Title
Toward a Generalized Model of Hardware Parallelism and Reconfigurability

Author
Black, Trevor

Publication Date
2017

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA

Los Angeles

Toward a Generalized Model of Hardware Parallelism and Reconfigurability

A thesis submitted in partial satisfaction

of the requirements for the degree

Master of Science in Electrical Engineering

by

Trevor David Black

2017

ABSTRACT OF THE THESIS

Toward a Generalized Model of Hardware
Parallelism and Reconfigurability

by

Trevor David Black

Master of Science in Electrical Engineering
University of California, Los Angeles, 2017
Professor Dejan Marković, Chair

The end of Moore's Law has created an increasing reliance on reconfiguration in the algorithm and in the hardware space to continue exponential speedup of application runtime. Increasingly exotic hardware designs such as GPUs, FPGAs, and Universal DSP devices are created with high levels of logical compute in mind, but necessitate increasingly complex control schemes to function as designed. This paper presents a unified mathematical model of Boolean computation for use in designing any reconfigurable or parallel hardware. Treating control signals as simple input signals, the bandwidth of a system's controls determines how quickly the device can reconfigure. The maximum amount of computation, and the set of all problems that a device can be compute, can be cheaply deduced by the bandwidth and span of the control network.

The thesis of Trevor David Black is approved.

William J. Kaiser

C.-K. Ken Yang

Dejan Marković, Committee Chair

University of California, Los Angeles

2017

To my sister Emily Black, and best friend Alexander Olson

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Acknowledgements

It would be all too remiss if I did not first and foremost thank my advisor Prof. Dejan Marković for his seemingly endless patience on this, my, endeavor. His aid over these last few years has not only pushed this thesis toward being a much stronger piece of literature, but has also pushed me as an academic and as a researcher. This simply would not have happened without his aid. It is also important to mention the great hivemind that Sumeet Singh, Uneeb Rathore, and myself have developed over the last 2 years of collaboration and discussion. It can be particularly strange sometimes, in that we'll all realize that we had been independently arguing the same thing to the professor across multiple different meetings. In truth, the hivemind seems to agree on 80% of everything. I am simply fortunate in that I am the first person to be writing most of this down, otherwise I would have needed to figure something else to write. A hearty thank-you and farewell to Sina Basir-Kazeruni for saving my bacon come tape-out time. You taught me how to do much of the nitty-gritty work for completing the tape-out, and have been a consistent source of Good Grad School Advice these past few years. You also pulled everyone together, including Hari Chandrakumar and Dejan Rozgić when the voltage rails for the memory layout suddenly mysteriously disappeared from the transistor files. And then those needed to be hand-laid for every single memory element on the chip. Lastly, I want to thank my father, David Black, my mother, Karen Black, my sister, Emily Black, and of course, Alec Olson. All of you have helped me in a million small ways and more, to get to this, today.

Thank You

CHAPTER 1

Introduction

Ultimately, Moore's Law remains a law of economics rather than strictly a law of engineering. While the engineering of smaller and smaller process nodes has grown increasingly difficult, the economics have changed little over the last decade. The world demands has demanded increased compute continuously has over the last two centuries. In truth, there has always been two versions of Moore's Law. The guess hedged by Gordon Moore in 1965 was that the number of transistors on die would double roughly every 2 years [1]. This doubling of transistors has slowed in recent years, but this slowing of the transistor count doubling is not what the collective community is referring to when the collective community exclaim that Moore's Law has ended. Moore's Law as entered the public consciousness is that the clock speed of processors would double every 2 years [2]. For much of early central processing unit (CPU) design, doubling your transistors had a linear speedup of core clock. But after hitting the power wall in 2005 [3], many companies, Intel and AMD specifically, pushed for an increase in instructions per cycle, the amount of work a CPU would accomplish in a clock [4].

Originally modeled after the Von Neumann Architecture [5], depicted in Figure 1-1, CPUs were able to increase their effectiveness by either increasing the native clock speed at which all operations ran, or increased the amount of operations that were run any given cycle. In CPUs all operations of interest were included in the logic unit, and the specific operations to execute were selected by the control unit. This simple architectural model has become the ubiquitous computer architecture model understood by the greater computer architecture community.

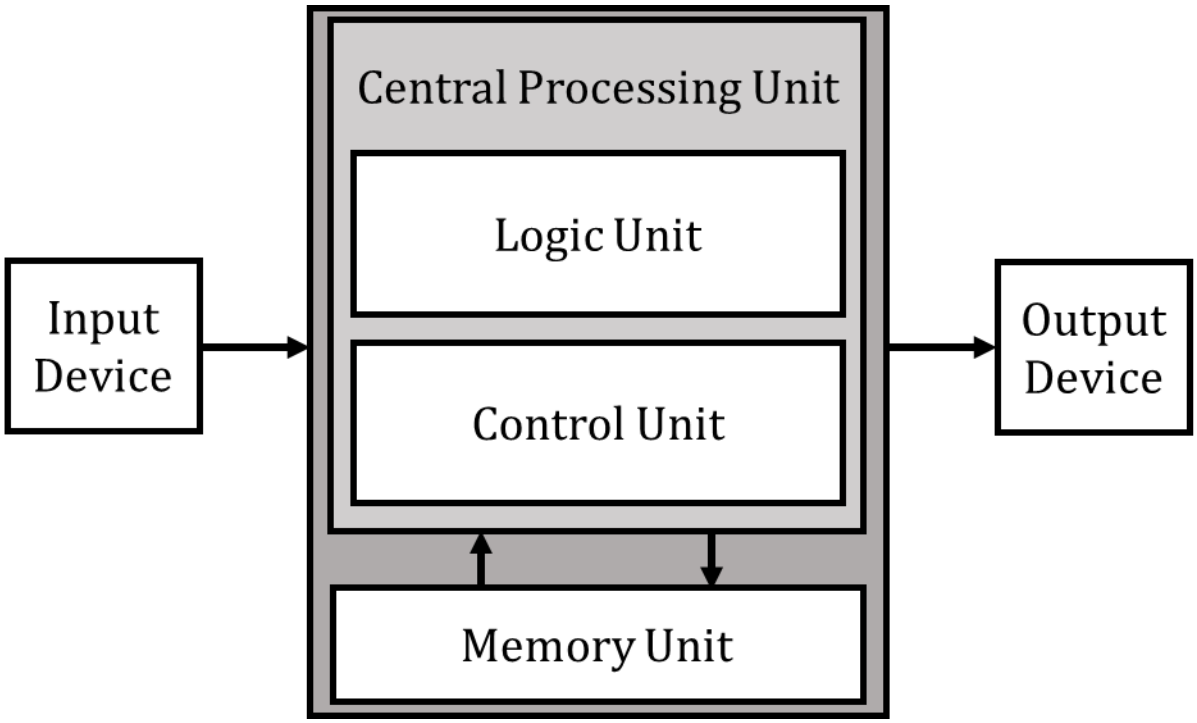


Figure 1-1: The Von Neumann Architecture

And for much of early computer architecture design, having a thorough understanding of the Von Neumann Architecture was sufficient. If you wanted to meaningfully improve the CPU, the easiest means to accomplish this was to speed up the logic unit. But as manufacturing techniques pushed transistor design deep into the submicron, the inherent complexities have proven too costly to economically continue the Consumers' Moore's Law. Meaningful improvements to compute power needed to come from elsewhere, with clock speeds having stagnated for over a decade. Designers have moved to maximizing the parallelism and reconfigurability of their existing compute devices to meet the compute demanded by the compute market. Massively parallel devices such as Field-Programmable Grid Arrays (FPGA) are increasingly becoming a part of the dataserver landscape where existing x86 CPU solutions cannot provide the necessary compute at acceptable efficiency levels.

Understandably, a model which contains only a single control unit and a single logic unit does not lend itself well to a compute environment which increasingly demands parallelism. Worse, as newer compute technologies with increasing reconfigurability become increasingly important sources of compute, the notion of what exists as a logic unit versus what exists as a control unit has become increasingly muddled. Devices like the Digital Signal Processor (DSP), the Graphics Processing Unit (GPU), and the FPGA move major control off to a separate compute devices running elsewhere. It is trivial to increase the logic unit count for increasing parallelism, but as the designer adds each new logic unit should the new unit demand its own control unit, or can a collection of logic units share a reduced number of control units? Indeed, for GPUs and DSPs, control units are shared across multiple logic units acting in parallel. The closest modern device which contains unique control units for each logic unit would be the multi-cored processors that are sold as most modern CPUs today. The class of FPGA compute devices are general purpose compute devices, but lack any onboard control logic for their logical units, where this control is offboard with the CPU programming (and controlling) the device.

The Von Neumann Architecture no longer functions as a representative model of the spectrum of compute devices as they exist today, and will only become less representative as newer compute devices are designed and implemented, devices such as the Universal Digital Signal Processors (UDSP) [6], Tensor Product Units (TPU) [7], and Quantum Computers, which use increasing reconfigurability and sophisticated control systems for specific computational problems. This thesis attempts to marry the foundations of computer architecture and hardware computation to mathematical models. Following this introduction, Chapter 2 (Discrete Spatial Logic Systems) attempts a very basic discussion after the establishment of an initial set of strong assumptions. Chapter 3 (Data Flow Graphs) uses this assumption set of generalized digital hardware to establish similarity between digital design and Data Flow Graphs, then establishes the important consequences of this insight. Chapter 4 through Chapter 6 each attempt to broaden the model by removing assumptions, starting with adding control signals in Chapter 4

(Reconfigurable Discrete Spatial Logic Systems), adding logical delays in Chapter 5 (Adding the Continuity of Time), and finally adding memory elements in Chapter 6 (Discretized Memory Elements). The final model is summed up and compared with the Von Neumann model as the thesis concludes in Chapter 7.

