University of South-Eastern Norway
Faculty of Technology, Natural Sciences, and Maritime Sciences
—
Master Thesis in Systems Engineering with Embedded Systems
Department of Science and Industry Systems
November 27, 2020

Halvor Nybø Risto

A study of CNTFET implementations for ternary logic and data radix conversion

© Halvor Nybø Risto, 2020

University of South-Eastern Norway
Faculty of Technology, Natural Sciences, and Maritime Sciences
Department of Science and Industry Systems
PO Box 235
NO-3603 Kongsberg, Norway

http://www.usn.no

This thesis is worth 30 study points

The undersigned have examined the thesis entitled ***A study of CNTFET implementations for ternary logic and data radix conversion*** presented by *Halvor Nybø Risto*, a candidate for the degree of *Master in Systems Engineering with Embedded Systems* and hereby certify that it is worthy of acceptance.

Date	Head of Department
Date	Supervisor's name
Date	Committee member's name
Date	Committee member's name

Abstract

Ternary logic theory and CNTFETs

The basic theory of ternary logic and CNTFETs are explored and explained, to set a theoretical context and build a base for the rest of the thesis. For ternary logic, this includes radix economy, ternary notations, ternary-valued logic functions, ternary algebra, and conversion overhead. For CNTFETs, topics discussed are the architecture, voltage threshold and characteristics, nanotube chirality, benefits over MOSFETs, and simulation models.

Ternary-valued CNTFET circuit design and logic synthesis

The methods used for transistor- and gate-level circuit design of ternary-valued CNTFET circuits are described, utilized, optimized, and automated with a logic synthesizer for generating simulation files with a research paper accepted in the SIMS 2020 conference, which includes proposed full adder circuits compared with simulation results.

Radix conversion of data between binary and ternary

Several methods of data radix conversion is discussed and explained. One method is implemented and optimized using the logic synthesizer, with circuit simulation results in HSPICE. It is shown that data radix conversion can be done at high speeds with a low transistor count and power consumption with the CNTFET circuits generated by the proposed logic synthesizer tool, with the proposed gate-level design.

Comparing binary-valued circuits with ternary-valued circuits for the purpose of basic arithmetic

Several ternary-valued full adder circuits are compared with binary-valued full adder circuits, including synthesized circuits as well as circuits found in related works. These circuits are compared in terms of transistor count, power consumption, and signal delay, while scaling for the digit ratio between binary and ternary. Despite any benefits of ternary logic, a ternary-valued adding circuit could not be found to outperform an equivalent binary-valued circuit for the purpose of basic arithmetic in terms of PDP and transistor count, with the CNTFET circuits used in this thesis.

Acknowledgements

I would like to thank the Ternary Research Group at USN for their help and cooperation with my research and guidance for this thesis, especially my supervisor Henning Gundersen and co-supervisor Steven Bos for taking their time to discuss in-depth the topics covered in this thesis, and help in writing multiple research papers. Thank you to my classmates, Yasin Yari, Mohanned Hadi, and Julian Breivold Nilsen, for your friendship through good times as well as trying times, and for the amazing experiences we've had together throughout this master's course.

I would also like to give special thanks to the discoverer of caffeine and inventor of caffeinated drinks, as many late nights were spent with coffee or an energy drink as my companion to complete this thesis.

Contents

1 Introduction **1**
1 What is ternary logic? . . . 1
2 Ternary computation in the past . . . 1
3 Why use ternary logic? . . . 1
4 What are CNTFETs? . . . 2
5 The need for efficient radix converters . . . 2
6 Research methodology . . . 2

2 Theory **4**
1 Ternary logic . . . 4
2 CNTFETs . . . 10

3 Related works **14**
1 Circuit design and logic synthesis . . . 14
2 Radix conversion between binary and ternary . . . 16
3 Comparing ternary logic with binary logic . . . 17
4 Conclusion of literature review . . . 18

4 Logic Synthesis and Circuit Design **19**
1 Circuit design methodologies . . . 19
2 Risto et al. 2020 . . . 25
3 Optimizing gate-level design . . . 29

5 Radix conversion circuits **30**
1 Radix conversion circuit architecture . . . 30
2 Circuit design . . . 32
3 Simulation results . . . 34

6 Ternary versus Binary logic circuits **35**
1 General overview of synthesized functions . . . 35
2 Use cases of ternary logic . . . 37
3 Design of adder circuits . . . 38
4 Simulation results . . . 41

7 Conclusion **43**
1 Ternary-logic circuit design and synthesis . . . 43
2 Radix conversion and inter-radix compatibility . . . 44
3 Comparison of ternary and binary for arithmetic circuits . . . 44

4 Discussions 44
5 Future work 46

BIBLIOGRAPHY **48**

APPENDICES **53**

A Risto et al., 2020, SIMS **53**

B Radix converter main simulation file, radixconverter.sp **57**

C Ternary logic function circuit synthesizer, main.cpp **66**

List of Figures

2.1 Overhead over datasize 9
2.2 (a)Typical CNTFET (b)SB-CNTFET (c)MOSFET-like (d)T-CNTFET 10
2.3 Voltage-current characteristic at various CNT diameters 10
2.4 Chirality vectors of the CNT on a sheet of graphene 11
2.5 CNTFET implementations of the NTI(a), PTI(b), and STI(c) inverters 12
2.6 DC analysis of 5 STI circuits in series, compared to the input voltage and its inverse . 13

4.1 Capacitor-aided ternary circuit 19
4.2 Static ternary gate design type A and type B 20
4.3 Transistor switching table 20
4.4 Basic gate-based balanced sum 21
4.5 General MUX-based circuit 22
4.6 1-trit MUX 22
4.7 Balanced sum (a) and carry MUX (b) circuits 23
4.8 Balanced full adder circuit (a) with any MUX circuit (b) 23
4.9 2-trit decoder extension of the 1-trit decoder 24
4.10 Truth tables for arity-2 functions with heptavintimal indexing 25
4.11 Truth table synthesis for pull-up and pull-down networks for the unbalanced SUM gate 26
4.12 Circuit implementation of a pull-up network truth table with 2 groups 26
4.13 Circuit implementation of the balanced carry 27
4.14 Three proposed balanced ternary full adders "compound"(a), "non-compound"(b), "hybrid"(c) 28

5.1 Radix converter with unoptimized adders 32
5.2 Transient analysis of the optimized radix converter 34

6.1 Transistor count for all ternary-valued functions of arity 2, by index (a), by count (b) 35
6.2 Transistor count(a)(c) and PDP(b)(d) for binary-valued(a)(b) and common ternary-valued(c)(d) functions 37
6.3 Common gate-level design of binary full adder 38
6.4 Proposed ternary full adder 39
6.5 Gate-based sum 39
6.6 Capacitor-aided ternary full adder 40

7.1 Secondary optimization of the full-V_{DD} pull-up transistor network of function B7P7PBPB7 46

List of Tables

1.1 Parameters for CNTFET model . . . 3

2.1 Average radix economies for various bases . . . 4
2.2 Unbalanced and balanced ternary compared to decimal and binary . . . 5
2.3 Number of possible logic functions in binary-valued logic versus ternary-valued logic . 6
2.4 Ternary-valued truth table for unbalanced sum . . . 6
2.5 Binary-valued truth table for "XOR" . . . 6
2.6 Truth table for MAX and MIN . . . 6
2.7 Truth table for increment, decrement, NTI, PTI, STI, equalities . . . 6
2.8 Truth table for CON and ANY . . . 7
2.9 Data capacity for ternary and binary . . . 8
2.10 Ternary conversion overheads with standard binary data sizes . . . 9
2.11 Binary conversion overheads with standard binary data sizes . . . 9
2.12 Truth tables for the ternary inverters . . . 12

4.1 Truth table of balanced sum . . . 21
4.2 The heptavintimal notation . . . 25
4.3 Simulation results with 2fF load capacitor . . . 28

5.1 Digit relation . . . 31
5.2 Radix relation matrix . . . 31
5.3 Expected carry signal values from each block in the radix converter circuit . . . 33
5.4 Indices of optimized sum functions for the radix converter circuit . . . 33
5.5 Indices of optimized carry functions for the radix converter circuit . . . 33
5.6 Performance of 8-bit to 6-trit radix converter . . . 34
5.7 Input value sequences for simulation . . . 34

6.1 The 16 binary-valued functions . . . 36
6.2 Commonly used ternary-valued functions . . . 36
6.3 Simulation results of binary and ternary full adders . . . 41
6.4 Minimum digits required for a minimum counting range . . . 41
6.5 Equivalent comparison of full adders with adjusted values for the ternary circuits . . . 42

Chapter 1

Introduction

In recent years, the development of processors with higher processing power has been slowing down [1]. To continue exponential growth and miniaturization, chip manufacturers must shift to new technologies. One promising technology is Carbon Nanotube Field Effect Transistors (CNTFETs) [2] which may allow Moore's law to continue in some form, and to go beyond the limitations of current technology. The current outlook, based on the past 70 years, is that processor technology will advance while keeping the binary-value computing paradigm in place, with little consideration to multi-valued logic with more than 2 logical values for the purpose of general computation [3], outside of research. In recent years, there has been a renewed interest in ternary research with CNTFET circuits.

1 What is ternary logic?

Today, most computation and data storage is done with binary, the individual unit of which is called a bit. These bits are represented by the two values 0 and 1, hence the name "binary", or base 2. However, multi-valued logic with a radix higher than 2 can be done as well, and in fact it is used in some cases such as in digital signal processing [4][5][6]. Ternary, represented by trits, makes use of three values for each digit with the logical values of 0, 1, and 2.

2 Ternary computation in the past

In 1958, a balanced ternary-valued computer was developed at Moscow State University. This was the Setun [7]. This computer was developed to fulfill the computational needs of the university, and fifty computers were built, until production was halted in 1965. In recent years, there has still been a small amount of research into software development for these types of computers [8]. One reason that binary-valued logic has been dominating over ternary-valued logic in the past is the difficulty and cost of engineering silicon-based transistor circuits as these require large components such as resistors to achieve three stable voltage levels[3]. However, recent work such as [9] demonstrate by circuit simulation that CNTFET-based logic circuits can be engineered to have three stable voltage levels by varying the width of the carbon nanotubes, with no need of extra components other than transistors.

3 Why use ternary logic?

It has been suggested in theory that a ternary-value computer, or ternary data processing components in general, may be more cost-efficient than an equivalent binary version. This has been suggested with the calculation of the radix economy [10][11], and the more compact representation of information itself (eg. 8 bit information can be stored in 6 trits, a saving of 25%). Another possible advantage is the much higher number of 19683 possible binary operators in 3-valued logic, compared to 16 in 2-valued logic [12]. This larger number of binary operators may allow for further-reaching abstractions than in

binary logic. Some functions such as ternary state machines might benefit from natively ternary states. In safety-critical systems, ternary-value computation may provide a benefit in allowing digits to be in an "uninitialized" stage, with the two remaining digit values functioning as binary values, to ensure that uninitialized values are not read as data values [13]. For safety-critical systems, ternary logic may also be used with ternary decision diagrams [14][15].

4 What are CNTFETs?

Carbon NanoTube Field-effect Transistors (CNTFETs) are a type of transistor which can be built with different voltage thresholds by changing the diameter of the carbon tube. This allows for multi-valued logic circuits such as ternary, without the need for extra components such as resistors to create the middle voltage level [9].

Example theorized or realized ternary chips are AI chips [16], graphical co-processors in a GPU, ternary based security [13], or ternary blockchain chips for IOTA [17].

5 The need for efficient radix converters

It is imaginable that specialized ternary chips will be integrated next to binary chips and pre-existing technology. This strongly benefits from a hardware radix conversion solution to minimize the conversion penalty of delay, as opposed to converting the inputs and outputs via microprocessors and software. To enable ternary technologies to develop in the near future, solutions for efficient radix conversion must first be developed, so that ternary components may be integrated in a mixed-radix system without too much conversion penalty.

Potentially, in later adoption stages of ternary-valued logic, the need for binary to ternary conversion chips will still be relevant for backwards compatibility or for mixed radix systems where binary and ternary values may coexist. To facilitate the feasibility of mixed radix systems, the signal delay through the proposed converters must be as low as possible, and should be scalable, to allow for fast and large-scale radix conversion of large quantities of data on the fly.

6 Research methodology

6.1 Research questions

A set of research questions is asked to guide the research project.

- 1. How can CNTFETs circuits perform multi-valued logic computation, and what methods are there to design the ternary circuits on the transistor and gate level?
- 2. How can compatibility be achieved between ternary-valued and binary-valued circuits? To enable ternary technology to develop into practical uses, it has to be compatible with existing technology. How can the data of the established binary radix be converted into the ternary radix efficiently, and what are the conversion penalties in terms of signal-delay, power consumption, transistor count?
- 3. How does ternary-valued logic circuits perform compared to binary-valued circuits? Can the benefits be shown to exist with circuits such as basic arithmetic circuits in terms of PDP?

To answer the research questions posed in this thesis, the primary methodological approach was to automate a logic synthesis algorithm, and simulate the circuits in HSPICE, after which they can be examined objectively. Using this same method for both binary-valued and ternary-valued circuits results in a more fair comparison, as binary-valued technologies have the benefit of many decades of aggressive optimization. Additionally, related research was also recreated and compared.

6.2 CNTFET circuit simulation

For simulating CNTFET circuits two components are integral, a SPICE circuit simulator and a CNTFET simulation model. Two SPICE simulators were considered: T-Spice from the Tanner software suite, [18] and HSPICE from the Synopsys software suite [19]. Two simulation models of CNTFETs are provided by Stanford University: the standard 32nm CNFET model [20], and the more compact VS-CNFET model [21]. The VS-CNFET was attempted to be used with T-Spice, with some functional circuits achieved, however did not produce sufficient correctness and functionality in circuit measurements when compared with existing research. As the models are optimized for use with HSPICE, T-Spice produced frequent convergence issues. Through experiment, the best combination was found to be the standard 32nm CNFET model in combination with HSPICE, which is also the most commonly used CNTFET model and SPICE software in research. The CNFET parameters used in this thesis is shown in table 1.1.

Table 1.1: Parameters for CNTFET model

Parameter	Value	Description
L_{ch}	32e-9	Physical channel length
L_{geff}	100e-9	The mean free path in the intrinsic CNT channel region due to non-ideal elastic scattering
L_{ss}	32e-9	The length of doped CNT source-side extension region
L_{dd}	32e-9	The length of doped CNT drain-side extension region
T_{ox}	4e-9	The thickness of high-k top gate dielectric material
Kox	16	Gate oxide dielectric constant
Efi	0.6	The Fermi level of the doped S/D tube
C_{sub}	40e-12	The coupling capacitance between the channel region and the substrate
Pitch	20e-9	The distance between the centers of two adjacent CNTs
Tubes	3	Number of CNTs
Supply	0.9	Gate supply voltage
Temp	25	Temperature

6.3 Circuit measurement methodology

With H-SPICE, the performance characteristics or a circuit can be measured.

Transistor count is the number of transistors used in a circuit. For sub-circuits that require extra external components such as input inverters, both the sub-circuit count and total count is included. Current measurement is measured at the voltage supply, with an ideal 0V supply in series with the circuit. To find the power in Watt, multiply by the supply voltage (0.9V). As the inputs may have half-voltage (0.45V), the currents should not be directly added. However, in most circuits the inputs do not produce a significant current, as they are only used as a transistor gate voltage. In the event that they do produce power, they should be measured separately to find the correct power consumption. In the event of alternating current direction, only one way of the current is measured for each voltage source, to prevent duplicate measurements of the same current going through multiple voltage sources. The worst-case delay is the maximum time the circuit takes from a change in inputs to a stable and correct output. It is the worst-case which limits the frequency at which the circuit can be functional. The PDP is the power-delay product, which is found by multiplying the power consumption with the delay of the circuit. For the best-case PDP, the worst-case delay is used. As a circuit output load, a capacitor can be used, or alternatively an FO4 fan out load consisting of 4 inverters in series, to simulate the sub-circuit of interest in the context of a larger circuit with external capacitance.

Chapter 2

Theory

This chapter will cover the basic theory of ternary logic and CNTFETs.

1 Ternary logic

In the past, there have been claims of ternary being the optimal radix [3], as well as some counterarguments against these claims [22]. There are several reasons ternary-valued logic might have benefits over binary-valued logic.

1.1 Radix economy

The radix economy is a measure of the cost of expressing numbers in a given base. This is found by taking the number of digits required to express a number and multiplying by the base [23]. For the general case of base b and number N, this can be expressed as in equation 2.1.

$$E(b, N) = b * \lfloor log_b(N) + 1 \rfloor \tag{2.1}$$

The radix economy of different bases can be compared by taking the average of the radix economy for a given base b with an increasing N up to a large number. The average is used, since data overhead will cause noise in the data. As N increases, the significance of this overhead will decrease.

Table 2.1: Average radix economies for various bases

Base b	Avg. E(b,N) N = 1 to 6	Avg. E(b,N) N = 1 to 43	Avg. E(b,N) N = 1 to 182	Avg. E(b,N) N = 1 to 5329
1	3.5	22.0	91.5	2665.0
2	4.7	9.3	13.3	22.9
e	4.5	9.0	12.9	22.1
3	5.0	9.5	13.1	22.2
4	6.0	10.3	14.2	23.9
5	6.7	11.7	15.8	26.3

According to these equations, the optimal radix can be found to be e, or 2.718, with base-3 and base-2 following in optimality, as seen in table 2.1. Since digital logic operates on integer bases, it follows that ternary-valued logic is the optimal base for computation. One flaw with this logic is the assumption that the number of required components scales proportionally with the base, i.e. an increase by a factor of 1.5 from binary to ternary. This will be investigated further in later chapters.

1.2 Balanced and unbalanced ternary notation

There are mainly two methods for expressing a number in ternary. The most straight-forward notation is unbalanced ternary, which is analogous to unsigned binary numbers. It is expressed with the symbols '0', '1', '2'. The other, more elegant notation, is balanced ternary, which is expressed with the symbols '-1', '0', '+1', or '-', '0', '+' as a short form [24][25]. Donald Knuth, a renowned computer scientist, writes in his book "The Art of Computer Programming", that "Perhaps the prettiest number system of all is the balanced ternary notation, which consists of radix-3 representation using −1, 0, and +1 as "trits" (ternary digits) instead of 0, 1, and 2". Furthermore, he stated that "If it would have been possible to build reliable ternary architecture, everybody would be using it." [26].

Table 2.2: Unbalanced and balanced ternary compared to decimal and binary

Decimal	Binary	Unbalanced	Balanced
-3			0 - 0
-2			0 - +
-1			0 0 -
0	000	000	0 0 0
1	001	001	0 0 +
2	010	002	0 + -
3	011	010	0 + 0
4	100	011	0 + +
5	101	012	+ - -
6	110	020	+ - 0

A comparison of unsigned binary, unbalanced and balanced ternary is shown in table 2.2. While binary requires a sign-bit to express negative numbers, balanced ternary can express both positive and negative numbers, while utilizing the full capacity of the trits. A result of this is a halved range in the positive value range compared to unbalanced, analogous to signed and unsigned binary numbers. Note that balanced ternary sometimes requires one more digit than the unbalanced ternary. To convert balanced ternary between positive to negative, all the digits are inverted, or in other words $+$ is swapped with $-$ and vice versa [27].

Despite the fault of having half the capacity of unsigned ternary numbers, balanced ternary makes up for it with arithmetic convenience[28][29], such as simpler addition and subtraction, and the capability of representing negative numbers.

Since the established norm with binary logic is that 1 represents 'true', it is also sometimes used as 'true' in unbalanced ternary values, with 2 representing 'unknown' or 'undefined'. It is more common to use 2 as 'true', as it is easier to operate on full binary voltage, while the middle voltage level may be used either to represent an uninitialized value, i.e. "unknown" [10], or a logical middle value with radix-3 data encoding.

1.3 Logic functions

A logic function, also known as a logic gate, is an abstract element which takes a number of digit inputs and outputs one digit output. With these logic functions, any logical operation can be done [30]. The number of parameters or inputs a function has, is known as the functions "arity". This arity can for example be 1, 2, or 3, which are known as unary, binary, and ternary functions respectively [31]. Note that the arity of a function does not imply the radix of the digit inputs to the function.

The number of possible functions for a specific arity and radix can be calculated as in equation 2.2, where R is the radix and A is the arity.

$$F_{range} = R^{(R^A)} \tag{2.2}$$

An increase of the radix by one offers an extreme increase of number of possible functions as the arity increases, as shown in Table 2.3.

Table 2.3: Number of possible logic functions in binary-valued logic versus ternary-valued logic

Arity	Radix 2	Radix 3
1	$2^{2^1} = 4$	$3^{3^1} = 27$
2	$2^{2^2} = 16$	$3^{3^2} = 19683$
3	$2^{2^3} = 256$	$3^{3^3} = 7{,}625{,}597{,}484{,}987$

A logic function can be expressed in terms of a truth table. Tables 2.4 and 2.5 show the truth tables for the ternary-valued unbalanced sum, and the binary-valued XOR, both of which are used to find the sum of two digits, a and b.

Table 2.4: Ternary-valued truth table for unbalanced sum

a	b	Sum
2	2	1
2	1	0
2	0	2
1	2	0
1	1	2
1	0	1
0	2	2
0	1	1
0	0	0

Table 2.5: Binary-valued truth table for "XOR"

a	b	XOR
1	1	0
1	0	1
0	1	1
0	0	0

1.4 Ternary logic algebra

Like with boolean algebra for binary logic, a similar ruleset can be defined for decomposing ternary logic into smaller functions. With some base functions, such as MIN, MAX, increment, decrement, equalities, as well as the inverters NTI, PTI, STI, any ternary-valued logic function can be constructed [32]. The truth table of these functions are shown in tables 2.6 and 2.7. Note that here, the unbalanced notation is used, however these rules apply to both balanced and unbalanced, with $-0+$ being interchangable with 012 for the purposes of a logic truth table.

Table 2.6: Truth table for MAX and MIN

a	b	MAX	MIN
2	2	2	2
2	1	2	1
2	0	2	0
1	2	2	1
1	1	1	1
1	0	1	0
0	2	2	0
0	1	1	0
0	0	0	0

Table 2.7: Truth table for increment, decrement, NTI, PTI, STI, equalities

a	incr.	decr.	NTI	PTI	STI	eq0	eq1	eq2
2	0	1	0	0	0	0	0	2
1	2	0	0	2	1	0	2	0
0	1	2	2	2	2	2	0	0

From table 2.7, it can be seen that NTI is the same function as equal 0, and that PTI is the inverse of equal 2, and STI is the inverse. Thus, we can write the functions used in ternary logic algebra as in equations 2.3 to 2.11

$$STI(a) = -a \tag{2.3}$$

$$NTI(a) = (a = 0) \tag{2.4}$$

$$Equal1(a) = (a = 1) \tag{2.5}$$

$$Equal2(a) = (a = 2) \tag{2.6}$$

$$PTI(a) = -(a = 2) \tag{2.7}$$

$$Increment(a) = a + 1 \tag{2.8}$$

$$Decrement(a) = a - 1 \tag{2.9}$$

$$MAX(a, b) = a \vee b \tag{2.10}$$

$$MIN(a, b) = a \wedge b \tag{2.11}$$

As in boolean algebra, DeMorgan's theorem still holds for ternary-valued logic [25], shown in equation 2.12.

$$-(a \wedge B) = -a \vee -b \tag{2.12}$$

Furthermore, MAX and MIN are distributive, the same way AND and OR is in boolean algebra [25], shown in equation 2.13

$$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c) \tag{2.13}$$

With these rules, any ternary-valued logic function can be implemented with the basic functions listed. One method to achieve this is to write out the canonical sum of products for the functions, and optimize the equation. An example of this for the balanced sum function is given in chapter 4.

Two more commonly used basic logic functions, Consensus and Any, can be defined here. They can be defined in terms of the other basic functions as in equation 2.14 and 2.15, and therefore they are not strictly needed as basic functions. However due to their ease of direct implementation they are included here nonetheless. Their truth tables are shown in table 2.8.

Table 2.8: Truth table for CON and ANY

a	b	CON	ANY
2	2	2	2
2	1	1	2
2	0	1	1
1	2	1	2
1	1	1	1
1	0	1	0
0	2	1	1
0	1	1	0
0	0	0	0

$$\begin{aligned} CON(a, b) = a \otimes b = (a \wedge b) \\ \vee((-(a = 0)) \wedge 1) \\ \vee((-(b = 0)) \wedge 1) \end{aligned} \tag{2.14}$$

$$\begin{aligned} ANY(a, b) = a \oplus b = ((a = 0) \wedge (b = 2) \wedge 1) \\ \vee((a = 1) \wedge b) \\ \vee((a = 2) \wedge ((b = 0) - 1)) \end{aligned} \tag{2.15}$$

1.5 Data sizes and overhead

One clear benefit of ternary-valued logic compared to binary-valued logic is the higher data capacity per digit, which is a given since it is a higher base. This has implications for data storage, transmission, and processing, as data may be expressed more efficiently. The data capacity for binary and ternary scales exponentially as the number of digits increases, as can be seen in equation 2.16 where b is the base, and n is the data size. A comparison between binary and ternary is shown in table 2.9.

$$b^n \qquad b, n \in \mathbb{N} \tag{2.16}$$

Table 2.9: Data capacity for ternary and binary

Data size	Binary	Ternary
1	$2^1 = 2$	$3^1 = 3$
2	$2^2 = 4$	$3^2 = 9$
3	$2^3 = 8$	$3^3 = 27$
4	$2^4 = 16$	$3^4 = 81$
5	$2^5 = 32$	$3^5 = 243$
6	$2^6 = 64$	$3^6 = 729$
7	$2^7 = 128$	$3^7 = 2187$
8	$2^8 = 256$	$3^8 = 6561$

When converting between binary and ternary data, one might want to consider different data sizes to convert. One aspect to consider is the overhead, which is the difference in capacity of the binary number and the ternary number. The capacity of an unsigned number can be expressed as the base to the power of the number of digits. If signed, the capacity is roughly halved. Examine the equation (2.17).

$$2^n = 3^m \qquad m, n \in \mathbb{N} \tag{2.17}$$

One consequence of this equivalence is the ratio between the number of digits required for the two bases. From equation 2.17, equation 2.18 can be derived, which implies that one trit can replace approximately 1.5849 bits.

$$n = \frac{m * log(3)}{log(2)} \approx 1.5849 * m \tag{2.18}$$

Another consequence is that while both m and n are integers above zero, there are no integer solutions to equation 2.17, since 2^n will always be even, and 3^m will always be odd. This means that some overhead is unavoidable when converting between binary and ternary numbers. However, the overhead can be minimized by choosing the right data sizes.

