   if (pass == 1) {
      io_line(DRIVER, doall);
      *done = 1;
   }
   else {
      dum_parse(buffer);
      if (pass == 3 && doall) ilines++;    /* Error checking only */
   }
```

```
    return;
 }
 else if (strcmp(ptr,".outputs") == 0) {
    if (pass == 2) {
       io_line(RECEIVER, doall);
       *done = 1;
    }
    else {
       dum_parse(buffer);
       if (pass == 3 && doall) olines++;  /* Make sure only one .output line
*/
    }                              /* For error checking only */
    return;
 }
 else if (strcmp(ptr,".end") == 0) {
    if (pass == 3 && doall) endlines++;   /* Error checking only */
    return;
 }
/* Could have numbers following a .names command, so not matching any *
 * of the tokens above is not an error.                                */
//Everything else is assumed to be a part of Truth Table (numbers)
 else {
       if (doall && pass == 3)
       {
             add_truth_table(ptr,buffer);
             return;
       }
 }
}
static void dum_parse (char *buf) {
/* Continue parsing to the end of this (possibly continued) line. */
 while (my_strtok(NULL,TOKENS,blif,buf) != NULL)
       ;
}
static void add_lut (int doall, int lut_size) {
/* Adds a LUT (.names) currently being parsed to the block array.  Adds *
 * its pins to the nets data structure by calling add_net.  If doall is *
 * zero this is a counting pass; if it is 1 this is the final (loading) *
 * pass.                                                                */
 char *ptr, saved_names[MAXLUT+2][BUFSIZE], buf[BUFSIZE];
 int i, j, len;
 num_blocks++;
```

```
/* Count # nets connecting */
 i=0;
 while ((ptr = my_strtok(NULL,TOKENS,blif,buf)) != NULL)  {
    if (i == MAXLUT+1) {
       fprintf(stderr,"Error:  LUT #%d has %d inputs.  Increase MAXLUT or"
            " check the netlist, line %d.\n",num_blocks-1,i-1,linenum);
       exit(1);
    }
    strcpy (saved_names[i], ptr);
    i++;
 }
 if (!doall) {          /* Counting pass only ... */
    for (j=0;j<i;j++)
       add_net(saved_names[j],RECEIVER,num_blocks-1,doall);
    return;
 }
 block[num_blocks-1].num_nets = i;
 block[num_blocks-1].type = LUT;
 for (i=0;i<block[num_blocks-1].num_nets-1;i++)   /* Do inputs */
    block[num_blocks-1].nets[i+1] = add_net (saved_names[i],RECEIVER,
                                    num_blocks-1,doall);
 block[num_blocks-1].nets[0] = add_net (
       saved_names[block[num_blocks-1].num_nets-1], DRIVER,num_blocks-
1,doall);

 for (i=block[num_blocks-1].num_nets; i<lut_size+2; i++)
    block[num_blocks-1].nets[i] = OPEN;
 len = strlen (saved_names[block[num_blocks-1].num_nets-1]);
 block[num_blocks-1].name = (char *) my_malloc ((len+1) * sizeof (char));
 strcpy(block[num_blocks-1].name, saved_names[block[num_blocks-1].num_nets-
1]);
 num_luts++;
}
static void add_latch (int doall, int lut_size) {
/* Adds the flipflop (.latch) currently being parsed to the block array.  *
 * Adds its pins to the nets data structure by calling add_net.  If doall *
 * is zero this is a counting pass; if it is 1 this is the final          *
 * (loading) pass.  Blif format for a latch is:                           *
 * .latch <input> <output> <type (latch on)> <control (clock)> <init_val> *
 * The latch pins are in .nets 0 to 2 in the order: Q D CLOCK.            */
 char *ptr, buf[BUFSIZE], saved_names[6][BUFSIZE];
 int i, len;
 num_blocks++;
```

```
/* Count # parameters, making sure we don't go over 6 (avoids memory corr.)
*/
/* Note that we can't rely on the tokens being around unless we copy them.
*/
 for (i=0;i<6;i++) {
    ptr = my_strtok (NULL,TOKENS,blif,buf);
    if (ptr == NULL)
       break;
    strcpy (saved_names[i], ptr);
 }
 if (i != 5) {
    fprintf(stderr,"Error:  .latch does not have 5 parameters.\n"
            "check the netlist, line %d.\n",linenum);
    exit(1);
 }
 if (!doall) {   /* If only a counting pass ... */
    add_net(saved_names[0],RECEIVER,num_blocks-1,doall);  /* D */
    add_net(saved_names[1],DRIVER,num_blocks-1,doall);    /* Q */
    add_net(saved_names[3],RECEIVER,num_blocks-1,doall);  /* Clock */
    return;
 }
 block[num_blocks-1].num_nets = 3;
 block[num_blocks-1].type = LATCH;

 block[num_blocks-1].nets[0] = add_net(saved_names[1],DRIVER,num_blocks-1,
     doall);                                                        /* Q */
 block[num_blocks-1].nets[1] = add_net(saved_names[0],RECEIVER,num_blocks-1,
     doall);                                                        /* D */
 block[num_blocks-1].nets[lut_size+1] = add_net(saved_names[3],RECEIVER,
     num_blocks-1,doall);                                           /* Clock
*/
 for (i=2;i<lut_size+1;i++)
    block[num_blocks-1].nets[i] = OPEN;

 len = strlen (saved_names[1]);
 block[num_blocks-1].name = (char *) my_malloc ((len+1) * sizeof (char));
 strcpy(block[num_blocks-1].name,saved_names[1]);
 num_latches++;
}
static void io_line(int in_or_out, int doall) {
/* Adds an input or output block to the block data structures.           *
 * in_or_out:  DRIVER for input, RECEIVER for output.                    *
 * doall:  1 for final pass when structures are loaded.  0 for           *
```

```
 * first pass when hash table is built and pins, nets, etc. are counted. */
 char *ptr;
 char buf2[BUFSIZE];
 int nindex, len;

 while (1) {
    ptr = my_strtok(NULL,TOKENS,blif,buf2);
    if (ptr == NULL) return;
    num_blocks++;

    nindex = add_net(ptr,in_or_out,num_blocks-1,doall);
                /* zero offset indexing */
    if (!doall) continue;   /* Just counting things when doall == 0 */
    len = strlen (ptr);
    if (in_or_out == RECEIVER) {  /* output pads need out: prefix
                                     to make names unique from LUTs */
       block[num_blocks-1].name = (char *) my_malloc ((len+1+4) *
            sizeof (char));          /* Space for out: at start */
       strcpy(block[num_blocks-1].name,"out:");
       strcat(block[num_blocks-1].name,ptr);
    }
    else {
       block[num_blocks-1].name = (char *) my_malloc ((len+1) *
            sizeof (char));
       strcpy(block[num_blocks-1].name,ptr);
    }

    block[num_blocks-1].num_nets = 1;
    block[num_blocks-1].nets[0] = nindex;  /* Put in driver position for */
   /*  OUTPAD, since it has only one pin (even though it's a receiver */

    if (in_or_out == DRIVER) {             /* processing .inputs line */
       num_p_inputs++;
       block[num_blocks-1].type = INPAD;
    }
    else {                                 /* processing .outputs line */
       num_p_outputs++;
       block[num_blocks-1].type = OUTPAD;
    }
 }
}
static int add_net (char *ptr, int type, int bnum, int doall) {
```

```
/* This routine is given a net name in *ptr, either DRIVER or RECEIVER *
 * specifying whether the block number given by bnum is driving this   *
 * net or in the fan-out and doall, which is 0 for the counting pass   *
 * and 1 for the loading pass.  It updates the net data structure and  *
 * returns the net number so the calling routine can update the block  *
 * data structure.                                                     */
 struct hash_nets *h_ptr, *prev_ptr;
 int index, j, nindex;
 index = hash_value(ptr);
 h_ptr = hash[index];
 prev_ptr = h_ptr;
 while (h_ptr != NULL) {
    if (strcmp(h_ptr->name,ptr) == 0) { /* Net already in hash table */
       nindex = h_ptr->index;

       if (!doall) {   /* Counting pass only */
          (h_ptr->count)++;
          return (nindex);
       }
       net[nindex].num_pins++;
       if (type == DRIVER) {
          num_driver[nindex]++;
          j=0;           /* Driver always in position 0 of pinlist */
       }
       else {
          j = net[nindex].num_pins - num_driver[nindex];
   /* num_driver is the number of signal drivers of this net. *
    * should always be zero or 1 unless the netlist is bad.   */
          if (j >= temp_num_pins[nindex]) {
             printf("Error:  Net #%d (%s) has no driver and will cause\n",
                nindex, ptr);
             printf("memory corruption.\n");
             exit(1);
          }
       }
       net[nindex].pins[j] = bnum;
       return (nindex);
    }
    prev_ptr = h_ptr;
    h_ptr = h_ptr->next;
 }
 /* Net was not in the hash table. */
```

```
 if (doall == 1) {
    printf("Error in add_net:  the second (load) pass could not\n");
    printf("find net %s in the symbol table.\n", ptr);
    exit(1);
 }
/* Add the net (only counting pass will add nets to symbol table). */
 num_nets++;
 h_ptr = (struct hash_nets *) my_malloc (sizeof(struct hash_nets));
 if (prev_ptr == NULL) {
    hash[index] = h_ptr;
 }
 else {
    prev_ptr->next = h_ptr;
 }
 h_ptr->next = NULL;
 h_ptr->index = num_nets - 1;
 h_ptr->count = 1;
 h_ptr->name = (char *) my_malloc((strlen(ptr)+1)*sizeof(char));
 strcpy(h_ptr->name,ptr);
 return (h_ptr->index);
}
static int hash_value (char *name) {
 int i,k;
 int val=0, mult=1;
 i = strlen(name);
 k = max (i-7,0);
 for (i=strlen(name)-1;i>=k;i--) {
    val += mult*((int) name[i]);
    mult *= 10;
 }
 val += (int) name[0];
 val %= HASHSIZE;
 return(val);
}
void echo_input (char *blif_file, int lut_size, char *echo_file) {
/* Echo back the netlist data structures to file input.echo to *
 * allow the user to look at the internal state of the program *
 * and check the parsing.                                      */
 int i, j, max_pin;
 FILE *fp;
 printf("Input netlist file: %s  Model: %s\n", blif_file, model);
```

```
 printf("Primary Inputs: %d.  Primary Outputs: %d.\n", num_p_inputs,
num_p_outputs);
 printf("LUTs: %d.  Latches: %d.\n", num_luts, num_latches);
 printf("Total Blocks: %d.  Total Nets: %d\n", num_blocks, num_nets);
 fp = my_fopen (echo_file,"w",0);
 fprintf(fp,"Input netlist file: %s  Model: %s\n",blif_file,model);
 fprintf(fp,"num_p_inputs: %d, num_p_outputs: %d, num_luts: %d,"
            " num_latches:
%d\n",num_p_inputs,num_p_outputs,num_luts,num_latches);
 fprintf(fp,"num_blocks: %d, num_nets: %d\n",num_blocks,num_nets);

 fprintf(fp,"\nNet\tName\t\t#Pins\tDriver\tRecvs.\n");
 for (i=0;i<num_nets;i++) {
    fprintf(fp,"\n%d\t%s\t", i, net[i].name);
    if (strlen(net[i].name) < 8)
       fprintf(fp,"\t");         /* Name field is 16 chars wide */
    fprintf(fp,"%d", net[i].num_pins);
    for (j=0;j<net[i].num_pins;j++)
        fprintf(fp,"\t%d",net[i].pins[j]);
 }
 fprintf(fp,"\n\n\nBlocks\t\t\tBlock Type Legend:\n");
 fprintf(fp,"\t\t\tINPAD = %d\tOUTPAD = %d\n", INPAD, OUTPAD);
 fprintf(fp,"\t\t\tLUT = %d\t\tLATCH = %d\n", LUT, LATCH);
 fprintf(fp,"\t\t\tEMPTY = %d\tLUT_AND_LATCH = %d\n\n", EMPTY,
       LUT_AND_LATCH);
 fprintf(fp,"\nBlock\tName\t\tType\t#Nets\tOutput\tInputs");
 for (i=0;i<lut_size;i++)
    fprintf(fp,"\t");
 fprintf(fp,"Clock\n\n");
 for (i=0;i<num_blocks;i++) {
    fprintf(fp,"\n%d\t%s\t",i, block[i].name);
    if (strlen(block[i].name) < 8)
       fprintf(fp,"\t");         /* Name field is 16 chars wide */
    fprintf(fp,"%d\t%d", block[i].type, block[i].num_nets);
   /* I'm assuming EMPTY blocks are always INPADs when I print
*
    * them out.  This is true right after the netlist is read in, and again
*
    * after ff_packing and compression of the netlist.  It's not true after
*
    * ff_packing and before netlist compression.
*/
    if (block[i].type == INPAD || block[i].type == OUTPAD ||
          block[i].type == EMPTY)
```