CHAPTER 2

Discrete Spatial Logic Systems

Before a thorough analysis of hardware reconfigurability can be undertaken, a set of assumptions and definitions need to be established. In this thesis, all systems of interest will be purely digital systems. Many of the ideas established here can be extended to continuous analog systems, but the typical voltage and current nonlinearity of analog circuits, alongside the prevalence of feedback represents additional complexity which is of questionable use to a purely digital system. Digital systems of interest include classical CMOS logic, binary memory cells, and both fuses and anti-fuses. While much of modern computing is devoted solely to CMOS and memory cells, fuses are also included because of their possible placement in various FPGAs and reprogrammable hardware devices. The simplicity of the binary alphabet of Boolean Algebra likely maps well to the readers background, and simplifies the amount of material necessary to cover. The explicit choice of using only the binary alphabet simplified the models' explanation and the actual models themselves.

First, the internal state function remains constant for all time, is deterministic for a set of inputs, and contains no random effects – this, can be stated thusly: The internal state function of the system is well defined for all time. Second, all input signals represent independently joint random variables. Additionally, the random distribution of these independent input signals is uniformly distributed over the {0,1} alphabet. Put another way, an input has a 50% chance of being a 1 and a 50% chance of being a 0. An independent and uniformly distributed set of input variables maximizes the entropy of the input space, which will be necessary for determining

maximum bounds. Throughout the body of the thesis these three assumptions will be held: All systems will be purely digital, the internal state function is well defined for all time, and the input set entropy is maximized. There are further assumptions which are included in initial analysis, which are to be removed in later chapters. First, the logic system contains no controls signals. Second, the delay from any one input to any one output has a time of zero seconds, or put simply, all logic paths have zero delay. The last initial assumption to be removed is that the state function is considered memoryless, such that the output of the system is wholly defined by the state function and present inputs.

Assumptions

1. All systems will be purely digital
2. The internal state function is well defined for all time
3. Maximum entropy for the input set
4. Logic system contains no control signals
5. All logic paths have zero delay
6. Memoryless state function

Constrained by the given set of assumptions, the digital systems of interest to the text can be formalized. The systems are purely digital and can be summed up as having a set of Boolean inputs (lacking control signal inputs), having a set of Boolean outputs, and an internal state function which is instantaneous and has no information of the past. A simple illustration of this abstraction can be seen in Figure 2-1. No real hardware satisfies these conditions when practical considerations are taken into account, and many of these practical considerations are included in subsequent chapters, but for the time being an expanded assumption set simplifies the initial models.

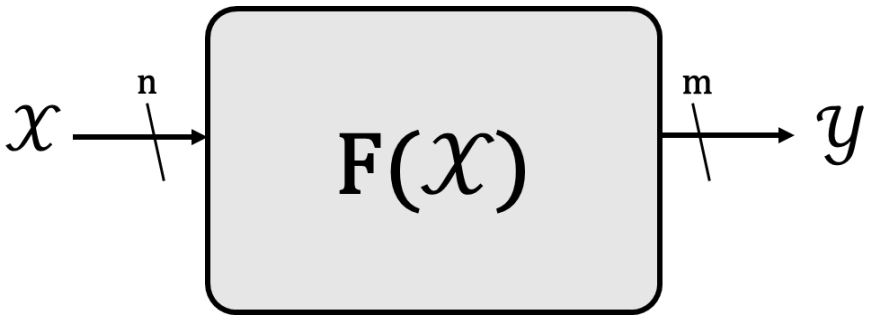


Figure 2-1: The Discrete Spatial Logic System

From Figure 2-1, all computational systems which satisfy the strict assumptions can be completely defined by **n**, the number of inputs to the system, **m**, the number of outputs from the system, and **F**, the instantaneous memoryless state function of the system. The state function takes as input **n** binary signals, with each signal capable of independently representing a 0 or 1. The union of these **n** Boolean symbols produces an input set of cardinality 2^n, or put simply, there are 2^n possible distinct inputs to the system. Under assumptions 2, 5, and 6: the state machine is well defined for all time, and the state function contains no information about the past. The internal state logic is defined as having a function, meaning that any one distinct input can map to exactly one output. The functional definition, alongside the total distinct number of inputs, means that the state function can produce a total of 2^n meaningfully unique function outputs. Looking to the output of the whole digital system, there includes **m** Boolean outputs, which following from the logic for the input signals, means that there is a total of 2^m possible distinct outputs of the digital system.

The max amount of information that the state function can output is **n** bits of information selected from the set of 2^n possible input combinations. Yet the system output is defined as having **m** bits. In all cases the max amount of information that the state function can output is wholly determined by the **n** signals at the input. However, the total information that the system can output is also bounded by the **m** signals at the output. Therefore, the total

information that can be conveyed at the output is: The minimum of the cardinality of the input set and the cardinality of the output set, regardless of the specifics of the internal state function.

$$X = \{x : 0 \leq x < 2^n\}$$
$$Y = \{y : 0 \leq x < 2^m and\ y = F(x), x \in X\}$$
$$|Y| = min(2^n, 2^m)$$

Another way to think of this is that the entropy of the output of the system cannot have greater entropy than the input of the system. Hence the entropy is bound both by input and output set size. Functions in digital systems can be onto (surjective), one-to-one (injective), both (bijective), or neither. The internal state function of the digital system maps an input set of cardinality 2^n to an output set of cardinality 2^m. For any given digital system which has a greater number of outputs than inputs, these systems can only be injective. The total number of possibilities of the output is bound by the reduced input set, so the complexity of the internal state function is bound by the input set. For any given digital system which has a greater number of inputs, these systems can only by surjective. The output set must have reduced entropy than the input set. meaning that the internal state function must correlate some of the output information and that the internal state function complexity is bounded by the output set.

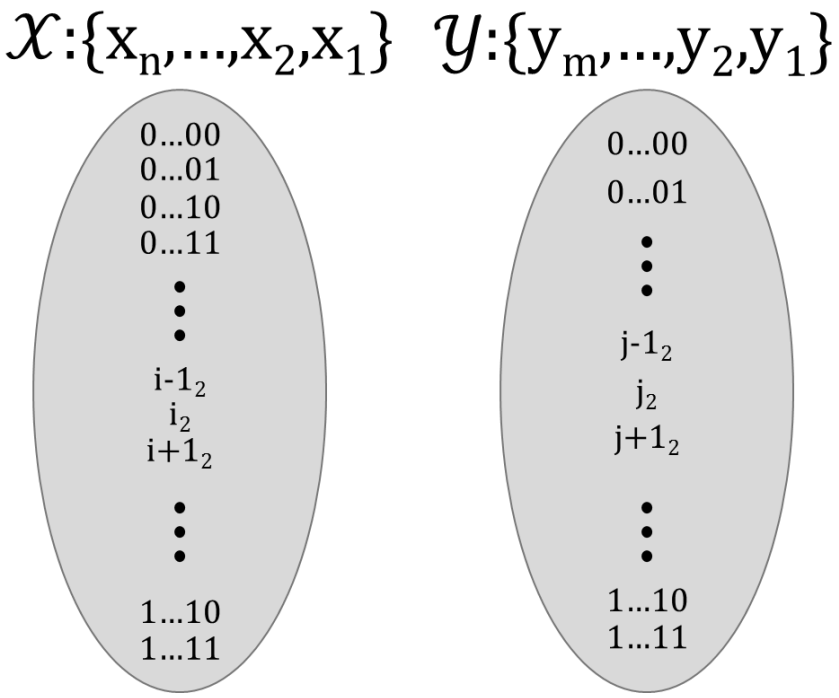


Figure 2-2: A generalized depiction of a function between **n** input bits and **m** output bits, lacking any mappings.

For a depiction of another input set and output set, look to Figure 2-2. Presented as the generalized case, Figure 2-2 depicts a function with **n** Boolean inputs, which span $2^{\mathbf{n}}$ elements, **m** Boolean outputs, which span $2^{\mathbf{m}}$ elements, but lacks any mapping from input to output. As a parsable example, Figure 2-3 presents a specific case. Containing exactly 3 Boolean inputs and 2 Boolean outputs, there are 8 mappings from the input set to the output set. The function depicted is onto the output set, but is not one-to-one. Indeed, the function depicted should be familiar to any digital designer, as it is the functional depiction of a Full-Adder. It is, perhaps, strange to depict the Full-Adder in this manner as a function which passes a 3-bit number to a 2-bit number, e.g. the decimal numbers 1, 2, and 4 all map to an output of 1. But the order of the bits presented is arbitrary, this ordering may align well with a classical depiction of the Full-Adder, but it is arbitrary. Altering the order of the bits would change the function's depiction,

but the Full-Adder is still fundamentally a function. The functional depiction of Figure 2-3 is as good a depiction as any other.