Optimized data sizes

When considering radix conversion, it is here assumed that the ternary data size is larger in capacity than the binary. Data sizes with little overhead may be considered, such as 84 bit to 53 trit. This would have a mere 0.2 percentage overhead, as shown in (2.19).

$$100\frac{3^{53} - 2^{84}}{3^{53}} = 0.208595 \tag{2.19}$$

These data sizes could be used in cases where it is integral to keep overhead to a minimum, such as data storage, or efficiency-critical data processing components. There are also larger pairs of data sizes with even lower overhead. If a large increase of data sizes is allowable, an overhead arbitrarily close to 0 can be achieved. This can be shown to be evident with the following method.

$$overhead(m) = \frac{3^{\lfloor m \rfloor} - 2^{\lfloor log_2 3^{\lfloor m \rfloor} \rfloor}}{3^{\lfloor m \rfloor}} \tag{2.20}$$

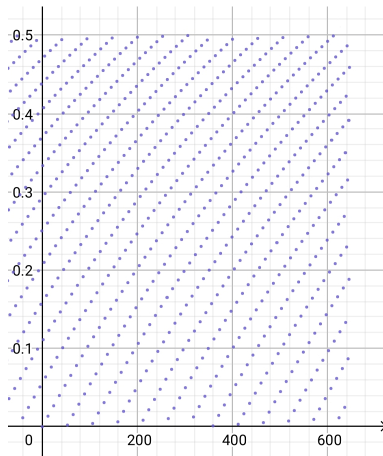


Figure 2.1: Overhead over datasize

The graph in figure 2.1 can be plotted from equation 2.20. The points in this graph each represent the conversion overhead, represented by the y-axis, of a ternary data size, represented by the x axis. It is clear that this is not a function which converges to one value, and therefore taking the limit of this function is a dead end. Instead, it is observed that all the points land on a specific set of arcs. Furthermore, it is known that the points are only on whole numbers along the x-axis. Thereby, the off-set positioning of the points on each arc is defined by the distance between the arcs. If this distance is irrational, then there are no repeating patterns, and it can be assumed that every rational overhead will be covered as the data size approaches infinite. This is most likely the case, since the log of integers produces irrational numbers.

However, a generally useful converter to be used in standard binary systems should adhere to the standard binary data sizes, which generally have a number of bits which are a power of two.

Compatibility with powers of two

If this overhead exceeds the benefits of ternary, it might not be advantageous to use binary data sizes in a ternary setting. Depending on the choice of data sizes, the overhead may be on the ternary side, or the binary side. The radix with overhead must be limited in range to not exceed the limit of the other radix. For this thesis, it is assumed that the ternary data size is bigger than the binary, however here overheads are given for both cases. Tables 2.10 and 2.11 show the data conversion overhead for standard binary data sizes, for the cases of $2^n < 3^m$, and $2^n > 3^m$ respectively.

Table 2.10: Ternary conversion overheads with standard binary data sizes

Bits	Trits	Overhead
2^8	3^6	64.88%
2^{16}	3^{11}	63.00%
2^{32}	3^{21}	58.94%
2^{64}	3^{41}	49.42%

Table 2.11: Binary conversion overheads with standard binary data sizes

Bits	Trits	Overhead
2^8	3^5	5.08%
2^{16}	3^{10}	9.90%
2^{32}	3^{20}	18.82%
2^{64}	3^{40}	34.09%

2 CNTFETs

Carbon Nanotube Field-Effect Transistors (CNTFET) is a highly promising transistor technology subject to a lot of research in recent years. One recent major milestone in CNTFET research was MIT's RISC-V processor, RV16XNano, which contains 14 000 CNTFETs[2].

2.1 Architecture of CNTFETs

Carbon Nanotube Field-effect Transistors (CNTFET) is a type of FET, which uses semi-conducting carbon nanotubes (CNTs) as the channel, depicted in figure 2.2 from [33], instead of bulk silicon like in traditional field-effect transistors like MOSFETs. These CNTs have the same atomic structure as graphene, formed into a tube, usually with one wall a single atom thick. While similar in function to a MOSFET, they've been shown to have significant benefits in terms of performance [34], which may facilitate more compact processor designs, such as 3D-VLSI [35]. A CNT acts as a one-dimensional electron path, resulting in no electron scattering. CNTs have certain benefits over bulk silicon, like a better threshold voltage and sub-threshold slope, a high electron mobility due to the one-dimensional nature of the single-atom thick nanotube resulting in quasi-ballistic transport of electrons, high current density compared to bulk silicon, high linearity in the relationship between voltage and current, and high transconductance [36][37]. The manufacturing of these CNTFETs can use a similar process as MOSFETs, with the exception of the carbon nanotubes, which can be grown or deposited during the manufacturing process [38]. Like with MOSFET transistors, n-type and p-type transistors can be achieved by different chemical dopings in the channel [39].

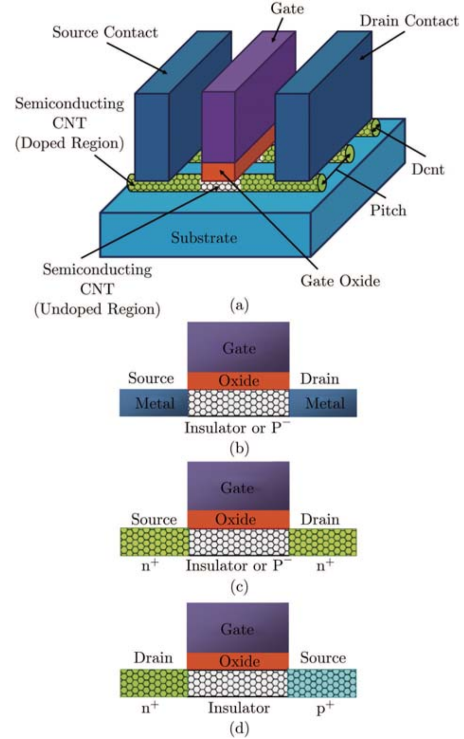


Figure 2.2: (a)Typical CNTFET (b)SB-CNTFET (c)MOSFET-like (d)T-CNTFET

2.2 Voltage characteristic and threshold

CNTFETs are useful for ternary-valued logic because of the fact that the voltage threshold of a specific transistor is dependant on the diameter of its carbon nanotubes. By using a combination of different voltage thresholds in the same circuit, multi-valued logic can be achieved. The voltage threshold V_{th} can be found with equation 2.21, where d_0 is the carbon-to-carbon bond distance in carbon nanotubes, and V_π is the carbon π-π bond energy [40]. Figure 2.3 shows the voltage-current characteristic of CNTFETs of various diameters, simulated in HSPICE. The simulations show that the voltage-current characteristic of the CNTFET matches up with the equation 2.21.

$$V_{th} \approx \frac{d_0 V_\pi}{e D_{CNT}} \approx \frac{0.43v}{D_{CNT}[nanometer]} \tag{2.21}$$

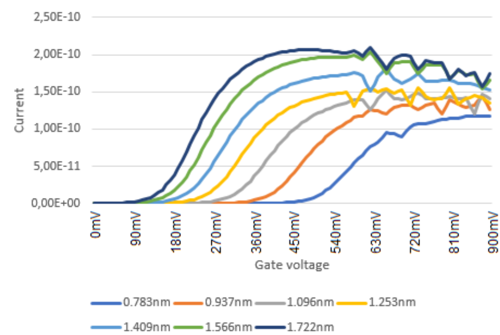


Figure 2.3: Voltage-current characteristic at various CNT diameters

2.3 Carbon nanotube chirality

The voltage threshold of a transistor can be chosen by specifying the nanotube diameter. The diameter and structure of the CNT can be expressed in terms of a chirality vector. The chirality vector $\vec{C}_h$ symbolizes the two points which connect when the graphene is formed into a nanotube, i.e. the circumference of the tube. It is defined by the base vectors $\vec{a}_1$ and $\vec{a}_2$ and integer scalars (n,m) [41], as in equation 2.22.

$$\vec{C}_h = n\vec{a}_1 + m\vec{a}_2 \tag{2.22}$$

This chirality vector also indicates if the carbon nanotube is metallic or semi-conducting, which is a result of the structure of the tube [20]. For the purpose of CNTFETs, the chirality vector is restricted to semi-conducting values. The metallic CNTs have a structure called "Armchair", which is the case when n and m are equal, while the structure of semiconducting CNTs are called "Zigzag", which are when one scalar is 0, while the other is free to vary to specify the CNT diameter. For CNTFETs, the Zigzag pattern is utilized to produce semiconducting CNTs with variable diameter. Figure 2.4 shows the base vector, with the "Armchair" and "Zigzag" patterns on the atomic structure of flat graphene.

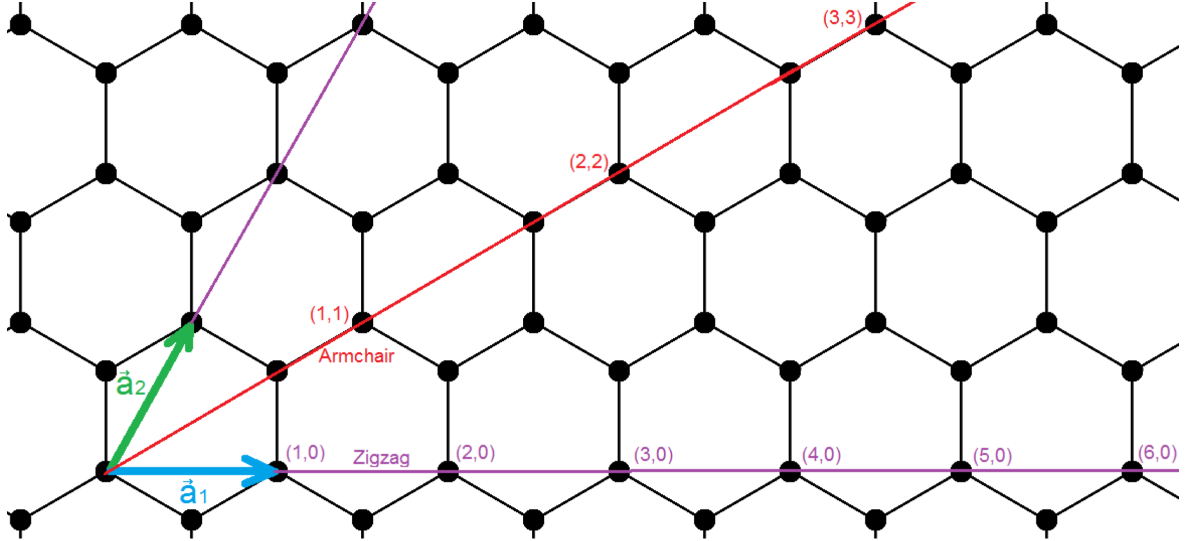


Figure 2.4: Chirality vectors of the CNT on a sheet of graphene

Since the chirality vector represents the circumference of the tube, equation 2.23 shows the relation between diameter of the CNT and the base vector scalars, where d_0 is the carbon-carbon atomic bonding distance.

$$D_{CNT} = \frac{\sqrt{3}d_0}{\pi}\sqrt{m^2 + mn + n^2} \approx 0.0783\sqrt{m^2 + mn + n^2} \quad [nanometer] \tag{2.23}$$

To ensure the semi-conductivity of the CNT, m is set to 0 while n varies from transistor to transistor. Thus the equation for the diameter can be simplified to equation 2.24.

$$D_{CNT} \approx 0.0783 * n \quad [nanometer] \tag{2.24}$$

2.4 Viability for ternary logic circuits

While the transistors themselves are inherently binary (on or off), due to the properties of the carbon nanotube, the voltage threshold of individual transistors can be specified by selecting a precise carbon nanotube diameter. With this knowledge, in combination with knowing how a binary inverter functions, we can build two of the three ternary inverters; the Positive Ternary Inverter (PTI), and the Negative Ternary Inverter (NTI). These two circuits can be combined to produce the Standard Ternary Inverter (STI) [9]. Their truth tables are shown in table 2.12, and circuit implementations in figure 2.5.

Table 2.12: Truth tables for the ternary inverters

a	NTI	PTI	STI
2	0	0	0
1	0	2	1
0	2	2	2

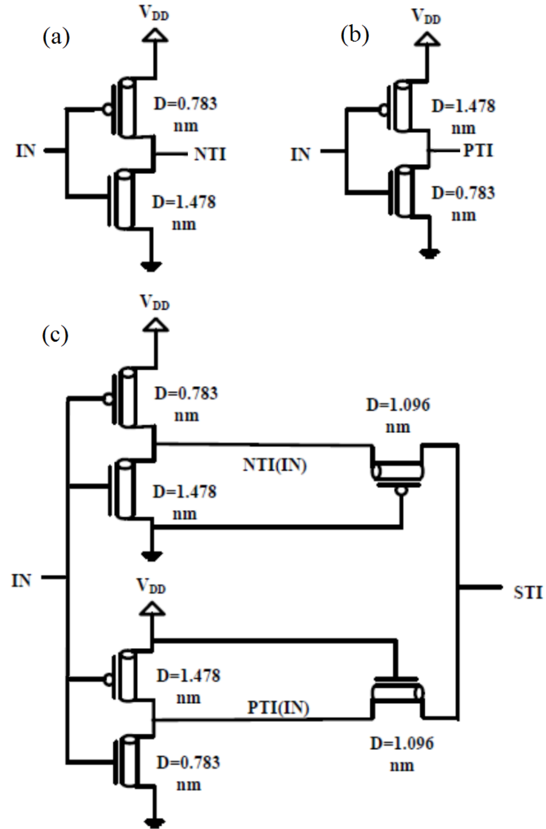


Figure 2.5: CNTFET implementations of the NTI(a), PTI(b), and STI(c) inverters

This circuit demonstrates that ternary-valued logic circuits can be achieved with complementary circuits with binary-valued outputs, such as NTI and PTI. The middle voltage is achieved through what is essentially voltage division, which naturally results in a higher power consumption. When NTI outputs '0', and PTI outputs '2', there is a path between V_{DD} and GND. This current is limited by the series resistance of the two 1.096nm transistors connecting NTI and PTI. This has been cited as one of the main flaws of ternary-valued logic in CNTFET logic circuits [22].

Next, a question of the stability of the middle value comes up. What happens if many STIs are connected in series? Will the voltage drift from the middle value over time? To show that CNTFET circuits such as the STI can produce a stable middle value, the input/output voltage characteristic of the STI can be generated in simulation using the 32nm CNFET model, which can be compared with that of an ideal linear voltage inverter, which in theory would have no voltage drift in any direction when connected in series. For these circuits, the voltage range used is 0V to 1V. By overlaying the two voltage curves, it becomes clear in which areas the voltage is serially pushed upwards or downwards by the STIs in series. For example, while the ideal voltage inverter would in theory invert 900mV to 100mV, the curve of the input/output characteristic of the STI lies below this line, meaning 900mV outputs a value lower than 100mV, and similarly for an input of 100mV, is inverted by the STI to a voltage higher than 900mV. The consequence of this, is that voltages outside the thresholds of the middle value, which is the area between the two furthest points where the two curves cross, will drift towards the outer values, while the voltages within these thresholds will drift towards the value at the middle crossing point. A comparison of 5 serial STIs and the inverse voltage of the input is show in figure 2.6.

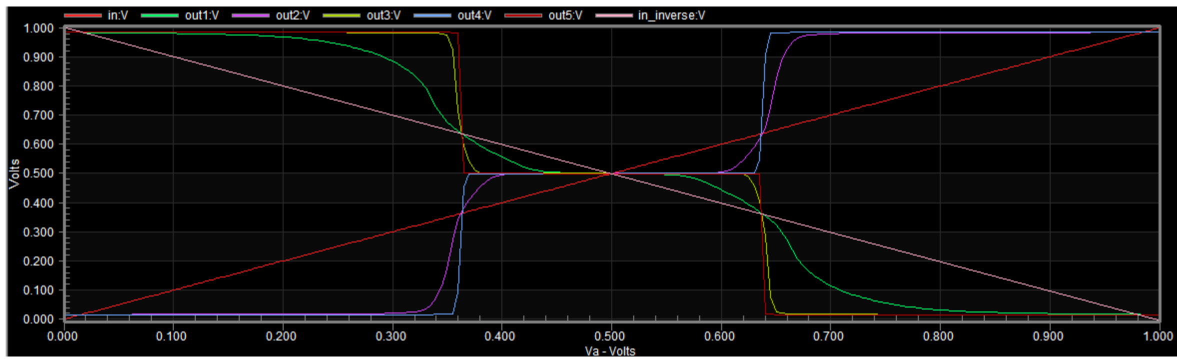


Figure 2.6: DC analysis of 5 STI circuits in series, compared to the input voltage and its inverse

The DC analysis shown in figure 2.6 demonstrates that for complementary ternary-valued circuits such as the STI inverter, the voltage levels are self-regulating, meaning voltage drift from the middle value to the high or low value is not an issue. In this example, the voltage thresholds are at 360mV and 630mV, inside which the voltage will drift towards the middle value of 0.5V, and outside of which the voltage will drift towards 0V and 1V.

Chapter 3

Related works

Before we continue, a brief literature review is done, which summarizes some pre-existing research and what their significances are in relation to this thesis. Like this thesis, the literature review is divided into three categories; Circuit design and logic synthesis, Binary-ternary data radix conversion, Comparisons of ternary-valued and binary-valued logic.

1 Circuit design and logic synthesis

Six papers or studies are identified, which present different methods of ternary-valued circuit design.

1.1 Stanley L. Hurst, 1984

Multiple-Valued Logic - Its Status and Its Future [3] surveys the possibility of MVL logic as opposed to binary, to increase information density in digital signals. pre-existing circuit implementations are discussed, realized with a CMOS-resistor design, which can be used to create combinatorial logic circuits with the Min, Max, and Unary logic gates. They consider future possible technologies, however this paper was published before the first demonstration of a CNTFET in 1998 [42][43]. They call the fact that no true multistate device higher than binary exists the major restraint to MVL technologies.

1.2 Douglas W. Jones, 2012

The Ternary Manifesto [25], by Douglas Jones from the University of Iowa Department of Computer Science, is a blog detailing ternary logic and how it could be implemented. He takes security-through-obscurity as a fundamental principle, to say that ternary should have security benefits over binary. He postulates that ternary logic, due to the fewer number of digits needed, should reduce the density of interconnect wiring, which is one of the main limiting factors of today's technology [3][44][45]. This benefit may possibly outweigh any additional cost of implementing more complex ternary circuits compared to binary.

The blog covers ternary logic, gate implementations with ternary logic algebra, ternary arithmetic, number and character encoding, heptavintimal encoding of triplets of trits, and proposes an instruction set computer architecture for ternary logic.

1.3 Chetan Vudadha et al. 2013

2:1 Multiplexer Based Design for Ternary Logic Circuits [46] investigates the usage of multiplexers to design a ternary full adder. They connect 1-trit multiplexers to produce the circuits used in a half adder, and by extension a full adder. In combination with ternary logic algebra, this method can be used to create combinatorial logic circuits to achieve any logic function.

1.4 Mohammad Hossein Moaiyeri et. al. 2015

An efficient ternary serial adder based on carbon nanotube FETs [47] proposes a ternary serial full adder with a clocked flip-flop gate. Interestingly, the full adder is implemented with the unary logic gates of an STI inverter, with shifted voltage thresholds, and capacitors to take averages of voltages for the input to the unary gates. This allows for a very low transistor count, however it relies on extra components which take up space in a circuit, and which may cause a larger power consumption due to high-frequency current leakage through the capacitors, as well as high delay due to the capacitance required to produce a functional voltage in the nodes of the circuit. Therefore, its efficiency is debatable.

1.5 Sunmean Kim et al 2017

An Optimal Gate Design for the Synthesis of Ternary Logic Circuits [48] proposes a method of synthesising ternary-valued logic gates using a static gate design, based on pull-up and pull-down networks The paper claims that their method produces a minimal number of transistors for logic gates. The optimality is debatable, as other papers have produced full adders with a lower transistor count [49]. They show an decrease in power-delay product compared to earlier work. They claim that the reason ternary logic has not been shown to outperform binary logic is due to the lack of optimization of the ternary circuits. However, they do not show a comparison of their results with binary arithmetical circuits. This method is applicable for the synthesis of any ternary-valued logic gate with a good PDP performance.

1.6 Mingqiang Huang et. al. 2019

Design and Implementation of Ternary Logic Integrated Circuits by Using Novel Two-Dimensional Materials [50] investigates design of ternary-valued logic through ternary algebra with functions such as MIN and MAX, with the use of two-dimensional materials as opposed to CNTFETs. They argue that in large scale circuits, approximately 70 percent of the chip area is designated for wire interconnections, thus a significant decrease of interconnections can give a strong boost to the performance of a chip.

They show a 50% reduction in number of transistors in 2-dimensional-based transistor circuits compared with research done with silicon-based and CNT-based transistor circuits for an unbalanced ternary full adder, which may give a very significant decrease in interconnection density.

2 Radix conversion between binary and ternary

Papers which cover radix conversion are reviewed, particularly ones which relate to binary-ternary radix conversion.

2.1 Fu-Qiang et al., 1995

A binary to balanced ternary converter based on Josephson junctions [51] makes use of Superconducting Quantum Interference Device (SQUID) gates to construct a binary to ternary radix converter circuit. This specific circuit requires superconductors at low temperatures. However, their method of binary to ternary radix conversion can be applied to CNTFET circuit design. By decomposing the arithmetic relation between binary and ternary digits, each ternary output can be defined as the sum of a set of binary inputs, with carry signals propagating from the least significant trit to the most significant trit.

2.2 Sasao, 2005

Radix converters: complexity and implementation by LUT cascades [52] assesses the design methods of general n-ary to q-ary data radix converters with the use of look-up table cascades. They use column multiplicity to mathematically deduce the lower bound of complexity for such a circuit.

This work was continued in [53] and [54], where arithmetic decomposition is used to design the converters.

2.3 Arjmand et al., 2012

In [55] an unbalanced ternary to unsigned binary converter for Quantum-dot Cellular Automata (QCA) is proposed. There are four possible states of a ternary QCA cell, two of which are zero, each of which are mapped on to two bits. While this same method could generally be implemented in a ternary logic circuit, this is not a capacity efficient conversion method, as two bits are used to store one trit, while the optimal tends towards approximately 1.58496 bits per trit, and therefore is not applicable to an efficient binary-ternary radix converter based on CNTFETs.

2.4 Shahangian et al., 2019

Design of a multi-digit binary-to-ternary converter based on cntfets [56] proposes a design method for unsigned binary to unbalanced binary, using the arithmetic decomposition method to produce the binary-ternary digit relation. Optimized adders are constructed with some unary circuits such as increment, decrement, and the inverters, as well as compact optimized 1-trit MUX circuits, and carry generator. These circuits are optimized in regards to the expected values of the inputs. As an example, the mux may only need to decode the select trit to two possible values instead of the full three, depending on the exact circuit. They show performance results for various digit sizes, including 8-bit to 6-trit.

2.5 Shahangian et al., 2020

Universal Method for Designing Multi-Digit Ternary to Binary Converter Using CNT-FET [57] is a continuation of [56], and proposes a method of converting unbalanced ternary to unsigned binary. They use similar methods, except that this circuit decodes the individual trits to binary signals to process them with binary adder circuits. They show simulation results, which are significantly better than their binary-ternary results.

3 Comparing ternary logic with binary logic

3.1 Hande Alemdar et al. 2017

Ternary Neural Networks for Resource-Efficient AI Applications [58] proposes a TNN (ternary neural network) with ternary neuron activations using a step function with two thresholds. With the levels used for weights and activation of -1, 0, and +1, they prune smaller weights by setting them to 0 during training, which makes them sparser, and thus makes the neural network more energy-efficient, compared to the binary counterpart of -1 and +1. Instead of encoding this data directly in balanced ternary with MVL circuits, the balanced ternary digits is encoded in binary data [59]. They consider efficient encoding and decompression of the balanced ternary digits in binary [60]. This will involve some overhead between the data space used and the actual symbols represented with bits. They do not consider the use of natively ternary-valued circuits.

3.2 Daniel Etiemble, 2019

Ternary circuits: why R = 3 is not the Optimal Radix for Computation [22] claims to have disproven that ternary-valued computation is superior to binary-valued computation. They review other existing research without providing simulation data themselves for power consumption or signal delay, with mainly comparisons of transistor count of equivalent binary and ternary circuits, which could possibly outweigh the higher transistor count needed to implement a more complex truth table. They argue that a higher transistor count overall can only lead to a higher interconnection length, however studies such as [61] show that the interconnection length can be reduced with ternary logic circuits.

While they refute the claim of radix economy implying ternary to be optimal, they do not refute the potential of ternary technologies over binary technologies in every aspect, as they do not properly consider factors such as PDP and interconnection wire length densities, and only cover some operators such as sum, and not multipliers. They have only shown that ternary has not yet been shown to outperform binary, due to engineering challenges.

3.3 Kiyung Kim et al., 2020

Extreme Low Power Technology using Ternary Arithmetic Logic Circuits via Drastic Interconnect Length Reduction [61] implements cell layouts for a balanced full adder and a multiplier, using the static gate design of [48]. Their balanced full adder is identical to the one proposed in [48], as well as the "compound" balanced full adder described in this thesis in chapter 4 section 2.3. They measure the interconnect wire length of the adder and the multiplier, for 5 trits, and compare it with the wire length for an 8-bit binary-valued adder and multiplier. They show that the ternary 5-trit multiplier significantly outperforms the 8-bit binary multiplier with a 37% reduction in length, however the ternary adder is slightly outperformed by the binary adder. They do not investigate the PDP of the circuits with circuit simulation, or consider the transistor count.

4 Conclusion of literature review

There are multiple methods for designing circuits, such as the static gate design with pull-up and pull-down networks, circuits built from unary gates and voltage-averaging capacitor, circuits based on MIN and MAX gates, and circuits built on ternary-valued 1-trit multiplexers. Additionally, circuits may be designed using LUT-cascades such as the radix converter design method proposed in [52]. The research done in regards to arithmetic is mainly focusing on unbalanced ternary as opposed to balanced ternary. An alternative technology, the novel two-dimensional material of [50], may challenge the CNTFET circuits, as they claim to have a 50% reduction in transistor count compared to circuits with CNT and silicon based devices.

Mainly two methods for radix conversion has been proposed, which are the LUT-cascade design and the arithmetic decomposition design. Binary to unbalanced ternary radix conversion circuits and reverse have been implemented in simulation. No CNTFET circuit implementations of 8-bit binary to 6-trit **balanced** ternary was found.

The papers examined in the literature review have not shown a proven improvement over binary logic, although some research indicate that interconnect density may be significantly lower than binary-valued circuits for some ternary-valued circuits such as a multiplier circuit. Furthermore, there are be technologies which encode ternary values in binary, and which could possibly benefit from implementing natively ternary values. However, existing ternary-valued logic circuits in CNTFET circuit research generally have a higher PDP and transistor count compared to equivalent binary circuits, for the purposes of adding.

Chapter 4

Logic Synthesis and Circuit Design

In this chapter, the different designs and methods of constructing ternary-valued circuits are explored. Furthermore, the synthesis of the static gate design is automated, along with our SIMS 2020 paper.

1 Circuit design methodologies

Multiple design methods to achieve a circuit for logic functions have been considered. These design methods may rely on some gate-level circuit design, as well as transistor-level circuit design. As an example, ternary logic algebra may be used to break a large function into smaller function in the gate-level design, however it does not take into account the transistor-level design or optimizations thereof.

1.1 Capacitor-aided circuits

If we allow the use of capacitors, we can achieve circuits such as a ternary full adder with unary functions. These types of circuits work by using capacitors to take the average of multiple voltages, which can be processed by simple unary functions with shifted voltage threshold to achieve the specific function. Figure 4.1 shows one possible example to achieve different types of full adder circuits, such as unbalanced or balanced.

While the transistor count and delay of this circuit may outperform other designs, it has a very severe flaw of a high current leakage through the capacitors at high input frequencies. Current can flow between the inputs, as well as from the inputs to V_{DD} and GND within the unary functions, and through the two C_2 capacitors. Furthermore, capacitors take up quite a lot of space on a chip. Both the size and the current can be remedied by reducing the size and capacitance of the capacitors, however this limits their ability to produce sufficient output voltages.