```
      max_pin = 1;
   else
      max_pin = lut_size+2;
   for (j=0;j<max_pin;j++) {
       if (block[i].nets[j] == OPEN)
          fprintf(fp,"\tOPEN");
       else
          fprintf(fp,"\t%d",block[i].nets[j]);
   }
 }
 fprintf(fp,"\n");
 fclose(fp);
}
static void check_net (int lut_size) {
/* Checks the input netlist for obvious errors. */
 int i, error, iblk;
 error = 0;
 if (ilines != 1) {
     printf("Warning:  found %d .inputs lines; expected 1.\n",
        ilines);
     error++;
 }
 if (olines != 1) {
     printf("Warning:  found %d .outputs lines; expected 1.\n",
        olines);
     error++;
 }
 if (model_lines != 1) {
     printf("Warning:  found %d .model lines; expected 1.\n",
        model_lines);
     error++;
 }
 if (endlines != 1) {
     printf("Warning:  found %d .end lines; expected 1.\n",
        endlines);
     error++;
 }
 for (i=0;i<num_nets;i++) {
    if (num_driver[i] != 1) {
       printf ("Warning:  net %s has"
         " %d signals driving it.\n",net[i].name,num_driver[i]);
       error++;
```

```
    }
    if ((net[i].num_pins - num_driver[i]) < 1) {
/* If this is an input pad, it is unused and I just remove it with  *
 * a warning message.  Lots of the mcnc circuits have this problem. */

       iblk = net[i].pins[0];
       if (block[iblk].type == INPAD) {
          printf("Warning:  Input %s is unused; removing it.\n",
             block[iblk].name);
          net[i].pins[0] = OPEN;
          block[iblk].type = EMPTY;
       }

       else {
          printf("Warning:  net %s has no fanout.\n",net[i].name);
          error++;
       }
    }
    if (strcmp(net[i].name, "open") == 0) {
       printf("Warning:  net #%d has the reserved name %s.\n",i,net[i].name);
       error++;
    }
 }
 for (i=0;i<num_blocks;i++) {
    if (block[i].type == LUT) {
       if (block[i].num_nets < 2) {
          printf("Warning:  logic block #%d with output %s has only %d
pin.\n",
             i,block[i].name,block[i].num_nets);
/* LUTs with 1 pin (an output)  can be a constant generator.  Warn the   *
 * user, but don't exit.                                                 */
          if (block[i].num_nets != 1) {
             error++;
          }
          else {
             printf("\tPin is an output -- may be a constant generator.\n");
             printf("\tNon-fatal error.\n");
          }
       }
       if (block[i].num_nets > lut_size + 1) {
          printf("Warning:  logic block #%d with output %s has %d pins.\n",
             i,block[i].name,block[i].num_nets);
```

```
            error++;
         }
      }
      else if (block[i].type == LATCH) {
         if (block[i].num_nets != 3) {
            printf("Warning:  Latch #%d with output %s has %d pin(s).\n",
               i, block[i].name, block[i].num_nets);
            error++;
         }
      }

      else {
         if (block[i].num_nets != 1) {
            printf("Warning:  io block #%d with output %s of type %d"
               "has %d pins.\n", i, block[i].name, block[i].type,
               block[i].num_nets);
            error++;
         }
      }
   }
   if (error != 0) {
      printf("Found %d fatal errors in the input netlist.\n",error);
      exit(1);
   }
}
static void free_parse (void) {
/* Release memory needed only during blif network parsing. */
   int i;
   struct hash_nets *h_ptr, *temp_ptr;
   for (i=0;i<HASHSIZE;i++) {
      h_ptr = hash[i];
      while (h_ptr != NULL) {
         free ((void *) h_ptr->name);
         temp_ptr = h_ptr->next;
         free ((void *) h_ptr);
         h_ptr = temp_ptr;
      }
   }
   free ((void *) num_driver);
   free ((void *) hash);
   free ((void *) temp_num_pins);
}
```

```
//Adding Truth Table Function
static void add_truth_table(char *input,char *buffer)
{
	char *output;
	if ((output = my_strtok(NULL,TOKENS,blif,buffer)) == NULL) {
		output = input;
		if(strcmp(output,"1") == 0)
			fill_table(1);    //Char to Int
		else if(strcmp(output,"0") == 0)
			fill_table(0);
	}else //If there are more than one values in truth table
	{
		//Decode charaters (Binary) into dec address(array Index)
		int address[16];
		int i;
		for (i = 0; i < 16; i++)
			address[i] = 0;
		decode_address(input,address);
		for (i = 0; i < 16; i++)
		{
			if (address[i] == 1)
			{
				if (strcmp(output,"1") == 0)
					block[num_blocks-1].truth_table[i] = 1;
				else if(strcmp(output,"0") == 0)
					block[num_blocks-1].truth_table[i] = 0;
			}
		}
	}
	return;
}
//Fill the whole truth table with input
static void fill_table(int value)
{
	int n = 0;
	for (n = 0; n < 16; n++)
		block[num_blocks-1].truth_table[n] = value;
	return;
}
//Decode Address Function
static void decode_address(char *input, int *address)
{
```

```
if (address_valid(input) == 1)
      address[bin2dec(atoi(input))] = 1;
else if ((strcmp(input,"-1")) == 0)
{
      address[bin2dec(1)] = 1;
      address[bin2dec(11)] = 1;
}
else if ((strcmp(input,"1-")) == 0)
{
      address[bin2dec(10)] = 1;
      address[bin2dec(11)] = 1;
}
else if ((strcmp(input,"--1")) == 0)
{
      address[bin2dec(1)] = 1;
      address[bin2dec(11)] = 1;
      address[bin2dec(101)] = 1;
      address[bin2dec(111)] = 1;
}
else if ((strcmp(input,"-1-")) == 0)
{
      address[bin2dec(10)] = 1;
      address[bin2dec(11)] = 1;
      address[bin2dec(110)] = 1;
      address[bin2dec(111)] = 1;
}
else if ((strcmp(input,"1--")) == 0)
{
      address[bin2dec(100)] = 1;
      address[bin2dec(101)] = 1;
      address[bin2dec(110)] = 1;
      address[bin2dec(111)] = 1;
}
else if ((strcmp(input,"---1")) == 0)
{
      address[bin2dec(1)] = 1;
      address[bin2dec(11)] = 1;
      address[bin2dec(101)] = 1;
      address[bin2dec(111)] = 1;
      address[bin2dec(1001)] = 1;
      address[bin2dec(1011)] = 1;
      address[bin2dec(1101)] = 1;
```

```
      address[bin2dec(1111)] = 1;
}
else if ((strcmp(input,"--1-"))     == 0)
{
      address[bin2dec(10)] = 1;
      address[bin2dec(11)] = 1;
      address[bin2dec(110)] = 1;
      address[bin2dec(111)] = 1;
      address[bin2dec(1010)] = 1;
      address[bin2dec(1011)] = 1;
      address[bin2dec(1110)] = 1;
      address[bin2dec(1111)] = 1;
}
else if ((strcmp(input,"-1--"))     == 0)
{
      address[bin2dec(100)] = 1;
      address[bin2dec(101)] = 1;
      address[bin2dec(110)] = 1;
      address[bin2dec(111)] = 1;
      address[bin2dec(1100)] = 1;
      address[bin2dec(1101)] = 1;
      address[bin2dec(1110)] = 1;
      address[bin2dec(1111)] = 1;
}
else if ((strcmp(input,"1---"))     == 0)
{
      address[bin2dec(1000)] = 1;
      address[bin2dec(1001)] = 1;
      address[bin2dec(1010)] = 1;
      address[bin2dec(1011)] = 1;
      address[bin2dec(1100)] = 1;
      address[bin2dec(1101)] = 1;
      address[bin2dec(1110)] = 1;
      address[bin2dec(1111)] = 1;
}
else if ((strcmp(input,"--11")) == 0)
{
      address[bin2dec(11)] = 1;
      address[bin2dec(111)] = 1;
      address[bin2dec(1011)] = 1;
      address[bin2dec(1111)] = 1;
}
```

```
else if ((strcmp(input,"-1-1")) == 0)
{
      address[bin2dec(101)] = 1;
      address[bin2dec(111)] = 1;
      address[bin2dec(1101)] = 1;
      address[bin2dec(1111)] = 1;
}
else if ((strcmp(input,"1--1")) == 0)
{
      address[bin2dec(1001)] = 1;
      address[bin2dec(1011)] = 1;
      address[bin2dec(1101)] = 1;
      address[bin2dec(1111)] = 1;
}
else if ((strcmp(input,"1-1-")) == 0)
{
      address[bin2dec(1010)] = 1;
      address[bin2dec(1011)] = 1;
      address[bin2dec(1110)] = 1;
      address[bin2dec(1111)] = 1;
}
else if ((strcmp(input,"11--")) == 0)
{
      address[bin2dec(1100)] = 1;
      address[bin2dec(1101)] = 1;
      address[bin2dec(1110)] = 1;
      address[bin2dec(1111)] = 1;
}
else if ((strcmp(input,"-111")) == 0)
{
      address[bin2dec(111)] = 1;
      address[bin2dec(1111)] = 1;
}
else if ((strcmp(input,"1-11")) == 0)
{
      address[bin2dec(1011)] = 1;
      address[bin2dec(1111)] = 1;
}
else if ((strcmp(input,"11-1")) == 0)
{
      address[bin2dec(1101)] = 1;
      address[bin2dec(1111)] = 1;
```