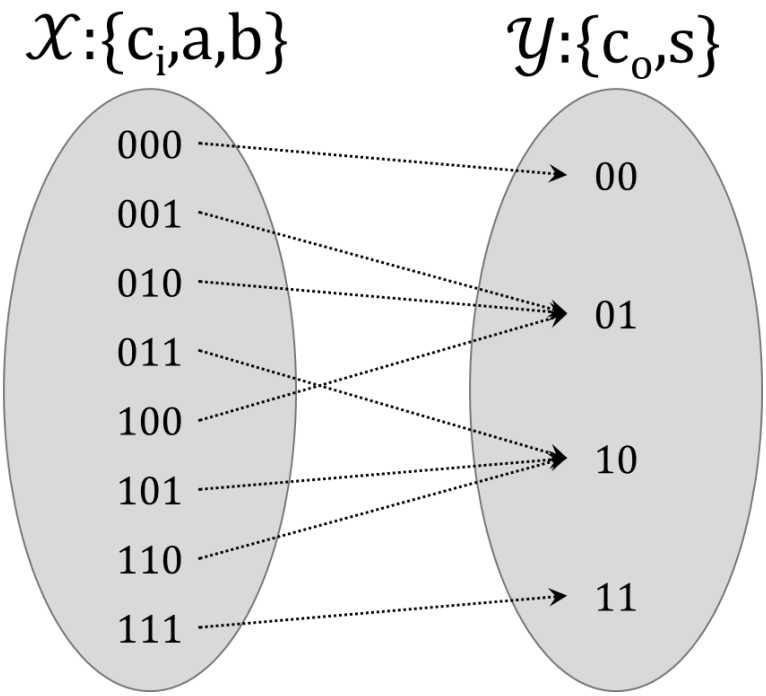


Figure 2-3: A functional depiction of the Full-Adder.

For a logic system which follows our stated assumptions, has **n** inputs, and has **m** outputs, what is the total possible number of unique functions? Two functions which are not unique will, for all elements of the input set, map to identical elements of the output set. Effectively, two nonunique functions will share 2^n input-output mapping pairs. Two functions which have multiple identical input-output pairs, but which differ by a single input-output pair, are unique functions. From the stated assumptions the system has maximum entropy for the input space, if the assumption is made that the output space has maximum entropy, then every output signal is independent and uniformly distributed. This is an unreasonable assumption in general given that the output signals could be correlated by the internal state function, and are guaranteed to be correlated for a system with fewer output signals than inputs signals, but this assumption is included, again, for the purposes of setting maximum bounds.

By assumption every output signal is independent, so we can look at a single output signal for analysis, independent from all the others. For any element of the input space, which contains 2^n elements, the input element can map that single output signal to either a 1 or to a 0. The space that encompasses all mappings from the input space of cardinality 2^n to the output space of cardinality 2, is a function space of cardinality 2^{2^n} or, written alternatively 2^{2^n}. When there exists **m** outputs signals, as opposed to only 1, the output space is of cardinality 2^m. Any entry in the input space can now map to any entry in the output space. For every entry in the input set, there is exactly 2^m possible outputs which can now be taken, as opposed to the case of 2 for one signal. Since all of these output signals are solved independently, per the assumption, the function space in total expands to $2^{m^{2^n}}$.

For the set of all digital systems which have **n** inputs and **m** outputs there exists an exponentially growing number of functions which map the input set to the output set. The exact number of unique functions grows quite rapidly with input and output count. Indeed, even for small numbers of Boolean inputs and outputs the number of unique functions can be quite large. For a single output, the number of unique possible functions begins with 2 unique functions for no inputs, 4 unique functions for 1 input, 16 unique functions for 2 inputs, 256 unique functions for 3 inputs, 65,536 for 4, 4.3 million for 5 inputs, and 18.4 quintillion for 6. Extending this idea, there can be an infinite number of implementations of any one individual logic function. This space is often incomprehensibly large for even small numbers of inputs and outputs. The typical designer approach to this incomprehensibility is to dissect the design into manageable chunks and solve each individually.

In digital systems there is a locus of useful operations upon which most of modern hardware design is based on. Memoryless logical operations of particular utility are referred to as logic gates, where every logic gate is given a classical truth table and a comparable word in English to describe its operation. For a single input to a single output: there exists the NOT gate

and the PASS gate. For two inputs to a single output: there exists the AND gate, the OR gate, the XOR gate, and their opposites, the NAND gate, the NOR gate and the XNOR gate. For three inputs and one output there is simply the multiplexor (MUX) gate. While not explicitly counted as gates, for zero inputs and one output there is tying a line to HIGH or to LOW through the power or ground rails. Lastly, logic gates with an internal memory include all kinds of latches and all kinds of different registers.

Continuing the discussion on functions and the accepted number of unique functions for a given set of inputs and outputs, it is simple to give the numbers for unique function count at small numbers of external signals. Starting with the first useful arrangement, zero inputs and one output, there are two possible unique functions: Tying the output to HIGH or to LOW. This trivial case is worth mentioning because tying an output signal is an extremely ubiquitous operation, indeed for every input count an operation must exist to tie a single output to HIGH or to LOW, otherwise the system lacks complete logic. Tying a signal has differing names in the literature, typically as HIGH-LOW or 1-0.

With the addition of a single input the number of meaningfully unique functions increases from two to four. All four possible functions are presented in functional depiction in Figure 2-4. As before, the single output can always be tied to HIGH or to LOW as unique functions. In addition, the single input means the output can be passed the input as-is, in the form of a PASS Gate, or can be inverted, in the form of a NOT gate. Again, tying the output signal is a kernel of the functional possibilities.

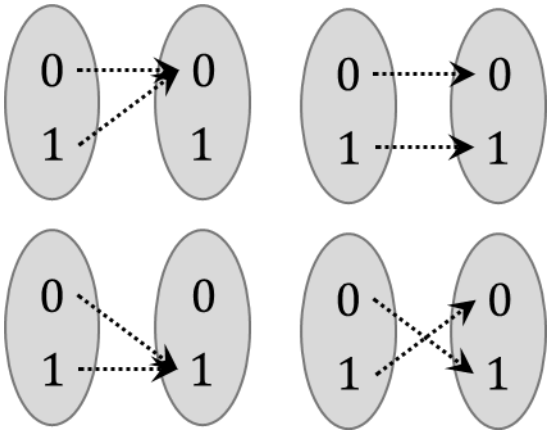


Figure 2-4: All possible unique functions for a digital system with one input and one output.

Adding another input signal for a total of two inputs and one output increases the number of meaningfully unique functions to 16. All possible unique configurations are depicted in Figure 2-5. The locus of useful digital functions containing 2 inputs and 1 output include the AND gate, OR gate, XOR gate, NAND gate, NOR gate, and XNOR gate. Not included in this group, is tying the output signal to HIGH or to LOW, though given the ubiquity of the operations, should be included. In total, that includes 8 named logic gates of the 16 meaningfully unique functional possibilities for a 2 by 1 logic gate.

The reason that only 8 of the 16 functions are given descriptors is a trivial one: the remaining 8 functions can be comprised of combinations of the named 8 functions. The utility of this simple fact should not be lost on the reader. The 8 functions of AND, OR, XOR, NAND, NOR, XNOR, HIGH, and LOW are sufficient to represent any possible function with 2 inputs and 1 output. If the outputs are solved for independently, then these 8 functions are also sufficient for solving any digital system which has 2 inputs, and any amount of outputs. As a stronger argument, these 8 functions are capable of solving any function with any number of inputs and outputs. They represent a complete set of logical functions, they are complete logic.

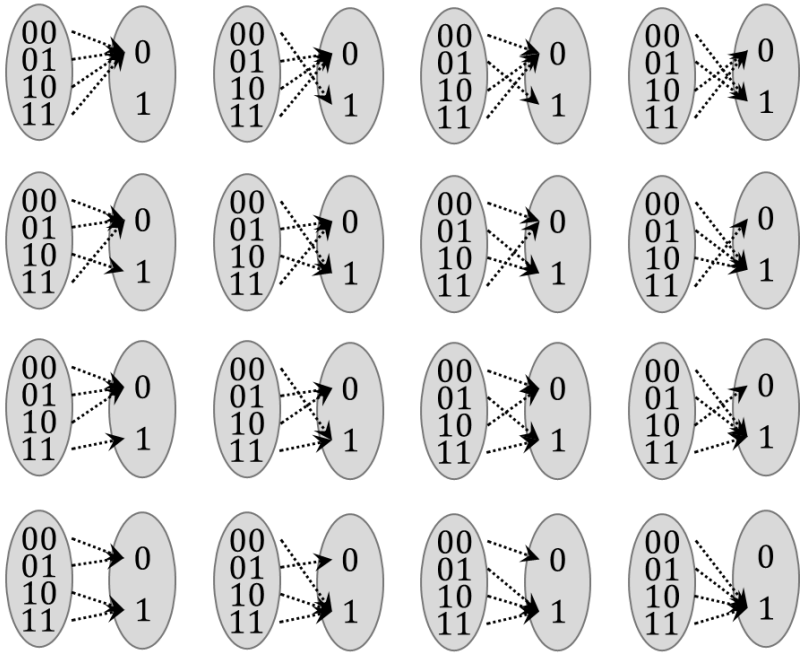


Figure 2-5: Increasing the number of input signals from one to two increases the total unique functions from 4 to 16. All 16 unique functions are depicted here.