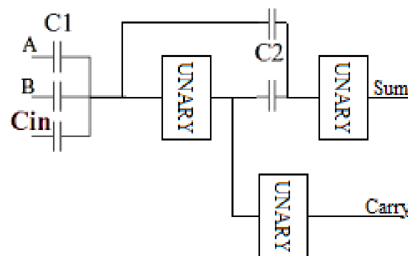


Figure 4.1: Capacitor-aided ternary circuit

1.2 Static gate design with pull-up and pull-down

The static gate design is a highly versatile and reliable circuit design which can achieve any ternary-valued logic function. It uses complementary transistor networks to pull the output voltage to high or low, or the middle voltage by dividing the voltage between the two transistors in the half-V_{DD} path, which is from an active pull-up network to an active pull-down network.
These 4 transistor networks are binary in function in the sense that they either do or do not connect the output to V_{DD} or GND. However, with variations in the diameters of the transistors within the network, in combination with NTI and PTI inverters on the inputs, a binary-valued truth table encompassing all combination of the ternary-valued inputs can be achieved.
Two types of the static gate design were identified [48][32], as shown in figure 4.2. Type A utilizes two transistor networks purely for the middle value, and two for the binary high and low output values. To limit the leakage current, the two pairs of networks have to be cross-wise exclusive of each other, i.e. no open path between V_{DD} and GND with the exception of voltage division, which results in a higher transistor count required. Type B functions in a similar manner to the commonly used design for the STI inverter. The top pair of networks generate an output analogous to the NTI part of an STI, while the bottom pair of networks are as the PTI part. Unlike type A, the networks are not at risk for cross-wise current leakage, however as neither pair of transistor networks can be active simultaneously, the same restrictions occur, giving the same increase in required transistors.

Type A and type B of the static gate design are found to be approximately equivalent in terms of transistor count, power consumption, and signal delay, and both designs can commonly be found in ternary-valued circuit design and research.
Within each network, the binary-output truth table with ternary inputs is achieved through the 4 transistor configurations shown in figure 4.3 from from [48]. For pull-up networks, p-type transistors are used, while n-type transistors are used for pull-down networks. Within each transistor network, 4 configurations of transistors can be connected in series to form a n-dimensional rectangle on the circuit truth table, which in parallel builds the whole truth table.

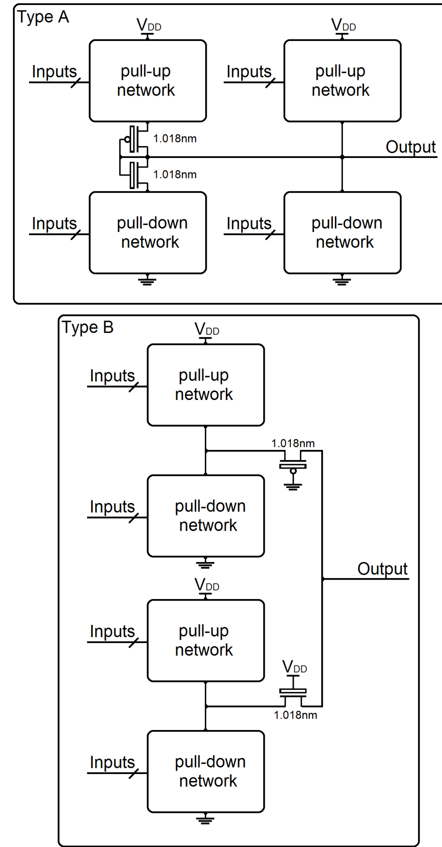


Figure 4.2: Static ternary gate design type A and type B

Ternary device switching table				
Pull-up network	(19, 0) I 0.289 V 1.487 nm	(10, 0) I 0.559 V 0.783 nm	(19, 0) I_N 0.289 V 1.487 nm	(19, 0) I_P 0.289 V 1.487 nm
Pull-down network	(19, 0) I_P 0.289 V 1.487 nm	(19, 0) I_N 0.289 V 1.487 nm	(19, 0) I 0.289 V 1.487 nm	(10, 0) I 0.559 V 0.783 nm
Switching operation				
Input = 0	ON state	ON state	OFF state	OFF state
Input = 1	ON state	OFF state	ON state	OFF state
Input = 2	OFF state	OFF state	ON state	ON state
Operator	A_0+A_1	A_0	A_1+A_2	A_2

Figure 4.3: Transistor switching table

1.3 Ternary Algebra with static gate base-functions

With the use of select base-functions, all ternary logic can be achieved. This method relies on the basic logic functions used in ternary logic algebra, the transistor circuits of which must be designed using some other method. Ternary logic algebra allows gate-level design of circuits, building on these base functions. As an example, the balanced sum gate can be constructed. The truth table is shown in table 4.1. While ANY and CON are simple to implement, the sum function inherently has a much higher complexity, due to the alternating values in the truth table. Therefore, we can use basic functions to achieve the sum function. The SOP form can be written as in equation 4.1. As 0 is equivalent to false, terms 2, 4, and 9 can be removed, as they will not contribute to the output. From equation 4.2, in terms 5 and 6, $(b = 1) \wedge 1$ and $(b = 2) \wedge 2$ can both be replaced with b. In a similar fashion, in terms 1 and 3, $(b = 0) \wedge 2$ and $(b = 2) \wedge 1$ can be replaced with $(b - 1)$, and in terms 7 and 8, $(b = 0) \wedge 1$ and $(b = 1) \wedge 2$ is replaced with $(b + 1)$, which results in equation 4.3. From here, there are 3 terms being repeated twice. Thus it is reduced to equation 4.4. The resulting gate-level design of the sum gate is shown in figure 4.4.

Table 4.1: Truth table of balanced sum

a	b	sum
2	2	0
2	1	2
2	0	1
1	2	2
1	1	1
1	0	0
0	2	1
0	1	0
0	0	2

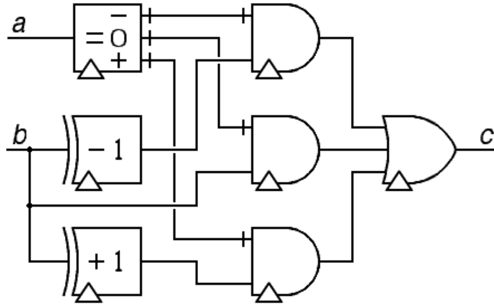


Figure 4.4: Basic gate-based balanced sum

$$
\begin{aligned}
a + b = &((a = 0) \wedge (b = 0) \wedge 2) \quad [term1]\\
\vee&((a = 0) \wedge (b = 1) \wedge 0) \quad [term2]\\
\vee&((a = 0) \wedge (b = 2) \wedge 1) \quad [term3]\\
\vee&((a = 1) \wedge (b = 0) \wedge 0) \quad [term4]\\
\vee&((a = 1) \wedge (b = 1) \wedge 1) \quad [term5]\\
\vee&((a = 1) \wedge (b = 2) \wedge 2) \quad [term6]\\
\vee&((a = 2) \wedge (b = 0) \wedge 1) \quad [term7]\\
\vee&((a = 2) \wedge (b = 1) \wedge 2) \quad [term8]\\
\vee&((a = 2) \wedge (b = 2) \wedge 0) \quad [term9]
\end{aligned}
\tag{4.1}
$$

$$
\begin{aligned}
a + b = &((a = 0) \wedge (b = 0) \wedge 2) \quad [term1]\\
\vee&((a = 0) \wedge (b = 2) \wedge 1) \quad [term3]\\
\vee&((a = 1) \wedge (b = 1) \wedge 1) \quad [term5]\\
\vee&((a = 1) \wedge (b = 2) \wedge 2) \quad [term6]\\
\vee&((a = 2) \wedge (b = 0) \wedge 1) \quad [term7]\\
\vee&((a = 2) \wedge (b = 1) \wedge 2) \quad [term8]
\end{aligned}
\tag{4.2}
$$

$$
\begin{aligned}
a + b = &((a = 0) \wedge (b - 1)) \quad [term1]\\
\vee&((a = 0) \wedge (b - 1)) \quad [term3]\\
\vee&((a = 1) \wedge b) \quad [term5]\\
\vee&((a = 1) \wedge b) \quad [term6]\\
\vee&((a = 2) \wedge (b + 1)) \quad [term7]\\
\vee&((a = 2) \wedge (b + 1)) \quad [term8]
\end{aligned}
\tag{4.3}
$$

$$
a + b = ((a = 0) \wedge (b - 1)) \quad \vee \quad ((a = 1) \wedge b) \quad \vee \quad ((a = 2) \wedge (b + 1)) \tag{4.4}
$$

The basic functions needed, being the unary functions of equality, increment and decrement, and the binary functions of MAX and MIN, can be generated with the logic synthesizer using the static gate circuit design method.

1.4 MUX-based LUT circuit

By utilizing multiplexer(MUX) circuits on the inputs as selection digits, the multi-digit output can be connected to a selection of sets of predefined values in the form of a look-up table(LUT). This method allows for circuits with n inputs and m outputs. The selection of outputs can be hard-wired voltages, outputs from other circuits, or the inputs themselves. Alternatively, it can be read from a configurable memory, making it a universal programmable function. Figure 4.5 shows the concept of a MUX-based n-input m-output circuit. While these types of circuits are generally fast, they require a large circuit which may result in a high passive current. In the general case, the number of selectable outputs is equivalent to the number of possible input combinations, and each selectable output combination can have a variable number of digits regardless of number of digit inputs. Since the multiplexing circuit can become exponentially complex for a high number of inputs, this circuit is mainly applicable for a smaller number of inputs. The MUX consists of decoders of the inputs, which check if each digit input is a specific value. This is done with the 3 equality functions. These decoders can give 3 outputs, to activate one of 3 inputs, as shown in figure 4.6.

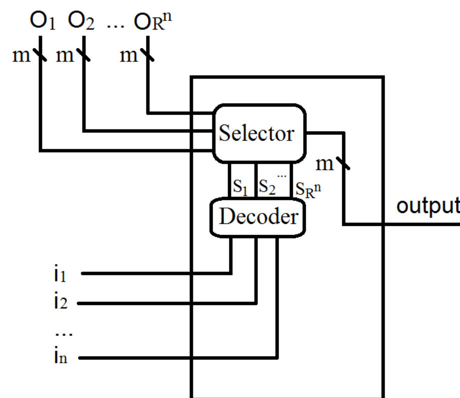


Figure 4.5: General MUX-based circuit

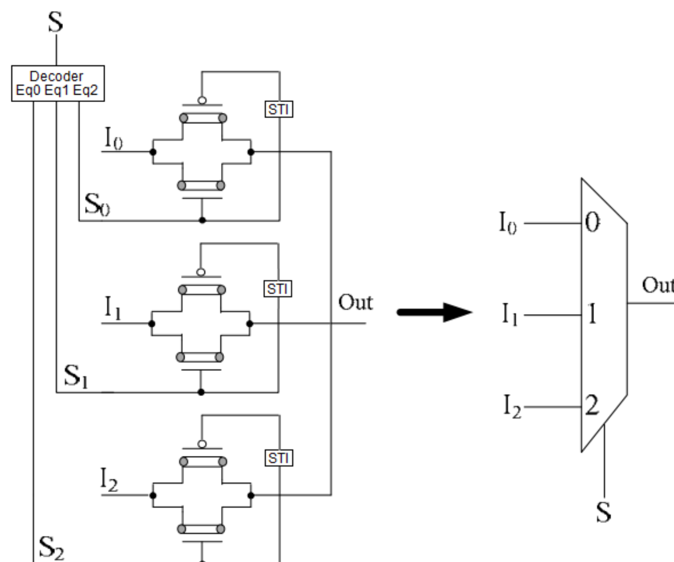


Figure 4.6: 1-trit MUX

These 1-trit MUX circuits can be connected to create more complex functions. As an example, the sum function can be constructed. Based on the logic algebra equation for the sum function, shown in equation 4.4, the MUX circuit can be constructed, and similarly the same can be done for the carry, also known as consensus, based on equation 2.14 as in figure 4.7.

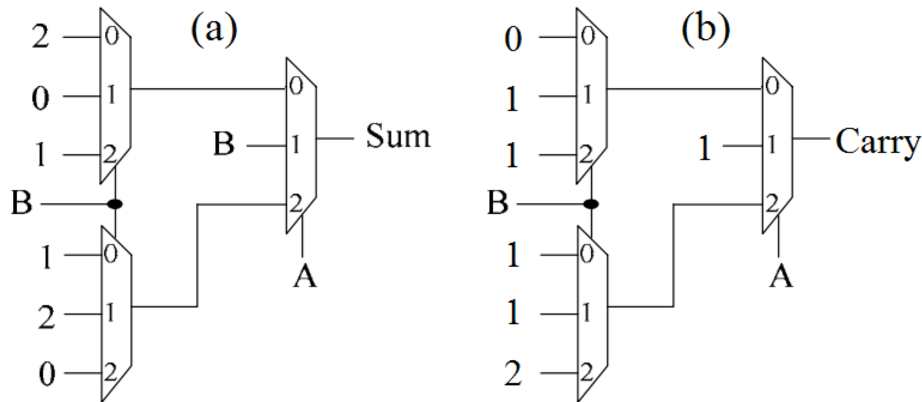


Figure 4.7: Balanced sum (a) and carry MUX (b) circuits

With these two circuits, a half adder can be constructed, and by extension a full adder with the "Accept Any" circuit, as shown in figure 4.8.

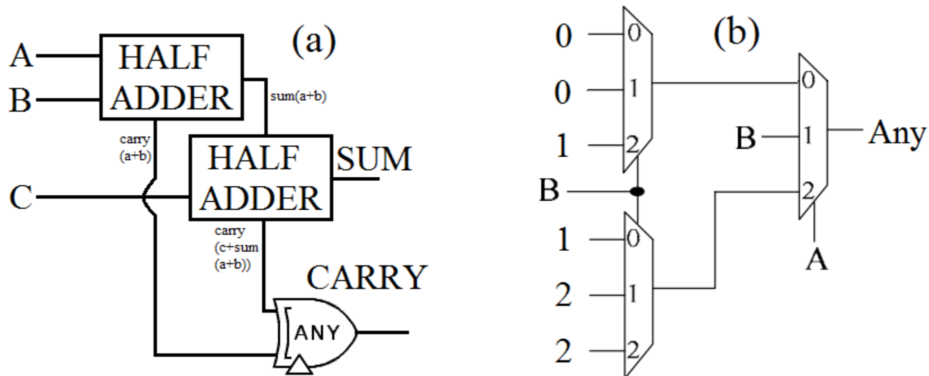


Figure 4.8: Balanced full adder circuit (a) with any MUX circuit (b)

One interesting observation to make here is that any 2-input function can be created with this method, however with a relatively high transistor count, as each 1-trit MUX requires 36 transistors to operate.

The decoder circuit for a 1-trit MUX consists of the 3 equality functions. This decoder circuit can be extended to n trits with the addition of 1-trit MUX circuits, as shown in figure 4.9 with an example of a 2-trit decoder with 9 output signals.

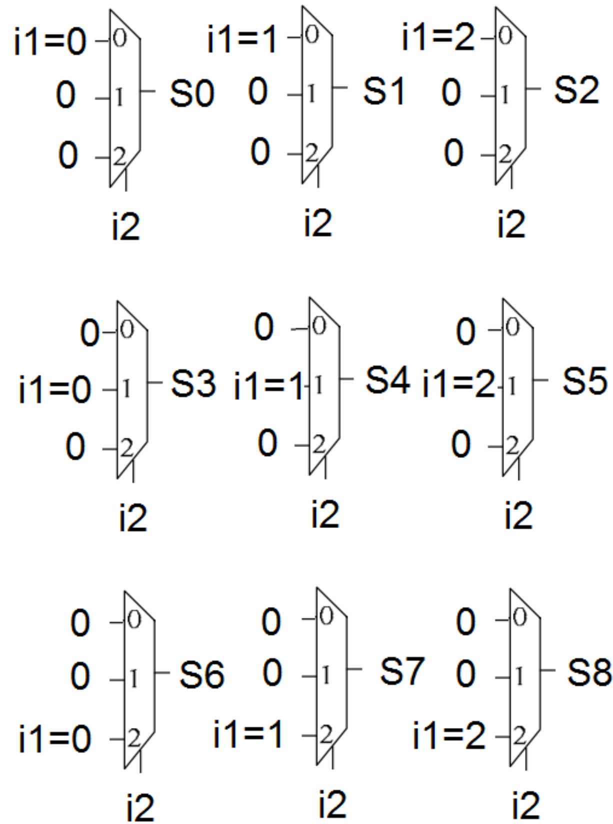


Figure 4.9: 2-trit decoder extension of the 1-trit decoder

It is clear that the increase of trits in the decoder brings an exponential growth of circuit complexity, and it's therefore ill-advised to use these multi-trit decoders over the alternative of a combination of 1-trit MUXes based on the ternary logic algebra of the function, as was demonstrated with figures 4.7 and 4.8.

Paper [52] utilizes a cascade of LUT circuit, which are optimized using column multiplicity of the truth table, i.e. how many duplicate columns can be found in different configurations of the specific function truth table. This method can also be used to design a circuit.

2 Risto et al. 2020

In our research paper "Automated synthesis of netlists for ternary-valued n-ary logic functions in CNTFET circuits", accepted in the SIMS 2020 research conference[62], we presented an open-source ternary-valued logic synthesizer for the transistor-level design using the static gate design, which generates design files for simulation, with CNTFET logic circuits from a truth table with up to 7 inputs, i.e. functions with arity up to 7. We also proposed an indexing system for the logic functions generated, which makes gate-level design much more convenient, as any circuit generated by a specific logic synthesizer can be referenced by an index without the need for a truth table or circuit diagram. Furthermore, we proposed the usage of a combination of different arities of function in a circuit. As an example, we compared three balanced full adders; Purely non-compound, traditional compound, and a hybrid of the two. Using simulation results to compare the three, the hybrid circuit outperformed the two others.

2.1 Function Indexing

Due to the large quantity of possible logic functions in ternary logic, we proposed an indexing system, which makes use of the base-27 heptavintimal notation shown in table 4.2.

With the heptavintimal notation, 3 trits can be expressed with a single character, which is analogous to hexadecimal encoding of bits in binary logic.

Table 4.2: The heptavintimal notation

Weight(Decimal)	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Ternary	000	001	002	010	011	012	020	021	022	100	101	102	110	111
Heptavintimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D
Weight(Decimal)	14	15	16	17	18	19	20	21	22	23	24	25	26	
Ternary	112	120	121	122	200	201	202	210	211	212	220	221	222	
Heptavintimal	E	F	G	H	K	M	N	P	R	T	V	X	Z	

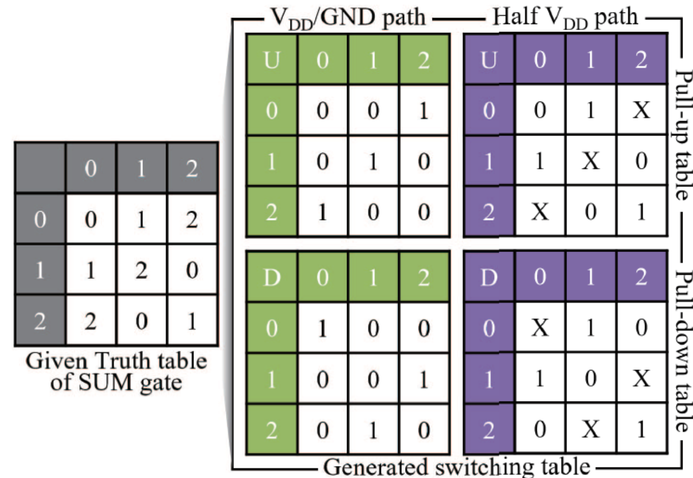
The function indexing system is a method of referring to specific truth tables with a string of heptavintimal characters. Since these truth tables have 3^m elements, it is divisible into groups of 3 ternary values, allowing for a convenient indexing of logic functions with the heptavintimal notation. As an example, the 19683 ternary-valued logic functions of arity 2 can be expressed with 3 heptavintimal characters. The functions are ordered by the values in the truth table, as shown in figure 4.10.

i1	i0	f_0		f_1		f_2		f_3		f_4		f_5		...	f_{ZZP}		f_{ZZR}		f_{ZZT}		f_{ZZV}		f_{ZZX}		f_{ZZZ}	
0	0	0		1		2		0		1		2			0		1		2		0		1		2	
0	1	0	0	0	1	0	2	1	3	1	4	1	5	...	1	P	1	R	1	T	2	V	2	X	2	Z
0	2	0		0		0		0		0		0			2		2		2		2		2		2	
1	0	0		0		0		0		0		0			2		2		2		2		2		2	
1	1	0	0	0	0	0	0	0	0	0	0	0	0	...	2	Z	2	Z	2	Z	2	Z	2	Z	2	Z
1	2	0		0		0		0		0		0			2		2		2		2		2		2	
2	0	0		0		0		0		0		0			2		2		2		2		2		2	
2	1	0	0	0	0	0	0	0	0	0	0	0	0	...	2	Z	2	Z	2	Z	2	Z	2	Z	2	Z
2	2	0		0		0		0		0		0			2		2		2		2		2		2	

Figure 4.10: Truth tables for arity-2 functions with heptavintimal indexing

2.2 Logic synthesis algorithm

By taking the truth table for the entire circuit, 4 binary truth tables are made, to represent the 4 transistor networks with the pull-up and pull-down network design method. These truth tables are represented by n-dimensional tables, which are 3 elements long in each dimension. An example of these 4 truth tables generated from a function truth table is shown in figure 4.11 from [48].



	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

U	0	1	2
0	0	0	1
1	0	1	0
2	1	0	0

U	0	1	2
0	0	1	X
1	1	X	0
2	X	0	1

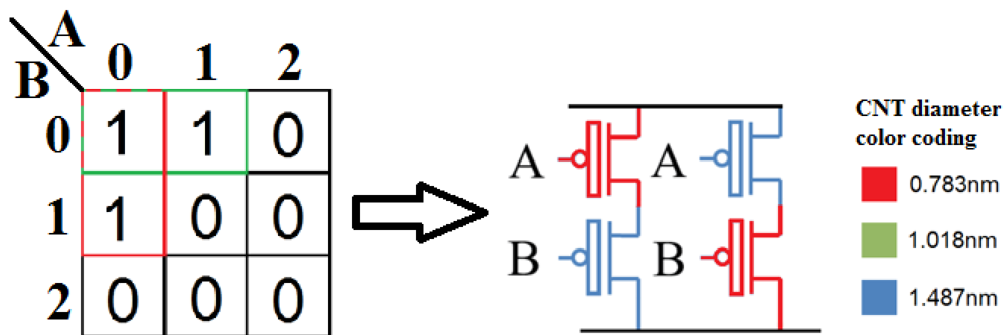
D	0	1	2
0	1	0	0
1	0	0	1
2	0	1	0

D	0	1	2
0	X	1	0
1	1	0	X
2	0	X	1

Figure 4.11: Truth table synthesis for pull-up and pull-down networks for the unbalanced SUM gate

Then, for each of the 4 networks, the truth table is drawn in n-dimensional space, and the largest n-dimensional rectangular groupings of 1s are found, to cover all the 1s with as few groups as possible.

Each group is then interpreted as a transistor path within the network. Using the 4 different modes for pull-up and pull-down transistors, as shown in figure 4.3, transistor paths can be constructed from groupings on the truth table, as in the example in figure 4.12. The nanotube diameter of the transistors are color coded with red, green, and blue, for 0.783nm, 1.018nm, and 1.487nm respectively.



B \ A	0	1	2
0	1	1	0
1	1	0	0
2	0	0	0

Figure 4.12: Circuit implementation of a pull-up network truth table with 2 groups

The code listing below shows the pseudocode for a general description of the logic synthesizer algorithm. For more detail of the synthesizer, read the source code with comments in appendix C, or on github[63].

```
take circuit truth table inputs
for each network
    generate transistor network truth tables
    for every possible grouping in truth table
        generate n-dimensional mask for the group
        compare mask with truth table
        if mask covers no '0's and at least one '1'
            store the mask group in the groups vector
    for each group in the groups vector
        compare the group with the sum of all other groups
        if group is covered by the sum of other groups
            remove group from the groups vector
    for groups in the group vector
        generate transistor path within the network
```

Using this synthesis algorithm, circuits for any ternary-valued logic function can now be generated. Figure 4.13 shows the circuit for the arity-3 balanced full carry function, as it is synthesized by the code.

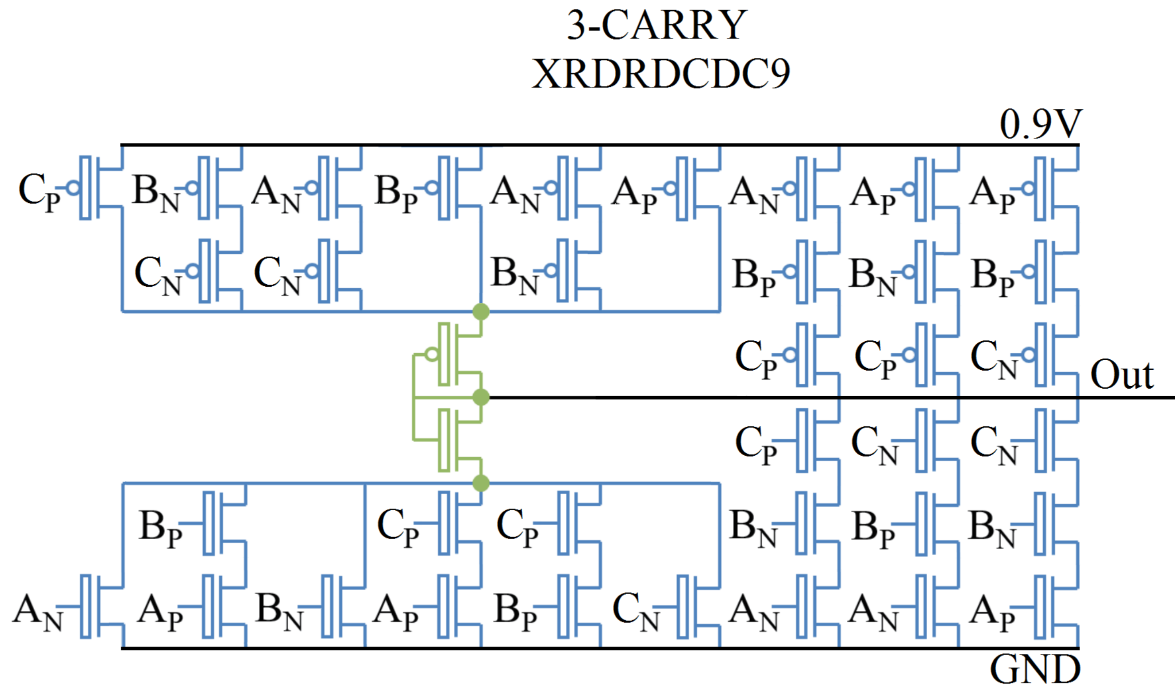


Figure 4.13: Circuit implementation of the balanced carry

2.3 Full adder architecture

To demonstrate the usage of the logic function netlist synthesizer, we compared three full adder gate-level designs, shown in figure 4.14.