```
	}
	else if ((strcmp(input,"111-")) == 0)
		address[bin2dec(1111)] = 1;
}
//Check if the address doesn't contain "-"(don't care)
static int address_valid(char *input)
{
	while (*input != '\0')
	{
		if (*input == '-')
			return 0;
		else
			input++;
	}
	return 1;
}
```

util.c:

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "util.h"
/* This file contains utility functions widely used in *
 * my programs.  Many are simply versions of file and  *
 * memory grabbing routines that take the same         *
 * arguments as the standard library ones, but exit    *
 * the program if they find an error condition.        */
int linenum;  /* Line in file being parsed. */
FILE *my_fopen (char *fname, char *flag, int prompt) {
 FILE *fp;   /* prompt = 1: prompt user.  prompt=0: use fname */
 while (1) {
    if (prompt)
       scanf("%s",fname);
    if ((fp = fopen(fname,flag)) != NULL)
       break;
    printf("Error opening file %s for %s access.\n",fname,flag);
    if (!prompt)
       exit(1);
    printf("Please enter another filename.\n");
 }
 return (fp);
}
int my_atoi (const char *str) {
/* Returns the integer represented by the first part of the character       *
 * string.  Unlike the normal atoi, I return -1 if the string doesn't       *
 * start with a numeric digit.
*/
 if (str[0] < '0' || str[0] > '9')
    return (-1);
 return (atoi(str));
}
void *my_calloc (size_t nelem, size_t size) {
 void *ret;
 if ((ret = calloc (nelem,size)) == NULL) {
    fprintf(stderr,"Error:  Unable to calloc memory.  Aborting.\n");
    exit (1);
 }
 return (ret);
```

```
}
void *my_malloc (size_t size) {
 void *ret;
 if ((ret = malloc (size)) == NULL) {
    fprintf(stderr,"Error:  Unable to malloc memory.  Aborting.\n");
    abort ();
    exit (1);
 }
 return (ret);
}
void *my_realloc (void *ptr, size_t size) {
 void *ret;
 if ((ret = realloc (ptr,size)) == NULL) {
    fprintf(stderr,"Error:  Unable to realloc memory.  Aborting.\n");
    exit (1);
 }
 return (ret);
}
void *my_chunk_malloc (size_t size, struct s_linked_vptr **chunk_ptr_head,
       int *mem_avail_ptr, char **next_mem_loc_ptr) {
/* This routine should be used for allocating fairly small data             *
 * structures where memory-efficiency is crucial.  This routine allocates   *
 * large "chunks" of data, and parcels them out as requested.  Whenever     *
 * it mallocs a new chunk it adds it to the linked list pointed to by       *
 * chunk_ptr_head.  This list can be used to free the chunked memory.       *
 * If chunk_ptr_head is NULL, no list of chunked memory blocks will be kept *
 * -- this is useful for data structures that you never intend to free as   *
 * it means you don't have to keep track of the linked lists.               *
 * Information about the currently open "chunk" must be stored by the       *
 * user program.  mem_avail_ptr points to an int storing how many bytes are *
 * left in the current chunk, while next_mem_loc_ptr is the address of a    *
 * pointer to the next free bytes in the chunk.  To start a new chunk,      *
 * simply set *mem_avail_ptr = 0.  Each independent set of data structures  *
 * should use a new chunk.
*/
/* To make sure the memory passed back is properly aligned, I must *
 * only send back chunks in multiples of the worst-case alignment  *
 * restriction of the machine.  On most machines this should be    *
 * a long, but on 64-bit machines it might be a long long or a     *
 * double.  Change the typedef below if this is the case.          */
 typedef long Align;
#define CHUNK_SIZE 32768
```

```
#define FRAGMENT_THRESHOLD 100
 char *tmp_ptr;
 int aligned_size;
 if (*mem_avail_ptr < size) {       /* Need to malloc more memory. */
    if (size > CHUNK_SIZE) {      /* Too big, use standard routine. */
       tmp_ptr = my_malloc (size);
/*#ifdef DEBUG
       printf("NB:  my_chunk_malloc got a request for %d bytes.\n",
          size);
       printf("You should consider using my_malloc for such big
requests.\n");
#endif */
       if (chunk_ptr_head != NULL)
          *chunk_ptr_head = insert_in_vptr_list (*chunk_ptr_head, tmp_ptr);
       return (tmp_ptr);
    }
    if (*mem_avail_ptr < FRAGMENT_THRESHOLD) {  /* Only a small scrap left.
*/
       *next_mem_loc_ptr = my_malloc (CHUNK_SIZE);
       *mem_avail_ptr = CHUNK_SIZE;
       if (chunk_ptr_head != NULL)
          *chunk_ptr_head = insert_in_vptr_list (*chunk_ptr_head,
                            *next_mem_loc_ptr);
    }
/* Execute else clause only when the chunk we want is pretty big,  *
 * and would leave too big an unused fragment.  Then we use malloc *
 * to allocate normally.                                           */
    else {
       tmp_ptr = my_malloc (size);
       if (chunk_ptr_head != NULL)
          *chunk_ptr_head = insert_in_vptr_list (*chunk_ptr_head, tmp_ptr);
       return (tmp_ptr);
    }
 }
/* Find the smallest distance to advance the memory pointer and keep *
 * everything aligned.                                               */
 if (size % sizeof (Align) == 0) {
    aligned_size = size;
 }
 else {
    aligned_size = size + sizeof(Align) - size % sizeof(Align);
 }
 tmp_ptr = *next_mem_loc_ptr;
```

```
 *next_mem_loc_ptr += aligned_size;
 *mem_avail_ptr -= aligned_size;
 return (tmp_ptr);
}
void free_chunk_memory (struct s_linked_vptr *chunk_ptr_head) {
/* Frees the memory allocated by a sequence of calls to my_chunk_malloc. */
 struct s_linked_vptr *curr_ptr, *prev_ptr;
 curr_ptr = chunk_ptr_head;
 while (curr_ptr != NULL) {
    free (curr_ptr->data_vptr);   /* Free memory "chunk". */
    prev_ptr = curr_ptr;
    curr_ptr = curr_ptr->next;
    free (prev_ptr);              /* Free memory used to track "chunk". */
 }
}
struct s_linked_vptr *insert_in_vptr_list (struct s_linked_vptr *head,void
*vptr_to_add) {
/* Inserts a new element at the head of a linked list of void pointers. *
 * Returns the new head of the list.                                     */
 struct s_linked_vptr *linked_vptr;
 linked_vptr = (struct s_linked_vptr *) my_malloc (sizeof(struct
                 s_linked_vptr));
 linked_vptr->data_vptr = vptr_to_add;
 linked_vptr->next = head;
 return (linked_vptr);     /* New head of the list */
}
t_linked_int *insert_in_int_list (t_linked_int *head, int data, t_linked_int
**
      free_list_head_ptr) {
/* Inserts a new element at the head of a linked list of integers.  Returns
*
 * the new head of the list.  One argument is the address of the head of
*
 * a list of free ilist elements.  If there are any elements on this free
*
 * list, the new element is taken from it.  Otherwise a new one is malloced.
*/
 t_linked_int *linked_int;
 if (*free_list_head_ptr != NULL) {
    linked_int = *free_list_head_ptr;
    *free_list_head_ptr = linked_int->next;
 }
 else {
    linked_int = (t_linked_int *) my_malloc (sizeof (t_linked_int));
```

```
 }
 linked_int->data = data;
 linked_int->next = head;
 return (linked_int);
}
void free_int_list (t_linked_int **int_list_head_ptr) {
/* This routine truly frees (calls free) all the integer list elements    *
 * on the linked list pointed to by *head, and sets head = NULL.          */
 t_linked_int *linked_int, *next_linked_int;
 linked_int = *int_list_head_ptr;
 while (linked_int != NULL) {
    next_linked_int = linked_int->next;
    free (linked_int);
    linked_int = next_linked_int;
 }
 *int_list_head_ptr = NULL;
}
void alloc_ivector_and_copy_int_list (t_linked_int **list_head_ptr,
            int num_items, struct s_ivec *ivec, t_linked_int
            **free_list_head_ptr) {
/* Allocates an integer vector with num_items elements and copies the       *
 * integers from the list pointed to by list_head (of which there must be   *
 * num_items) over to it.  The int_list is then put on the free list, and   *
 * the list_head_ptr is set to NULL.
*/
 t_linked_int *linked_int, *list_head;
 int i, *list;
 list_head = *list_head_ptr;
 if (num_items == 0) {    /* Empty list. */
    ivec->nelem = 0;
    ivec->list = NULL;
    if (list_head != NULL) {
       printf ("Error in alloc_ivector_and_copy_int_list:\n Copied %d "
           "elements, but list at %p contains more.\n", num_items,
list_head);
       exit (1);
    }
    return;
 }
 ivec->nelem = num_items;
 list = (int *) my_malloc (num_items * sizeof (int));
 ivec->list = list;
 linked_int = list_head;
```

```
 for (i=0;i<num_items-1;i++) {
    list[i] = linked_int->data;
    linked_int = linked_int->next;
 }
 list[num_items-1] = linked_int->data;
 if (linked_int->next != NULL) {
    printf ("Error in alloc_ivector_and_copy_int_list:\n Copied %d elements,
"
            "but list at %p contains more.\n", num_items, list_head);
    exit (1);
 }
 linked_int->next = *free_list_head_ptr;
 *free_list_head_ptr = list_head;
 *list_head_ptr = NULL;
}
static int cont;  /* line continued? */
char *my_fgets(char *buf, int max_size, FILE *fp) {
 /* Get an input line, update the line number and cut off *
  * any comment part.  A \ at the end of a line with no   *
  * comment part (#) means continue.                      */
 char *val;
 int i;
 cont = 0;
 val = fgets(buf,max_size,fp);
 linenum++;
 if (val == NULL) return(val);

/* Check that line completely fit into buffer.  (Flags long line   *
 * truncation).                                                    */
 for (i=0;i<max_size;i++) {
    if (buf[i] == '\n')
       break;
    if (buf[i] == '\0') {
       printf("Error on line %d -- line is too long for input buffer.\n",
          linenum);
       printf("All lines must be at most %d characters long.\n",BUFSIZE-2);
       printf("The problem could also be caused by a missing newline.\n");
       exit (1);
    }
 }
 for (i=0;i<max_size && buf[i] != '\0';i++) {
    if (buf[i] == '#') {
```

```
        buf[i] = '\0';
        break;
    }
 }
 if (i<2) return (val);
 if (buf[i-1] == '\n' && buf[i-2] == '\\') {
    cont = 1;   /* line continued */
    buf[i-2] = '\n';  /* May need this for tokens */
    buf[i-1] = '\0';
 }
 return(val);
}
char *my_strtok(char *ptr, char *tokens, FILE *fp, char *buf) {
/* Get next token, and wrap to next line if \ at end of line.    *
 * There is a bit of a "gotcha" in strtok.  It does not make a   *
 * copy of the character array which you pass by pointer on the  *
 * first call.  Thus, you must make sure this array exists for   *
 * as long as you are using strtok to parse that line.  Don't    *
 * use local buffers in a bunch of subroutines calling each      *
 * other; the local buffer may be overwritten when the stack is  *
 * restored after return from the subroutine.                    */

 char *val;
 val = strtok(ptr,tokens);
 while (1) {
    if (val != NULL || cont == 0) return(val);
   /* return unless we have a null value and a continuation line */
    if (my_fgets(buf,BUFSIZE,fp) == NULL)
       return(NULL);
    val = strtok(buf,tokens);
 }
}
void free_ivec_vector (struct s_ivec *ivec_vector, int nrmin, int nrmax) {
/* Frees a 1D array of integer vectors.                              */
 int i;
 for (i=nrmin;i<=nrmax;i++)
    if (ivec_vector[i].nelem != 0)
       free (ivec_vector[i].list);
 free (ivec_vector + nrmin);
}
void free_ivec_matrix (struct s_ivec **ivec_matrix, int nrmin, int nrmax,
       int ncmin, int ncmax) {
```

```
/* Frees a 2D matrix of integer vectors (ivecs).                    */
 int i, j;
 for (i=nrmin;i<=nrmax;i++) {
    for (j=ncmin;j<=ncmax;j++) {
       if (ivec_matrix[i][j].nelem != 0) {
          free (ivec_matrix[i][j].list);
       }
    }
 }
 free_matrix (ivec_matrix, nrmin, nrmax, ncmin, sizeof (struct s_ivec));
}
void free_ivec_matrix3 (struct s_ivec ***ivec_matrix3, int nrmin, int nrmax,
       int ncmin, int ncmax, int ndmin, int ndmax) {
/* Frees a 3D matrix of integer vectors (ivecs).                    */
 int i, j, k;
 for (i=nrmin;i<=nrmax;i++) {
    for (j=ncmin;j<=ncmax;j++) {
       for (k=ndmin;k<=ndmax;k++) {
          if (ivec_matrix3[i][j][k].nelem != 0) {
             free (ivec_matrix3[i][j][k].list);
          }
       }
    }
 }
 free_matrix3 (ivec_matrix3, nrmin, nrmax, ncmin, ncmax, ndmin,
               sizeof (struct s_ivec));
}
void **alloc_matrix (int nrmin, int nrmax, int ncmin, int ncmax,
   size_t elsize) {
/* allocates an generic matrix with nrmax-nrmin + 1 rows and ncmax - *
 * ncmin + 1 columns, with each element of size elsize. i.e.        *
 * returns a pointer to a storage block [nrmin..nrmax][ncmin..ncmax].*
 * Simply cast the returned array pointer to the proper type.       */
 int i;
 char **cptr;
 cptr = (char **) my_malloc ((nrmax - nrmin + 1) * sizeof (char *));
 cptr -= nrmin;
 for (i=nrmin;i<=nrmax;i++) {
    cptr[i] = (char *) my_malloc ((ncmax - ncmin + 1) * elsize);
    cptr[i] -= ncmin * elsize / sizeof(char);  /* sizeof(char) = 1 */
 }
 return ((void **) cptr);
```

```
}
/* NB:  need to make the pointer type void * instead of void ** to allow   *
 * any pointer to be passed in without a cast.                              */
void free_matrix (void *vptr, int nrmin, int nrmax, int ncmin, size_t elsize)
{
 int i;
 char **cptr;
 cptr = (char **) vptr;
 for (i=nrmin;i<=nrmax;i++)
    free (cptr[i] + ncmin * elsize / sizeof (char));
 free (cptr + nrmin);
}
void ***alloc_matrix3 (int nrmin, int nrmax, int ncmin, int ncmax,
      int ndmin, int ndmax, size_t elsize) {
/* allocates a 3D generic matrix with nrmax-nrmin + 1 rows, ncmax -  *
 * ncmin + 1 columns, and a depth of ndmax-ndmin + 1, with each      *
 * element of size elsize. i.e. returns a pointer to a storage block *
 * [nrmin..nrmax][ncmin..ncmax][ndmin..ndmax].  Simply cast the      *
 *  returned array pointer to the proper type.                       */
 int i, j;
 char ***cptr;
 cptr = (char ***) my_malloc ((nrmax - nrmin + 1) * sizeof (char **));
 cptr -= nrmin;
 for (i=nrmin;i<=nrmax;i++) {
    cptr[i] = (char **) my_malloc ((ncmax - ncmin + 1) * sizeof (char *));
    cptr[i] -= ncmin;
    for (j=ncmin;j<=ncmax;j++) {
       cptr[i][j] = (char *) my_malloc ((ndmax - ndmin + 1) * elsize);
       cptr[i][j] -= ndmin * elsize / sizeof(char); /* sizeof(char) = 1) */
    }
 }
 return ((void ***) cptr);
}
void free_matrix3 (void *vptr, int nrmin, int nrmax, int ncmin, int ncmax,
        int ndmin, size_t elsize) {
 int i, j;
 char ***cptr;
 cptr = (char ***) vptr;
 for (i=nrmin;i<=nrmax;i++) {
    for (j=ncmin;j<=ncmax;j++)
       free (cptr[i][j] + ndmin * elsize / sizeof (char));
    free (cptr[i] + ncmin);
```

```
 }
 free (cptr + nrmin);
}
/* Portable random number generator defined below.  Taken from ANSI C by  *
 * K & R.  Not a great generator, but fast, and good enough for my needs. */
#define IA 1103515245u
#define IC 12345u
#define IM 2147483648u
#define CHECK_RAND
static unsigned int current_random = 0;
void my_srandom (int seed) {
 current_random = (unsigned int) seed;
}
int my_irand (int imax) {
/* Creates a random integer between 0 and imax, inclusive.  i.e. [0..imax] */
 int ival;
/* current_random = (current_random * IA + IC) % IM; */
 current_random = current_random * IA + IC;  /* Use overflow to wrap */
 ival = current_random & (IM - 1);  /* Modulus */
 ival = (int) ((float) ival * (float) (imax + 0.999) / (float) IM);
#ifdef CHECK_RAND
 if ((ival < 0) || (ival > imax)) {
    printf("Bad value in my_irand, imax = %d  ival = %d\n",imax,ival);
    exit(1);
 }
#endif
 return(ival);
}
float my_frand (void) {
/* Creates a random float between 0 and 1.  i.e. [0..1).        */
 float fval;
 int ival;
 current_random = current_random * IA + IC;  /* Use overflow to wrap */
 ival = current_random & (IM - 1);  /* Modulus */
 fval = (float) ival / (float) IM;
#ifdef CHECK_RAND
 if ((fval < 0) || (fval > 1.)) {
    printf("Bad value in my_frand, fval = %g\n",fval);
    exit(1);
 }
#endif
 return(fval);
```

```
}
//Binary to Decimal Conversion
int bin2dec(int n)
{
	int dec = 0, i = 0, rem;
	while (n != 0)
	{
		rem = n % 10;
		n /= 10;
		dec += rem*pow(2,i++);
	}
	if (dec >=0 && dec <= 15)
		return dec;
	else return -1;
}
int measure_data_mem(struct s_node* head, int num_nodes)
{
	int x = 1;
	//moving the current pointer to the last block
	struct s_node* n_ptr = head;
	while (n_ptr->n_node != NULL)
	{
		n_ptr = n_ptr->n_node;
		x++;
	}
	printf("num_nodes: %d, x: %d\n",num_nodes,x);
	int life = num_nodes;
	int i = 0;
	//Fill all the nodes with their life times
	while (n_ptr->p_node != NULL)
	{
		for (i = 0; i < 4; i++)
		{
			if (n_ptr->inputs[i] != NULL)
			{
				if ((n_ptr->inputs[i])->life_time < life)
				{
					(n_ptr->inputs[i])->life_time = life;
				}
			}
		}
		n_ptr = n_ptr->p_node;
```

```
	life--;
}
//Now calculate data memory by counting the maximum number
//of variables alive at any perticular time durin simulation
n_ptr = head;
life = 1;
struct l_node* l_head = NULL;
struct l_node* l_tail = NULL;
struct l_node* l_ptr = NULL;
int data_mem = 0;
while (n_ptr != NULL)
{
	if (n_ptr->life_time > 0 && n_ptr->life_time > life)
	{
		l_ptr = (struct l_node*)malloc(sizeof(struct l_node));
		l_ptr->num = n_ptr->life_time;
		l_ptr->p_node = l_tail;
		l_ptr->n_node = NULL;

		if (l_head == NULL)
		{
			l_head = l_ptr;
			l_tail = l_ptr;
		}else
		{
			l_tail->n_node = l_ptr;
			l_tail = l_tail->n_node;
		}
	}
	//Cleaning the list
	l_ptr = l_head;
	while (l_ptr != NULL)
	{
		if (l_ptr->num <= life)
		{
			if (l_ptr == l_head && l_ptr == l_tail)
			{
				l_ptr = NULL;
				l_head = NULL;
				l_tail = NULL;
			}else if (l_ptr == l_head && l_ptr != l_tail)
			{
```

```
                        l_ptr = l_ptr->n_node;
                        l_head = l_ptr;
                        free(l_ptr->p_node);
                        l_ptr->p_node = NULL;
                  }else if (l_ptr != l_head && l_ptr == l_tail)
                  {
                        l_ptr = l_ptr->p_node;
                        l_tail = l_ptr;
                        free(l_ptr->n_node);
                        l_ptr->n_node = NULL;
                  }else if (l_ptr != l_head && l_ptr != l_tail)
                  {
                        (l_ptr->p_node)->n_node = l_ptr->n_node;
                        (l_ptr->n_node)->p_node = l_ptr->p_node;
                        struct l_node* tmp = l_ptr;
                        l_ptr = l_ptr->n_node;
                        free(tmp);
                  }

            }else
                  l_ptr = l_ptr->n_node;
      }
      life++;
      int tmp_num = measure_size(l_head);
      if (tmp_num >= data_mem)
            data_mem = tmp_num;

      n_ptr = n_ptr->n_node;
  }
  printf("Data Mem size: %d\n",data_mem);
  return data_mem;
}
int measure_size(struct l_node* head)
{
  int size = 0;
  while(head != NULL)
  {
      size++;
      head = head->n_node;
  }
  return size;
}
```