All that is necessary for complete logic is that the set of logical operations can span the entire space of possible unique functions, which, again, for **n** inputs and **m** outputs, is a set of cardinality $2^m \wedge 2^n$. In the example of 2 inputs and 1 output, all 16 possible logical functions are enumerated in Table 2-1. And, with an inductive step, these 8 functions of interest can solve a system with an arbitrary number of inputs. But, these 8 functions are actually a super-complete set of logical operations, meaning that they contain additional operations unnecessary for completeness. There are multiple sets of these 8 operations which on their own will satisfy completeness.

Complete Logic Gate Sets:

NAND gates

NOR gates

AND, NOT gates

OR, NOT gates

Table 2-1: Starting with an index of 0, every functional depiction (from left-to-right, top-to-bottom) in Figure 2-5 is assigned a number. A logically equivalent formulation is given across three complete sets of operations, first is ANDs, ORs, NOTs, then solely NANDs, and finally solely NORs

Index	Gate Name	ANDs, ORs, NOTs	NANDs	NORs
0	LOW	¬AA	¬(¬(A ¬(AA)) ¬(A ¬(AA)))	¬(A+ ¬(A+A))
1	NOR	¬(A+B)	¬(¬(¬(AA) ¬(BB)) ¬(¬(AA) ¬(BB)))	¬(A+B)
2		¬AB	¬(¬(AA) B)	¬(A + ¬(B+B))
3		¬A	¬(AA)	¬(A+A)
4		A¬B	¬(A ¬(BB))	¬(¬(A+A)+B)
5		¬B	¬(BB)	¬(B+B)
6	XOR	¬AB + A¬B	¬(¬(¬(AA) B) ¬(A ¬(BB)))	¬(¬(A + ¬(B+B)) + ¬(¬(A+A) + B)))
7	NAND	¬(AB)	¬(AB)	¬(¬(¬(A+A) + ¬(B+B)) + ¬(¬(A+A) + ¬(B+B)))
8	AND	AB	¬(¬(AB) ¬(AB))	¬(¬(A+A) + ¬(B+B))
9	XNOR	¬A¬B+AB	¬(¬(¬(AA) ¬(BB)) ¬(AB))	¬(¬(A+B) + ¬(¬(A+A) + ¬(B+B)))
10		B	¬(¬(BB) ¬(BB))	¬(¬(B+B) + ¬(B+B))
11		¬A + B	¬(A ¬(BB))	¬(¬(¬(A+A) + B) + ¬(¬(A+A) + B))
12		A	¬(¬(AA) ¬(AA))	¬(¬(A+A) + ¬(A+A))
13		A + ¬B	¬(¬(AA) B)	¬(¬(A + ¬(B+B)) + ¬(A + ¬(B+B)))
14	OR	A+B	¬(¬(AA) ¬(BB))	¬((¬(A+B) + ¬(A+B))
15	HIGH	¬A+A	¬(A ¬(AA))	¬(¬(A+ ¬(A+A)) + ¬(A+ ¬(A+A))))

Along with an indexing of the functional operation, and the logic gate (if applicable), every function in Table 2-1 is given a formulation according to the set of AND gates, OR gates, and NOT gates. These three gates are a super-complete logic set, having solely AND gates and NOT gates, or OR gates and NOT gates, are sufficient. It is simple to observe that the super-complete set of logic operations can much more compactly represent a function than solely through the complete set of NAND gates or NOR gates. The super complete set of 8 operations can represent any function that much more compactly.

A logical system containing only ANDs, ORs, and NOTs alone is sufficient in representing all possible unique functions. But, this discussion also includes NAND and NOR

gates due to their simplified expression in CMOS transistor configurations. NANDs, NORs, and NOTs can all be constructed simply with PMOS and NMOS transistors in configuration. Indeed, in general CMOS the construction of OR and AND gates is accomplished with a NOT gate following their opposite logic. Other commonly represented hardware gates are the XOR gate and the MUX gate through the use of pass gates. In general, nearly all modern CMOS design consists of the NAND, NOR, NOT, XOR, and MUX alphabet. This is entirely owing to their simplicity in implementing these gates in modern CMOS. There is an entire area of logic and logic completeness which is rarely discussed in digital design discussions.

Another complete set of logic consists of the NOT operation and the Imply operation. These two operations are logically complete. For the same set of 16 functions, Table 2-2 provides the implication logic necessary for satisfying completeness condition. Logically, A implies B is equivalent to B OR NOT A. Completeness condition is satisfied by any set of logical operations that, one interacts between two inputs, and two can negate an input. The set of AND and NOT gates satisfies these conditions, as does the set of NAND gates solely, as a NAND gate can also operate as a NOT gate.

Table 2-2: Every function is given an index (following the logic of Table 2-1) and the function is now formulated by complete implication logic

Index	Gate Name	ANDs, ORs, NOTs	Implies, NOTs
0	LOW	¬AA	¬(A ⇒ A)
1	NOR	¬(A+B)	¬(¬A ⇒ B)
2		¬AB	¬(B ⇒ A)
3		¬A	¬A
4		A¬B	¬(A ⇒ B)
5		¬B	¬B
6	XOR	¬AB + A¬B	(A ⇒ B) ⇒ ¬(B ⇒ A)
7	NAND	¬(AB)	A ⇒ ¬B
8	AND	AB	¬(A ⇒ ¬B)
9	XNOR	¬A¬B+AB	(A ⇒ ¬B) ⇒ ¬(¬A ⇒ B)
10		B	B
11		¬A + B	A ⇒ B
12		A	A
13		A + ¬B	B ⇒ A
14	OR	A+B	¬A ⇒ B
15	HIGH	¬A+A	A ⇒ A

For the simplifying assumptions made, we have a defined set of logic operations with each logic operation having a well-defined (for all time) operation. For even this small set of logic operations, equivalences between gates can be made. Combined logic operations can be made to operate identically to a different logic operation or a different combination of logic operations. Using our hardware approved collection of NANDs, NORs, and Inverters we can build equivalences for other gates.

Based on our assumption that all digital systems are a collection of these small logical gates, and our knowledge that this collection of small logic gates themselves have well defined equivalences, any digital system can have a set of all the possible combinations of small logic gates. This set has an extremely large cardinality. The cardinality increases when the repetition of negate operations are taken, such as cascaded NOT gates, as a series of $2n$ inverters is logically equivalent to a PASS gate and has no logical consequence.

CHAPTER 3

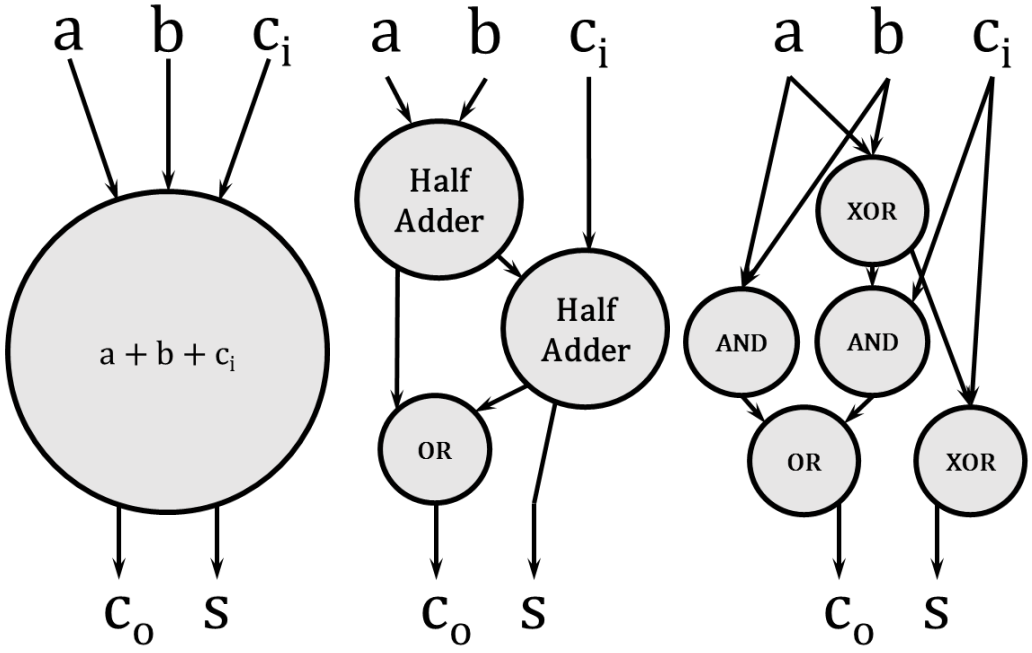
Data Flow Graphs

Continuing from the discussion of the most frequently used digital logic gates and their natural sufficiency for complete logic. The discussion stopped with the idea that a small set of logical operations could span the entire function space. Leading to a purely theoretical result. The actual building of digital systems is always made up of the composite of multiple smaller digital systems, which in the discussion of complete logic rose no higher than the logical gate level. The purpose of this chapter is to include the analysis of collections of these smaller digital systems. If any digital system following our initial set of assumptions can be produced by a small set of complete logic, then a graph, specifically a data flow graph of these operations can also represent any digital system.