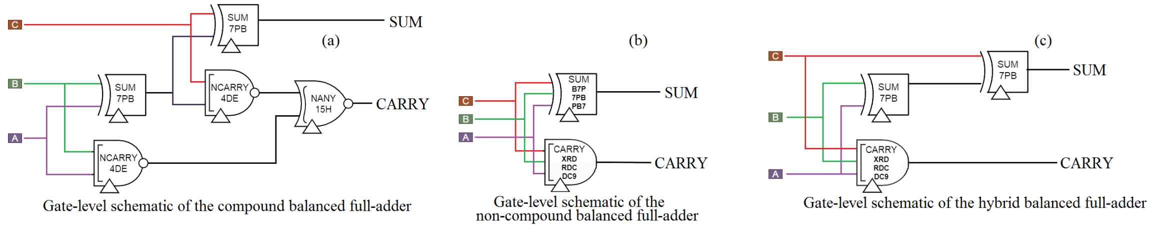


Figure 4.14: Three proposed balanced ternary full adders "compound" (a), "non-compound" (b), "hybrid" (c)

The compound adder only makes use of functions with an arity of 2. This is the traditional design method most commonly used [48][50][46].

The non-compound design method represents a more holistic view of the circuit as a relation between the inputs and the outputs, as each output can be achieved with a single function.

The hybrid design method is a combination of these two, which takes advantage of the benefits of lower and higher arity functions. In this example, the sum output is achieved in the same way as the compound full adder, however the carry output is generated by a single holistic function.

2.4 Simulation results

The circuits, as generated by the logic synthesizer, was simulated in HSPICE, and compared in terms of delay, power consumption, and number of transistors. The PDP (power-delay-product), represents the product of the worst-case delay and the power consumption. The results are shown in table 4.3.

Table 4.3: Simulation results with 2fF load capacitor

Circuit	Transistors	Avg. power 50MHz	Worst Delay	PDP 50MHz
2-sum (7PB)	40 (32)	$0.35\mu W$	530ps	0.185e-15 J
2-ncarry (4DE)	10	$0.80\mu W$	20ps	0.016e-15 J
2-nany (15H)	18	$0.32\mu W$	40ps	0.013e-15 J
3-sum (B7P7PBPB7)	150 (138)	$0.34\mu W$	1530ps	0.526e-15 J
3-carry (XRDRDCDC9)	50 (38)	$0.82\mu W$	30ps	0.024e-15
[64] Unbalanced FA	106	$0.47\mu W$	0.89ns	**0.421e-15 J**
[49] Unbalanced FA	98	$\mathbf{0.43\mu W}$*	1.57ns*	0.667e-15 J*
Proposed Compound Balanced FA	118 (102)	$2.29\mu W$	**0.55ns**	1.262e-15 J
Proposed Non-compound Balanced FA	188 (176)	$1.18\mu W$	1.53ns	1.805e-15 J
Proposed Hybrid Balanced FA	118 (102)	$1.50\mu W$	0.56ns	0.840e-15 J

* see [64]

We compare our results with some data-points from other researchers, however it should be noted that this is not entirely an equivalent comparison, as the results were not generated in the same simulation environment. Despite this, it is interesting to see that the performance is in a similar ball-park of numbers as cutting edge research.

More interestingly, we compare our own circuits, with simulation results of both individual functions, and full adder circuits. The results show that the hybrid design outperforms both the compound and

non-compound full adder designs. Based on the results of the individual functions, this is expected; the higher arity functions have lower power consumption but worse delay. In the hybrid full adder, the critical path of delay are the sum functions, while the carry function can expense some delay in exchange for less power consumption with the same number of transistors, resulting in a full adder with the same delay as the non-compound design, with the same number of transistors, but lower power consumption. This result shows that the higher arity functions generated by the synthesizer can provide benefits over lower arity functions in the right context.

2.5 Contributions

In this paper, we had 3 major contributions:

- 1. An open source netlist synthesizer for n-ary ternary-logic functions in CNTFET circuits, available on GitHub.
- 2. A proposed function indexing system
- 3. Three proposed gate-level full adder methods were simulated and compared

3 Optimizing gate-level design

Using this logic synthesizer, ternary logic gates can now be generated with ease. This facilitates gate-level design of logic circuits, on the higher abstraction level. That said, there is still a question of which functions should be used.

3.1 Expected input values

One strategy is to consider the expected values on each input to the function. As an example, if one of the inputs is a binary-valued input, that input will never be the middle value. When generating the function, all outputs in the case of that specific input being the middle value can be written as 'x', i.e. "don't care". The synthesizer will then optimize with this in mind and generate the most simple function to achieve the wanted functionality.

3.2 Compound vs non-compound

As shown in the results of our paper, the choice of arity and compoundness of the larger circuit can have an effect on the performance. Higher arity functions generally have a lower power consumption, but higher delay and transistor count. Therefore, a circuit can be optimized by only using higher arity functions in signal paths with non-critical delay.

Chapter 5

Radix conversion circuits

Radix conversion is important not only for compatibility with binary-valued technology, but it also opens up for the possibility of mixed-radix systems. It is imaginable that specialized ternary circuits may be integrated into a binary-valued system, and therefore it must be shown that the costs of converting the radix of data does not outweigh the benefits of using ternary-valued circuits for either computation or data storage.

1 Radix conversion circuit architecture

In this subchapter, every method conceived or discovered is described. Some of these methods are quite trivial, however they are discussed here for completeness. Software solutions are not considered, as this conversion is meant to take place at the hardware interface between binary and ternary components.

1.1 Simultaneous iteration method

One conceivable method, although highly inefficient, is to synchronously iterate a counter in each radix from zero until the counter in the radix to be converted from reaches the value to be converted. Then, the counter in the other radix will have the translated value. While it is clear that this is an time-inefficient method, its non-linear time complexity can be shown to be $O(2^n)$ for n bits, which is proportional to the value range of the input data.

1.2 The analog intermediate method

If a binary-to-analog and analog-to-ternary converter has high enough resolution, it may be used to convert between these two radices. A time-efficient ADC with parallel comparators for each measurable value may be used, however this would be a very complex circuit with high requirements for resolution and accuracy, and may be rendered useless by electrical noise for larger converters. This method has the issue of voltage stabilization and signal noise, and an exponentially increasing circuit-complexity. It could be possible to implement for smaller converters, such as 8-bit, as only 256 voltage levels needs to be differentiated. However, the circuits for digital-analog conversion is relatively complex and not entirely reliable with narrow voltage bands [65].

1.3 Look-up table

A look-up table can be used to map any input value to a hard-wired output value. This may also be used to convert between binary to ternary numbers, if every possible input is extensively covered with mapping to a corresponding output. By the use of multiplexing, the correct value can be accessed from a look-up table given its input. Each added binary input would require one additional layer of multiplexing, and double the size of the look-up table. Due to the exponentially increasing circuit complexity of this type of circuit, this method is not feasible for a large circuit such as a binary-ternary radix converter.

1.4 Adder-based arithmetic decomposition method

A set of equation can be defined for each binary digit of the input, by decomposing the binary digit into its arithmetic values represented in balanced ternary form, with regards to its digit place. This set of equations show the influence each binary input has on each ternary output, in other words the relation between the input and output digits. Table 5.1 shows this process for 8-bit binary to 6-trit balanced ternary.

Table 5.1: Digit relation

$2^0 =$						$+ 3^0$
$2^1 =$					$+ 3^1$	$- 3^0$
$2^2 =$					$+ 3^1$	$+ 3^0$
$2^3 =$				$+ 3^2$		$- 3^0$
$2^4 =$			$+ 3^3$	$- 3^2$	$- 3^1$	$+ 3^0$
$2^5 =$			$+ 3^3$	$+ 3^2$	$- 3^1$	$- 3^0$
$2^6 =$		$+ 3^4$	$- 3^3$	$+ 3^2$		$+ 3^0$
$2^7 =$	$+ 3^5$	$- 3^4$	$- 3^3$	$- 3^2$	$+ 3^1$	$- 3^0$
t_n	t_5	t_4	t_3	t_2	t_1	t_0

As each digit t_n represents the 3^n component of the number to be converted, each output digit can be found from their component of the sum of the contributions of the binary digits in balanced ternary form. Thus, the radix relation matrix can be made from table 5.1.

Table 5.2: Radix relation matrix

3^0	$+b_0$	$-b_1$	$+b_2$	$-b_3$	$+b_4$	$-b_5$	$+b_6$	$-b_7$
3^1		$+b_1$	$+b_2$		$-b_4$	$-b_5$		$+b_7$
3^2				$+b_3$	$-b_4$	$+b_5$	$+b_6$	$-b_7$
3^3					$+b_4$	$+b_5$	$-b_6$	$-b_7$
3^4							$+b_4$	$-b_7$
3^5								$+b_7$
	2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7

Each row in table 5.2 represents contribution each binary input has at each digit place of the balanced ternary output. Each input may influence each output through a matrix of adders in accordance with the matrix derived from the radix relation matrix between binary and ternary. Note that these full adders will sum up each row, and produce a carry signal which propagates downwards to the higher significance digit place rows[54][53]. This type of circuit has been achieved in CNTFET simulations for both binary to unbalanced ternary, and unbalanced ternary to binary [56][57]. However it has not been done for balanced ternary numbers.

2 Circuit design

Based on the arithmetic decomposition method of radix conversion, the logic circuits can be constructed for the purpose of simulation. The indices used in this chapter refers to synthesized circuits.

2.1 Pure adder-based

As a starting point, a radix converter circuit is built by using the ”hybrid” full adder circuits proposed in chapter 4. The circuit schematic is shown in figure 5.1. Each block represents an adder component, labeled SUM, HA, FA, for summation, half adder, and full adder respectively. The blocks are also numbered by rows and columns for easy reference when optimizing. The sum output of each block propagates to the right, while the carry output propagates downwards. The inputs counts from i_0 to i_n from the lowermost input upwards for each block. The full adder uses the hybrid balanced full adder architecture described earlier, while the half adder uses one 2-input sum and one 2-input carry. For the SUM blocks, either one or two 2-input sum functions are used, depending on the number of inputs. For this version, inverters are used on the binary inputs to achieve the negative sign, with 450mV representing 0, to make binary 0 and 1 analogue to balanced ternary 0 and +. Carry signals from the lowermost blocks could be utilized to signify overflow, which for this circuit should never happen.

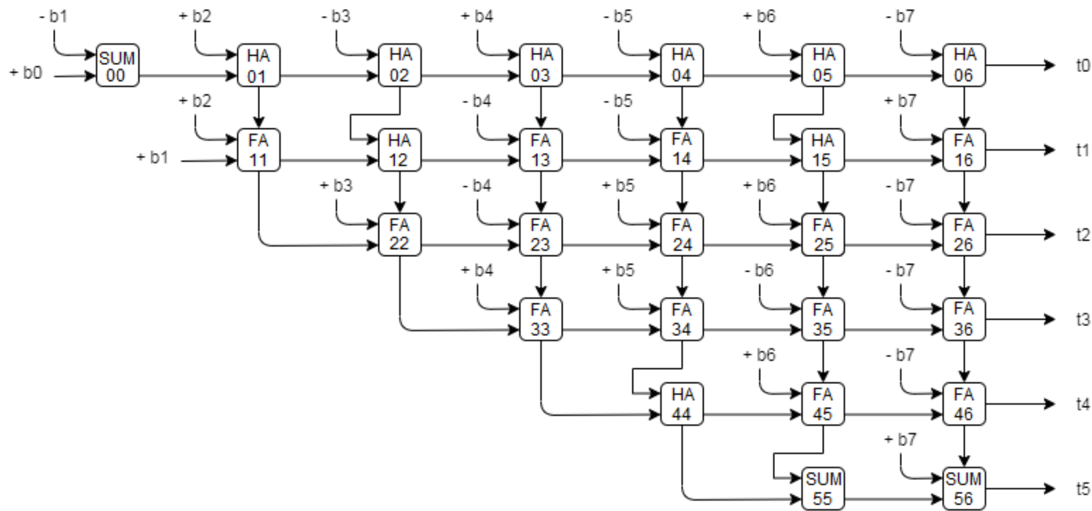


Figure 5.1: Radix converter with unoptimized adders

2.2 Optimized functions

As mentioned in Chapter 4 Section 3, the gate-level design can be optimized if the expected input values for each function are known. For example, in this circuit many of the inputs are binary-valued, meaning they have the values of either low or high, or in this case ’0’ or ’2’. Furthermore, the carry output of many of these functions are limited, which gives further information on which input values are expected for the functions further down the line. By starting from the top row working downwards to the right of the pure adder-based radix conversion circuit, a table of expected values at every point in the circuit can be constructed. Based on this table, most of the adders can be replaced with simpler ones, while still achieving the same expected function. In addition, all negative signs on binary inputs, resulting from the digit relation, can be accounted for in the selection of the optimized functions. For this specific circuit, all sum outputs for the blocks is expected to be any of the three values, however the carry outputs are often limited. The expected values for the carry signal for each block is listed in table 5.3.

Table 5.3: Expected carry signal values from each block in the radix converter circuit

Row	Column 0	Column 1	Column 2	Column 3	Column 4	Column 5	Column 6
0	0	0/+	-/0	0/+	-/0	0/+	-/0
1		0/+	-/0	-/0/+	-/0	0/+	-/0/+
2			0/+	-/0/+	-/0/+	0/+	-/0/+
3				0/+	-/0/+	-/0/+	-/0/+
4					0/+	-/0/+	-/0/+

Since this method does not account for the concurrent values of the carry signals, but instead every expected possible value for each of them, some expected values might be a result of concurrent expected values which never coincide. Therefore, it is clearly possible to optimize this circuit further. This method can also show the sub-optimality of the circuit, as the 5th row should never output a carry signal, as $2^8 < 3^6$, yet this method implies a possible non-zero carry output from the 5th row.

From here, the logic synthesizer can be used to optimize the individual functions used. The wanted output for input values which are not expected to occur can be written as 'x', or "don't care", which will allow the grouping algorithm to construct a simpler circuit. The binary inputs $b_0..b_7$ are either the high value or the low value, however with the synthesizer, we can choose how they are interpreted. For the middle value, 'x' can be written as output, as it is not expected to occur with binary values. For the high value, it can be interpreted as addition or subtraction, to account for the signs on the inputs. The resulting functions are listed in table 5.4 and table 5.5, for sum and carry respectively. Functions connected serially to produce a 3-input functions are notated with + between the indices.

Table 5.4: Indices of optimized sum functions for the radix converter circuit

Row	Column 0	Column 1	Column 2	Column 3	Column 4	Column 5	Column 6
0	CRX	77P	BBP	77P	BBP	77P	BPP
1		55X + 7PP	PPB	BPP + 7PP	BPP + PPB	7PP	77P + PPB
2			88R + PPB	BPP + 7PB	77P + PPB	77P + 7PP	BPP + 7PB
3				88R + 7PB	77P + 7PB	BPP + 7PP	BPP + 7PB
4					8R9	77P + 7PB	BPP + 7PB
5						8R9	77P + 7PB

Table 5.5: Indices of optimized carry functions for the radix converter circuit

Row	Column 1	Column 2	Column 3	Column 4	Column 5	Column 6
0	RRD	CDD	RRD	CDD	RRD	CDD
1	ZZXXXDXXD	DDC	DRRCDDCDD	CDDCDD9CC	RDD	RRDRRDDDC
2		RRDRRDDDD	DRRCDD9CC	RRDRRDDDC	XXRRRDRRD	DRRCDD9CC
3			ZZRRRDDDD	XXRRRDDDC	DRRCDDCDD	DRRCDD9CC
4				RDD	XXRRRDDDC	DRRCDD9CC

The netlist implementation of this optimized circuit is shown in Appendix B.

Table 5.6: Performance of 8-bit to 6-trit radix converter

Circuit	Transistors	Current	Worst-case delay	Best-case PDP	1GHz PDP
Pure adder-based	2452	$70.1\mu A$	640ps	4.037e-14	6.309e-14
Optimized	1587	$55.4\mu A$	395ps	1.969e-14	4.986e-14
[56]	700	$465.9\mu A$	137ps	6.370e-14	n/a

3 Simulation results

Table 5.7: Input value sequences for simulation

Input	0ns	1ns	2ns	3ns	4ns	5ns
b_0	1	1	0	1	1	0
b_1	1	1	1	0	1	0
b_2	1	0	0	1	0	1
b_3	1	0	1	0	1	0
b_4	1	1	0	1	0	1
b_5	1	1	1	0	0	1
b_6	1	0	0	1	1	0
b_7	1	0	1	0	1	0

With the input value combination pattern in table 5.7, a transient circuit simulation was done in HSPICE for the radix converter circuits. The transient output for the optimized radix converter circuit is shown in figure 5.2, with t_0, t_1, t_2, t_3, t_4, t_5 shown as dark blue, orange, gray, yellow, light blue, and green respectively. All the outputs consistently inhabit three well-defined voltage levels with clear separation and little to no voltage drift. The transition period at each input change is relatively short, implying that it could run comfortably at a higher input frequency. The data from these transient simulations were analysed to produce assessable data, displayed in table 5.6. These results show the increase in performance for the optimized circuit, and shows that one GigaByte per second of 8-bit binary-valued data can be converted to balanced ternary, at roughly $50\mu W$. These circuits can be connected in parallel, which means data can be converted at a rate of $50\mu W$ and 1587 transistors per GB/s. For example, a 20GB/s radix converter has a power consumption of $1mW$, and a component count of 31740 transistors. In conclusion, conversion of data radix between binary and ternary is very feasible with CNTFET circuits.

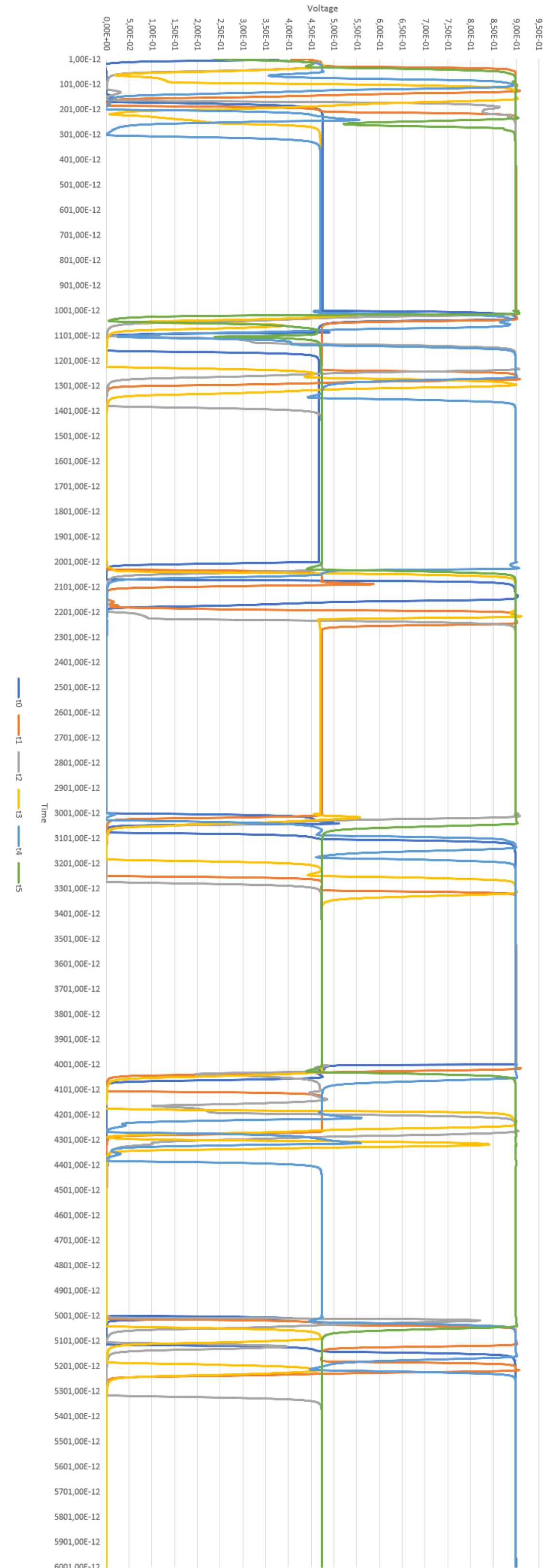


Figure 5.2: Transient analysis of the optimized radix converter

Chapter 6

Ternary versus Binary logic circuits

In this chapter, the application of ternary-valued logic is compared with binary-valued logic, in terms of efficiency and component count. Both synthesized circuits and some alternative circuits are investigated.

1 General overview of synthesized functions

The individual functions for binary and ternary logic can be examined in terms of transistor count, as well as power consumption and signal delay. Figure 6.1 shows the transistor count for all 19683 of the ternary functions with an arity of 2, ordered by index and by transistor count, as they are generated by the logic circuit synthesizer.

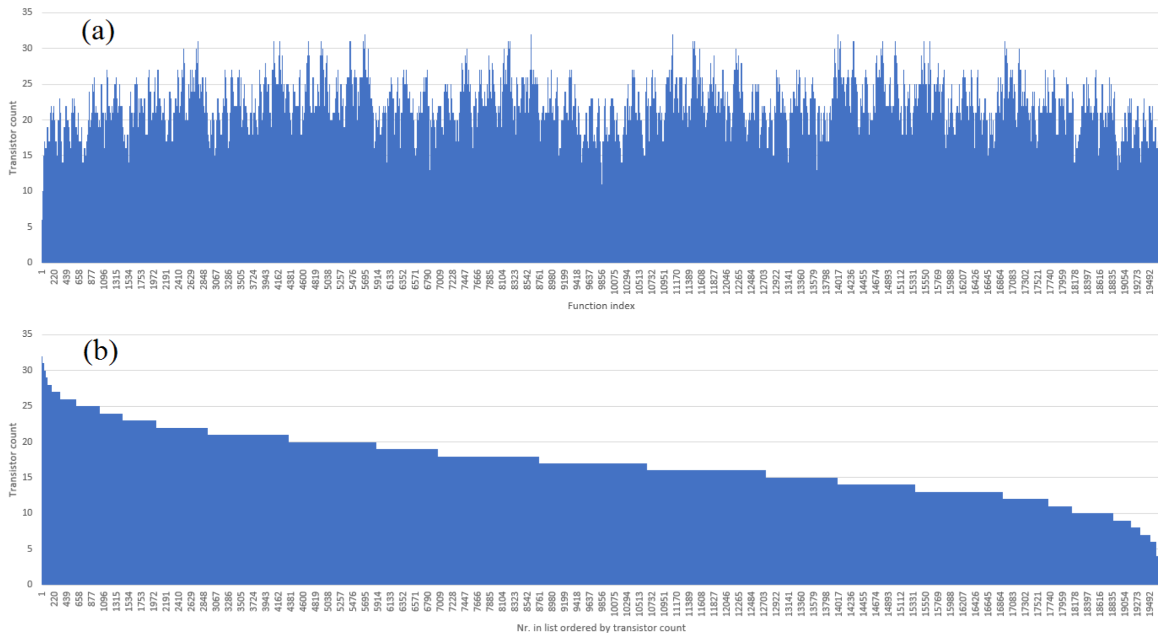


Figure 6.1: Transistor count for all ternary-valued functions of arity 2, by index (a), by count (b)

Although the synthesizer is intended for generating circuits for ternary-valued logic functions, the binary-valued functions are a subset of the functions that can be synthesized. The 16 binary-valued functions with an arity of 2 is listed in table 6.1 with simulation results at 2GHz input frequency, with 4 inverters in series on the output as a fanout load (FO4). These binary-valued functions is compared with some commonly used ternary-valued functions, listed in table 6.2 with simulation results.

Table 6.1: The 16 binary-valued functions

Index	Name	Transistors	With inverters	Current	Worst-case delay	PDP
f_000	Constant 0	0	0	0	0	0
f_002	NOR	4	4	8.32E-08	13E-12	9.73E-19
f_00K	A and not B	4	6	4.26E-08	20E-12	7.67E-19
f_00Z	not B	2	2	3.75E-08	7E-12	2.36E-19
f_200	not A and B	4	6	3.16E-08	19E-12	5.39E-19
f_222	not A	2	2	3.72E-08	7E-12	2.34E-19
f_20K	XOR	8	12	9.50E-08	23E-12	1.96E-18
f_22Z	NAND	4	4	4.29E-09	5E-12	1.93E-20
f_K00	AND	4	8	2.94E-08	14E-12	3.70E-19
f_K02	NOT XOR	8	12	1.53E-07	19E-12	2.61E-18
f_KKK	A buffer	2	4	6.16E-08	13E-12	7.20E-19
f_KKZ	A or not B	4	6	8.59E-08	15E-12	1.16E-18
f_Z00	B buffer	2	4	6.16E-08	13E-12	7.20E-19
f_Z22	not A or B	4	6	6.55E-08	9E-12	5.30E-19
f_ZKK	OR	4	8	1.06E-07	14E-12	1.33E-18
f_ZZZ	Constant 1	0	0	0	0	0

Table 6.2: Commonly used ternary-valued functions

Index	Name	Transistors	With inverters	Current	Worst-case delay	PDP
f_PC0	MIN / AND	10	18	2.54E-07	16E-12	3.66E-18
f_ZRP	MAX / OR	10	18	3.26E-07	15E-12	4.40E-18
f_8	PTI	2	2	6.27E-09	12E-12	6.77E-20
f_2	NTI	2	2	4.30E-08	6E-12	3.32E-19
f_5	STI	6	6	3.24E-07	6E-12	1.75E-18
f_RDC	Consensus	10	18	6.26E-07	15E-12	8.45E-18
f_7PB	Balanced sum	32	40	3.47E-07	33E-12	1.03E-17
f_C90	Unbalanced carry	8	16	2.64E-07	21E-12	4.99E-18
f_VK0	Unbalanced carry 2	8	16	4.80E-08	16E-12	6.92E-19
f_B7P	Unbalanced sum	31	39	3.41E-07	36E-12	1.11E-17
f_2	Equal 0	2	2	2.62E-08	17E-12	4.01E-19
f_6	Equal 1	4	6	5.66E-08	12E-12	6.11E-19
f_K	Equal 2	2	4	4.71E-08	11E-12	4.66E-19
f_7	Increment	7	9	4.64E-07	15E-12	6.27E-18
f_B	Decrement	7	9	4.77E-07	17E-12	7.30E-18

The transistor counts and PDP of the ternary-valued and binary-valued functions are graphed in figure 6.2.