graph.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "vpack.h"
//List of Functions
struct s_node* generate_graph(struct s_block*, struct s_net*, int num_blocks,
int num_p_outputs, int graph_size);
struct s_node* generate_schedule(struct s_node*, struct s_block* blocks,
struct s_net* nets);
void print_schedule(struct s_node*,char*,struct s_net*);
void push(struct s_node**,struct s_node**, struct s_node*);
struct s_node* pop(struct s_node**,struct s_node**);
int search(struct s_node*,struct s_node*);
int empty(struct s_node*, struct s_node*);      //returns TRUE if queue empty
void cp_node_block(struct s_node*,struct s_block*,struct s_node*,struct
s_node*,struct s_node*,int,int,int);
void cp_node_node(struct s_node*, struct s_node*);    //output, input
void add_node(struct s_node*, struct s_node*, struct s_node*);  //node to be
added, head of list and position in array
void swap_node(struct s_node*, struct s_node*);
struct s_node* find_position(struct s_node*, struct s_node*, int graph_size);
//returns the position of node in main array
void print_graph(struct s_node*,int,int,char*);
void print_node_to_file(FILE*, struct s_node*, int);
void count_inputs(struct s_node*,int);
void free_graph(struct s_node*,int);
void free_list(struct s_node*);
//Searches the list, input arguments: Head Pointer to list, Block Name;
Returns a pointer to the node
struct s_node* search_list(struct s_node*, char*);
//Generates a graph and return root
struct s_node* generate_graph(struct s_block* blocks, struct s_net* nets, int
num_blocks, int num_p_outputs, int graph_size)
{
      struct s_block* my_blocks = blocks;
      }


      printf("Everything Copied to Parent\n");fflush(stdout);
      //sorting array, moving LATCH and Primary INPAD up in the system
      for (i = 0; i < (graph_size-2); i++)
      {
            ptr = parent+1;
```

```
            struct s_node* next_ptr = parent+2;
            int j = 0;
            for (j = 0; j < (graph_size-2); j++)
            {
                  if (((next_ptr->block)->type == INPAD || (next_ptr->block)-
>type == LATCH))
                        {
                              if ((ptr->block)->type != INPAD && (ptr-
>block)->type != LATCH)
                              {
                                    swap_node(ptr,next_ptr);
                              }
                        }
                  ptr++, next_ptr++;
            }
      }
      printf("Everthing Sorted and LUT and INPAD are on TOP
now\n");fflush(stdout);
      //adding root node
      ptr = parent + 1;
      root->block = NULL; root->array_position = root; root->n_node = NULL;
root->p_node = NULL;
            root->num_inputs = 0; root->scheduled = 0; root->visited = 0;
      printf("\nparent:%p,ptr:%p,end:%p,graph_size:%d,parent+graph_size:%p",p
arent,ptr,end,graph_size,parent+graph_size);fflush(stdout);
      i = 0;
      while (ptr != end)
      {
            if ((ptr->block)->type == INPAD || (ptr->block)->type == LATCH)
            {
                  struct s_node* tmp = (struct s_node*)malloc(sizeof(struct
s_node));
                  cp_node_node(tmp,ptr);
                  add_node(tmp,root,ptr);
            }
            ptr++;
      }
      printf("Rooted Added Successfully\n");fflush(stdout);
      //creating rest of graph
      ptr = root + 1;
      int j = 0;
      for (j = 1; j < graph_size; j++)
      {
```

```
            int num_blks = nets[(ptr->block)->nets[0]].num_pins;
      //number of blocks connected to this block output
      for (i = 1; i < num_blks; i++)
            {
                  if ((blocks[nets[(ptr->block)->nets[0]].pins[i]]).type !=
OUTPAD)
                  {
                        //create an empty new node
                        struct s_node* tmp_node = (struct
s_node*)malloc(sizeof(struct s_node));
                        tmp_node->block = &blocks[nets[(ptr->block)-
>nets[0]].pins[i]];
                        ptr->p_node = NULL;
                        while(ptr->n_node != NULL)
                              ptr = ptr->n_node;
                        ptr->n_node = tmp_node;
                        tmp_node->p_node = ptr;
                        tmp_node->n_node = NULL;
                        tmp_node->array_position =
find_position(parent,tmp_node,graph_size);
                        tmp_node->scheduled = 0;
                        tmp_node->visited = 0;
                        tmp_node->num_inputs = 0;
                        while(ptr->p_node != NULL)
                              ptr = ptr->p_node;
//generate schedule and return head pointer for the list
struct s_node* generate_schedule(struct s_node* root, struct s_block* blocks,
struct s_net* nets)
{
      //creating a queue and schedule list
      struct s_node* q_head = NULL;
      struct s_node* q_tail = NULL;
      struct s_node* v_head = NULL;
      struct s_node* v_tail = NULL;
      struct s_node* ptr = NULL;
      struct s_node* q_tmp = NULL;
      push(&q_head,&q_tail,root);
      (root->array_position)->scheduled = 1;
      while(empty(q_head,q_tail) != 1)
      {
            q_tmp = pop(&q_head,&q_tail);
            (q_tmp->array_position)->visited++;
            if (q_tmp->block != NULL)
            {
```