Any data flow graph of interest contains a set of arrows and a set of logical nodes. The arrows are unidirectional edges which pass data from one logical node to another. The logical nodes take data in the form of Boolean numbers inputted from arrows and present data through their output arrows. Within a single grand data flow graph there can exist specific input and output arrows that represent the getting and setting of external data signals.

It is easy to see that such a system is equivalent to the Discrete Spatial Logic System of Chapter 2. The graph of logic nodes and arrows represent an alternative mathematical construction for the internal state function. Chapter 2 includes the functional depiction of a Full-Adder as a parsable example, continuing this analysis, a Full-Adder is depicted in triplicate in Figure 3-1. The simplest depiction of the Full-Adder is that of a gray-box: A function with 3 inputs and 2 outputs solving the entire Full-Adder function. Indeed, this single logical node is

solving the function as depicted in Figure 2-4, only presented here as a data flow graph. Increasing the visibility of the internals of the Full-Adder can be made possible by simply moving an abstraction level down. In this fashion, the Full-Adder is then constituted by multiple logic nodes and additional arrows. Now comprising of two Half-Adders and a single OR gate, this graphic depiction is functionally identical to that of a single great Full-Adder logical node. In the final depiction the Full-Adder is reduced to its base logical gates. The final depiction of the Full-Adder consisting of 2 AND gates, 2 XOR gates, and 1 OR gate is functionally equivalent to all implementations of the Full-Adder. Within this Figure exists 3 separate implementations for a single unique function. Indeed, any gates generally, or the XOR gates more specifically, can be further reduced to NANDs, NORS, or any similar set of complete logic operations (gates).



a) b) c)

Figure 3-1: The Full-Adder logic operation is presented at three levels of hierarchy, a) treating the Full-adder as a gray box of inputs and outputs, b) increasing visibility by including Half-Adder gates, and finally c) showing the Full-Adder at its canonical form of XORs, ANDs, and an OR gate.

Recalling the analyses on the quantity of unique functions given a specified number of inputs and outputs, there exists a total 65,536 (2^2 ^ 2^3) unique functions for a system with 3 inputs and 2 outputs. Depicted in Figure 3-1 are three possible implementations of a single entry in that functional space. These three separate implementations occur across three levels of hierarchy within the Full-Adder function, but there exists additional means of differentiating functional implementation even within the same level of hierarchy.

In the Full-Adder example a function is given (i.e. the Full-Adder function) and a data flow graph is constructed to describe wholly describe it. This process can also work in reverse, i.e. given a data flow graph a function, or set of functions, can be constructed to wholly describe it. For an arbitrary data flow graph cutsets can be created which partition the data flow graph into specific functions of interest. This "functional decomposition" is depicted for an arbitrary set of undefined nodes in Figure 3-2. Four cutsets are selected out of the graph and each is assigned a function descriptor. An arbitrary data flow graph can be partitioned into multiple functions and multiple permutations of cutsets.

It is convenient to think of any logical node within a data flow graph as a submodule within the greater module of the larger data flow graph. In effect, the logical element is a submodule of the entire digital logic system. The greater module will at a given time set the input arrows for the specified logical element, while also reading from the logical element's output signals. When thinking hierarchically we may separate the implementation of the logical element's internal state function from that of the internal state function of the greater data flow graph, treating it as if it were a gray or black box. This approach to digital design enables thinking of any given logic system as if it were a number of smaller manageable chunks, each to be solved for separately.

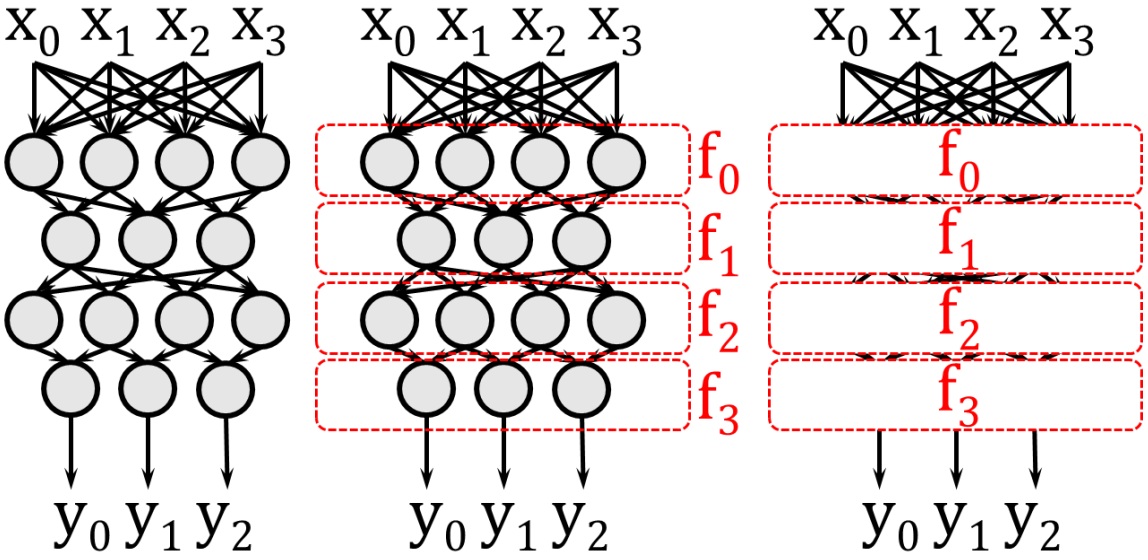


Figure 3-2: An arbitrary Data Flow Graph is presented and then decomposed into four functions, a) Is the four layer network data flow graph containing 4 inputs and 3 outputs, b) Is an arbitrary partitioning of the data flow graph into four cutsets, and c) depicts the data flow graph as a feed-forward serial graph with four functional nodes.

In the trivial case a data flow graph (module) has a single logical element (submodule) to which all inputs are driven and from which all outputs are driven. In a purely feed-forward system, one which lacks any kind of feedback, these logical elements can exist in parallel to one another, in series to one another, or both in parallel and in series.

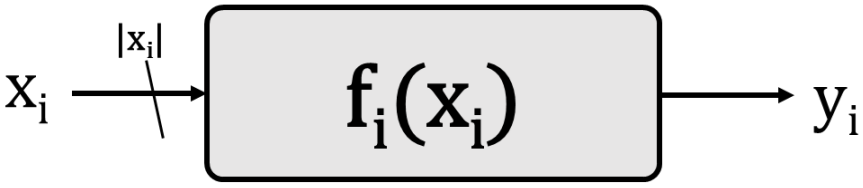


Figure 3-3: A parallel submodule which takes as input a subset of the input set, and drives exactly one output bit.

In solving for the total number of unique functions, the assumption is made that any one output signal can be assumed independent from any other output signal. As mentioned, this assumption does not hold in general as any two or more output signals can be made correlated by the internal state function. It is, however, fair to assume that all output signals can be solved independently from any other output signal. Given that each output signal is wholly defined by the current input signals and the internal state function (as per our initial assumptions), any one output of the system can be solved independently and in parallel with all other outputs. For a module which is split into parallel submodules, any one submodule can drive a unique output. Each output submodule will take as its input a subset of the module's input set. This set of inputs for one output submodule is not necessarily unique, and can overlap with the set used by any of the other output submodules. One of these parallel submodules is depicted in Figure 3-3. The figure depicts a single output which is driven by a unique internal state function, and a subset of the input set.

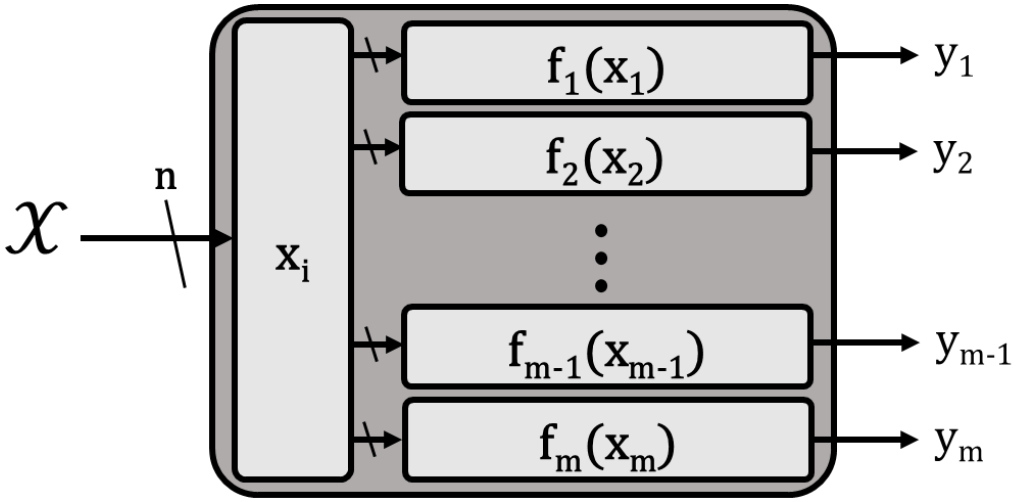


Figure 3-4: For the **m** outputs of a Discrete Spatial Logic System, each output can be solved independently through a distinct output function. These distinct functions can be implemented through parallel submodules, which each receive a distinct subset of the input set. This is equivalent to the general case.