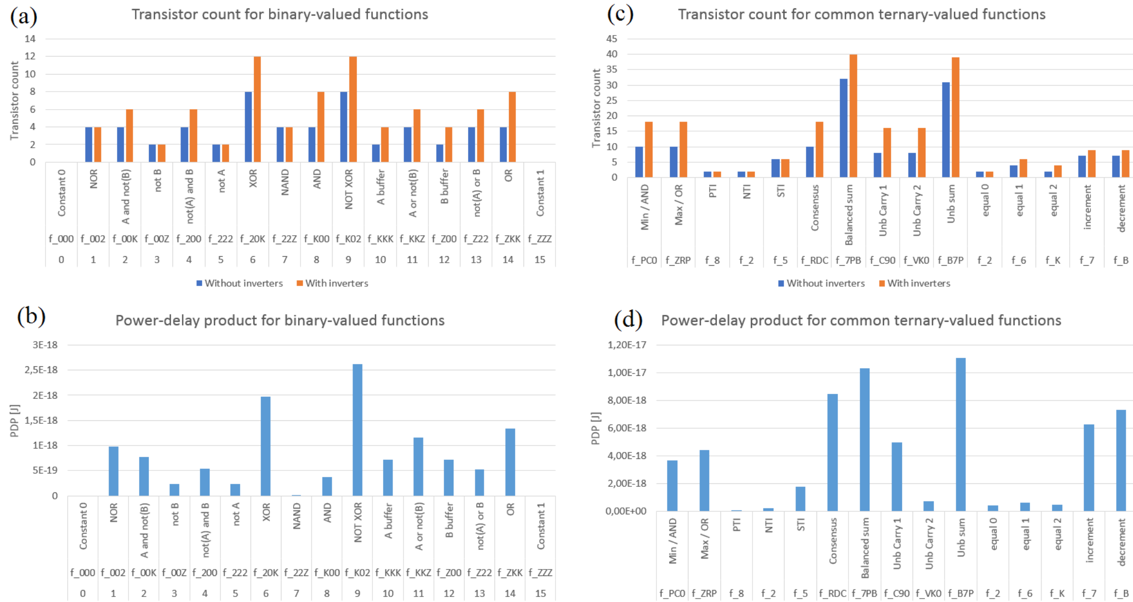


Figure 6.2: Transistor count(a)(c) and PDP(b)(d) for binary-valued(a)(b) and common ternary-valued(c)(d) functions

From these results, it is clear that individual synthesized ternary-valued CNTFET circuits tend to have a significantly higher transistor count and PDP compared to the synthesized binary-valued circuits. By examining the PDP of the ternary-valued functions, it becomes clear that the functions which only output 0 or 2 have a much lower PDP than the rest of the functions. This is due to a major flaw of the circuit, being the voltage division to create the middle voltage. It could be possible to remedy this with a second voltage source for the middle voltage, which removes the need to do voltage division in the logic circuit, improving the PDP. However, this would be a trade-off with a higher number of interconnections, as well as transistor count to keep the two voltages exclusive of each other. For a complete comparison, we need to consider the implementation of these functions to achieve a desired output, where an m-trit ternary output can be thought to be equivalent to an n-bit binary output.

2 Use cases of ternary logic

When comparing binary and ternary logic, it is important to keep in mind the possible use cases for ternary logic. While only the PDP and transistor count of adder circuits are investigated, there are other use cases as well to keep in mind.

2.1 Basic arithmetics

Basic arithmetics such as adding are essential operations for a general computing system. To show that ternary-logic is superior to binary logic in the aspect of general computing, basic arithmetics must be shown to be more efficient in ternary-logic. Since basic arithmetic operations on n bits are generally less complex than the radix conversion circuit for n bits, it's evident that it is not worthwhile to convert the radix to only do simple arithmetics in a mixed radix system. However, there may still be benefits to using a purely ternary-valued logic system. Therefore, the arithmetical function of adding is compared for ternary-valued and binary-valued logic.

2.2 Data storage and transmission

Although this research project did not cover data storage and transmission, it's worth noting that ternary-valued data have a higher information density. The advantage of ternary over binary in this aspect is 1.58 times better than binary, as shown in earlier Chapter 2 Section 1.5.

To enable ternary data to be stored, a special technology called memristors can be used. Research has been done on memristors and ternary data storage in the Ternary Research Group at USN [66][67].

While the exact performance of this data storage depends on the performance of memristors, the component count may be lower, and data transmission may be faster as a result of fewer digits needed to be stored and transmitted, however more research must be done on these topics.

2.3 Specialized ternary algorithms

While it is imaginable that special algorithms may exist which are more efficient to perform in ternary logic, as most if not all algorithms are based on the basic arithmetics such as adding, multiplying, and subtracting, basic arithmetics are the baseline comparison of ternary and binary logic in these circuit implementations.

One study [16] implements Ternary Neural Networks by encoding balanced ternary in binary values. It may have benefits from natively ternary circuits, however more research is needed. Another study [61] shows that the multiplication operation can be done in ternary with lower interconnection density. This implies that even if adding can not be done more efficiently in ternary, there may still exist other operations and algorithms which might benefit from ternary logic.

3 Design of adder circuits

Multi-digit adders for binary and ternary logic are compared. Both binary and ternary full adders are synthesized for a fair comparison. Additionally, some alternate commonly used circuits are compared.

3.1 Synthesized binary adder circuit

Using the synthesized functions, a binary full adder can be constructed with a common design such as in figure 6.3, shown with synthesizer index for binary XOR(20K), AND(K00), OR(ZKK) gates.

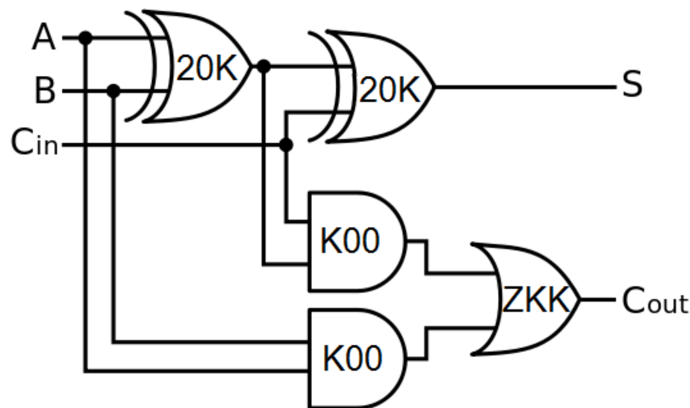


Figure 6.3: Common gate-level design of binary full adder

3.2 Synthesized balanced ternary adder circuit

The hybrid gate-level design proposed in our SIMS paper is used, as shown in figure 6.4.

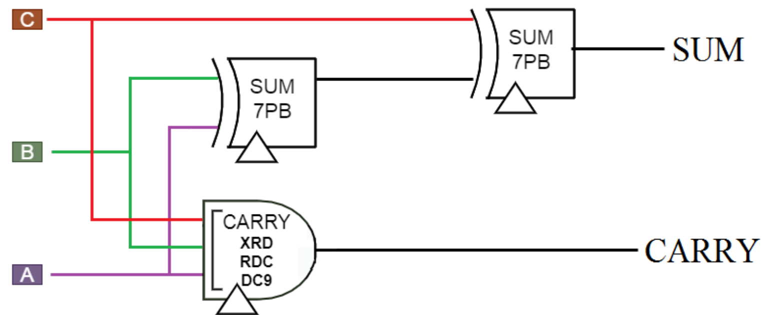


Figure 6.4: Proposed ternary full adder

3.3 Basic gate-based balanced ternary full adder

As an example of a circuit built with logic algebra from the basic gates, the sum function was constructed in chapter 4. The sum gate is shown with radices for each gate in figure 6.5. Each gate can be synthesized by the netlist generator with the index shown.

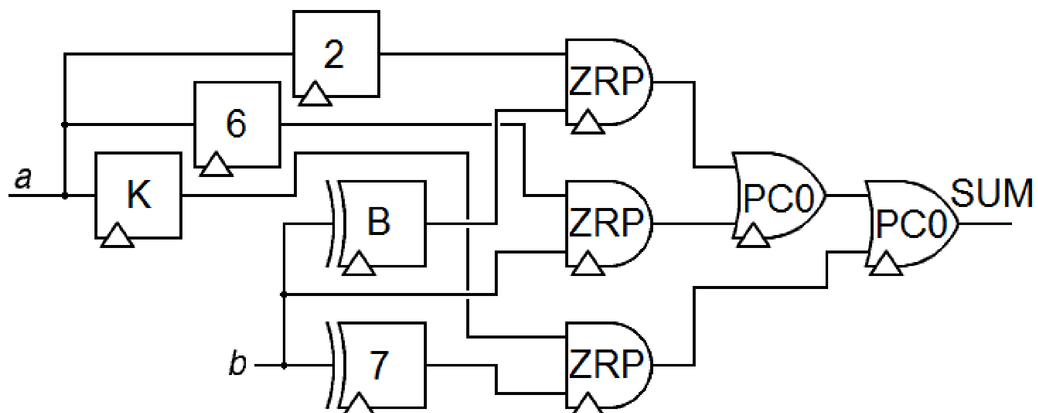


Figure 6.5: Gate-based sum

To achieve a full adder, the functions of consensus and any can be used, as in the compound design described in our paper, with the sum gate derived with logic algebra replacing the sum gate $7PB$.

3.4 Capacitor-aided balanced ternary full adder

Using the capacitor-aided circuit design, a balanced full adder can be constructed as in 6.6 from [47].

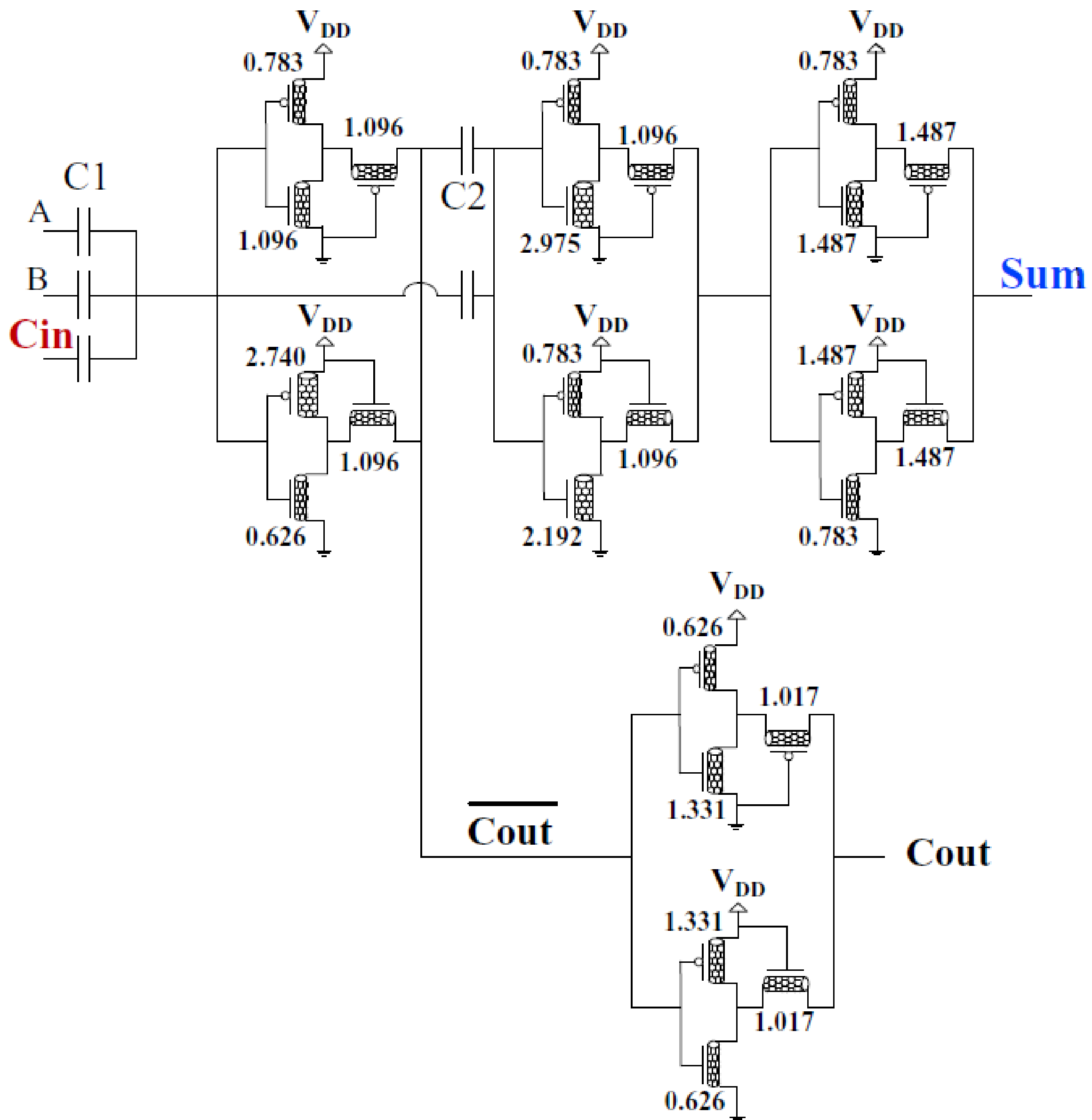


Figure 6.6: Capacitor-aided ternary full adder

Due to the voltage thresholds required for each inverter gate, and quite large current at 2GHz frequency through the capacitors, this circuit was found to not be applicable for high-performance adding. The paper [47] similarly report a high power consumption of 28.5μW, as well as a high delay time of 200ps for their serial adder. The current can be limited by reducing the size of the capacitors, however this causes the circuit to malfunction below a certain capacitance. Despite its low transistor count, it is an inefficient and unreliable circuit, whose function is better implemented with other circuits.

4 Simulation results

The single full adders are simulated individually, and are compared in terms of transistor count and power-delay product.

4.1 Results

Table 6.3 shows the simulation results for the full adders discussed above. Circuits are simulated in HSPICE, at 2GHz input frequency and FO4 output load. The best results for both the binary and ternary category are highlighted.

Table 6.3: Simulation results of binary and ternary full adders

Circuit	Transistors	w/ inverters	Current	Worst-case delay	PDP
Synthesized binary FA	46	52	7.02E-07	5.80E-11	3.67E-17
28-transistor binary FA	**28**	**28**	**1.63E-07**	**2.20E-11**	**3.22E-18**
Proposed balanced ternary FA	106	118	**2.55E-06**	**1.28E-10**	**2.94E-16**
Basic gate-based balanced ternary FA	258	270	1.29E-05	2.00E-10	2.32E-15
MUX-based balanced FA	**72**	**108**	5.75E-06	1.61E-10	8.33E-16

4.2 Comparison of results

Of the balanced ternary full adders discussed, the proposed balanced ternary full adder has the best PDP performance, although the capacitor-aided full adder far outperforms in terms of transistor count, although this is weighed against the choice of using capacitors which are large in size, and causes a much higher current than the transistor-based voltage division. Therefore, the proposed balanced ternary FA is the highest performing balanced ternary full adder that was discovered in this project.

When adding two numbers together, the number of digits in the two numbers is equivalent to the number of full adders required to add them together in parallel, with a PDP roughly equivalent to a serial adder. Therefore, to make the simulation results of performance equivalent between binary and ternary, the results for the ternary can be divided by the digit ratio for a specific counting range. The digit ratios for various data sizes are shown in table 6.4, for counting ranges of both $3^m/2$ and 2^n, which shows that for commonly used binary data sizes such as 32 and 64, a digit ratio close to the convergent ratio can be achieved.

Table 6.4: Minimum digits required for a minimum counting range

Positive range	Bits	Trits	Digit ratio
2^2	2	2	1
2^4	4	4	1
2^8	8	6	1.333
2^{16}	16	11	1.454
2^{32}	32	21	1.523
2^{64}	64	42	1.523
$3^3/2$	4	3	1.333
$3^9/2$	14	9	1.555
$3^{27}/2$	42	27	1.555
$3^{81}/2$	128	81	1.580
$3^{243}/2$	385	243	1.584
$\rightarrow \infty$	$\rightarrow \infty$	$\rightarrow \infty$	$\rightarrow 1.58496$

Table 6.5: Equivalent comparison of full adders with adjusted values for the ternary circuits

Circuit	Transistors	w/ inverters	Current	Worst-case delay	PDP
Synthesized binary FA	46	52	7.02E-07	5.80E-11	3.67E-17
28-transistor binary FA	**28**	**28**	**1.63E-07**	**2.20E-11**	**3.22E-18**
Proposed balanced ternary FA	70*	78*	**1.67E-06***	**8.40E-11***	**1.27E-16**
Basic gate-based balanced ternary FA	170*	178*	8.46E-06*	1.31E-10*	1.00E-15
MUX-based balanced FA	48*	71*	3.77E-06*	1.05E-10*	3.56E-16

*Values are divided by 1.523 for an equivalent comparison

The equivalent results are compared in table 6.5.

To compare the simulation results, the transistor count, current, and worst-case delay of the proposed balanced ternary full adders is divided by 1.523 to make the results equivalent. These results suggest that, with the circuits implemented here, ternary-valued computation might not be more efficient than binary-valued computation of basic arithmetics in terms of PDP, despite the benefit of fewer digits. One ternary-valued circuit is implemented with a very low transistor count, however it requires extra components and has a poor PDP performance.

Chapter 7

Conclusion

The conclusions of this thesis is grouped in three categories; those related to the circuit design and synthesis of ternary-valued CNTFET circuits, those related to radix conversion, and conclusions related to the comparison of ternary-valued circuits and binary-valued circuits. The main results from the three chapters related to these three topics are summarized here.

1 Ternary-logic circuit design and synthesis

Several design methods to achieve a ternary-valued circuits were identified and investigated. These include capacitor-aided circuits, the static gate design with pull-up and pull-down transistor networks, gate design from basic gates using ternary algebra, and MUX-based circuits. Ternary algebra was shown to be useful for designing mux-based circuits, as the mux gates are equivalent to the basic functions used. These were implemented in Chapter 6 for the purpose of full adders, to provide a comparison between the investigated circuits and binary-valued circuits. The highest-performing circuit was found to be the static gate design, due to its low power and signal delay, as well as versatility to achieve any function with a single gate.

The static gate architecture was automated with a logic synthesizer which generates CNTFET circuit simulation files from a given truth table. This synthesizer was shown to be capable of correctly generating commonly used ternary-valued circuits, as well as a slew of other functions used in this thesis, including binary-logic circuits. As a proof of concept for the circuit synthesizer, an 8-bit binary to 6-trit balanced ternary radix converter circuit was constructed and simulated using synthesized functions. Our research paper on this logic synthesizer was accepted in the SIMS 2020 conference. An alternative circuit architecture is the capacitor based circuit, but these circuits are limited in the functions it can achieve, have a much higher power consumption, and require a large space in the circuit for capacitors.

The commonly used balanced ternary full adder, the ”compound” design, was improved in terms of PDP with the proposed ”hybrid” design, with the same transistor count and negligible delay difference. The PDP was shown to be decreased by more than a third with the proposed low-power design.

2 Radix conversion and inter-radix compatibility

Using the logic synthesizer, an 8-bit binary to 6-trit balanced ternary radix converter was shown to be capable of conversion at $55.4\mu A$ and 1587 transistors per 1 GB/s of 8-bit data, or potentially a faster data rate. Using this circuit, compatibility between ternary-valued and binary-valued circuits can be achieved, e.g. arithmetic logic circuits or data storage circuits. In terms of PDP, these results are better than reported results in other research on binary to ternary radix converters, such as [56], with a decrease of more than two thirds of their reported PDP for an 8-bit to 6-trit radix converter, due to the low-power operation of the static gate design. However, their circuit has less than half the transistor count. Furthermore, the proposed radix converter of this thesis converts to balanced ternary as opposed to unbalanced ternary. This shows that binary to ternary radix conversion, and by extension ternary to binary radix conversion, is very feasible to implement at high speeds and low power with CNTFET circuits.

3 Comparison of ternary and binary for arithmetic circuits

A selection of binary-valued and ternary-valued full adders were identified, simulated, and compared. To achieve equivalence between ternary and binary, the values were adjusted according to the ratio between number of digits for ternary and binary at specific a data size with multi-digit adders. The results show that the CNTFET circuit ternary-valued multi-trit adders identified in this thesis do not outperform the binary equivalent for the purpose of arithmetic addition. However there is room for optimization, and other research has shown that ternary multiplier circuit can be implemented with a lower length of circuit interconnections [61] when compared to binary multiplier circuits.

4 Discussions

4.1 Validity of simulation results

As with most research into ternary-valued CNTFET circuits, the 32nm Stanford University CNFET model was used in combination with HSPICE to simulate these circuits. This model has been tuned according to the physics and performance of a CNTFET [68], implying that the use of this model should produce a realistic result.

The CNFET model parameters used in the simulations were given in Chapter 1 section 6.2, which are adhering to the specifications of the CNFET model. These parameters may vary between different research, and may result in differing measurements.

The voltage threshold of these CNTFETs were found in simulation, as shown in figure 2.3. Comparing this qualitative graph showing voltage thresholds for different CNT diameters with the quantitative theory of CNTFET voltage threshold in Equation 2.21, they appear to correspond to each other, implying that the voltage thresholds of the simulations are accurate, or close to accurate.

The circuits simulated produce the expected logic output based on the circuit design. For emulating the integration of sub-circuits in a larger circuit, an FO4 load or capacitance output load was used, to provide a more realistic simulation.

These simulations were done with an older version of the HSPICE simulator, with the exact simulation condition and environment to achieve convergence and functionality found through trial and error, which may affect the exact values measured. For absolute certainty in these results, these circuits must be physically implemented, and be recreated in simulation by other researchers.

4.2 Optimality of logic synthesizer

The logic circuit synthesizer produces optimized circuits, however a claim of total optimality can not be made. The circuits are optimized with transistor count in mind, and it is assumed that the lower transistor count will also produce a lower PDP, however this may not be an exact correlation. Furthermore, it does not optimize for the fewest number of inverters required to achieve the input voltages. The synthesizer only optimizes the number of components in the sub-circuit, and only to an extent.

By manual inspection of the synthesized ternary-logic circuits, it was found that a secondary optimization of transistor count might be possible. It could be argued that since these circuits are inherently larger than binary-valued circuits, there is more potential for optimizing the circuits, which may possibly lead to a performance close to or better than a binary circuit. However this is counterbalanced by the inherently higher complexity of a ternary-logic function needed to implement a more complex truth table. The sub-optimality may weaken the argument that ternary-valued circuits can not achieve a better performance than binary-valued circuits in terms of PDP.

4.3 Optimality of the proposed radix converter

While a better balance between power-consumption and signal delay was found compared to [56], albeit with a higher transistor count, the circuit can still be further optimized. This only strengthens the argument for the feasibility of high-speed low-power radix conversion between binary and ternary.

Balanced ternary to binary radix conversion was not implemented, however [57] reports simulation performance of an unbalanced ternary to binary radix converter, with a significantly higher performance than their reported performance of their binary to unbalanced ternary converter. This can likely be attributed to the fact that since the output is binary-valued, the circuit can be constructed with mostly binary-logic components.

4.4 Completeness of binary and ternary circuits comparison

Not every circuit has been covered, due to limited scope of the project. However a reasonable sample of the research is investigated, for the purposes of full adders.

The capacitor-aided full adder could not be implemented, and thus simulation results is not shown in this thesis. However, partial simulations of the circuit indicate a relatively high current and delay at high frequencies, similarly to [47]. Despite the lack of a full simulation and analysis of the circuit, it was concluded that this circuit design was not appropriate for high-efficiency adding, when compared to the other circuits implemented in this thesis. Despite this, a description of the circuit was included for completeness.

This thesis only considered circuits for full adders, which could not be shown to outperform the binary equivalent, however there are other operators, such as multiplication, which may outperform a binary multiplier [61].

While ternary logic could not be shown to outperform binary logic for high-performance logic processing, there are still uses for ternary-valued logic with data storage and transmission, as well as some other areas such as in safety-critical systems, where the middle value can be used as "unknown" or "uninitialized". Furthermore, technologies that encode ternary values with binary digits may benefit from natively ternary circuits. These topics were not investigated in this thesis, and is suggested as future work.

5 Future work

This section describes some suggested future work that was conceived of or discovered during this thesis.

5.1 Future work: Secondary optimization of synthesized circuits

A method to further optimize the circuits generated by the ternary circuit synthesizer is theorized. While the synthesizer produces discrete transistor paths for each optimal grouping found, it should be possible to make sub-group of these transistor paths within each transistor network to produce a more interconnected network. For circuits such as the 3-input sum function B7P7PBPB7, it might be possible for the transistor count in the full-V_{DD} pull-up and pull-down networks to each be reduced from 36 transistors to 23 transistors, as shown in figure 7.1.

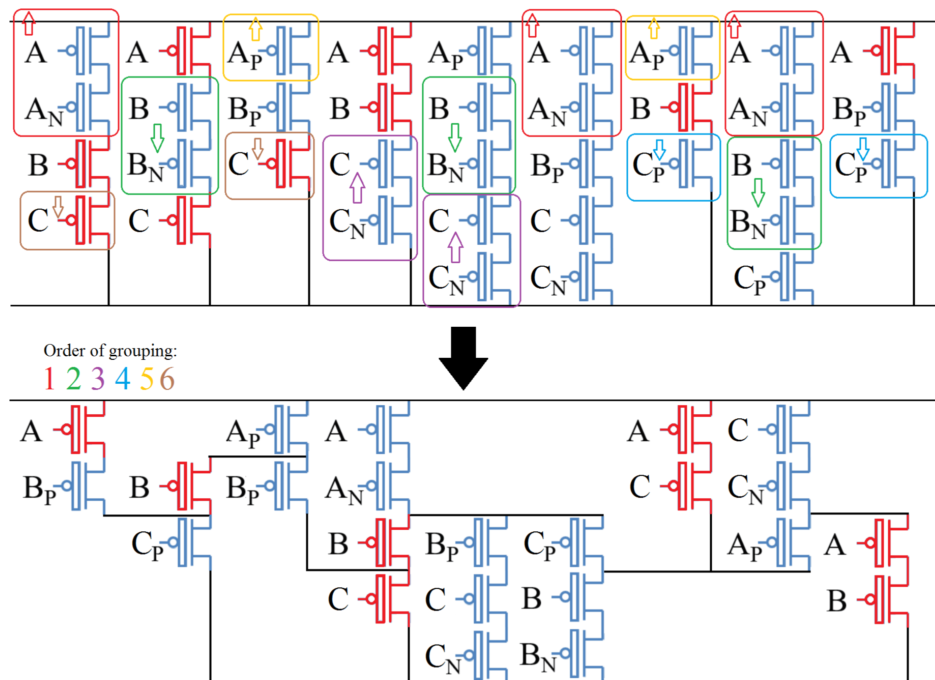


Figure 7.1: Secondary optimization of the full-V_{DD} pull-up transistor network of function B7P7PBPB7

For this example, 6 groupings were manually identified. What this method does is look for repetitions in the collection of transistor-paths, and groups them together at a grouping-space, such as the top or the bottom of the transistor network, specified with an up or down arrow. Each grouping will divide the grouping space into a sub-space, which means that one grouping may restrict others. Therefore, the order of the groupings is relevant. Finding the optimal grouping for the least amount of transistors is a complex search task. The method done here by hand is to first identify the largest groups within each grouping space, i.e. A and A_N for the top grouping space, and B and B_N for the bottom grouping space, which each reduce the transistor count by 4, and so on for each resulting sub-space.

This circuit was not implemented, however no reason could be found for why this should not be a functional optimization of the transistor networks within the synthesized circuits. Further research into these types of optimizations is needed and is suggested as future work for ternary-logic CNTFET circuit synthesis.

5.2 Future work: Full optimization of carry functions

Using the expected inputs method for optimizing the carry functions in the gate-level design of the radix converter, the circuit can be optimized further, with a full circuit analysis of expected node values. The method that was used did not consider the concurrent values of the circuit nodes, but instead lists every possible value for each node in the first row, and works downwards with the assumption that any combination of possible values can occur simultaneously. This leads to an optimized and fully functional circuit, however it is sub-optimal. For a full circuit analysis of the expected node values, the circuit can be emulated in code, which runs through every combination of binary inputs, and records every occurring logic value at each node. Then, a more simplified version of table 5.3 is produced, and the optimized circuit can be constructed in the same manner. Additionally, an efficient ternary to binary may also be constructed using the same methods used here. This is suggested as future work for binary to ternary radix conversion, although the performance shown with the proposed radix converter is arguably satisfactory for most purposes.