```
                  if(((q_tmp->block)->type == INPAD) || (q_tmp->block)->type
== LATCH)
                  {
                        (q_tmp->array_position)->visited = 0;
                        (q_tmp->array_position)->num_inputs = 0;
                  }
            }
            ptr = (q_tmp->array_position)->n_node;
            if ((q_tmp->array_position)->scheduled == 0 && ((q_tmp-
>array_position)->visited >= ((q_tmp->array_position)->num_inputs)))
            {
                  push(&v_head,&v_tail,q_tmp);
                  (q_tmp->array_position)->scheduled = 1;
            }
            while(ptr != NULL)
            {
                  if ((ptr->block)->type == LATCH)
                  {
                        if ((ptr->array_position)->num_inputs > (ptr-
>array_position)->visited)
                        {
                              push(&q_head,&q_tail,ptr);
                        }
                  }else
                  {
                        if ((ptr->array_position)->num_inputs >= (ptr-
>array_position)->visited)
                        {
                              push(&q_head,&q_tail,ptr);
                        }
                  }
                  ptr = ptr->n_node;
            }
      }
      //Fill all list nodes with its inputs
      ptr = v_head;
      int i = 0;
      while (ptr != NULL)
      {
            for (i = 0; i < MAXLUT; i++)
                  ptr->inputs[i] = NULL;
            for (i = 0; i < ((ptr->block)->num_nets)-1; i++)
            {
```

```
                  if ((ptr->block)->type != INPAD)
                        ptr->inputs[i] = search_list(v_head,nets[(ptr-
>block)->nets[i+1]].name);
            }
            ptr = ptr->n_node;
      }
      free_list(q_head);
      return v_head;
}
//printing the graph into a file
void print_graph(struct s_node* root, int num_blocks, int graph_size, char*
output_file)
{
      struct s_node* col = root;
      struct s_node* row = col->n_node;
      FILE *fptr;
      fptr = fopen(output_file,"w");
      if (fptr == NULL)
      {
            printf("\nCan't Open File to Print Graph");
            exit(1);
      }
      fprintf(fptr,"**************Printing Graph**************\n");
      int i = 0;
      for (i = 0; i < graph_size; i++)
      {
            fprintf(fptr,"\nParent Array Node : %d",i+1);
            print_node_to_file(fptr,col,i+1);
            fprintf(fptr,"\n");
            int j = 1;
            while (row != NULL)
            {
                  fprintf(fptr,"\nList Node : %d",j);
                  print_node_to_file(fptr,row,j++);
                  fprintf(fptr,"\n-----------------");
                  row = row->n_node;
            }
            fprintf(fptr,"\n*****************END of
List*****************\n");
            col++; row = col->n_node;
      }

      fprintf(fptr,"\nEnd of Graph");
```

```
	fclose(fptr);
}
//copy a block information into node
void cp_node_block(struct s_node* node,
		struct s_block* block,
		struct s_node* position,
		struct s_node* n,
		struct s_node* p,
		int inputs,
		int scheduled,
		int visited)
{
	node->block = block;
	node->array_position = position;
	node->n_node = n;
	node->p_node = p;
	node->num_inputs = inputs;
	node->scheduled = scheduled;
	node->visited = visited;
}
//copy a node into another node
void cp_node_node(struct s_node* output_node, struct s_node* input_node)
{
	output_node->block = input_node->block;
	output_node->array_position = input_node->array_position;
	output_node->n_node = input_node->n_node;
	output_node->p_node = input_node->p_node;
	output_node->num_inputs = input_node->num_inputs;
	output_node->scheduled = input_node->scheduled;
	output_node->visited = input_node->visited;
}
//add a new to the end of the list
void add_node(struct s_node* node, struct s_node* head, struct s_node*
position)
{
	while(head->n_node != NULL)
	{
		head = head->n_node;
	}
	head->n_node = node;
	node->p_node = head;
	node->n_node = NULL;
```

```
	node->block = position->block;
	node->array_position = position;
	node->num_inputs = 0;
	node->scheduled = 0;
	node->visited = 0;
}
//swap two nodes into eachother
void swap_node(struct s_node* node1, struct s_node* node2)
{
	struct s_node* tmp = (struct s_node*)malloc(sizeof(struct s_node));
	cp_node_node(tmp,node1);
	cp_node_node(node1,node2);
	cp_node_node(node2,tmp);
	free(tmp);
}
struct s_node* find_position(struct s_node* head, struct s_node* node, int
size)
{
	struct s_node* tmp = head;
	int i = 0;
	for (i = 0; i < size; i++)
	{
		if (tmp->block == node->block)
		{
			return tmp;
		}else tmp++;
	}
	return NULL;
}
//printing a node to the file
void print_node_to_file(FILE* fptr, struct s_node* tmp, int i)
{
	fprintf(fptr,"\nNode[%d] Add:%p",i,tmp);
	fprintf(fptr,"\nblock Add:%p",tmp->block);
	if (tmp->block != NULL)
	{
		fprintf(fptr,"\nblock name:%s",(tmp->block)->name);
		if ((tmp->block)->type == INPAD)
			fprintf(fptr,"\nblock type: INPAD");
		else if ((tmp->block)->type == OUTPAD)
			fprintf(fptr,"\nblock type: OUTPAD");
		else if ((tmp->block)->type == LUT)
```

```
                  fprintf(fptr,"\nblock type: LUT");
            else if ((tmp->block)->type == LATCH)
                  fprintf(fptr,"\nblock type: LATCH");
            else if ((tmp->block)->type == EMPTY)
                  fprintf(fptr,"\nblock type: EMPTY");
            else if ((tmp->block)->type == LUT_AND_LATCH)
                  fprintf(fptr,"\nblock type: LUT_AND_LATCH");
      }else if(tmp->block == NULL)
      {
            fprintf(fptr,"\nblock name: root");
      }
            fprintf(fptr,"\narray position:%p",tmp->array_position);
            fprintf(fptr,"\nnext node:%p",tmp->n_node);
            fprintf(fptr,"\nprevious node:%p",tmp->p_node);
            fprintf(fptr,"\nnum_inputs:%d",tmp->num_inputs);
            fprintf(fptr,"\nscheduled:%d",tmp->scheduled);
            fprintf(fptr,"\nvisited:%d",tmp->visited);
            fprintf(fptr,"\nTruth Table:");
            int n = 0;
            for (n = 0; n < 16; n++)
                  fprintf(fptr,"\nAdd[%d]: %d",n,(tmp->block)-
>truth_table[n]);
}
//traverse the whole graph and update num_inputs in each node in parent
void count_inputs(struct s_node* head, int size)
{
      struct s_node* col = head + 1;
      struct s_node* row = col->n_node;
      int i = 0;
      for (i = 0; i < (size-1); i++)
      {
            while(row != NULL)
            {
                  (row->array_position)->num_inputs++;
                  row = row->n_node;
            }col++; row = col->n_node;
      }
}
void push(struct s_node** head, struct s_node** tail, struct s_node* node)
{
      struct s_node* ptr = (struct s_node*)malloc(sizeof(struct s_node));
      cp_node_node(ptr,node);
```

```
	ptr->n_node = NULL;
	if(*tail == NULL && *head == NULL)
	{
		ptr->p_node = NULL;
		*head = ptr;
		*tail = ptr;
	}else
	{
		ptr->p_node = *tail;
		(*tail)->n_node = ptr;
		*tail = ptr;
	}
}
struct s_node* pop(struct s_node** head, struct s_node** tail)
{
	struct s_node* ptr = *head;
	if (*head == *tail)
	{
		*head = NULL;
		*tail = NULL;
	}else
	{
		*head = (*head)->n_node;
		(*head)->p_node = NULL;
	}
	ptr->p_node = NULL;
	ptr->n_node = NULL;
	return ptr;
}
int search(struct s_node* head, struct s_node* key)
{
	struct s_node* ptr = head;
	while (ptr != NULL)
	{
		if (ptr == key)
			return 1;
		ptr = ptr->n_node;
	}
	return 0;
}
int empty(struct s_node* head, struct s_node* tail)
{
```

```
	if (head == NULL && tail == NULL)
		return 1;
	else return 0;
}
void print_schedule(struct s_node* head, char* output_file, struct s_net*
net)
{
	struct s_node* ptr = head;
	FILE *fptr;
	fptr = fopen(output_file,"w");
	if (fptr == NULL)
	{
		printf("\nCan't Open File to Print Graph");
		exit(1);
	}
	fprintf(fptr,"**************schedule**************\n\n");
	int i = 0;
	while (ptr != NULL)
	{
		if ((ptr->block)->type == INPAD)
			fprintf(fptr,"Prev:%p\tBlock
No.%d:%p,\tBlock:%s,\tType:INPAD\tNext:%p\n",ptr->p_node,i++,ptr,(ptr-
>block)->name,ptr->n_node);
		else if ((ptr->block)->type == OUTPAD)
			fprintf(fptr,"Prev:%p\tBlock
No.%d:%p,\tBlock:%s,\tType:OUTPAD\tNext:%p\n",ptr->p_node,i++,ptr,(ptr-
>block)->name,ptr->n_node);
		else if ((ptr->block)->type == LUT)
		{
			fprintf(fptr,"Prev:%p\tBlock
No.%d:%p,\tBlock:%s,\tType:LUT\tNext:%p\n",ptr->p_node,i++,ptr,(ptr->block)-
>name,ptr->n_node);
			fprintf(fptr,"Truth Table:\n");
			int n = 0;
			for (n = 0; n < 16; n++)
				fprintf(fptr,"Add[%d]: %d\n",n,(ptr->block)-
>truth_table[n]);
			fprintf(fptr,"\n");
			for (n = 0; n < MAXLUT; n++)
			{
				if (ptr->inputs[n] != NULL)
					fprintf(fptr,"Input Pointers[%d]:
%s\n",n,((ptr->inputs[n])->block)->name);
			}
```

```
                  fprintf(fptr,"\n");
            }
            else if ((ptr->block)->type == LATCH)
            {
                  fprintf(fptr,"Prev:%p\tBlock
No.%d:%p,\tBlock:%s,\tType:LATCH\tNext:%p\n",ptr->p_node,i++,ptr,(ptr-
>block)->name,ptr->n_node);
                  int n = 0;
                  for (n = 0; n < MAXLUT; n++)
                  {
                        if (ptr->inputs[n] != NULL)
                              fprintf(fptr,"Input Pointers[%d]:
%s\n",n,((ptr->inputs[n])->block)->name);
                  }
                  fprintf(fptr,"\n");
            }
            else if ((ptr->block)->type == EMPTY)
                  fprintf(fptr,"Prev:%p\tBlock
No.%d:%p,\tBlock:%s,\tType:EMPTY\tNext:%p\n",ptr->p_node,i++,ptr,(ptr-
>block)->name,ptr->n_node);
            else if ((ptr->block)->type == LUT_AND_LATCH)
                  fprintf(fptr,"Prev:%p\tBlock
No.%d:%p,\tBlock:%s,\tType:LUT_AND_LATCH\tNext:%p\n",ptr-
>p_node,i++,ptr,(ptr->block)->name,ptr->n_node);
            int x = 0;
            for (x = 0; x < ((ptr->block)->num_nets); x++)
            {
                  if (x == 0)
                        fprintf(fptr,"Output using net: %s\n",net[((ptr-
>block)->nets[x])].name);
                  else
                        fprintf(fptr,"Input using net[%d]: %s\n",x,net[((ptr-
>block)->nets[x])].name);
            }
            fprintf(fptr,"\n");
            ptr = ptr->n_node;
      }
      fprintf(fptr,"Total Blocks scheduled: %d\n",i);
      fprintf(fptr,"\n***************the end***************\n\n\n");
      fclose(fptr);
}
void free_graph(struct s_node* head,int size)
{
      int i = 0;
```