Any given output submodule needs a subset of the input set, so a block is added which selects the input set for each distinct output submodule. The collection of the output submodules and the input set selection block is depicted in Figure 3-4. Within this figure every submodule has $|x_i|$ inputs and exactly one output unique to the submodule. From earlier analysis, this submodule has $2^{|x_i|}$ unique input states, has $\min(2^{|x_i|}, 2)$ meaningfully unique outputs, and $2^{2^{|x_i|}}$ possible unique internal state functions. We can observe these same statistics across the collection of **m** output submodules. Looking across all submodules the entire module has **n** binary inputs. The number of unique input states is unchanged from the case of a single module, this result should be unsurprising. Looking at the number of meaningfully unique outputs the number has changed to:

$$\prod_{i=1}^{m} 2^{min(x_i,1)} = 2^{\sum_{i=1}^{m} min(x_i,1)}$$

The maximum of this result is 2^m, which in general is increased from the $\min(2^n, 2^m)$ found for the case of a single module. But this result is not considering the correlation of the outputs caused by to the internal state function. For the case where the number of outputs is less than the number of inputs, the outputs are not guaranteed to be correlated by the internal state function. Under such conditions, the maximum number of outputs would be constrained by:

$$min(2^n, 2^m) = 2^m$$

Which agrees with the logic of the parallel submodules. In the case when the number of inputs is less than the number of outputs, such a case which guarantees output correlation, then the

number of meaningfully unique outputs is bounded by the established condition that the number of meaningfully unique outputs is bounded by:

$$min(2^n, 2^m) = 2^n$$

Two observations rise out of this, the first is that two implementations of an internal state function which are meaningfully equivalent do not in general share an identical complexity. The cost of solving for each output separately is generally higher than the cost of solving for the outputs as collections; in the case where the outputs are correlated (greater number of outputs than inputs), this generality is guaranteed. The second observation is that the cost of implementation has in general no relation to the complexity of the internal state function. For an optimally designed implementation of the internal state function the complexity scales with complexity of the internal state function, this is a trivial result. But the complexity of an implementation can scale ad infinitum and produce a meaningfully equivalent internal state function.

The most direct means of increasing the complexity of the implementation is through the addition of serialism in the design. For any given stage in this serial pipeline, the submodule will only be connected to the submodules directly before and after it by $|x_i|$ signals, excepting the submodule at the input and the submodule at the output. For the trivial case that the internal state function contains a single submodule acting as a single pipe stage the results are as expected. For a single submodule pipe stage, the characteristics are identical to those of a single module. The input set to the single pipe stage is of cardinality 2^n for a system with **n** inputs, the output set of the single pipe stage can represent a set of cardinality $\min(2^n, 2^m)$ meaningfully uniquely for **m** outputs of the whole system, and there exists $2^m \hat{} 2^n$ meaningfully unique

implementations of the internal state machine for the single pipe stage. Increasing the number of pipe stages, or submodules, from one to two has certain effects of note.

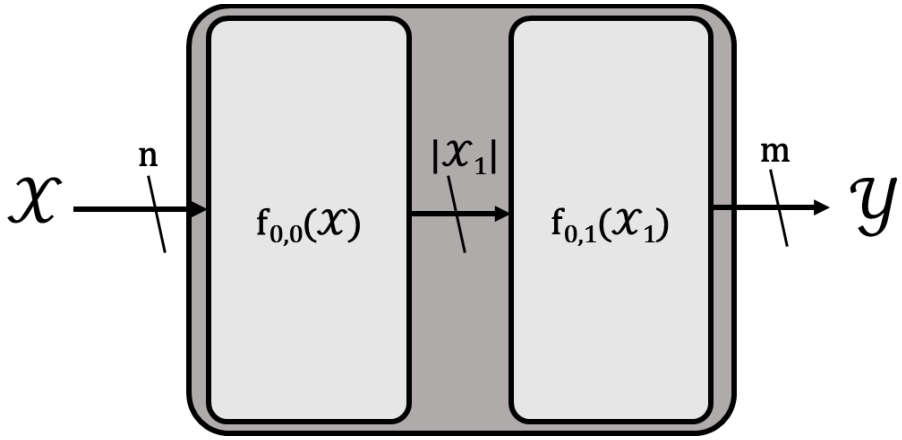


Figure 3-5: Splitting an internal state function into two separate serial pipe stages with a single unidirectional bus between them.

An example of a serial pipe stage with two submodules is depicted in Figure 3-5. Given that the submodules only transfer information through $|x_1|$ signals between them, it is possible to bifurcate the two submodules and treat these $|x_1|$ signals as output from one submodule and input to the other. For the first submodule there exist **n** inputs and $|x_1|$ outputs, for the second submodule there exists $|x_1|$ inputs and **m** outputs. Looking exclusively at the first submodule, the number of meaningfully unique internal state functions is determined by the minimum of the cardinality of the input set versus the cardinality of the output set. In the case of the first submodule this output set cardinality is determined by the $|x_1|$ intermediate signals, so $2^{|x_1|}$. For the case of the second submodule the input set cardinality is then set by these same $|x_1|$ signals, so also $2^{|x_1|}$. The output set cardinality would follow from the output set of the whole system, so 2^m.

This understanding is incomplete because the input set of the whole system is not $2^{|x_1|}$ in general. The number of meaningfully unique outputs of any submodule is the minimum of the

cardinality of the meaningfully unique input set and the meaningfully unique output set. For the second submodule the number of meaningfully unique outputs is obviously bounded by the **m** outputs, it is also obviously bounded by the $|x_1|$ intermediate signals. But the number of meaningfully unique inputs represented by the $|x_1|$ signals is also bounded by the number of meaningfully unique inputs to the first submodule, 2^n. In short, the number of meaningfully unique outputs is the minimum of the output set and the intermediate set, which is itself the minimum of the intermediate set and the input set. In total, the cardinality of meaningfully unique outputs is:

$$min(2^n, 2^m, 2^{|x_1|})$$

This is a well-observed information theoretic result when looking purely at the possible total entropy of any given function (submodule) in a serial pipe. Increasing the number of pipe stages from two to three has the expected consequences. The number of meaningfully unique outputs from the first stage is $\min(2^n, 2^{|x1|})$, the number of meaningfully unique outputs of the second stage is $\min(2^{|x2|}$, cardinality of meaningfully unique outputs of the first stage). The number of meaningfully unique outputs of the third stage, and whole system, is $\min(2^m$, meaningfully unique outputs of second stage). In total, the number of meaningfully unique outputs is:

$$min(2^n, 2^m, 2^{|x_1|}, 2^{|x_2|})$$

By induction, the number of meaningfully unique outputs for a system of k pipe stages is

$$min(2^n, 2^m, 2^{|x_1|}, \ldots, 2^{|x_{k-1}|})$$

The depiction of k serial submodule pipe stages is depicted in Figure 3-6. In the event that the minimum is not set by the number of output signals, then the minimum must either be at the input or somewhere along the pipeline. Given that any local pipe stage is the minimum of its respective inputs and outputs, any pipe stage following the global minimum will be held to the same minimum constraint. Ultimately, at any point past the global minimum number of signals the proceeding meaningfully unique output sets are bounded, any additional logic above this is considered waste. This can be used to reduce the complexity of the logic in the subsequent stages.

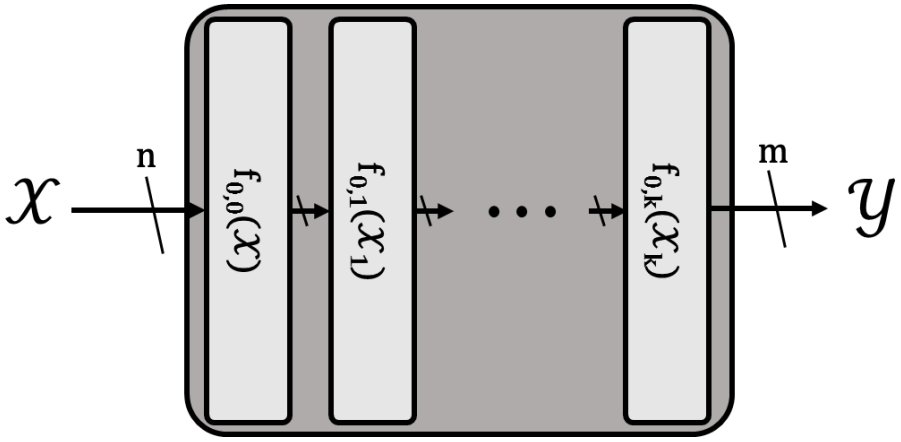


Figure 3-6: A pipe stage of **k** serial submodules.

The location of the minimum signals in the pipeline can be thought of as partitioning the pipeline stages as before and after the minimum. As stated, any pipeline stages following have a guaranteed maximum complexity. Any pipe stages that come before the minimum are subject to more local constraints. For the partition of stages that exist before the global minimum, the

minimum of these signals can also be determined by taking the minimum of **n**, $|x_1|$, ..., $|x_{i-1}|$ for an **i**th global minimum. The set of pipeline stages can be recursively partitioned into a set before the partition minimum and after the partition minimum. The complexity of the partition following a local (or global) minimum is always constrained by the partition local maximum. In this way, the complexity of the entire pipeline stage can be simplified by this maximum bound complexity for any given pipe stage subject to its partition set.