5.3 Future work: Optimizing interconnection density

In this thesis, the focus has been on transistor count and PDP performance. However, if the interconnection density can be shown to be significantly lower in a ternary logic circuit compared to an equivalent binary circuit, this may outweigh the extra costs of implementing ternary logic in terms of PDP and transistor count, as the interconnection density is cited as one of the main factors limiting modern processor technology [3][44][45]. Some research [61] has suggested that for certain circuits such as multipliers, ternary CNTFET circuits may outperform binary circuits. Interestingly, the full adder they implement, which is identical to the "compound balanced full adder" in chapter XX, is barely outperformed by the binary equivalent in terms of interconnect length. In this thesis, the "hybrid" design was proposed which reduces the PDP by a third with the same transistor count. It would be interesting to see how this design would fare in terms of interconnection wire length.

5.4 Future work: Natively ternary TNN

In the literature review, a paper on ternary neural networks was discussed. They encode balanced ternary values in binary values. To determine if technologies encoding ternary with binary would benefit from natively ternary logic, more research is needed. The same can be done for similar technologies which encode ternary values, such as ternary state machines, encoding of ternary trees, or ternary data in general, or similar.

5.5 Future work: Memristor-CNTFET circuits

As [3] stated in 1984 that the main restraint of MVL technologies is the lack of a true multistate device, this is still the case for CNTFETs. However, memristors have been shown to be capable of inhabiting three states [66]. Therefore, it is suggested for future work to investigate the possibility of memristor-transistor circuits such as in [69].

Bibliography

[1] M. M. Waldrop, "The chips are down for moore's law," *Nature News*, vol. 530, no. 7589, p. 144, 2016.

[2] G. Hills, C. Lau, A. Wright, S. Fuller, M. D. Bishop, T. Srimani, P. Kanhaiya, R. Ho, A. Amer, Y. Stein, D. Murphy, Arvind, A. Chandrakasan, and M. M. Shulaker, "Modern microprocessor built from complementary carbon nanotube transistors," *Nature*, vol. 572, no. 7771, pp. 595–602, 2019. [Online]. Available: https://doi.org/10.1038/s41586-019-1493-8

[3] Hurst, "Multiple-valued logic—its status and its future," *IEEE Transactions on Computers*, vol. C-33, no. 12, pp. 1160–1179, 1984.

[4] T. Hanyu, M. Kameyama, and T. Higuchi, "Beyond-binary circuits for signal processing," in *1993 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, 1993, pp. 134–135.

[5] N. I. Chernov, N. N. Prokopenko, V. Y. Yugai, and N. V. Butyrlagin, "The application of multi-valued logic elements "minimum" and "maximum" for processing current signals of sensors," in *2017 24th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2017, pp. 450–453.

[6] J. Astola and R. S. Stankovic, "Signal processing algorithms and multiple-valued logic design methods," in *36th International Symposium on Multiple-Valued Logic (ISMVL'06)*, 2006, pp. 16–16.

[7] N. P. Brousentsov, S. P. Maslov, R. A. J, and E. A. Zhogolev, "DEVELOPMENT OF TERNARY COMPUTERS AT MOSCOW STATE UNIVERSITY." [Online]. Available: https://computer-museum.ru/english/setun.htm

[8] J. R. Alvarez and J. Vladimirova, "Software for a small computer "setun"," in *2014 Third International Conference on Computer Technology in Russia and in the Former Soviet Union*, 2014, pp. 110–113.

[9] M. H. Moaiyeri, R. F. Mirzaee, A. Doostaregan, K. Navi, and O. Hashemipour, "A universal method for designing low-power carbonnanotube fet-based multiple-valued logic circuits," *IET Computers & Digital Techniques*, vol. 7, no. 4, pp. 167–181, 2013.

[10] B. Hayes, "Third base," *American scientist*, vol. 89, no. 6, pp. 490–494, 2001.

[11] H. S. Warren, *Hacker's Delight, Second Edition.* Pearson Education, 2013, ch. 12, p. 280.

[12] H. Gundersen, "Aspects of balanced ternary arithmetics implemented using cmos recharged semi-floating gate devices," *University of Oslo*, May 2008.

[13] B. Cambou, P. Flikkema, J. Palmer, D. Telesca, and C. Philabaum, "Can ternary computing improve information assurance?" *Cryptography*, vol. 2, no. 1, p. 6, 2018.

[14] T. Sasao, "Ternary decision diagrams. survey," in *Proceedings 1997 27th International Symposium on Multiple- Valued Logic*, 1997, pp. 241–250.

[15] Yangyang Yu and B. W. Johnson, "A novel safety-critical system modeling approach: ternary decision diagram," in *RAMS '06. Annual Reliability and Maintainability Symposium, 2006.*, 2006, pp. 582–587.

[16] H. Alemdar, V. Leroy, A. Prost-Boucle, and F. Pétrot, "Ternary neural networks for resource-efficient ai applications," 2016.

[17] K. Dasgupta and M. Babu, *A Review on Crypto-Currency Transactions Using IOTA (Technology).* Springer, 01 2019, pp. 67–81.

[18] "Tanner T-Spice analysis software." [Online]. Available: https://www.directindustry.com/prod/mentor-graphics/product-60189-1860234.html

[19] "HSPICE Reference Manual." [Online]. Available: http://www.synopsys.com/

[20] "Stanford CNFET model." [Online]. Available: https://nano.stanford.edu/stanford-cnfet-model

[21] "Stanford VS-CNFET model." [Online]. Available: https://nano.stanford.edu/stanford-cnfet2-model

[22] D. Etiemble, "Ternary circuits: why r=3 is not the optimal radix for computation," *ArXiv*, vol. abs/1908.06841, 2019.

[23] "Revisiting Radix Economy." [Online]. Available: https://sweis.medium.com/revisiting-radix-economy-8f642d9f3c6a

[24] F. G. Reinagel, "Trinary logic operations and circuitry with communication systems applications," in *MILCOM 1986 - IEEE Military Communications Conference: Communications-Computers: Teamed for the 90's*, vol. 2, 1986, pp. 19.4.1–19.4.9.

[25] D. W. Jones, "The Ternary Manifesto · Heptavintimal encoding of ternary values," 2012. [Online]. Available: http://homepage.divms.uiowa.edu/ jones/ternary/hept.shtml

[26] D. E. Knuth, *The Art of Computer Programming: Combinatorial Algorithms, Part 1*, 1st ed. Addison-Wesley Professional, 2011.

[27] H. Gundersen and Y. Berg, "A novel balanced ternary adder using recharged semi-floating gate devices," in *36th International Symposium on Multiple-Valued Logic (ISMVL'06)*, 2006, pp. 18–18.

[28] H. Gundersen, "On the potential of cmos recharged semi-floating gate devices used in balanced ternary logic," in *Proceedings ULSIWS 2008, Dallas*, May 2008.

[29] S. Ahmad and M. Alam, "Balanced-ternary logic for improved and advanced computing," *Int J Comput Sci Inf Technol*, vol. 5, no. 4, pp. 5157–5160, 2014.

[30] R. S. Stanković, J. T. Astola, and M. G. Karpovsky, "Some historical remarks on switching theory."

[31] W. V. O. Quine, *Mathematical logic.* Harvard University Press, 1981.

[32] N. Takagi and K. Nakashima, "Hyperoperations on 0, 1, 2 based on min, max, and universal literal operations," in *33rd International Symposium on Multiple-Valued Logic, 2003. Proceedings.*, 2003, pp. 11–16.

[33] M. Moaiyeri, R. Faghih Mirzaee, K. Navi, and O. Hashemipour, "Efficient cntfet-based ternary full adder cells for nanoelectronics," *Nano-Micro Letters*, vol. 3, 03 2011.

[34] M. M. Shulaker, G. Hills, N. Patil, H. Wei, H.-Y. Chen, H.-S. P. Wong, and S. Mitra, "Carbon nanotube computer," *Nature*, vol. 501, no. 7468, pp. 526–530, Sep 2013. [Online]. Available: https://doi.org/10.1038/nature12502

[35] Kabir, "Performance analysis of cntfet and mosfet focusing channel length, carrier mobility and ballistic conduction in high speed switching," https://pdfs.semanticscholar.org/e60c/34c805c77f154a1f69aea417d01e2f4d054b.pdf, October 2014, (Accessed on 11/02/2020).

[36] H. Dai, A. Javey, E. Pop, D. Mann, and Y. Lu, "Electrical transport properties and field-effect transistors of carbon nanotubes," *Nano*, vol. 01, 07 2006.

[37] T. Ravi and V. Kannan, "Modeling and performance analysis of ballistic carbon nanotube field effect transistor (cntfet)," in *Recent Advances in Space Technology Services and Climate Change 2010 (RSTS CC-2010)*, 2010, pp. 285–289.

[38] M. D. Bishop, G. Hills, T. Srimani, C. Lau, D. Murphy, S. Fuller, J. Humes, A. Ratkovich, M. Nelson, and M. M. Shulaker, "Fabrication of carbon nanotube field-effect transistors in commercial silicon manufacturing facilities," *Nature Electronics*, vol. 3, no. 8, pp. 492–501, Aug 2020. [Online]. Available: https://doi.org/10.1038/s41928-020-0419-7

[39] S.-K. Chang-Jian, J.-R. Ho, and J. Cheng, "Characterization of developing source/drain current of carbon nanotube field-effect transistors with n-doping by polyethylene imine," *Microelectronic Engineering - MICROELECTRON ENG*, vol. 87, pp. 1973–1977, 10 2010.

[40] S. Bari, S. Biswas, A. K. M. Arifuzzman, H. Ahmad, and N. Hasan, "Performance analysis of dynamic threshold-voltage cntfet for high-speed multi-level voltage detector," in *14th International Conference on Computer Modelling and Simulation*, 03 2012, pp. 643–648.

[41] J. Deng and H. . P. Wong, "A compact spice model for carbon-nanotube field-effect transistors including nonidealities and its application—part i: Model of the intrinsic channel region," *IEEE Transactions on Electron Devices*, vol. 54, no. 12, pp. 3186–3194, 2007.

[42] S. Tans, A. Verschueren, and C. Dekker, "Room-temperature transistor based on a single carbon nanotube," *Nature*, vol. 393, pp. 49–52, 1998.

[43] R. Martel, T. Schmidt, H. Shea, T. Hertel, and P. Avouris, "Single- and multi-wall carbon nanotube field-effect transistors," *Applied Physics Letters*, vol. 73, 10 1998.

[44] K. C. Smith, "A multiple valued logic: a tutorial and appreciation," *Computer*, vol. 21, no. 4, pp. 17–27, 1988.

[45] P. Tirumalai and J. T. Butler, "Analysis of minimization algorithms for multiple-valued programmable logic arrays," in *[1988] Proceedings. The Eighteenth International Symposium on Multiple-Valued Logic*, 1988, pp. 226–236.

[46] C. Vudadha, S. Katragadda, and P. S. Phaneendra, "2:1 multiplexer based design for ternary logic circuits," in *2013 IEEE Asia Pacific Conference on Postgraduate Research in Microelectronics and Electronics (PrimeAsia)*, 2013, pp. 46–51.

[47] M. H. Moaiyeri, M. Nasiri, and N. Khastoo, "An efficient ternary serial adder based on carbon nanotube fets," *Engineering Science and Technology, an In-*

ternational Journal, vol. 19, no. 1, pp. 271 – 278, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2215098615001214

[48] S. Kim, T. Lim, and S. Kang, "An optimal gate design for the synthesis of ternary logic circuits," in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2018, pp. 476–481.

[49] C. Vudadha, A. Surya, S. Agrawal, and M. B. Srinivas, "Synthesis of ternary logic circuits using 2:1 multiplexers," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 12, pp. 4313–4325, 2018.

[50] M. Huang, X. Wang, G. Zhao, P. Coquet, and B. Tay, "Design and implementation of ternary logic integrated circuits by using novel two-dimensional materials," *Applied Sciences*, vol. 9, p. 4212, 10 2019.

[51] Fu-Qiang Li, M. Morisue, and T. Ogata, "A proposal of josephson binary-to-ternary converter," *IEEE Transactions on Applied Superconductivity*, vol. 5, no. 2, pp. 2632–2635, June 1995.

[52] T. Sasao, "Radix converters: complexity and implementation by lut cascades," in *35th International Symposium on Multiple-Valued Logic (ISMVL'05)*, 2005, pp. 256–263.

[53] Y. Iguchi, T. Sasao, and M. Matsuura, "On designs of radix converters using arithmetic decompositions," in *36th International Symposium on Multiple-Valued Logic (ISMVL'06)*, 2006, pp. 3–3.

[54] Y. Iguchi, T. Sasao, and M. Matsuura, "Design methods of radix converters using arithmetic decompositions," *IEICE Transactions on Information and Systems*, vol. E90D, 06 2007.

[55] M. M. Arjmand, M. Soryani, K. Navi, and M. A. Tehrani, "A novel ternary-to-binary converter in quantum-dot cellular automata," in *2012 IEEE Computer Society Annual Symposium on VLSI*, Aug 2012, pp. 147–152.

[56] M. Shahangian, S. A. Hosseini, and S. H. Pishgar Komleh, "Design of a multi-digit binary-to-ternary converter based on cntfets," *Circuits, Systems, and Signal Processing*, vol. 38, no. 6, pp. 2544–2563, Jun 2019. [Online]. Available: https://doi.org/10.1007/s00034-018-0977-3

[57] M. Shahangian, S. Hosseini, and R. Faghih Mirzaee, "A universal method for designing multi-digit ternary-to-binary converter using cntfet," *Journal of Circuits, Systems and Computers*, 01 2020.

[58] H. Alemdar, V. Leroy, A. Prost-Boucle, and F. Pétrot, "Ternary neural networks for resource-efficient ai applications," in *2017 International Joint Conference on Neural Networks (IJCNN)*, 2017, pp. 2547–2554.

[59] A. Prost-Boucle, A. Bourge, F. Pétrot, H. Alemdar, N. Caldwell, and V. Leroy, "Scalable high-performance architecture for convolutional ternary neural networks on fpga," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–7.

[60] O. Muller, A. Prost-Boucle, A. Bourge, and F. Pétrot, "Efficient decompression of binary encoded balanced ternary sequences," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 8, pp. 1962–1966, 2019.

[61] K. Kim, S. Kim, Y. Lee, D. Kim, S.-Y. Kim, S. Kang, and B. H. Lee, "Extreme low power technology using ternary arithmetic logic circuits via drastic interconnect length reduction," in *ISMVL*, 2020.

[62] "Program for wednesday, september 23rd," https://easychair.org/smart-program/SIMS2020/2020-09-23.html, (Accessed on 11/26/2020).

[63] H. N. Risto, “Ternary Logic Function Circuit Generator · Halvor64/Ternary · GitHub.” [Online]. Available: https://doi.org/10.5281/zenodo.4015574

[64] S. Lee, S. Kim, and S. Kang, “Ternary logic synthesis with modified quine-mccluskey algorithm,” in *2019 IEEE 49th International Symposium on Multiple-Valued Logic (ISMVL)*, 2019, pp. 158–163.

[65] T. Tanoue, M. Nagatani, and T. Waho, “A ternary analog-to-digital converter system,” in *37th International Symposium on Multiple-Valued Logic (ISMVL'07)*, 2007, pp. 36–36.

[66] S. Bos, J. B. Nilsen, and H. Gundersen, “Post-binary robotics: Using memristors with ternary states for robotics control,” in *2020 IEEE 8th Electronics System-Integration Technology Conference (ESTC)*, 2020, pp. 1–6.

[67] S. Bos, H. Gundersen, and F. Sanfilippo, “uMemristorToolbox : Open source framework to control memristors in Unity for ternary applications,” *International Symposium on Multiple-ValuedLogic (ISMVL)*, vol. 50th, pp. 1–6, 2020.

[68] J. Deng and H. . P. Wong, “A compact spice model for carbon-nanotube field-effect transistors including nonidealities and its application—part ii: Full device model and circuit performance benchmarking,” *IEEE Transactions on Electron Devices*, vol. 54, no. 12, pp. 3195–3205, 2007.

[69] X. Y. Wang, P. F. Zhou, J. K. Eshraghian, C. Y. Lin, H. H. C. Iu, T. C. Chang, and S. M. Kang, “High-density memristor-cmos ternary logic family,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, pp. 1–11, 2020.

Appendix A

Risto et al., 2020, SIMS

Automated synthesis of netlists for ternary-valued n-ary logic functions in CNTFET circuits

Halvor Nybø Risto[1] Steven Bos[1] Henning Gundersen[1]

[1]Ternary Research Group, Department of Science and Industry Systems
[1]University of South-Eastern Norway, Norway, {henning.gundersen}@usn.no

Abstract

This paper is an investigation of automated netlist synthesis for ternary-valued n-ary logic functions, based on a static ternary gate design methodology. We present an open-source C++ implementation, which outputs a ready-to-simulate SPICE subcircuit netlist file for ternary-valued n-ary function circuits. A circuit schematic of the 3-operand carry is demonstrated as synthesized by the netlist generator.

We investigate a holistic (non-compound) approach to designing balanced full-adders by using 3-operand functions as compared to a traditional 2-operand compound design methodology. Three gate-level design approaches (compound, non-compound and hybrid) for the balanced full-adder have been simulated in HSPICE and are compared to each other and the state-of-the-art with simulation results.

Furthermore, we propose to standardize the ternary functions by indexing them. This indexing system allows for the convenience of referencing any possible logic function with no ambiguity. This indexing is necessary as most ternary functions do not have semantic names (e.g. AND, OR) and the amount of unique 3-valued functions grows exponentially with higher arity.

Keywords: ternary, netlist, synthesis, simulation

1 Introduction

In recent years, along with developments of the carbon nanotube field-effect transistor (CNTFET), there have been a handful of papers on the designs and synthesis of ternary or 3-valued logic gates implemented in simulations of CNTFET circuits. One paper in particular (Kim et al., 2018) proposes a design method for ternary logic gates, with the use of pull-up and pull-down networks constructed from a truth table for the circuit. In ternary logic, with only two operands, there are 19683 possible logic gates. With three operands, 7.6e12 logic gates are possible. The process of designing the circuit and writing the netlists of these circuits can be a tedious process, especially for circuits with more than two operands. Therefore, an open-source netlist synthesizer is of much use. The study (Lee et al., 2019) reports to have automated this process, however their code is not open-source.

2 Function Indexing

To unambiguously refer to any of the many logic functions, we propose a simple indexing system.

2.1 Range of index in arities

The number of possible functions for a specific arity and radix can be calculated as in Equation 1, where R is the radix and A is the arity.

$$F_{range} = R^{R^A} \tag{1}$$

Table 1. Range of functions in arities and radices

Arity	Radix 2	Radix 3
1	$2^{2^1} = 4$	$3^{3^1} = 27$
2	$2^{2^2} = 16$	$3^{3^2} = 19683$
3	$2^{2^3} = 256$	$3^{3^3} = 7{,}625{,}597{,}484{,}987$

While in binary, there are few enough functions that naming the useful functions (AND, OR, XOR, etc.) has been a feasible practice, for ternary logic the quantity of possible functions is more unwieldy, as can be seen in Table 1. Therefore, an indexing system is proposed to refer to specific ternary-radix functions.

The indexing system maps every truth table to an index by counting up from 0 to the function range, along with the values of the truth table. As an example, a function always outputting the low value would be the 0th index. Then, the first row of the truth table acts as the three lowest-significance trits of the index, and so on. With a truth table listed vertically, the output values for a specific function can be read in ternary as the function index.

2.2 Heptavintimal index encoding

We adopt the usage of the base-27 heptavintimal notation for ternary values (Jones, 2012), as it conveniently covers three ternary digits (trits) per symbol, as shown in Table 2. As one operand can have one of three values, the truth table of a function is three trits long in each dimension. Therefore, a logic function index can conveniently be encoded with the heptavintimal notation.

Table 2. The heptavintimal notation

Weight(Decimal)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
Ternary	000	001	002	010	011	012	020	021	022	100	101	102	110	111	112	120	121	122	200	201	202	210	211	212	220	221	222
Heptavintimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	G	H	K	M	N	P	R	T	V	X	Z

3 Methodology

Based on the static ternary gate design methodology of (Kim et al., 2018), we have implemented an algorithm in C++, which produces a ready-to-simulate SPICE subcircuit netlist file from a circuit truth table.

The program takes an n-dimensional truth table, and constructs the four truth tables for the pull-up and pull-down networks. Then, for each network, a set of n-dimensional rectangular groupings are found. Each of these groupings will provide a transistor path to the output within each pull-up and pull-down network.

3.1 Usage

To use the program, compile the open-source code in a C++ compiler. The netlists will be generated in the same file directory as the compiled program. When the program starts, it asks for the function arity, and the values of each element in the three-by-three n-dimensional truth table, with values low(0), middle(1), high(2), don't care(x). The filename will be generated as the function index of the specific function. The transistor parameters, as well as which CNTFET model is being used, can be specified with the string variables p0, p1, p2, n0, n1, n2.

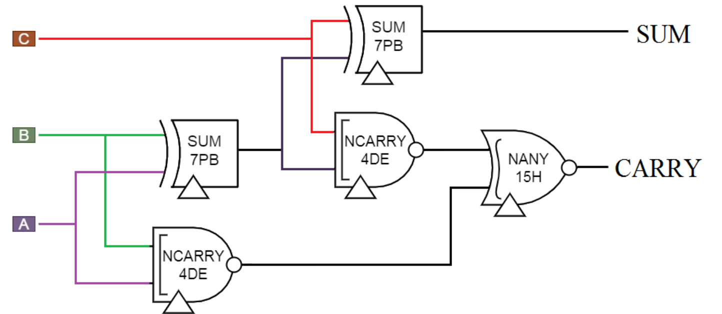
The program will produce a subcircuit which must be connected externally to a 0.9V voltage supply, and the operand inputs. The circuits rely on external 2-transistor Positive Ternary Inverters (PTI) and Negative Ternary Inverters (NTI) to achieve the four different transistor operations detailed in (Kim et al., 2018).

3.2 Logic minimization algorithm

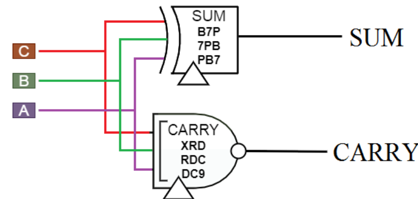
The logic minimization done to produce an optimized circuit is similar to karnaugh-mapping. The grouping algorithm takes the truth tables for each transistor network and draws n-dimensional rectangular groupings which covers every '1' on the truth table for each network, with as few groupings as possible. These groupings represent the transistor-paths in the circuit towards the output within each of the four transistor networks. Each transistor in series narrows down the throughput, until the logical rectangle of a grouping is achieved.

4 Circuit schematics

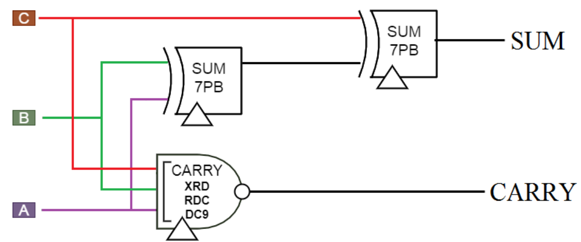
The common procedure for constructing functions with more than two operands is to combine smaller functions to create bigger compound functions. However, circuits with more operands can also be generated with our program. Therefore we investigate the usage of 3-operand functions in a 1-trit balanced full-adder circuit, in the form of a non-compound and a hybrid gate architecture. These architectures are a more holistic view of the function as a relation between input and output. Figure 1 shows three different balanced full-adder circuit design approaches.



Gate-level schematic of the compound balanced full-adder



Gate-level schematic of the non-compound balanced full-adder



Gate-level schematic of the hybrid balanced full-adder

Figure 1. Three approaches for a balanced full-adder

Figure 2 shows the circuit schematic for the 3-carry circuit used in the hybrid 1-trit full-adder, with diameters 1.487 *nm* and 1.018 *nm* being depicted as blue and green respectively.

Table 3. Simulation results with 2fF load capacitor

Circuit	Transistors	Avg. power 500MHz	Avg. power 50MHz	Worst Delay	PDP 500MHz	PDP 50MHz
2-sum (7PB)	40 (32)	$0.61\mu W$	$0.35\mu W$	530ps	0.327e-15 J	0.185e-15 J
2-ncarry (4DE)	10	$0.80\mu W$	$0.80\mu W$	20ps	0.016e-15 J	0.016e-15 J
2-nany (15H)	18	$0.33\mu W$	$0.32\mu W$	40ps	0.013e-15 J	0.013e-15 J
3-sum (B7P7PBPB7)	150 (138)	$0.56\mu W$	$0.34\mu W$	1530ps	0.856e-15 J	0.526e-15 J
3-carry (XRDRDCDC9)	50 (38)	$0.87\mu W$	$0.82\mu W$	30ps	0.026e-15 J	0.024e-15
(Lee et al., 2019) Unbalanced FA	106	(no data reported)	$0.47\mu W$	0.89ns	(no data reported)	**0.421e-15 J**
(Vudadha et al., 2018) Unbalanced FA	98	(no data reported)	**$0.43\mu W$***	1.57ns*	(no data reported)	0.667e-15 J*
Proposed Compound Balanced FA	118 (102)	$2.73\mu W$	$2.29\mu W$	**0.55ns**	1.44e-15 J	1.262e-15 J
Proposed Non-compound Balanced FA	188 (176)	**$1.67\mu W$**	$1.18\mu W$	1.53ns	2.55e-15 J	1.805e-15 J
Proposed Hybrid Balanced FA	118 (102)	$1.96\mu W$	$1.50\mu W$	0.56ns	**1.10e-15 J**	0.840e-15 J

* see (Lee et al., 2019)

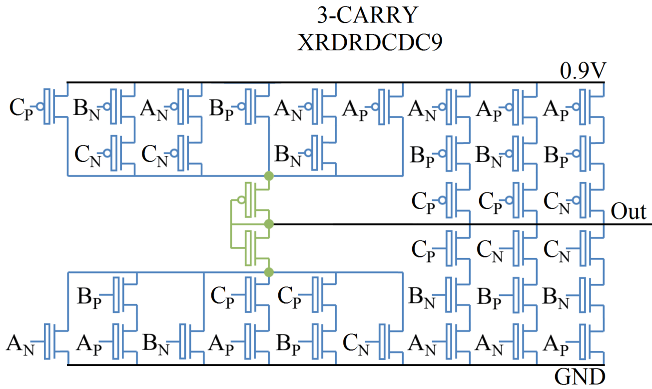


Figure 2. The balanced 3-operand carry function

5 Simulation results

The simulations were done in HSPICE, with the standard 32nm CNFET model technology from Stanford University. (Deng and Wong, 2007)

For the sake of these simulations, voltages below 200mV is considered "low", 250mV to 650mV is "middle", and above 700mV is "high".

Table 3 shows simulation results for some balanced functions, and compares three different circuit concepts of a balanced full-adder. The average current is measured at 500MHz and 50MHz. All measurements include the external PTI and NTI inverters of 2 transistors each where they are required. The transistor count is shown with and without the external inverters. A capacitive load of 2fF was put on the output to ground.

6 Discussion

The runtime performance of the synthesizer can be further optimized for > 7 arity. Under that condition circuit solutions can be found in reasonable time on standard hardware. Due to the sheer number of possible functions, only a minority of the circuit solutions were tested. However, all the tests produced the correct output values.

It should be possible to optimize the circuit solutions even further as we found by manual inspection. This is especially true for circuits with high arity functions. It is interesting to see that the performance of a 1-trit balanced ternary full-adder compared is comparable to an unbalanced version, commonly found in literature.