```
	struct s_node* tmp = NULL;
	struct s_node* col = head;
	struct s_node* row = col->n_node;
	for (i = 0; i < size; i++)
	{
		while(row != NULL)
		{
			tmp = row->n_node;
			free(row);
			row = tmp;
		}
		tmp = col++;
		free(col);
		col = tmp; row = col->n_node;
	}
}
void free_list(struct s_node* head)
{
	struct s_node* tmp = NULL;
	while (head != NULL)
	{
		tmp = head->n_node;
		free(head);
		head = tmp;
	}
}
struct s_node* search_list(struct s_node* head, char* block_name)
{
	if (block_name == NULL)
		return NULL;
	else {
		while (head != NULL)
		{
			if (strcmp((head->block)->name,block_name) == 0)
			{
				return head;
			}
			head = head->n_node;
		}
	}
	return NULL;
}
```

simulation.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include "simulation.h"
#include "util.h"
long double C_u = 0.0000000000000000064;
long double A_bit = 140;
long double F = 0.000000032;
void simulate(struct s_node*, int,int,int,float,int,int,float,float);
int evaluate_block(struct s_node*, int);
long double fpga_pwr(unsigned long,float);
long double rand_mem_cap(int,int);
long double seq_mem_cap(int,int);
//Memory Capacitances(Energies)
long double data_energy = 0.0;
long double instruction_energy = 0.0;
long double Total_Data_Mem_Access = 0.0;
long double Total_Inst_Mem_Access = 0.0;
//Capacitance to evaluate 4LUT
long double LUT_Cap = 0.0;
long double Data_Mem_Cap_per_Access = 0.0;
long double Inst_Mem_Cap_per_Access = 0.0;
long double Functional_Unit_Cap = 0.0;
//FPGA Total LUT Evaluation Cap
long double FPGA_LUT_Eval_Cap = 0.0;
void simulate(struct s_node* head, int sim_cycles, int num_luts, int num_latches,
float area_clb, int data_mem, int data_bits, float inst_comp, float data_comp)
{
	srand(time(NULL));
	struct s_node* c_ptr = head;

	unsigned long FPGA_wire_len = 0;
	//First time fill everything with random 0 or 1
	//This will remove any value other than 0 or 1 even for first evaluation
	while (c_ptr != NULL)
	{
		c_ptr->current_value = (rand()%2);
		c_ptr = c_ptr->n_node;
	}
	int tmp_val = -1;
	//Total CPU Cap
	long double mp_pwr = 0.0;
```

```
	//Values for W and M for Data and Instruction Memory
	int W_I = 16+5*(ceil(log2((num_luts+num_latches))));
	int M_I = num_luts*inst_comp;
	int W_D = data_bits;
	int M_D = data_mem*data_comp;
	//Measure Data and Instruction Memory Cap per Access
	Data_Mem_Cap_per_Access = rand_mem_cap(W_D,M_D);
	Inst_Mem_Cap_per_Access = seq_mem_cap(W_I,M_I);
	//Measure Total Instruction Memory Access
	Total_Inst_Mem_Access = inst_comp*(sim_cycles*(num_luts+num_latches));
	//Measure 4LUT Cap first
	LUT_Cap = rand_mem_cap(1,16);
	//Measure Total 4LUT Evaluation Cap FPGA
	FPGA_LUT_Eval_Cap = sim_cycles*data_bits*num_luts*LUT_Cap;
	c_ptr = head;
	int i = 0;
	for (i = 0; i < sim_cycles; i++)
	{
		//Fill Primary input nodes with random values
		while (((c_ptr->block)->type) == INPAD)
		{
			int rand_num = rand()%2;
			if (c_ptr->current_value != rand_num);
			{
				FPGA_wire_len += c_ptr->net_length;
				c_ptr->current_value = rand_num;
			}
			c_ptr = c_ptr->n_node;
		}
		//Do one simulation cycle
		while (c_ptr != NULL)
		{
			if ((c_ptr->block)->type == LUT)
			{
				tmp_val = evaluate_block(c_ptr,1);
				if (tmp_val != c_ptr->current_value)
				{
					c_ptr->current_value = tmp_val;
					Total_Data_Mem_Access += (c_ptr->block)->num_nets;
					FPGA_wire_len += c_ptr->net_length;
				}else
					Total_Data_Mem_Access += ((c_ptr->block)->num_nets)-
1;
				}else if ((c_ptr->block)->type == LATCH)
			{
				tmp_val = evaluate_block(c_ptr,0);
```

```
                        if (tmp_val != c_ptr->current_value)
                        {
                                c_ptr->current_value = tmp_val;
                                Total_Data_Mem_Access += ((c_ptr->block)->num_nets)-
1;
                        }
                }
                c_ptr = c_ptr->n_node;
        }
        c_ptr = head;
    }
    //Measure Total FPGA Cap
    long double FPGA_Routing_Cap = (data_bits)*(fpga_pwr(FPGA_wire_len,area_clb));
    long double fp_pwr = FPGA_Routing_Cap + FPGA_LUT_Eval_Cap;
    //Measure Total Data Memory Cap (CPU)
    data_energy = Data_Mem_Cap_per_Access*(Total_Data_Mem_Access*data_comp);
    //Measure Total Inst Memory Cap (CPU)
    instruction_energy = Inst_Mem_Cap_per_Access*Total_Inst_Mem_Access;
    //Measure Total Functional Unit Cap (CPU)
    Functional_Unit_Cap = fp_pwr/14;

    //Measure Total CPU Cap
    mp_pwr = Functional_Unit_Cap + data_energy + instruction_energy;
    printf("My_Data Instruction: %Lf ",instruction_energy);
    printf("Data: %Lf ",data_energy);
    printf("Funt_Unit: %Lf ",Functional_Unit_Cap);
    printf("Processor: %Lf ",mp_pwr);// fflush(stdout);
    printf("Routing: %Lf ",FPGA_Routing_Cap);
    printf("LUT: %Lf ",FPGA_LUT_Eval_Cap);
    printf("FPGA: %Lf ",fp_pwr);
    if (mp_pwr > fp_pwr)
    {
            float mp_x_fp = mp_pwr/fp_pwr;
            printf("uP is %fx times more than FPGA",mp_x_fp);
    }
    else if (fp_pwr > mp_pwr)
    {
            float fp_x_mp = fp_pwr/mp_pwr;
            printf("FPGA is %fx times more than uP",fp_x_mp);
    }
}
//Calculate FPGA Power consumed
long double fpga_pwr(unsigned long wire_len,float area_clb)
{
    long double cap = wire_len*sqrtl(area_clb*12*8*3);
    return cap;
```

```
}
//Calcualte Random Access Memory Capacitance
long double rand_mem_cap(int W,int M)
{
	long double tmp_log = (logl(M))/(logl(2));
	long double cap = (tmp_log+(2*(2*W+2)))*(sqrtl(W*M));
	return cap;
}
//Calculate Seq Access Memory Capacitance
long double seq_mem_cap(int W, int M)
{
	long double cap = ((2*(2*W+1))*sqrtl(W*M));
	return cap;
}
//Evaluate current block value based on its input values
int evaluate_block(struct s_node* c_ptr, int type)
{
	int n = 0;
	if (type == 0)
	{
		//If LATCH
	return (c_ptr->inputs[0])->current_value;
	}
	else if (type == 1) //For LUT
	{
		char zero[2];
		strcpy(zero,"0");
		char one[2];
		strcpy(one,"1");
		char address[MAXLUT+2];
		strcpy(address,"0");
		for (n = 0; n < ((c_ptr->block)->num_nets)-1; n++)
		{
			if ((c_ptr->inputs[n])->current_value == 0)
			{
				strcat(address,zero);
			}else if ((c_ptr->inputs[n])->current_value == 1)
			{
				strcat(address,one);
			}
		}
		return (c_ptr->block)->truth_table[bin2dec(atoi(address))];
	}
	return -1;
}
```

read_route.c:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "vpack.h"
#define BUFSIZE 100
#define TOKENS " \t\n"
#define FOUND 0
#define NOT_FOUND 1
struct node
{
      char *name;
      struct node* p_node;
      struct node* n_node;
};
//Add a node to the end of list
void add_node_r(struct node*,char*);
//Create a list
struct node* create_list_r();
//Remove a list
void remove_list_r(struct node*);
//Search list
int search_list_r(struct node*,char *);
//Clean the charater array
void clean_char_r(char*);
//Search schedule
struct s_node* search_schedule_r(struct s_node*, char*);
void update_net_lengths(struct s_node* schedule, char* input_file, char*
output_file)
{
      char buffer[BUFSIZE];
      FILE *fp_in,*fp_out;
      fp_in = fopen(input_file,"r");
      fp_out = fopen(output_file,"w");
      unsigned long total_wire_length = 0;
      unsigned long net_length = 0;
      unsigned long blah_wire = 0;

      char NET[BUFSIZE];
      clean_char_r(NET);
      char p_word[BUFSIZE];
      clean_char_r(p_word);
```

```
struct node* channel_list = NULL;
struct s_node* ptr = NULL;
//Read a single line from file
while (fgets(buffer,BUFSIZE,fp_in) != NULL)
{
	//Break the line into tokens
	char *word = strtok(buffer,TOKENS);
	if (word != NULL)
	{
		//If the first Token is Net
		if (strcmp(word,"Net") == 0)
		{
			ptr = search_schedule_r(schedule,NET);
			if (ptr != NULL)
				ptr->net_length = net_length;
			fprintf(fp_out,"Net Length: %lu\n\n",net_length);
			clean_char_r(NET);
			net_length = 0;
			fprintf(fp_out,"%s ",word);
			if (channel_list == NULL)
				channel_list = create_list_r();
			else if(channel_list != NULL)
			{
				free(channel_list);
				channel_list = create_list_r();
			}
			int done = 0;
			while ((word = strtok(NULL,TOKENS)) != NULL)
			{
				if (word != NULL && done == 1)
				{
					int len = strlen(word);
					int i = 0;
					for (i = 1; i < len-1; i++)
						NET[i-1] = word[i];
					fprintf(fp_out,"%s",NET);
				}
				done = 1;
			}
			fprintf(fp_out,"\n");
		}
		//If the first Token is CHANX or CHANY
```

```
                  else if(strcmp(word,"CHANX") == 0 || strcmp(word,"CHANY")
== 0)
                  {
                        int done = 0;
                        clean_char_r(p_word);
                        strcpy(p_word,word);
                        while ((word = strtok(NULL,TOKENS)) != NULL)
                        {
                              if (word != NULL && done == 0)
                              {
                                    strcat(p_word," ");
                                    strcat(p_word,word);
                                    if (search_list_r(channel_list,p_word) ==
NOT_FOUND)
                                    {
                                          net_length++;
                                          total_wire_length++;
                                          add_node_r(channel_list,p_word);
                                    }
                                    blah_wire++;
                                    fprintf(fp_out,"%s\n",p_word);
                                    done = 1;
                              }
                        }
                  }
            }
      }
      ptr = search_schedule_r(schedule,NET);
      if (ptr != NULL)
            ptr->net_length = net_length;

      fprintf(fp_out,"Net Length: %lu\n\nTotoal Wire Length: %lu\nBlah wire:
%lu\n",net_length,total_wire_length,blah_wire);
      fclose(fp_in);
      fclose(fp_out);
      printf("Net Length: %lu\n",net_length);
      printf("Total wire length: %lu\nBlah wire:
%lu\n",total_wire_length,blah_wire);
}
//Adding a node to list
void add_node_r(struct node* head,char *name)
{
      struct node* n_ptr = (struct node*)malloc(sizeof(struct node));
```

```
	n_ptr->name = (char*)malloc(10);
	while(head->n_node != NULL)
		head = head->n_node;
	strcpy(n_ptr->name,name);
	n_ptr->p_node = head;
	head->n_node = n_ptr;
	n_ptr->n_node = NULL;
	while (head->p_node != NULL)
		head = head->p_node;
}
//Create list
struct node* create_list_r()
{
	struct node* ptr = (struct node*)malloc(sizeof(struct node));
	ptr->name = (char*)malloc(5);
	strcpy(ptr->name,"Net");
	ptr->p_node = NULL;
	ptr->n_node = NULL;
	return ptr;
}
//Remove list
void remove_list_r(struct node* head)
{
	while (head->n_node != NULL)
	{
		free(head->name);
		head = head->n_node;
	}
	free(head->name);
	head = head->p_node;
	while(head->p_node != NULL)
	{
		free(head->n_node);
		head = head->p_node;
	}
	free(head);
}
//Search list
int search_list_r(struct node* head, char *name)
{
	while (head != NULL)
	{
```