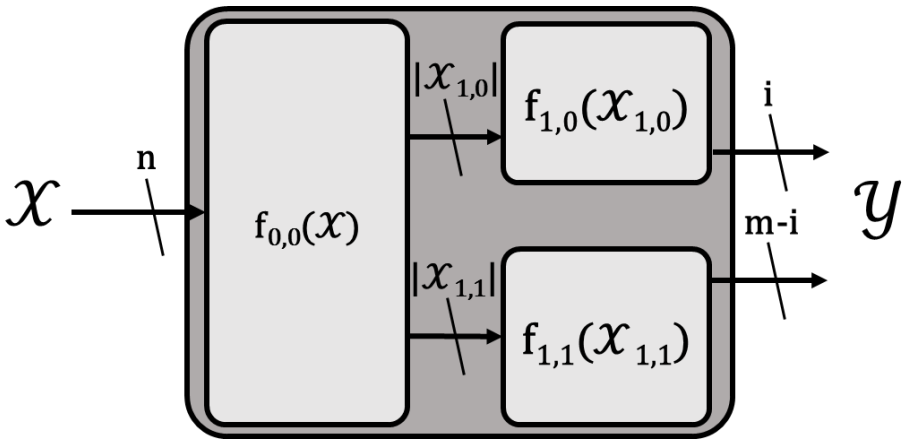


Figure 3-7: The mutated submodule graph which consists of serialism and parallelism.

Returning to the case of two pipe stages a mutation is introduced, the serial submodules have the parallelism added from earlier examples. The first stage of the pipeline remains unchanged with **n** inputs, a well-defined internal state function, and $|x_1|$ outputs. The second stage is split into two parallel submodules with each taking as input a subset of the $|x_1|$ intermediate signals, and each drives a non-overlapping set of the **m** outputs, this can be viewed in Figure 3-7. Just as in the parallelism case above, the intermediate signals can be shared by parallel submodules, but each output is driven exclusively by one submodule. For any given output signal the total number of meaningfully unique possibilities is 2 since it is a binary signal, but for a collection of **i** outputs coming from any given submodule in the second stage, the total number of meaningfully unique output bits is the minimum of **i**, **n**, and the $|x_{1,-}|$ inputs to that

submodule. And the cardinality of the meaningfully unique output set for the collection of both output sets can be calculated as:

$$min(2^n, min(2^n, 2^{|x_{1,0}|}, 2^i) * min(2^n, 2^{|x_{1,1}|}, 2^{m-i}))) =$$

$$min(2^n, min(2^{n+n}, 2^{n+|x_{1,1}|}, 2^{n+m-i}, 2^{n+|x_{1,0}|},$$

$$2^{|x_1|}, 2^{m-i+|x_{1,0}|}, 2^{n+i}, 2^{i+|x_{1,1}|}, 2^m) =$$

$$min(2^n, 2^m, 2^{|x_1|}, 2^{m-i+|x_{1,0}|}, 2^{i+|x_{1,1}|})$$

This analysis can be extrapolated to systems of greater complexity, but such analysis is not pursued in the body of this text. Moving on, divert this bundle of **i** output signals from the output of one submodule and now redirect them as input to the parallel submodule. The inputs are still driven to the first pipe stage, as before, the collection of inputs from the first stage to the second has remained unchanged. What is changed is that one of the parallel submodules on pipe stage two has added inputs driven from the other submodule.

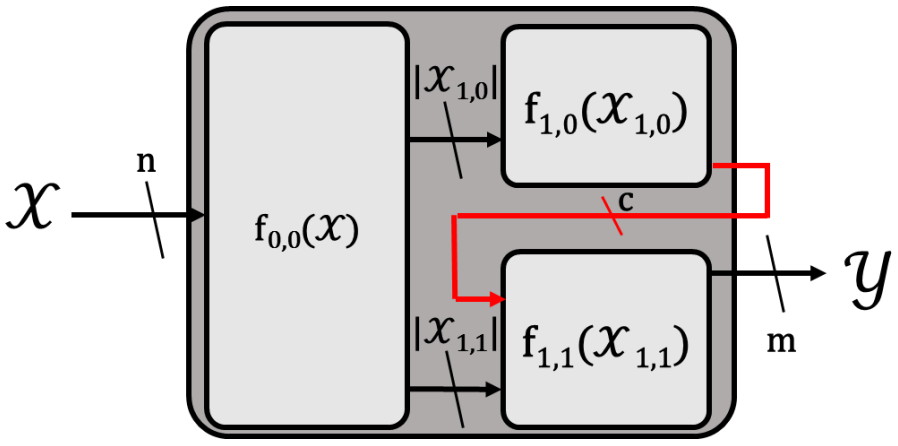


Figure 3-8: One submodule's outputs have been directed as inputs to it's sister submodule.

We now move the input stage of the pipeline and leave only the two submodules. They both take an arbitrary, and possibly overlapping, subset of the input signals. But, now, one submodule has **c** inputs which it takes as input from its sibling. For simplicity we assume that this only occurs in one direction, submodule zero provides input to submodule one, and submodule one does not also provide input to submodule zero. If the inputs were provided in both directions this would act as continuous looping feedback, which is not covered in detail here.

The analysis of the submodule providing input is unchanged from previous. The analysis of the other submodule has added complexity. Submodule one is taking as input up to **n** inputs from the external source, drives all **m** output signals, and now has the addition of **c** inputs provided by submodule zero. The cardinality of the input set is constructed by the union of the external inputs and provided inputs, which in total represents $2^{(|x_{1,1}| + c)}$. But the cardinality of the input set is meaningfully bounded by the meaningful uniqueness of those provided inputs. This uniqueness is determined simply by the minimum of the number of inputs submodule one is passed from the input selector alongside the number of inputs provided by submodule zero:

$$min(2^n, 2^{|x_{1,1}|} * min(2^c, 2^{|x_{1,0}|})) = min(2^n, 2^{|x_{1,1}|+c})$$

The number of meaningfully unique output sets is also determined by this adjoined input set. The trivial math leads to a total of:

$$min(2^n, 2^m, 2^{|x_1|}, 2^{|x_{1,1}|+c})$$

Meaningfully unique output possibility for the **m** output signals that submodule one drives.

Note that this cardinality is equivalent to two parallel submodules in the case when the number of **c** signals are greater than the number of $|x_{1,1}|$ signals. In the event that the provided inputs are greater than the data inputs to the one submodule, the space of the provided inputs can be thought of as a sparse space that the $|x_{1,1}|$ inputs map to. But the cardinality of the output space is not increased by this parallelism and feedback from one submodule to the other. At most, the outputs space is still bounded by the familiar $\min(2^n, 2^m)$. Parallelism and serialization are fundamentally tools to aid in the design of equivalent internal state functions. That these tools exist introduces additional implementations of an equivalent internal state function, effectively increasing the space of implementations for any given function in the massive function space. That mapping has not necessarily changed in the transition. There is an infinitely large set of implementations that can map from one input space to one output space, even small input and output spaces. Ultimately, this form of feedback in analysis does not fundamentally add to the output space that the logic system can map to, it is still bound by input and output size constraints.

This understanding is important when we give arbitrary names to our parallel submodules. For the submodule which provides feedback to the other, the name control unit is given. The submodule being provided inputs is given the name logic unit. The description given is equivalent to the Input/Output, control unit, and logic unit of the Von Neumann machine. And as established, the existence of the control unit does not fundamentally alter what the entire system is capable of computing.

CHAPTER 4

Reconfigurable Discrete Spatial Logic Systems

The duration of Chapters 2 and 3 are focused on creating the math for logical systems constrained by the initial six assumptions. These six assumptions were included initially to simplify the analysis, and aided in the construction of several firm definitions for an arbitrary digital system. For the body of text Chapter 3, assumption 4, “Logic System contains no control signals” is to be removed and consequences are to be discussed.

Assumptions

1. All systems will be purely digital
2. The internal state function is well defined for all time
3. Maximum entropy for the input set
4. Logic system contains no control signals
5. All logic paths have zero delay
6. Memoryless state function

The input to our logical system of interest is still defined by the count of its input signals, and it is important to note, as before, that the input signals can be collected to form a number in digital logic greater than 0 or 1. An interesting example of this input signal collection is the n-bit adder. An 8-bit adder has 8 bits for the A input, 8 bits for the B input, and a possible additional bit for the carry-in. An 8-bit adder has 16 or 17 binary inputs, and can represent an input decimal number up to 65,535 or 131,071. For any digital system with **n** inputs, it is trivial to

partition the input bits into two sets. In the case of the 8-bit adder (without carry-in) we as designers would partition the input signals into the set of input signals which comprise input A and those that comprise input B. In this way we partition the input set into two decimal numbers, A with range 0 to 255, and B with range 0 to 255. This trivial partitioning of the input set does not fundamentally change the internals of the logic system. The input set, internal function, and output set are unchanged. As an example, an A of 15 adding to a B of 15, produces an output of 30. Likewise, the collection of all input signals would represent A of 15 and B of 15 as the decimal number 3,855, and would still map to an output of 30. This bipartition of the input set is merely a matter of designer choice. Trivially, there exists 2^n possible unique bipartitions of the input signal set. In the case of the 8-bit adder, the great many of these input bipartitions would not be of much greater designer aid than nonsense.