7 Conclusion

For up to 7 operands, a circuit of any 1-output ternary-valued function can be produced. We show that 3-operand functions can be implemented in circuits such as a 1-trit balanced full-adder, and may in some cases outperform a traditional 2-operand design strategy, as the hybrid full-adder was shown to outperform the compound full-adder in terms of power-delay-product (PDP) performance.

This paper has provided three contributions:

1. An open-source implementation for synthesis of n-ary ternary-valued CNTFET circuits (Risto, 2020).
2. An indexing system has been proposed which allows for any possible ternary-valued logic function to be referenced unambiguously.
3. A novel 3 operand, classical 2 operand, and a hybrid 1-trit balanced full-adder circuits have been simulated and compared with simulation results.

References

J. Deng and H. . P. Wong. A compact spice model for carbon-nanotube field-effect transistors including nonidealities and its application—part i: Model of the intrinsic channel region. *IEEE Transactions on Electron Devices*, 54(12):3186–3194, 2007.

Douglas W. Jones. The Ternary Manifesto · Heptavintimal encoding of ternary values, 2012. URL `http://homepage.divms.uiowa.edu/~jones/ternary/hept.shtml`.

S. Kim, T. Lim, and S. Kang. An optimal gate design for the synthesis of ternary logic circuits. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 476–481, 2018.

S. Lee, S. Kim, and S. Kang. Ternary logic synthesis with modified quine-mccluskey algorithm. In *2019 IEEE 49th International Symposium on Multiple-Valued Logic (ISMVL)*, pages 158–163, 2019.

Halvor Nybø Risto. Automated synthesis of netlists for ternary-valued n-ary logic functions in cntfet circuits, September 2020. URL `https://doi.org/10.5281/zenodo.4015574`.

C. Vudadha, A. Surya, S. Agrawal, and M. B. Srinivas. Synthesis of ternary logic circuits using 2:1 multiplexers. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65 (12):4313–4325, 2018.

Appendix B

Radix converter main simulation file, radixconverter.sp

```
.TITLE 'binary to ternary radix converter'

*************************************************
*For optimal accuracy, convergence, and runtime
*************************************************
.options POST
.options AUTOSTOP
.options INGOLD=2       DCON=1
.options GSHUNT=1e-12 RMIN=1e-15
.options ABSTOL=1N   ABSVDC=1e-4
.options RELTOL=1e-2   RELVDC=1e-2
.options NUMDGT=4       PIVOT=13
.options VNTOL=1M
.OPTION CONVERGE=5
.options dcstep = 1
.OPTIONS METHOD=GEAR
.options runlvl=0
.param TEMP = 25

*************************************************
*Include relevant model files
*************************************************

.lib 'CNFET.lib' CNFET

*************************************************

.include 'f_CRX.sp'
.include 'f_77P.sp'
```

```
.include 'f_BBP.sp'
.include 'f_CDD.sp'
.include 'f_RRD.sp'
.include 'f_55X.sp'
.include 'f_7PP.sp'
.include 'f_8R9.sp'
.include 'f_PPB.sp'
.include 'f_BPP.sp'
.include 'f_7PB.sp'
.include 'f_88R.sp'
.include 'f_DDC.sp'
.include 'f_CDDCDD9CC.sp'
.include 'f_DRRCDD9CC.sp'
.include 'f_DRRCDDCDD.sp'
.include 'f_RDD.sp'
.include 'f_RRDRRDDDC.sp'
.include 'f_RRDRRDDDD.sp'
.include 'f_XXRRRDDDC.sp'
.include 'f_XXRRRDRRC.sp'
.include 'f_XXRRRDRRD.sp'
.include 'f_ZZRRRDDDD.sp'
.include 'f_ZZXXXDXXD.sp'
.include 'STI.sp'
.include 'nti.sp'
.include 'pti.sp'
**************************************************
*Beginning of circuit and device definitions
**************************************************
*Supplies and voltage params:
.param Supply=0.9
.param Vg='Supply'
.param Vd='Supply'

*Some CNFET parameters:
.param Ccsd=0        CoupleRatio=0
.param m_cnt=1       Efo=0.6
.param Wg=0          Cb=40e-12
.param Lg=32e-9      Lgef=100e-9
.param Vfn=0         Vfp=0
.param m=19          n=0
.param Hox=4e-9      Kox=16

******************************************************************
* Define power supply
******************************************************************
Vd   top     Gnd     0.9
Vm   top     vdd     0
```

```
*********************************************************************
* Main Circuits
*********************************************************************

Vin0    b0    gnd    PWL(0ps 0v 1ps 0.90v    1000ps 0.90v
+1001ps 0.90v   2000ps 0.90v    2001ps 0.00v    3000ps 0.00v
+3001ps 0.90v   4000ps 0.90v    4001ps 0.90v    5000ps 0.90v
+5001ps 0.00v   6000ps 0.00v)

Vin0n   b0n   gnd    PWL(0ps 0v 1ps 0.00v    1000ps 0.00v
+1001ps 0.00v   2000ps 0.00v    2001ps 0.90v    3000ps 0.90v
+3001ps 0.00v   4000ps 0.00v    4001ps 0.00v    5000ps 0.00v
+5001ps 0.90v 6000ps 0.90v)

Vin1    b1    gnd    PWL(0ps 0v 1ps 0.90v    1000ps 0.90v
+1001ps 0.90v   2000ps 0.90v    2001ps 0.90v    3000ps 0.90v
+3001ps 0.00v   4000ps 0.00v    4001ps 0.90v    5000ps 0.90v
+5001ps 0.00v 6000ps 0.00v)

Vin1n   b1n   gnd    PWL(0ps 0v 1ps 0.00v    1000ps 0.00v
+1001ps 0.00v   2000ps 0.00v    2001ps 0.00v    3000ps 0.00v
+3001ps 0.90v   4000ps 0.90v    4001ps 0.00v    5000ps 0.00v
+5001ps 0.90v 6000ps 0.90v)

Vin2    b2    gnd    PWL(0ps 0v 1ps 0.90v    1000ps 0.90v
+1001ps 0.00v   2000ps 0.00v    2001ps 0.00v    3000ps 0.00v
+3001ps 0.90v   4000ps 0.90v    4001ps 0.00v    5000ps 0.00v
+5001ps 0.90v 6000ps 0.90v)

Vin2n   b2n   gnd    PWL(0ps 0v 1ps 0.00v    1000ps 0.00v
+1001ps 0.90v   2000ps 0.90v    2001ps 0.90v    3000ps 0.90v
+3001ps 0.00v   4000ps 0.00v    4001ps 0.90v    5000ps 0.90v
+5001ps 0.00v 6000ps 0.00v)

Vin3    b3    gnd    PWL(0ps 0v 1ps 0.90v    1000ps 0.90v
+1001ps 0.00v   2000ps 0.00v    2001ps 0.90v    3000ps 0.90v
+3001ps 0.00v   4000ps 0.00v    4001ps 0.90v    5000ps 0.90v
+5001ps 0.00v 6000ps 0.00v)

Vin3n   b3n   gnd    PWL(0ps 0v 1ps 0.00v    1000ps 0.00v
+1001ps 0.90v   2000ps 0.90v    2001ps 0.00v    3000ps 0.00v
+3001ps 0.90v   4000ps 0.90v    4001ps 0.00v    5000ps 0.00v
+5001ps 0.90v 6000ps 0.90v)
```

```
Vin4     b4      gnd     PWL(0ps  0v  1ps  0.90v     1000ps  0.90v
+1001ps  0.90v   2000ps  0.90v    2001ps  0.00v     3000ps  0.00v
+3001ps  0.90v   4000ps  0.90v    4001ps  0.00v     5000ps  0.00v
+5001ps  0.90v  6000ps  0.90v)

Vin4n    b4n     gnd     PWL(0ps  0v  1ps  0.00v     1000ps  0.00v
+1001ps  0.00v   2000ps  0.00v    2001ps  0.90v     3000ps  0.90v
+3001ps  0.00v   4000ps  0.00v    4001ps  0.90v     5000ps  0.90v
+5001ps  0.00v  6000ps  0.00v)

Vin5     b5      gnd     PWL(0ps  0v  1ps  0.90v     1000ps  0.90v
+1001ps  0.90v   2000ps  0.90v    2001ps  0.90v     3000ps  0.90v
+3001ps  0.00v   4000ps  0.00v    4001ps  0.00v     5000ps  0.00v
+5001ps  0.90v  6000ps  0.90v)

Vin5n    b5n     gnd     PWL(0ps  0v  1ps  0.00v     1000ps  0.00v
+1001ps  0.00v   2000ps  0.00v    2001ps  0.00v     3000ps  0.00v
+3001ps  0.90v   4000ps  0.90v    4001ps  0.90v     5000ps  0.90v
+5001ps  0.00v  6000ps  0.00v)

Vin6     b6      gnd     PWL(0ps  0v  1ps  0.90v     1000ps  0.90v
+1001ps  0.00v   2000ps  0.00v    2001ps  0.00v     3000ps  0.00v
+3001ps  0.90v   4000ps  0.90v    4001ps  0.90v     5000ps  0.90v
+5001ps  0.00v  6000ps  0.00v)

Vin6n    b6n     gnd     PWL(0ps  0v  1ps  0.00v     1000ps  0.00v
+1001ps  0.90v   2000ps  0.90v    2001ps  0.90v     3000ps  0.90v
+3001ps  0.00v   4000ps  0.00v    4001ps  0.00v     5000ps  0.00v
+5001ps  0.90v  6000ps  0.90v)

Vin7     b7      gnd     PWL(0ps  0v  1ps  0.90v     1000ps  0.90v
+1001ps  0.00v   2000ps  0.00v    2001ps  0.90v     3000ps  0.90v
+3001ps  0.00v   4000ps  0.00v    4001ps  0.90v     5000ps  0.90v
+5001ps  0.00v  6000ps  0.00v)

Vin7n    b7n     gnd     PWL(0ps  0v  1ps  0.00v 1000ps  0.00v
+1001ps  0.90v   2000ps  0.90v    2001ps  0.00v     3000ps  0.00v
+3001ps  0.90v   4000ps  0.90v    4001ps  0.00v     5000ps  0.00v
+5001ps  0.90v  6000ps  0.90v)
```

```
**NOTE: circuits require both NTI and PTI of inputs.
**However with binary inputs, NTI and PTI are equivalent.

xsum00 b0n b0n b1 sum00 vdd f_CRX
xnti00 sum00 sum00n vdd nti
xpti00 sum00 sum00p vdd pti

xsum01 sum00 sum00p sum00n b2 b2n sum01 vdd f_77P
xnti01 sum01 sum01n vdd nti
xpti01 sum01 sum01p vdd pti

xsum02 sum01 sum01p sum01n b3 b3n sum02 vdd f_BBP
xnti02 sum02 sum02n vdd nti
xpti02 sum02 sum02p vdd pti

xsum03 sum02 sum02p sum02n b4 b4n sum03 vdd f_77P
xnti03 sum03 sum03n vdd nti
xpti03 sum03 sum03p vdd pti

xsum04 sum03 sum03p sum03n b5 b5n sum04 vdd f_BBP
xnti04 sum04 sum04n vdd nti
xpti04 sum04 sum04p vdd pti

xsum05 sum04 sum04p sum04n b6 b6n sum05 vdd f_77P
xnti05 sum05 sum05n vdd nti
xpti05 sum05 sum05p vdd pti

xsum06 sum05 sum05p sum05n b7 b7n sum06 vdd f_BBP

xcarry01 sum00p b2n carry01 vdd f_RRD
xcarry02 sum01n b3 carry02 vdd f_CDD
xcarry03 sum02p b4n carry03 vdd f_RRD
xcarry04 sum03n b5 carry04 vdd f_CDD
xcarry05 sum04p b6n carry05 vdd f_RRD
xcarry06 sum05n b7 carry06 vdd f_CDD

xsum10a b1 b1n b2 b2n sum10a vdd f_55X

xptisum10 sum10a sum10ap vdd pti
xntisum10 sum10a sum10an vdd nti
xcarry01pti carry01 carry01p vdd pti
xsum10b sum10a sum10ap sum10an carry01 carry01p sum10 vdd f_7PP

xsum10pti sum10 sum10p vdd pti
xsum10nti sum10 sum10n vdd nti
```

```
xcarry02nti carry02 carry02n vdd nti

xsum11 sum10 sum10p sum10n carry02 carry02n sum11 vdd f_PPB

xsum11pti sum11 sum11p vdd pti
xsum11nti sum11 sum11n vdd nti
xcarry03pti carry03 carry03p vdd pti

xsum12a sum11 sum11p sum11n b4 b4n sum12a vdd f_BPP
xsum12apti sum12a sum12ap vdd pti
xsum12anti sum12a sum12an vdd nti

xsum12b sum12a sum12ap sum12an carry03 carry03p sum12 vdd f_7PP

xsum12pti sum12 sum12p vdd pti
xsum12nti sum12 sum12n vdd nti

xsum13a sum12 sum12p sum12n b5 b5n sum13a vdd f_BPP
xsum13apti sum13a sum13ap vdd pti
xsum13anti sum13a sum13an vdd nti

xcarry04nti carry04 carry04n vdd nti

xsum13b sum13a sum13ap sum13an carry04 carry04n sum13 vdd f_PPB

xsum13pti sum13 sum13p vdd pti
xsum13nti sum13 sum13n vdd nti

xcarry05pti carry05 carry05p vdd pti

xsum14 sum13 sum13p sum13n carry05 carry05p sum14 vdd f_7PP

xsum14pti sum14 sum14p vdd pti
xsum14nti sum14 sum14n vdd nti

xsum15a sum14 sum14p sum14n b7 b7n sum15a vdd f_77P
xsum15apti sum15a sum15ap vdd pti
xsum15anti sum15a sum15an vdd nti
xcarry06nti carry06 carry06n vdd nti
xsum15b sum15a sum15ap sum15an carry06 carry06n sum15 vdd f_PPB

xcarry10 b1n b2n carry01p carry10 vdd f_ZZXXXDXXD
xcarry11 sum10n carry02n carry11 vdd f_DDC
xcarry12 sum11p sum11n b4 carry03p carry12 vdd f_DRRCDDCDD
xcarry13 sum12p sum12n b5 carry04n carry13 vdd f_CDDCDD9CC
xcarry14 sum13p carry05p carry14 vdd f_RDD
xcarry15 sum14p sum14n b7n carry06n carry15 vdd f_RRDRRDDDC
```

```
xcarry10pti carry10 carry10p vdd pti
xcarry11nti carry11 carry11n vdd nti
xcarry12pti carry12 carry12p vdd pti
xcarry12nti carry12 carry12n vdd nti
xcarry13nti carry13 carry13n vdd nti
xcarry14pti carry14 carry14p vdd pti
xcarry15pti carry15 carry15p vdd pti
xcarry15nti carry15 carry15n vdd nti

xsum20a carry10 carry10p b3 b3n sum20a vdd f_88R
xsum20apti sum20a sum20ap vdd pti
xsum20anti sum20a sum20an vdd nti
xsum20b sum20a sum20ap sum20an carry11 carry11n sum20 vdd f_PPB
xsum20pti sum20 sum20p vdd pti
xsum20nti sum20 sum20n vdd nti
xsum21a sum20 sum20p sum20n b4 b4n sum21a vdd f_BPP
xsum21apti sum21a sum21ap vdd pti
xsum21anti sum21a sum21an vdd nti
xsum21b sum21a sum21ap sum21an carry12 carry12p carry12n sum21 vdd f_7PB
xsum21pti sum21 sum21p vdd pti
xsum21nti sum21 sum21n vdd nti
xsum22a sum21 sum21p sum21n b5 b5n sum22a vdd f_77P
xsum22apti sum22a sum22ap vdd pti
xsum22anti sum22a sum22an vdd nti
xsum22b sum22a sum22ap sum22an carry13 carry13n sum22 vdd f_PPB
xsum22pti sum22 sum22p vdd pti
xsum22nti sum22 sum22n vdd nti

xsum23a sum22 sum22p sum22n b6 b6n sum23a vdd f_77P
xsum23apti sum23a sum23ap vdd pti
xsum23anti sum23a sum23an vdd nti
xsum23b sum23a sum23ap sum23an carry14 carry14p sum23 vdd f_7PP

xsum23pti sum23 sum23p vdd pti
xsum23nti sum23 sum23n vdd nti

xsum24a sum23 sum23p sum23n b7 b7n sum24a vdd f_BPP
xsum24apti sum24a sum24ap vdd pti
xsum24anti sum24a sum24an vdd nti
xsum24b sum24a sum24ap sum24an carry15 carry15p carry15n sum24 vdd f_7PB

xcarry20 carry10p b3n carry11n carry20 vdd f_RRDRRDDDD
xcarry21 sum20p sum20n b4 carry12p carry12n carry21 vdd f_DRRCDD9CC
xcarry22 sum21p sum21n b5n carry13n carry22 vdd f_RRDRRDDDC
xcarry23 sum22p sum22n b6n carry14p carry23 vdd f_XXRRRDRRD
xcarry24 sum23p sum23n b7 carry15p carry15n carry24 vdd f_DRRCDD9CC
```

```
xcarry20pti carry20 carry20p vdd pti
xcarry21pti carry21 carry21p vdd pti
xcarry21nti carry21 carry21n vdd nti
xcarry22pti carry22 carry22p vdd pti
xcarry22nti carry22 carry22n vdd nti
xcarry23pti carry23 carry23p vdd pti
xcarry24pti carry24 carry24p vdd pti
xcarry24nti carry24 carry24n vdd nti

xsum30a carry20 carry20p b4 b4n sum30a vdd f_88R
xsum30anti sum30a sum30an vdd nti
xsum30apti sum30a sum30ap vdd pti
xsum30b sum30a sum30ap sum30an carry21 carry21p carry21n sum30 vdd f_7PB
xsum30nti sum30 sum30n vdd nti
xsum30pti sum30 sum30p vdd pti
xsum31a sum30 sum30p sum30n b5 b5n sum31a vdd f_77P
xsum31anti sum31a sum31an vdd nti
xsum31apti sum31a sum31ap vdd pti
xsum31b sum31a sum31ap sum31an carry22 carry22p carry22n sum31 vdd f_7PB
xsum31nti sum31 sum31n vdd nti
xsum31pti sum31 sum31p vdd pti
xsum32a sum31 sum31p sum31n b6 b6n sum32a vdd f_BPP
xsum32anti sum32a sum32an vdd nti
xsum32apti sum32a sum32ap vdd pti
xsum32b sum32a sum32ap sum32an carry23 carry23p sum32 vdd f_7PP
xsum32nti sum32 sum32n vdd nti
xsum32pti sum32 sum32p vdd pti
xsum33a sum32 sum32p sum32n b7 b7n sum33a vdd f_BPP
xsum33anti sum33a sum33an vdd nti
xsum33apti sum33a sum33ap vdd pti
xsum33b sum33a sum33ap sum33an carry24 carry24p carry24n sum33 vdd f_7PB

xcarry30 carry20p b4n carry21p carry21n carry30 vdd f_ZZRRRDDDD
xcarry31 sum30p sum30n b5n carry22p carry22n carry31 vdd f_XXRRRDDDC
xcarry32 sum31p sum31n b6 carry23p carry32 vdd f_DRRCDDCDD
xcarry33 sum32p sum32n b7 carry24p carry24n carry33 vdd f_DRRCDD9CC

xcarry30pti carry30 carry30p vdd pti
xcarry31nti carry31 carry31n vdd nti
xcarry31pti carry31 carry31p vdd pti
xcarry32pti carry32 carry32p vdd pti
xcarry32nti carry32 carry32n vdd nti
xcarry33pti carry33 carry33p vdd pti
xcarry33nti carry33 carry33n vdd nti

xsum40 carry30 carry30p carry31 carry31p carry31n sum40 vdd f_8R9
```

```
xsum40pti sum40 sum40p vdd pti
xsum40nti sum40 sum40n vdd nti
xsum41a sum40 sum40p sum40n b6 b6n sum41a vdd f_77P
xsum41apti sum41a sum41ap vdd pti
xsum41anti sum41a sum41an vdd nti
xsum41b sum41a sum41ap sum41an carry32 carry32p carry32n sum41 vdd f_7PB
xsum41pti sum41 sum41p vdd pti
xsum41nti sum41 sum41n vdd nti

xsum42a sum41 sum41p sum41n b7 b7n sum42a vdd f_BPP
xsum42pti sum42a sum42ap vdd pti
xsum42nti sum42a sum42an vdd nti
xsum42b sum42a sum42ap sum42an carry33 carry33p carry33n sum42 vdd f_7PB

xcarry40 carry30p carry31p carry40 vdd f_RDD
xcarry41 sum40p sum40n b6n carry32p carry32n carry41 vdd f_XXRRRDDDC
xcarry42 sum41p sum41n b7 carry33p carry33n carry42 vdd f_DRRCDD9CC

xcarry40pti carry40 carry40p vdd pti
xcarry41nti carry41 carry41n vdd nti
xcarry41pti carry41 carry41p vdd pti
xcarry42pti carry42 carry42p vdd pti
xcarry42nti carry42 carry42n vdd nti

xsum50 carry40 carry40p carry41 carry41p carry41n sum50 vdd f_8R9
xsum50pti sum50 sum50p vdd pti
xsum50nti sum50 sum50n vdd nti
xsum51a sum50 sum50p sum50n b7 b7n sum50a vdd f_77P
xsum50apti sum50a sum50ap vdd pti
xsum50anti sum50a sum50an vdd nti
xsum51b sum50a sum50ap sum50an carry42 carry42p carry42n sum51 vdd f_7PB

**********************************************************************
* Measurements
**********************************************************************
.measure tran iavgsum avg i(vm) from=0p to=6000p
.tran 1p 6000p

.print V(sum06) ***t0
.print V(sum15) ***t1
.print V(sum24) ***t2
.print V(sum33) ***t3
.print V(sum42) ***t4
.print V(sum51) ***t5

.end
```