```
            if (strcmp(head->name,name) == FOUND)
                  return FOUND;
            head = head->n_node;
      }
      return NOT_FOUND;
}
//Clean charater array
void clean_char_r(char* word)
{
      int i = 0;
      for (i = 0; i < BUFSIZE; i++)
            word[i] = '\0';
}
//Search Schedule
struct s_node* search_schedule_r(struct s_node* schedule, char *Net_name)
{
      struct s_node* ptr = schedule;
      while (ptr != NULL)
      {
            if (strcmp(((ptr->block)->name),Net_name) == 0)
                  return ptr;
            ptr = ptr->n_node;
      }
      return NULL;
}
```

H-Files:

globals.h:

```
/* Netlist description data structures. */
extern int num_nets, num_blocks;
extern int num_p_inputs, num_p_outputs;

extern int graph_size;

extern struct s_net *net;
extern struct s_block *block;

/* Number in original netlist, before FF packing. */
extern int num_luts, num_latches;

/* Graph Nodes. */
extern struct s_node* graph;

/* Queue Variables
extern struct s_node* q_head;
extern struct s_node* q_tail;
extern struct s_node* v_head;
extern struct s_node* v_tail;*/
```

graph.h:

```
#ifndef __GRAPH_H
#define __GRAPH_H
#include "vpack.h"
extern struct s_node* generate_graph(struct s_block*, struct s_net*, int
num_blocks, int num_p_outputs, int graph_size);
extern void print_graph(struct s_node*, int num_blocks, int graph_size,
char*);
extern struct s_node* generate_schedule(struct s_node*, struct s_block*,
struct s_net*);
extern void print_schedule(struct s_node*,char*,struct s_net*);
extern void free_graph(struct s_node*,int);
#endif

read_blif.h:
void read_blif (char *blif_file, int lut_size);
```

```
void echo_input (char *blif_file, int lut_size, char *echo_file);

read_route.h
#ifndef __READ_ROUTE_H
#define __READ_ROUTE_H
#include "vpack.h"
extern void update_net_lengths(struct s_node*, char*, char*);
#endif

simulation.h:
#ifndef __SIMULATION_H
#define __SIMULATION_H
#include "vpack.h"
void simulate(struct s_node*, int, int, int, float, int, int, float, float);
#endif
```

util.h:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "vpack.h"

/*************** Global variables exported by this module
********************/

extern int linenum;  /* line in file being parsed */


/******************* Types and defines exported by this module
***************/

#ifndef TRUE      /* Some compilers predefine TRUE, FALSE */
typedef enum {FALSE, TRUE} boolean;
#else
typedef int boolean;
#endif

#define BUFSIZE 300     /* Maximum line length for various parsing proc. */
#define max(a,b) (((a) > (b))? (a) : (b))
#define min(a,b) ((a) > (b)? (b) : (a))
#define nint(a) ((int) floor (a + 0.5))
```

```
/* Linked lists of void pointers and integers, respectively.
*/

struct s_linked_vptr {void *data_vptr; struct s_linked_vptr *next;};

struct s_linked_int {int data; struct s_linked_int *next;};
typedef struct s_linked_int t_linked_int;


/* Integer vector.  nelem stores length, list[0..nelem-1] stores list of    *
 * integers.
*/

struct s_ivec {int nelem; int *list;};


/************************ Memory allocation routines
************************/

void *my_calloc (size_t nelem, size_t size);
void *my_malloc (size_t size);
void *my_realloc (void *ptr, size_t size);

void *my_chunk_malloc (size_t size, struct s_linked_vptr **chunk_ptr_head,
         int *mem_avail_ptr, char **next_mem_loc_ptr);

void free_chunk_memory (struct s_linked_vptr *chunk_ptr_head);


/******************* Linked list, matrix and vector utilities
***************/

void free_ivec_vector (struct s_ivec *ivec_vector, int nrmin, int nrmax);

void free_ivec_matrix (struct s_ivec **ivec_matrix, int nrmin, int nrmax,
         int ncmin, int ncmax);

void free_ivec_matrix3 (struct s_ivec ***ivec_matrix3, int nrmin, int nrmax,
         int ncmin, int ncmax, int ndmin, int ndmax);

void **alloc_matrix (int nrmin, int nrmax, int ncmin, int ncmax,
      size_t elsize);
```

```
void ***alloc_matrix3 (int nrmin, int nrmax, int ncmin, int ncmax, int ndmin,
      int ndmax, size_t elsize);

void free_matrix (void *vptr, int nrmin, int nrmax, int ncmin, size_t
elsize);

void free_matrix3 (void *vptr, int nrmin, int nrmax, int ncmin, int ncmax,
      int ndmin, size_t elsize);

struct s_linked_vptr *insert_in_vptr_list (struct s_linked_vptr *head,
         void *vptr_to_add);

t_linked_int *insert_in_int_list (t_linked_int *head, int data, t_linked_int
**
         free_list_head_ptr);

void free_int_list (t_linked_int **int_list_head_ptr);

void alloc_ivector_and_copy_int_list (t_linked_int **list_head_ptr,
            int num_items, struct s_ivec *ivec, t_linked_int
            **free_list_head_ptr);


/****************** File and parsing utilities
*****************************/

FILE *my_fopen (char *fname, char *flag, int prompt);

char *my_strtok(char *ptr, char *tokens, FILE *fp, char *buf);

char *my_fgets(char *buf, int max_size, FILE *fp);

int my_atoi (const char *str);


/********************** Portable random number generators
******************/

void my_srandom (int seed);
int my_irand (int imax);
float my_frand (void);
```

```
extern int bin2dec(int);
extern int measure_data_mem(struct s_node*,int);

int measure_size(struct l_node*);
```

vpack.h:

```
#ifndef __VPACK_H
#define __VPACK_H
#define MAXLUT 7        /* Maximum number of inputs per LUT */
#define HASHSIZE 4095
#define NAMELENGTH 16   /* Length of the name stored for each net */
#define DEBUG 1         /* Echoes input & checks error conditions */
/*#define VERBOSE 1*/   /* Prints all sorts of intermediate data */


#define NO_CLUSTER -1
#define NEVER_CLUSTER -2
#define NOT_VALID -10000  /* Marks gains that aren't valid */
                          /* Ensure no gain can ever be this negative! */
#define UNDEFINED -1

#define DRIVER 0     /* Is a pin driving a net or in the fanout? */
#define RECEIVER 1
#define OPEN -1      /* Pin is unconnected. */

#define TABLESIZE 16    /* Truth Table Size */

enum block_types {INPAD = -2, OUTPAD, LUT, LATCH, EMPTY, LUT_AND_LATCH};
enum e_cluster_seed {TIMING, MAX_INPUTS};

struct hash_nets {      char *name;
                  int index;
                  int count;
                  struct hash_nets *next; };
/* count is the number of pins on this net so far. */

struct s_net {    char *name;
            int num_pins;
            int *pins;  };
/* name:  ASCII net name for informative annotations in the output.  *
```

```
 * num_pins:  Number of pins on this net.                              *
 * pins[]: Array containing the blocks to which the pins of this net *
 *         connect.  Output in pins[0], inputs in other entries.     */

struct s_block {char *name;
            enum block_types type;
            int num_nets;
                int nets[MAXLUT+2];
            int truth_table[TABLESIZE];   };
/* name:  Taken from the net which it drives.                          *
 * type:  LUT, INPAD, OUTPAD or LATCH.                                 *
 * num_nets:  number of nets connected to this block.                  *
 * nets[]:  List of nets connected to this block.  Net[0] is the       *
 *          output, others are inputs, except for OUTPAD.  OUTPADs     *
 *          only have an input, so this input is in net[0].           */

struct s_node {   struct s_block *block;
            struct s_node *array_position;
            struct s_node *n_node;
            struct s_node *p_node;
            struct s_node *inputs[MAXLUT];
            int num_inputs;
            int scheduled;
            int visited;
            int current_value;
            unsigned long net_length;
            int life_time;                };
/* block: Pointer to block contained by this node                *
 * n_node: Pointer to next node                                  */


//This structure will be used in different small operations
struct l_node { int num;
            struct l_node* n_node;
            struct l_node* p_node;        };
#endif
```

Blif files:

8_bit_compressor_tree:

```
.model top
.inputs a0 a1 a2 a3 a4 a5 a6 a7 b0 b1 b2 b3 b4 b5 b6 b7 c0 c1 c2 c3 c4 c5 c6
c7 d0 d1 d2 d3 d4 d5 d6 d7 e0 e1 e2 e3 e4 e5 e6 e7 f0 f1 f2 f3 f4 f5 f6 f7 g0
g1 g2 g3 g4 g5 g6 g7 cin_1 cin_2
.outputs s0 s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 s14 s15 cout_0 cout_1
cout_6_4
.names a0 b0 c0 cout_0_1
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names a0 b0 c0 sum_0_1
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names d0 e0 f0 cout_0_2
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names d0 e0 f0 sum_0_2
000 0
001 1
010 1
011 0
```

```
100 1
101 0
110 0
111 1
.names sum_0_1 sum_0_2 g0 cout_0_3
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names sum_0_1 sum_0_2 g0 sum_0_3
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names cout_0_1 cout_0_2 cout_0_3 cout_0_4
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names cout_0_1 cout_0_2 cout_0_3 sum_0_4
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names sum_0_3 cin_1 cin_2 s1
000 0
```

```
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names sum_0_3 cin_1 cin_2 s0
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names a1 b1 c1 cout_1_1
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names a1 b1 c1 sum_1_1
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names d1 e1 f1 cout_1_2
000 0
001 0
010 0
011 1
100 0
101 1
110 1
```

```
111 1
.names d1 e1 f1 sum_1_2
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names sum_1_1 sum_1_2 g1 cout_1_3
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names sum_1_1 sum_1_2 g1 sum_1_3
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names cout_1_1 cout_1_2 cout_1_3 cout_1_4
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names cout_1_1 cout_1_2 cout_1_3 sum_1_4
000 0
001 1
010 1
011 0
```

```
100 1
101 0
110 0
111 1
.names sum_1_3 sum_0_4 cin_1 s3
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names sum_1_3 sum_0_4 cin_1 s2
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names a2 b2 c2 cout_2_1
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names a2 b2 c2 sum_2_1
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names d2 e2 f2 cout_2_2
000 0
```

```
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names d2 e2 f2 sum_2_2
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names sum_2_1 sum_2_2 g2 cout_2_3
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names sum_2_1 sum_2_2 g2 sum_2_3
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names cout_2_1 cout_2_2 cout_2_3 cout_2_4
000 0
001 0
010 0
011 1
100 0
101 1
110 1
```

```
111 1
.names cout_2_1 cout_2_2 cout_2_3 sum_2_4
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names sum_2_3 sum_1_4 cout_0_4 s5
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names sum_2_3 sum_1_4 cout_0_4 s4
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names a3 b3 c3 cout_3_1
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names a3 b3 c3 sum_3_1
000 0
001 1
010 1
011 0
```

```
100 1
101 0
110 0
111 1
.names d3 e3 f3 cout_3_2
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names d3 e3 f3 sum_3_2
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names sum_3_1 sum_3_2 g3 cout_3_3
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names sum_3_1 sum_3_2 g3 sum_3_3
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names cout_3_1 cout_3_2 cout_3_3 cout_3_4
000 0
```

```
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names cout_3_1 cout_3_2 cout_3_3 sum_3_4
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names sum_3_3 sum_2_4 cout_1_4 s7
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names sum_3_3 sum_2_4 cout_1_4 s6
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names a4 b4 c4 cout_4_1
000 0
001 0
010 0
011 1
100 0
101 1
110 1
```

```
111 1
.names a4 b4 c4 sum_4_1
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names d4 e4 f4 cout_4_2
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names d4 e4 f4 sum_4_2
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names sum_4_1 sum_4_2 g4 cout_4_3
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names sum_4_1 sum_4_2 g4 sum_4_3
000 0
001 1
010 1
011 0
```

```
100 1
101 0
110 0
111 1
.names cout_4_1 cout_4_2 cout_4_3 cout_4_4
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names cout_4_1 cout_4_2 cout_4_3 sum_4_4
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names sum_4_3 sum_3_4 cout_2_4 s9
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names sum_4_3 sum_3_4 cout_2_4 s8
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names a5 b5 c5 cout_5_1
000 0
```

```
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names a5 b5 c5 sum_5_1
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names d5 e5 f5 cout_5_2
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names d5 e5 f5 sum_5_2
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names sum_5_1 sum_5_2 g5 cout_5_3
000 0
001 0
010 0
011 1
100 0
101 1
110 1
```

```
111 1
.names sum_5_1 sum_5_2 g5 sum_5_3
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names cout_5_1 cout_5_2 cout_5_3 cout_5_4
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names cout_5_1 cout_5_2 cout_5_3 sum_5_4
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names sum_5_3 sum_4_4 cout_3_4 s11
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names sum_5_3 sum_4_4 cout_3_4 s10
000 0
001 1
010 1
011 0
```

```
100 1
101 0
110 0
111 1
.names a6 b6 c6 cout_6_1
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names a6 b6 c6 sum_6_1
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names d6 e6 f6 cout_6_2
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names d6 e6 f6 sum_6_2
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names sum_6_1 sum_6_2 g6 cout_6_3
000 0
```

```
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names sum_6_1 sum_6_2 g6 sum_6_3
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names cout_6_1 cout_6_2 cout_6_3 cout_6_4
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names cout_6_1 cout_6_2 cout_6_3 sum_6_4
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names sum_6_3 sum_5_4 cout_4_4 s13
000 0
001 0
010 0
011 1
100 0
101 1
110 1
```

```
111 1
.names sum_6_3 sum_5_4 cout_4_4 s12
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names a7 b7 c7 cout_7_1
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names a7 b7 c7 sum_7_1
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names d7 e7 f7 cout_7_2
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names d7 e7 f7 sum_7_2
000 0
001 1
010 1
011 0
```

```
100 1
101 0
110 0
111 1
.names sum_7_1 sum_7_2 g7 cout_7_3
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names sum_7_1 sum_7_2 g7 sum_7_3
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names cout_7_1 cout_7_2 cout_7_3 cout_1
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names cout_7_1 cout_7_2 cout_7_3 cout_0
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names sum_7_3 sum_6_4 cout_5_4 s15
000 0
```

```
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names sum_7_3 sum_6_4 cout_5_4 s14
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.end
```

8_Full_Adders:

```
.model top
.inputs x1 x2 x3 x4 x5 x6 x7 x8 y1 y2 y3 y4 y5 y6 y7 y8 z1 z2 z3 z4 z5 z6 z7
z8
.outputs s1 s2 s3 s4 s5 s6 s7 s8 c1 c2 c3 c4 c5 c6 c7 c8
.names x1 y1 z1 c1
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names x2 y2 z2 c2
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names x3 y3 z3 c3
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names x4 y4 z4 c4
000 0
001 0
010 0
011 1
100 0
101 1
110 1
```

```
111 1
.names x5 y5 z5 c5
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names x6 y6 z6 c6
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names x7 y7 z7 c7
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names x8 y8 z8 c8
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names x1 y1 z1 s1
000 0
001 1
010 1
011 0
```

```
100 1
101 0
110 0
111 1
.names x2 y2 z2 s2
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names x3 y3 z3 s3
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names x4 y4 z4 s4
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names x5 y5 z5 s5
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names x6 y6 z6 s6
000 0
```

```
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names x7 y7 z7 s7
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.names x8 y8 z8 s8
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.end
```

Full Adder:

```
.model top
.inputs a b c
.outputs sum carry
.names a b c carry
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
.names a b c sum
000 0
```

```
001 1
010 1
011 0
100 1
101 0
110 0
111 1
.end
```

Makefile:

```
# This makefile is written for gcc running under SLinux 6.0 on x86 arch.
# To compile T-VPack on other systems, you may have to change:
# (1) CC to the name of your C compiler.
# (2) OPT_FLAGS should be changed to whatever options turn on maximum
#     optimization in your compiler.

CC = gcc
#CC = purify gcc
#CC = g++


# Overly strict flags line below.  Lots of useless warnings, but can
# let you look for redudant declarations.
# To avoid redundant declarations here I use -D__STDC instead of
# -D__USE_FIXED_PROTOTYPES, but that means some prototypes are missing.

#FLAGS = -Wall -Wpointer-arith -Wcast-qual -Wstrict-prototypes -O -D__STDC__
-ansi -pedantic -Wredundant-decls -Wmissing-prototypes -Wshadow -Wcast-align
-D_POSIX_SOURCE

#Flags to be passed to the compiler.  First is for strict warnings,
#second for interactive debugging and third for optimization.

#-D_POSIX_SOURCE stops extra declarations from being included in math.h
#and causing -Wshadow to complain about conflicts with y1 in math.h
#(Bessel function 1 of the second kind)
```

```
WARN_FLAGS = -Wall -Wpointer-arith -Wcast-qual -Wstrict-prototypes -O -
D__USE_FIXED_PROTOTYPES__ -ansi -pedantic -Wmissing-prototypes -Wshadow -
Wcast-align -D_POSIX_SOURCE

DEBUG_FLAGS = -g -Wall
#OPT_FLAGS = -O2

FLAGS = $(DEBUG_FLAGS)
#FLAGS = $(OPT_FLAGS)
#FLAGS = $(WARN_FLAGS)



#Useful flags on HP machines.
#DEBUG_FLAGS = -Aa -g
#OPT_FLAGS = -Aa +O3




EXE = t-vpack

OBJ = main.o util.o read_blif.o graph.o simulation.o read_route.o

SRC = main.c util.c read_blif.c graph.c simulation.c read_route.c

H = util.h vpack.h globals.h read_blif.h graph.h simulation.h read_route.h

LIB = -lm

# Add purify in front of CC below to run purify on the code.

$(EXE): $(OBJ)
      $(CC) $(FLAGS) $(OBJ) -o $(EXE) $(LIB)

main.o: main.c $(H)
      $(CC) -c $(FLAGS) main.c

read_blif.o: read_blif.c $(H)
      $(CC) -c $(FLAGS) read_blif.c

graph.o:  graph.c $(H)
      $(CC) -c $(FLAGS) graph.c
```

```
simulation.o: simulation.c $(H)
	$(CC) -c $(FLAGS) simulation.c

util.o: util.c $(H)
	$(CC) -c $(FLAGS) util.c

read_route.o: read_route.c $(H)
	$(CC) -c $(FLAGS) read_route.c
```

Bibliography

[1] J. Rose, A. E. Gamal and A. S. Vincentell, "Architecture of Field-Programmable Gate Arrays," *Proceedings of the IEEE,* vol. 81, no. 7, pp. 1013-1029, 1993.

[2] J. Fowers, G. Brown, P. Cooke and G. Stitt, "A Performance and Energy Comparison of FPGAs, GPUs, and Multicores for Sliding Window Applications," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, 2012.

[3] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE,* vol. 96, no. 5, pp. 879-899, 2008.

[4] Technologies, Qualcomm, "Qualcomm Snapdragon 820 Processor," [Online]. Available: https://www.qualcomm.com/documents/snapdragon-820-processor-product-brief.

[5] J. Rose, "Hard vs. Soft: The Central Question of Pre-Fabricated Silicon," in *Proceedings of the 34th International Symposium on Multiple-Valued Logic (ISMVL'04)*, 2004.

[6] I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2007.

[7] A. Dehon, "Wordwidth, Instructions, Looping, and Virtualization The Role of Sharing in Absolute Energy Minimization," in *FPGA '14 Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, 2014.

[8] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P.-C. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson and M. B. Taylor, "The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future," *IEEE Micro,* vol. 31, no. 2, pp. 86-95, 2011.

[9] Altera Corporation, "Standard Cell ASIC to FPGA Design Methodology and Guidelines," Altera Corporation, 2009.

[10] A. Marquardt, V. Betz and J. Rose, "Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density," in *FPGA '99 Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, 1999.

[11] V. Betz and J. Rose, "Cluster-based logic blocks for FPGAs: area-efficiency vs. input sharing and size," in *IEEE 1997 Custom Integrated Circuits Conference*, 1997.

[12] J. Rose, R. J. Francis, D. Lewis and P. Chow, "Architecture of Field Programmable Gate Arrays: The Effect of Logic Block Complexity on Area Efficiency," *IEEE Journal of Soild State Circuits*, vol. 25, no. 5, pp. 1217-1225, 1990.

[13] D. Chen, J. Cong and P. Pan, "FPGA Design Automation: A Survey," *Foundations and Trends in Electronic Design Automation*, vol. 1, no. 3, pp. 139-169, 2006.

[14] K. E. Murray, "Divide-and-Conquer Techniques for Large Scale FPGA Design," University of Toronto, 2015.

[15] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown and T. Czajkowski, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, 2011.

[16] Altera Corporation, "Implementing FPGA Design with the OpenCL Standard," 2012.

[17] J. Lamoureux and S. J. Wilton, "On the Interaction Between Power-Aware FPGA CAD Algorithms," in *Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, 2003.

[18] A. Marquardt, V. Betz and J. Rose, "Timing-Driven Placement for FPGAs," in *Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*, 2000.

[19] S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671-680, 13 May 1983.

[20] F. Rubin, "The Lee Path Connection Algorithm," *IEEE Transactions on Computers,* Vols. c-23, no. 9, pp. 907-914, 1974.

[21] S. M. Sait and H. Youssef, VLSI Physical Design Automation: Theory and Practice, World Scientific Publishing Co. Pte. Ltd., 1999.

[22] E. Bozorgzadeh, S. Ogrenci-Memik and M. Sarrafzadeh, "RPack: Routability-Driven packing for cluster-based FPGAs," in *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, 2001.

[23] A. Singh, G. PARTHASARATHY and M. MAREK-SADOWSKA, "Efficient Circuit Clustering for Area and Power Reduction in FPGAs," *ACM Transactions on Design Automation of Electronic Systems (TODAES),* vol. 7, no. 4, pp. 643-663, 2002.

[24] A. W. Burks, H. H. Goldstine and J. v. Neumann, "Preliminary Discussion of The Logical Design of an Electronic Computing Instrument," Princeton, 1946.

[25] M. D. Blasi, Computer Architecture, Addison Wesley, 1990.

[26] J. P. Hayes, Computer Architecture and Organization, McGraw-Hill, 1998.

[27] W. Wolf, Computers as Components, Morgan Kaufmann Publishers, 2008.

[28] I. Koren, Computer Arithmatic Algorithms, Universities Press, 2002.

[29] N. Weste and D. M. Harris, CMOS VLSI Design: A Circuits and Systems Perspective, Addison-Wesley, 2011.

[30] V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research," in *International Workshop on Field Programmable Logic*, London, UK, 1997.

[31] B. University of California, "Berkeley logic interchange format (BLIF)," *Tool Distribution,* vol. 2, pp. 197-247, 1992.

[32] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, The MIT Press, 2009.

[33] A. V. Aho, M. S. Lam, R. Sethi and J. D. Ullman, Compilers: Principles, Techniques and Tools, Pearson: Addison-Wesley, 2007.

[34] V. Betz, J. Rose and A. Marquardt, Architecture and CAD for Deep-Submicron FPGAs, Kluwer Academic Publishers, 1999.

[35] F. F. Khan and A. Ye, "Measuring the Accuracy of Minimum Width Transistor Area in Estimating FPGA Layout Area," in *IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2015.

[36] I. Kuon, R. Tessier and J. Rose, "FPGA Architecture: Survey and Challenges," *Foundations and Trends in Electronic Design Automation*, vol. 2, no. 2, pp. 135-253, 2008.

Glossary

FPGA	Field Programmable Gate Array
LUT	Look up Table
GPU	Graphics Processing Unit
DSP	Digital Signal Processor
CPU	Central Processing Unit
ASIC	Application Specific Integrated Circuits
MCNC	Microelectronics Centre of North Carolina
IC	Integrated Circuit
NRE	Non-Recurring Engineering
BRAM	Block Random Access Memory
DFF	D-Flip Flop
BLE	Basic Logic Element
LB	Logic Block
HDL	Hardware Description Language
CAD	Computer Aided Design
HLS	High Level Synthesis
FA	Full Adder
BLIF	Berkeley Logic Interchange Format
PE	Processing Element
VPR	Versatile Place and Route
SOC	System on Chip