Appendix C

Ternary logic function circuit synthesizer, main.cpp

```
// Written by Halvor Nyboe Risto for a student short paper for SIMS 2020
// Research group website: http://www.ternaryresearch.com/
// GPL-3 license
// github: www.github.com/halvor64/Ternary-logic-function-circuit-generator/blob/
    master/main.cpp



using namespace std;

#include <iostream>
#include <vector>
#include <math.h>
#include <fstream>
#include <string>
#include <algorithm>
#include <functional>
#include <cctype>
#include <direct.h>
#include <stdlib.h>
#include <stdio.h>


vector<char> truthtable;  //the truthtable for the entire circuit
vector<char> tempVect;
vector<vector<char>> networks;  // the four truth tables, pull-up and pull-down
    networks for 0.9v and 0.45v
vector<char> upvddgnd;    //network[0][x]
vector<char> downvddgnd;  //network[1][x]
vector<char> uphalfvdd;   //network[2][x]
vector<char> downhalfvdd; //network[3][x]
vector<vector<vector<string>>> circuit; //network, group, series. The transistor types
     and their connections are encoded in this vector
vector<char> mask;  // the rectangular groupings are first generated here, then
    compared to the truthtable.
vector<vector<char>> groups;  //groupnr, values. The valid rectangular groupings are
```

```
    stored here.
35
36 int dimensions = -1; // the number of inputs
37 int maskIndex = 0;
38
39
40 int dimensionLevel(int index, int dimension) {  //returns the level a specific
    dimension is for a given index (not its value) NOT ZERO INDEXED
41   return ((index % int((pow(3, dimension)))) / int(pow(3, (dimension - 1))));
42 }
43
44 void maskRecurs(int n, int p1, int p2) {
45   //recursivly goes through all the dimensions an fills in the mask vector between the
      two opposing corner points
46   for (int i = 0; i < 3; i++) {
47     if (n == 1) {
48       maskIndex += 1;
49     }
50     if (!(i > dimensionLevel(p2, n)) && !(i < dimensionLevel(p1, n))) { //"current
    point" not smaller than p1,
51       if (n > 1) {                          //not bigger than p2 in the current
    dimension
52         maskRecurs(n - 1, p1, p2);
53       }
54       else {
55         mask[maskIndex - 1] = '1';
56       }
57     }
58     else {
59       if (n > 1) maskIndex += int(pow(3, n - 1));
60
61     }
62
63   }
64 }
65
66 void drawMask(int p1, int p2) {   // draws an n-dimensional rectangle between two
    corner points
67   fill(mask.begin(), mask.end(), '0');
68   maskIndex = 0;
69   bool error = false;
70   for (int i = 1; i < dimensions + 1; i++) {
71     if (dimensionLevel(p2, i) < dimensionLevel(p1, i)) {
72       error = true;
73     }
74   }
75
76   if (error) {
77     cout << "\nError: one of p2's dimensions is lower than p1's.\n"; // The starting
    corner of the rectangle must be smaller in all dimensions compared to the end
    corner
78
79   }
80   else {
81     maskRecurs(dimensions, p1, p2); // calls the recursive function to draw the
    rectangle in the mask vector
82   }
83 }
84
```

```
85 int main() {
86
87   cout << "\nEnter the function arity(number of inputs, 1~7): ";
88   cin >> dimensions;
89   while (!((dimensions < 8) && (dimensions > 0)) || cin.fail()) {   // Higher arities
     are possible but not recommended
90     cout << "\nEnter the function arity(number of inputs, 1~7): ";
91     cin.clear();
92     cin.ignore(256, '\n');
93     cin >> dimensions;
94   }
95
96   int maxGrpNr = dimensions * dimensions * 100;//dimensions * 1000; // This number
     must be higher for more inputs. Program will crash if it is too low. Must be
     higher than number of groups found.
97   int grpExp = 1.64;
98
99   cout << "\ngenerating vectors...\n";
100  circuit.resize(4, vector<vector<string>>(maxGrpNr, vector<string>(dimensions)));
101
102  for (int i = 0; i < pow(3, dimensions); i++) {
103    truthtable.push_back('0');
104    tempVect.push_back('0');
105    networks.resize(4);
106    networks[0].push_back('0');
107    networks[1].push_back('0');
108    networks[2].push_back('0');
109    networks[3].push_back('0');
110    mask.push_back('0');
111
112    groups.resize(int(pow(pow(3, dimensions), grpExp)));
113    for (int j = 0; j < int(pow(pow(3, dimensions), grpExp)); j++) {
114      groups[j].push_back('0');
115    }
116  }
117
118  char indexyn;
119  cout << "\nWould you like to generate the circuit from an index? (y/n): ";
120  cin >> indexyn;
121  string index = "";
122  if (indexyn == 'y') {
123    bool valid = false;
124    while (!valid) {
125      valid = true;
126      cout << "\nEnter the index ("<< truthtable.size()/3 <<" characters) : ";
127      cin >> index;
128      transform(index.begin(), index.end(), index.begin(), ptr_fun<int, int>(toupper))
     ; // converting to uppercase
129
130      for (int i = 0; i < truthtable.size() - 2; i = i + 3) { //checking if the index
     is valid
131
132        string base27 = "0123456789ABCDEFGHKMNPRTVXZ";
133        bool lettervalid = false;
134        for (int j = 0; j < 27; j++) {                // if a letter in the index is not
       found in base27, the index is not valid.
135          if (index[i/3] == base27[j]) lettervalid = true;
136        }
137        if (!lettervalid) {
```

```
138         cout << "\n" << index[i / 3] << " is not a valid heptavintimal character!\
    nThe valid characters are " << base27 << "\n";
139         valid = false;
140       }
141     }
142   }
143
144   for (int i = 0; i < index.length(); i++) {
145     // THE BASE-27 HEPTAVINTIMAL NOTATION
146     // 000 001 002 010 011 012 020 021 022 100 101 102 110 111 112 120 121 122 200
    201 202 210 211 212 220 221 222
147     //  0 1 2 3 4 5 6 7 8 9 A B C D E F G H K M N P R T V X Z
148
149     // converts the inputted index to truthtable values, starting from lower
    significance
150     if (index[index.length() - i - 1] == '0') { truthtable[i * 3] = '0'; truthtable[
    i * 3 + 1] = '0'; truthtable[i * 3 + 2] = '0'; }
151     if (index[index.length() - i - 1] == '1') { truthtable[i * 3] = '1'; truthtable[
    i * 3 + 1] = '0'; truthtable[i * 3 + 2] = '0'; }
152     if (index[index.length() - i - 1] == '2') { truthtable[i * 3] = '2'; truthtable[
    i * 3 + 1] = '0'; truthtable[i * 3 + 2] = '0'; }
153     if (index[index.length() - i - 1] == '3') { truthtable[i * 3] = '0'; truthtable[
    i * 3 + 1] = '1'; truthtable[i * 3 + 2] = '0'; }
154     if (index[index.length() - i - 1] == '4') { truthtable[i * 3] = '1'; truthtable[
    i * 3 + 1] = '1'; truthtable[i * 3 + 2] = '0'; }
155     if (index[index.length() - i - 1] == '5') { truthtable[i * 3] = '2'; truthtable[
    i * 3 + 1] = '1'; truthtable[i * 3 + 2] = '0'; }
156     if (index[index.length() - i - 1] == '6') { truthtable[i * 3] = '0'; truthtable[
    i * 3 + 1] = '2'; truthtable[i * 3 + 2] = '0'; }
157     if (index[index.length() - i - 1] == '7') { truthtable[i * 3] = '1'; truthtable[
    i * 3 + 1] = '2'; truthtable[i * 3 + 2] = '0'; }
158     if (index[index.length() - i - 1] == '8') { truthtable[i * 3] = '2'; truthtable[
    i * 3 + 1] = '2'; truthtable[i * 3 + 2] = '0'; }
159     if (index[index.length() - i - 1] == '9') { truthtable[i * 3] = '0'; truthtable[
    i * 3 + 1] = '0'; truthtable[i * 3 + 2] = '1'; }
160     if (index[index.length() - i - 1] == 'A') { truthtable[i * 3] = '1'; truthtable[
    i * 3 + 1] = '0'; truthtable[i * 3 + 2] = '1'; }
161     if (index[index.length() - i - 1] == 'B') { truthtable[i * 3] = '2'; truthtable[
    i * 3 + 1] = '0'; truthtable[i * 3 + 2] = '1'; }
162     if (index[index.length() - i - 1] == 'C') { truthtable[i * 3] = '0'; truthtable[
    i * 3 + 1] = '1'; truthtable[i * 3 + 2] = '1'; }
163     if (index[index.length() - i - 1] == 'D') { truthtable[i * 3] = '1'; truthtable[
    i * 3 + 1] = '1'; truthtable[i * 3 + 2] = '1'; }
164     if (index[index.length() - i - 1] == 'E') { truthtable[i * 3] = '2'; truthtable[
    i * 3 + 1] = '1'; truthtable[i * 3 + 2] = '1'; }
165     if (index[index.length() - i - 1] == 'F') { truthtable[i * 3] = '0'; truthtable[
    i * 3 + 1] = '2'; truthtable[i * 3 + 2] = '1'; }
166     if (index[index.length() - i - 1] == 'G') { truthtable[i * 3] = '1'; truthtable[
    i * 3 + 1] = '2'; truthtable[i * 3 + 2] = '1'; }
167     if (index[index.length() - i - 1] == 'H') { truthtable[i * 3] = '2'; truthtable[
    i * 3 + 1] = '2'; truthtable[i * 3 + 2] = '1'; }
168     if (index[index.length() - i - 1] == 'K') { truthtable[i * 3] = '0'; truthtable[
    i * 3 + 1] = '0'; truthtable[i * 3 + 2] = '2'; }
169     if (index[index.length() - i - 1] == 'M') { truthtable[i * 3] = '1'; truthtable[
    i * 3 + 1] = '0'; truthtable[i * 3 + 2] = '2'; }
170     if (index[index.length() - i - 1] == 'N') { truthtable[i * 3] = '2'; truthtable[
    i * 3 + 1] = '0'; truthtable[i * 3 + 2] = '2'; }
171     if (index[index.length() - i - 1] == 'P') { truthtable[i * 3] = '0'; truthtable[
    i * 3 + 1] = '1'; truthtable[i * 3 + 2] = '2'; }
```

```
172       if (index[index.length() - i - 1] == 'R') { truthtable[i * 3] = '1'; truthtable[
    i * 3 + 1] = '1'; truthtable[i * 3 + 2] = '2'; }
173       if (index[index.length() - i - 1] == 'T') { truthtable[i * 3] = '2'; truthtable[
    i * 3 + 1] = '1'; truthtable[i * 3 + 2] = '2'; }
174       if (index[index.length() - i - 1] == 'V') { truthtable[i * 3] = '0'; truthtable[
    i * 3 + 1] = '2'; truthtable[i * 3 + 2] = '2'; }
175       if (index[index.length() - i - 1] == 'X') { truthtable[i * 3] = '1'; truthtable[
    i * 3 + 1] = '2'; truthtable[i * 3 + 2] = '2'; }
176       if (index[index.length() - i - 1] == 'Z') { truthtable[i * 3] = '2'; truthtable[
    i * 3 + 1] = '2'; truthtable[i * 3 + 2] = '2'; }
177
178     }
179
180   }
181   else {
182     // takes inputs for each truthtable value
183     for (int i = 0; i < pow(3, dimensions); i++) {
184       truthtable[i] = 'y';
185       while (truthtable[i] != '0' && truthtable[i] != '1' && truthtable[i] != '2' &&
    truthtable[i] != 'x') {
186         cout << "Enter the function output (0,1,2,x) when ";
187         for (int j = 1; j < dimensions + 1; j++) {
188           cout << "i" << j << " = " << dimensionLevel(i, j) << " ";
189         }
190         cin >> truthtable[i];
191       }
192
193     }
194   }
195
196
197
198   // generates the truthtables for the 4 transistor networks based on the full
    truthtable
199   for (int i = 0; i < truthtable.size(); i++) {
200
201     if (truthtable[i] == 'x') {
202       networks[0][i] = 'x';
203       networks[1][i] = 'x';
204       networks[2][i] = 'x';
205       networks[3][i] = 'x';
206     }
207     if (truthtable[i] == '0') {
208       networks[0][i] = '0';
209       networks[1][i] = '1';
210       networks[2][i] = '0';
211       networks[3][i] = 'x';
212     }
213     if (truthtable[i] == '1') {
214       networks[0][i] = '0';
215       networks[1][i] = '0';
216       networks[2][i] = '1';
217       networks[3][i] = '1';
218     }
219     if (truthtable[i] == '2') {
220       networks[0][i] = '1';
221       networks[1][i] = '0';
222       networks[2][i] = 'x';
223       networks[3][i] = '0';
```

```
224     }
225   }
226 
227 
228   fill(truthtable.begin(), truthtable.end(), '0'); // empties the full truthtable for
        later use
229 
230 
231   for (int n = 0; n < 4; n++) {
232     cout << "\n\nNETWORK " << n << ": \n";
233     for (int i = 0; i < truthtable.size(); i++) {
234       if (i % 3 == 0) cout << "\n";
235       if (i % 9 == 0) cout << "\n\n";
236       cout << networks[n][i];
237     }
238 
239   }
240 
241   int groupNr = 0;
242   bool lessthan = false; // if p2 is lower in any dimension than p1, it is not a valid
         rectangle
243 
244   for (int n = 0; n < 4; n++) {
245   // For each of the 4 network, a set of optimal groupings of 1s are found.
246   // Each grouping represents a transistor-path towards the output.
247     if (n == 0) cout << "\nBuilding the 0.9V pull-up circuit...\n";
248     if (n == 1) cout << "\nBuilding the 0.9V pull-down circuit...\n";
249     if (n == 2) cout << "\nBuilding the 0.45V pull-up circuit...\n";
250     if (n == 3) cout << "\nBuilding the 0.45V pull-down circuit...\n";
251 
252     groupNr = 0;
253     for (int f = 0; f < truthtable.size(); f++) {
254       fill(groups[f].begin(), groups[f].end(), '0');
255     }
256     for (int p1 = 0; p1 < truthtable.size(); p1++) { //for each point in the network
        which is 1 or x
257 
258       if ((networks[n][p1] == '1') || (networks[n][p1] == 'x')) {
259         for (int p2 = p1; p2 < truthtable.size(); p2++) { //  for each point after the
         1 or x
260           lessthan = false;
261 
262           for (int j = 1; j < dimensions + 1; j++) {          // check if it's lower
        in any dimension (it would result in a 0-mask)
263             if (dimensionLevel(p2, j) < dimensionLevel(p1, j)) {
264               lessthan = true;
265             }
266           }
267 
268           if (!lessthan) {      // if it is a valid rectangle, compare it with the
        truthtable of the network
269             drawMask(p1, p2);
270             bool equalMask = true;
271             for (int j = p1; j < p2 + 1; j++) {   // for every point, see if a 1 in
        the mask is a 0 in the network truth table
272               if (mask[j] == '1') {
273                 if (networks[n][j] == '0') {
274                   equalMask = false;
275                 }
```

```
276             }
277           }
278 
279           if (equalMask == true) {        // if there are no 0s compared to the mask
280             bool written = false;
281             bool covered = true;
282 
283             for (int g = 0; g < groupNr; g++) {       // check if a group would be
    covered by the next group, and overwrite it if it does
284                                     //NOTE: This can overwrite multiple groups,
    resulting in duplicate groups
285               covered = true;
286               for (int c = 0; c < p2; c++) {
287                 if ((groups[g][c] == '1') && (mask[c] == '0')) {
288                   covered = false;
289                 }
290               }
291               if (covered) {
292                 for (int j = p1; j < p2 + 1; j++) {
293                   groups[g][j] = mask[j];
294                 }
295                 written = true;
296               }
297             }
298 
299             if (!written) {
300 
301               fill(tempVect.begin(), tempVect.end(), '0');      // checks if the sum
     of the pre-existing groups would cover the mask
302               for (int g = 0; g < groupNr; g++) {
303                 for (int j = 0; j < truthtable.size(); j++) {
304                   if (groups[g][j] == '1') {
305                     tempVect[j] = '1';
306                   }
307                 }
308               }
309 
310               covered = true;
311               for (int j = 0; j < truthtable.size(); j++) {
312                 if (mask[j] == '1') {
313                   if (tempVect[j] == '0') {
314                     covered = false;
315                   }
316                 }
317               }
318               if (!covered) {
319                 for (int j = p1; j < p2 + 1; j++) {
320                   groups[groupNr][j] = mask[j];
321                 }
322                 groupNr += 1;
323                 cout << ".";
324               }
325             }
326 
327           }
328         }
329       }
330     }
331   }
```

```
332
333     bool duplicate = false;
334
335     for (int g = groupNr - 1; g > 0; g--) {        // checks for duplicate groups
336       for (int g2 = 0; g2 < g; g2++) {
337         duplicate = true;
338         for (int c = 0; c < truthtable.size(); c++) {
339           if ((groups[g2][c] == '1') && (groups[g][c] == '0')) {
340             duplicate = false;
341           }
342         }
343       }
344
345       if (duplicate) {
346         for (int j = 0; j < truthtable.size(); j++) {
347           groups[g][j] = '0';
348         }
349         //for () // shift all groups above this one down and subtract groupNr (not implemented)
350
351       }
352
353     }
354
355
356     //for each group, see if the sum of the other groups would cover all the 1s (WITHOUT x)
357     for (int i = 0; i < groupNr; i++) {
358       fill(tempVect.begin(), tempVect.end(), '0');
359       for (int j = 0; j < groupNr; j++) {
360         if (i != j) {
361           for (int k = 0; k < truthtable.size(); k++) {
362             if (groups[j][k] == '1') {
363               tempVect[k] = '1';
364
365             }
366           }
367
368
369         }
370       }
371
372       bool covered = true;
373       for (int k = 0; k < truthtable.size(); k++) {
374         if ((networks[n][k] == '1') && (tempVect[k] != '1')) {
375           covered = false;
376         }
377       }
378       if (covered) {  // if the group is not needed, set it to 0
379         for (int k = 0; k < truthtable.size(); k++) {
380           groups[i][k] = '0';
381         }
382       }
383     }
384
385
386     fill(tempVect.begin(), tempVect.end(), '0');
387     for (int j = 0; j < groupNr; j++) {
388
```

```
389       for (int k = 0; k < truthtable.size(); k++) {
390         if (groups[j][k] == '1') {
391           tempVect[k] = '1';
392 
393         }
394       }
395 
396     }
397 
398 
399     // reconstructing the final full truthtable (if it had x's in it to begin with, it
     don't anymore!)
400     for (int i = 0; i < truthtable.size(); i++) {
401       if (tempVect[i] == '1') {
402         if (n == 0) {
403           truthtable[i] = '2';
404         }
405         else if (n == 1) {
406           truthtable[i] = '0';
407         }
408         else if (n == 2) {
409           if (truthtable[i] != '2') {
410             truthtable[i] = '1';
411           }
412 
413         }
414         //else if (n == 3) {
415         // don't need this one, it's covered by the others
416         //}
417       }
418     }
419 
420 
421     // build the circuit
422     for (int g = 0; g < groupNr; g++) {
423       for (int d = 0; d < dimensions; d++) {
424 
425         bool cut = true;
426         string transType = "111"; //transtype represents the types of transistors. For
     example, "100" is open for low voltage, but not for medium or high
427                        // "010" is a connection of "110" and "011" in series
428         // the circuit is build by covering every group with the use of these
    transistor types
429         for (int L = 0; L < 3; L++) {
430           for (int i = 0; i < truthtable.size(); i++) {
431             if (dimensionLevel(i, d + 1) == L) {
432               if (groups[g][i] != '0') {
433                 cut = false;
434               }
435             }
436           }
437           if (cut) {
438             transType[L] = '0';
439           }
440           else cut = true;
441         }
442         circuit[n][g][d] = transType;
443       }
444     }
```

```
445
446 }
447
448
449 /* NOTE:
450 The circuit can be optmized further at this point.
451 If two transistors of the same type and input in the same network are both connected
      to vdd on one side, they can be merged.
452 The spaces along VDD, GND, OUT can hold mergings, which then produces sub-spaces for
      further merging.
453 This optimization was not implemented here.
454 (the order of transistors in each branch can be swapped around for maximum
    optimization)
455 high-arity functions can be optimized more than low-arity functions
456 (The high arity functions are highly unoptimized in this synthesizer)
457 */
458
459
460 cout << "\n\n final circuit truthtable: \n";
461
462 for (int i = 0; i < truthtable.size(); i++) {
463   if (i % 3 == 0) cout << "\n";
464   if (i % 9 == 0) cout << "\n";
465   if (i % 27 == 0) cout << "\n";
466   cout << truthtable[i];
467 }
468
469 // THE BASE-27 HEPTAVINTIMAL NOTATION
470 // 000 001 002 010 011 012 020 021 022 100 101 102 110 111 112 120 121 122 200 201
    202 210 211 212 220 221 222
471 //  0 1 2 3 4 5 6 7 8 9 A B C D E F G H K M N P R T V X Z
472 index = "";
473 string hept;
474 for (int i = truthtable.size() - 1; i > 0; i -= 3) { // the heptavintimal function
    index is generated
475   hept = truthtable[i];
476   hept += truthtable[i - 1];
477   hept += truthtable[i - 2];
478   if (hept == "000") { index += "0"; }
479   else if (hept == "001") { index += "1"; }
480   else if (hept == "002") { index += "2"; }
481   else if (hept == "010") { index += "3"; }
482   else if (hept == "011") { index += "4"; }
483   else if (hept == "012") { index += "5"; }
484   else if (hept == "020") { index += "6"; }
485   else if (hept == "021") { index += "7"; }
486   else if (hept == "022") { index += "8"; }
487   else if (hept == "100") { index += "9"; }
488   else if (hept == "101") { index += "A"; }
489   else if (hept == "102") { index += "B"; }
490   else if (hept == "110") { index += "C"; }
491   else if (hept == "111") { index += "D"; }
492   else if (hept == "112") { index += "E"; }
493   else if (hept == "120") { index += "F"; }
494   else if (hept == "121") { index += "G"; }
495   else if (hept == "122") { index += "H"; }
496   else if (hept == "200") { index += "K"; }
497   else if (hept == "201") { index += "M"; }
498   else if (hept == "202") { index += "N"; }
```

```
499     else if (hept == "210") { index += "P"; }
500     else if (hept == "211") { index += "R"; }
501     else if (hept == "212") { index += "T"; }
502     else if (hept == "220") { index += "V"; }
503     else if (hept == "221") { index += "X"; }
504     else if (hept == "222") { index += "Z"; }
505 
506   }
507 
508   cout << "\nheptavintimal function index: " << index;
509   cout << "\n\n";
510 
511 
512   if (dimensions>4)cout << "Custom filename is recommended for high-arity functions\n"
        ;
513   cout << "Would you like to use the index as the filename? (y/n): ";
514   char nameyn = 'n';
515   cin.ignore(100000, '\n');
516   cin >> nameyn;
517   string filename;
518   if (nameyn == 'y') {
519     filename = "f_";
520     for (int i = 0; i < (int(pow(3, dimensions - 1))) - index.length(); i++) {
        filename += "0"; }
521     filename += index;
522   } else {
523     cout << "Enter the filename: ";
524     cin >> filename;
525   }
526 
527 
528 
529   if (_mkdir("./functions") == 0){
530     printf("Directory './functions' was successfully created\n");
531   } //else printf("Problem creating directory './functions'\n");
532 
533 
534   ofstream myfile;
535   string path = "functions/";
536   path += filename;
537   path += ".sp";
538   myfile.open(path);
539 
540 
541 
542 
543   // SPECIFY TRANSISTOR MODEL AND PARAMETERS HERE
544   string p0 = " gnd PCNFET Lch=Lg  Lgeff='Lgef' Lss=32e-9  Ldd=32e-9 \n+Kgate = 'Kox'
        Tox = 'Hox' Csub = 'Cb' Vfbp = 'Vfp' Dout = 0  Sout = 0  Pitch = 20e-9 tubes = 3
        n2 = n  n1 = 13 "; //" ptype 1.018nm";
545   string n0 = " gnd NCNFET Lch=Lg  Lgeff='Lgef' Lss=32e-9  Ldd=32e-9 \n+Kgate = 'Kox'
        Tox = 'Hox' Csub = 'Cb' Vfbn = 'Vfn' Dout = 0  Sout = 0  Pitch = 20e-9 tubes = 3
        n2 = n  n1 = 13 "; //" ntype 1.018nm";
546 
547   string n1 = " gnd NCNFET Lch=Lg  Lgeff='Lgef' Lss=32e-9  Ldd=32e-9 \n+Kgate = 'Kox'
        Tox = 'Hox' Csub = 'Cb' Vfbn = 'Vfn' Dout = 0  Sout = 0  Pitch = 20e-9 tubes = 3
        n2 = n  n1 = 10 "; //" ntype 0.783nm";
548   string n2 = " gnd NCNFET Lch=Lg  Lgeff='Lgef' Lss=32e-9  Ldd=32e-9 \n+Kgate = 'Kox'
        Tox = 'Hox' Csub = 'Cb' Vfbn = 'Vfn' Dout = 0  Sout = 0  Pitch = 20e-9 tubes = 3
```

```
    n2 = n  n1 = 19 "; //" ntype 1.487nm";
549
550  string p1 = " gnd PCNFET Lch=Lg  Lgeff='Lgef' Lss=32e-9  Ldd=32e-9 \n+Kgate = 'Kox'
    Tox = 'Hox' Csub = 'Cb' Vfbp = 'Vfp' Dout = 0  Sout = 0  Pitch = 20e-9 tubes = 3
    n2 = n  n1 = 10  "; //" ptype 0.783nm";
551  string p2 = " gnd PCNFET Lch=Lg  Lgeff='Lgef' Lss=32e-9  Ldd=32e-9 \n+Kgate = 'Kox'
    Tox = 'Hox' Csub = 'Cb' Vfbp = 'Vfp' Dout = 0  Sout = 0  Pitch = 20e-9 tubes = 3
    n2 = n  n1 = 19  "; //" ptype 1.487nm";
552
553
554
555  myfile << ".subckt " << filename << " "; //<< " i0 i0_p i0_n i1 i1_p i1_n out vdd\n
    "; // circuit relies on external PTI and NTI
556
557  for (int i = 0; i < dimensions; i++) {   // CREATING THE SUBCIRCUIT INTERFACE. will
    only require PTI and NTI when necessary
558    bool bI = false;
559    bool bIP = false;
560    bool bIN = false;
561    for (int n = 0; n < 4; n++) {
562      for (int g = 0; g < maxGrpNr; g++) {
563        if (n % 2 == 0) {
564          if (circuit[n][g][i] == "100" || circuit[n][g][i] == "110" || circuit[n][g][
    i] == "010") {
565            bI = true;
566          }
567          if (circuit[n][g][i] == "001") {
568            bIP = true;
569          }
570          if (circuit[n][g][i] == "011" || circuit[n][g][i] == "010") {
571            bIN = true;
572          }
573        }
574        else {
575          if (circuit[n][g][i] == "001" || circuit[n][g][i] == "011" || circuit[n][g][
    i] == "010") {
576            bI = true;
577          }
578          if (circuit[n][g][i] == "110" || circuit[n][g][i] == "010") {
579            bIP = true;
580          }
581          if (circuit[n][g][i] == "100") {
582            bIN = true;
583          }
584        }
585
586      }
587    }
588    if (bI) myfile << "i" << i << " ";
589    if (bIP) myfile << "i" << i << "_p ";
590    if (bIN) myfile << "i" << i << "_n ";
591  }
592
593  myfile << "out vdd\n"; // end of inputs/outputs interface
594
595  myfile << "\n\nxp0 up out out" << p0;
596  myfile << "\nxn1 out out down" << n0 << "\n";
597  int connections = 0; //counts number of connection nodes
598  int transistors = 2; //counts number of transistors
```

```
599
600
601 string connect1 = ""; // connection variables (these depend on the network and group
      number)
602 string connect2 = "";
603 string connect3 = "";
604 string out = "";    // the connection to the output (out, up, down)
605 string vsource = "";  // the first connection (gnd, vdd)
606
607 for (int n = 0; n < 4; n++) {
608
609
610   if (n == 0) {
611     out = "out";
612     vsource = "vdd";
613     myfile << "\n\n***pullup full" << endl;
614   }
615   if (n == 1) {
616     out = "out";
617     vsource = "gnd";
618     myfile << "\n\n***pulldown full" << endl;
619   }
620   if (n == 2) {
621     out = "up";
622     vsource = "vdd";
623     myfile << "\n\n***pullup half" << endl;
624   }
625
626   if (n == 3) {
627     out = "down";
628     vsource = "gnd";
629     myfile << "\n\n***pulldown half" << endl;
630   }
631
632
633   for (int g = 0; g < maxGrpNr; g++) {
634     // the first and last groups to be implemented indicates when the connections
    should be at vsource and out
635     // in circuit[n][g][d], what number d is the first and last valid one? (empty,
    111, 000 are not valid)
636     int firstDimension = -1;
637     int lastDimension = 0;
638     for (int dd = 0; dd < dimensions; dd++) {
639
640       if (circuit[n][g][dd] != "000" && circuit[n][g][dd] != "111" && !circuit[n][g
    ][dd].empty()) {
641         lastDimension = dd;
642         if (firstDimension == -1) {
643           firstDimension = dd;
644         }
645       }
646     }
647
648     for (int d = 0; d < dimensions; d++) {
649       if (!circuit[n][g][d].empty() && circuit[n][g][d] != "000" && circuit[n][g][d]
     != "111") {
650
651         // connection variables are defined
652         if (d == firstDimension) {
```

```
653             myfile << "\n";
654             connect1 = vsource;
655           }
656           else {
657             connect1 = 'p' + to_string(connections);
658             connections += 1;
659           }
660 
661           if (d == lastDimension) {
662             if (circuit[n][g][d] == "010") {
663               connect2 = 'p' + to_string(connections);
664               connections += 1;
665               connect3 = out;
666             }
667             else {
668               connect2 = out;
669             }
670           }
671           else {
672             connect2 = 'p' + to_string(connections);
673             if (circuit[n][g][d] == "010") {
674               connections += 1;
675               connect3 = 'p' + to_string(connections);
676 
677             }
678           }
679 
680           // circuit is built using the "transistor types" in the circuit vector and
    the connection variables
681           if (n % 2 == 0) {
682             if (circuit[n][g][d] == "100") {  // small ptype I
683               myfile << "\nxp" << transistors << " " << connect1 << " i" << d << " "
    << connect2 << p1;
684               transistors += 1;
685             }
686             if (circuit[n][g][d] == "110") {  // big ptype I
687               myfile << "\nxp" << transistors << " " << connect1 << " i" << d << " "
    << connect2 << p2;
688               transistors += 1;
689             }
690             if (circuit[n][g][d] == "001") {  // big ptype I_P
691               myfile << "\nxp" << transistors << " " << connect1 << " i" << d << "_p "
     << connect2 << p2;
692               transistors += 1;
693             }
694             if (circuit[n][g][d] == "011") {  // big ptype I_N
695               myfile << "\nxp" << transistors << " " << connect1 << " i" << d << "_n "
     << connect2 << p2;
696               transistors += 1;
697             }
698             if (circuit[n][g][d] == "010") {  // big ptype I + big ptype I_N
699               myfile << "\nxp" << transistors << " " << connect1 << " i" << d << " "
    << connect2 << p2;
700               transistors += 1;
701               myfile << "\nxp" << transistors << " " << connect2 << " i" << d << "_n "
     << connect3 << p2;
702               transistors += 1;
703             }
704           }
```

```
705           else {
706             if (circuit[n][g][d] == "100") {  // big ntype I_N
707               myfile << "\nxn" << transistors << " " << connect1 << " i" << d << "_n "
     << connect2 << n2;
708               transistors += 1;
709             }
710             if (circuit[n][g][d] == "110") {  // big ntype I_P
711               myfile << "\nxn" << transistors << " " << connect1 << " i" << d << "_p "
     << connect2 << n2;
712               transistors += 1;
713             }
714             if (circuit[n][g][d] == "001") {  // small ntype I
715               myfile << "\nxn" << transistors << " " << connect1 << " i" << d << " "
    << connect2 << n1;
716               transistors += 1;
717             }
718             if (circuit[n][g][d] == "011") {  // big ntype I
719               myfile << "\nxn" << transistors << " " << connect1 << " i" << d << " "
    << connect2 << n2;
720               transistors += 1;
721             }
722 
723             if (circuit[n][g][d] == "010") {  // big ntype I_P + big ntype I
724               myfile << "\nxn" << transistors << " " << connect1 << " i" << d << "_p "
     << connect2 << n2;
725               transistors += 1;
726               myfile << "\nxn" << transistors << " " << connect2 << " i" << d << " "
    << connect3 << n2;
727               transistors += 1;
728             }
729           }
730         }
731       }
732     }
733   }
734 
735   myfile << "\n\n.ends\n\n";
736   myfile.close();
737 
738   cout << "\n\n Circuit outputted into functions/" << filename << ".sp\n\n";
739 
740   system("pause");
741 
742 
743 
744   return 0;
745 }
```