For any logical design of interest, we can trivially partition its input signal set without fundamental alteration to logical operation. Specifically, we partition the **n** input bits to a set containing **c** bits, and a set containing **l** bits. This process, as a matter of course, leaves the internal state function unchanged and leaves our **m** output bits unchanged. The notation slightly changes:

$$F(X^n) = F(\{X^l, X^c\}) = F(X^l, X^c)$$

The logic system depiction would also need to change to accommodate the alteration of the input signals, see Figure 4-1.

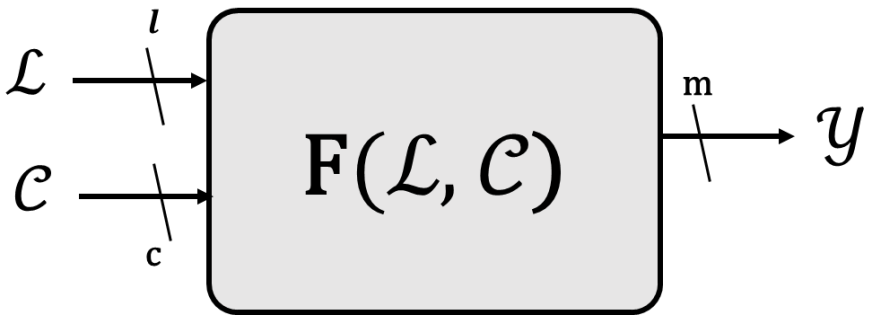


Figure 4-1: Bifurcating the input signals does not alter functional operation, this is an equivalent depiction of a logic system to that of Figure 2-1.

But our system remains fundamentally unchanged. This system is still equivalent to our initial system. For the purposes of design or documentation, we can also arbitrarily rename our system inputs according to the partition defined above. We maintain that the **l** input bits are still referred to as "inputs" but the **c** inputs we are now referring to as our "Controls". After the silicon returns with dedicated transistors, or the FPGA has already been programmed, naming any of the input signals as controls signals is simply a matter of perspective, it is not a matter of engineering. It is convenient to refer to the **l** input signals as input logic signals.

There is a common refrain in modern digital hardware that the hardware in question contains an element of flexibility. Flexibility, referring not to physical deformation, but rather a flexibility with respect to the math that the hardware is capable of solving. This notion of hardware flexibility is a fiction. The transistors as laid out do not change by design, the internal state function will remain unchanged. The exception to this maxim is with unintended destruction of operation. Even with intended physical changes, such as those with fuses and anti-fuses, the math that the state function solves is fundamentally set at design time, is unchanged. The only perceived, if not genuinely functional, change is the alteration in the state of our arbitrarily declared control inputs. The space that these control inputs can occupy is not fundamentally altered, with a cardinality of 2^c. Deviations from this cardinality, or, alternatively, alterations in the state space of the control inputs is not functional flexibility. Changing the

control space occurs either from changing our old, arbitrary, selection of control inputs to a different, arbitrary, selection of control inputs, which is not a functional change. Or, changing the value of the control inputs within the control space, which is nothing more than a change in the inputs to the logic system, this is also not a functional change.

For the remainder of this text, hardware will not be referred to as flexible, hardware of interest may be referred to as being reconfigurable. This notion of reconfigurability is also fictitious. It can aid to think of reconfigurability as a multiplexor at the output of the internal state function. If the digital system has a single control input representing either a 0 or a 1, then the conventional wisdom is that this control signal would be multiplexing between two (possibly similar) internal state functions. For the example of an n-bit adder and an n-bit subtractor which can controllably be multiplexed between, look to Figure 4-2.

For two separate control signals, there can exist up to a possibility of four distinct internal state functions to multiplex between. Note, the true internal state function remains unchanged. The total design of the function includes its **k** multiplexed possibilities, but the union of all possible functions and the multiplexor combine to represent the original internal state function, which is wholly defined at design time. That we refer to such systems as reconfigurable is a convenient handoff from common English.

With this thorough grounding, we will proceed with the understanding that digital systems being referred to as reconfigurable are not in fact runtime-defined, rather they are multiplexing between well-defined subsets of the internal state function. These “reconfigurable” systems have a bipartitioned input set with a select few inputs having been arbitrarily called control inputs. These digital systems which are defined as being reconfigurable have control signals which are programmable by some higher master digital system.

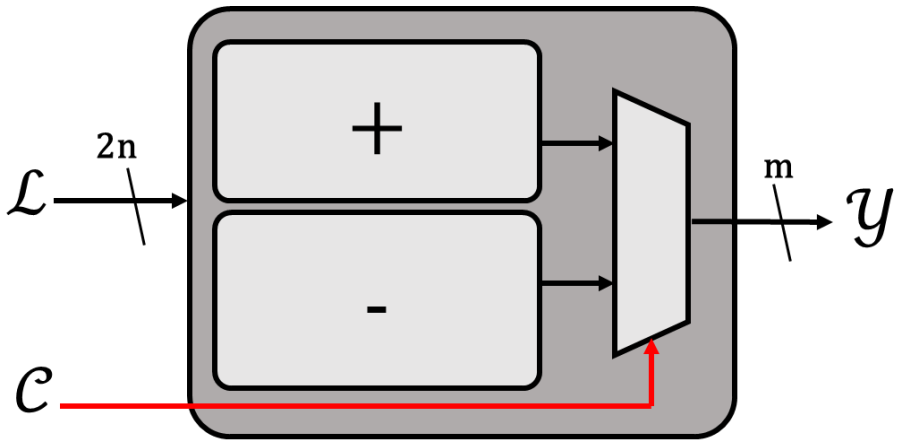


Figure 4-2: The control input to the system selects between an n-bit adder or an n-bit subtractor. Per the vernacular, this system is reconfigurable.

Following from the previous summary of hierarchy in digital systems, we can categorize any logic system when viewed from the top of its hierarchy on down. A potentially useful, if deeply compromised categorization, is that a logic system is reconfigurable if it contains input signals which the top of the logic system defines to be noteworthy of the control moniker. Effectively, a logic system is reconfigurable if it exists within a hierarchy, and exposes "control" signals to its top-most parent module. Unfortunately, this categorization is deeply flawed in that, it fails all previous discussion on the complete arbitrariness of naming input signals as control signals. Following this categorization any logic system within a hierarchy could be referred to as being both reconfigurable and as being not reconfigurable, depending on the choice of control signals.

An AND logic gate is not reconfigurable because it is a dedicated logic gate. An AND logic gate is reconfigurable because it's controllably a PASS gate. An 8-bit adder is not reconfigurable because it will add two 8-bit numbers for all time. An 8-bit adder is reconfigurable because it is controllably a 7-bit adder with overflow detection. A computer is not reconfigurable because all its inputs are taken from the memory which belongs within the computer. A computer is reconfigurable because it is controlled by a human providing control signals at the keyboard. A human is not reconfigurable because the human contains no external control signals.

The top of this logic system, for most practical purposes, would be the human designer. The human designer would always be the determinant of whether a logic system's inputs are controls, and whether the system as a whole is reconfigurable. Another compromise of the classification is that it lacks certainty and may not be transferrable across different human designers. A different human designer may disagree on what would constitute a control signal, producing a different set of what constitutes a reconfigurable system.

Accepting the compromises for this categorization, we can still make some useful and distinctive statements. A CPU may be reconfigurable if it has inputs which are human-controlled, and an FPGA may be reconfigurable if it has inputs which are human-controlled. These statements both map to conventional wisdom with the keyboard and mouse acting as control signals to the CPU, and the JTAG probe acting as control signals for the FPGA. Note the important caveat in that last statement about FPGAs being directly controlled by the human through the JTAG. The JTAG is a signal from the computer to the FPGA through the motherboard. The JTAG is definitely controlled by the human designer, but is so indirectly. An offhanded explanation for this difference is that the CPU has its control logic on the chip itself, whereas the control logic of the FPGA is "off-chip" and must be determined somewhere else.

Up to now we have not elaborated on the possibility of having more than one internal state function within a module. A hierarchy can exist within the internal state function of a module. Any internal state function can be partitioned into **k** distinct state functions which are multiplexed by control signals. We can now ponder the existence of these control signals. Within a hierarchy, the control signals must come from the same level in the hierarchy or passed down from above. From before, we know that control signals are inputs of specific significance, but they are still just input signals. Which means they are still logical outputs from some other logical system. In the Von Neumann model the control unit is treated separately to the logic unit and provides meaningful controls to the logic unit.

It is trivial to say that what the Von Neumann Architecture refers to as a control unit can be substituted for a logic unit with the same inputs, outputs, and internal state function. In this way, we can represent the Von Neumann Architecture as Memory, Input/Output, and two logic units with one providing additional logic inputs to the other. These two logic units can be bucketed into a larger single logic unit, and say that this greater single unit represents the internal state function of the entire CPU.

The Von Neumann Architecture represents a specific case of the more generalized model outlined in this thesis. The statement that a CPU is a Von Neumann machine is a stricter statement than that a CPU is a machine with inputs, outputs, internal memory, and an internal state function. A CPU's control signals are, in effect, contained within the greater CPU. That the CPU has control signals is a useful descriptor by humans, but these signals are contained within the CPU. An additional classification for hardware reconfiguration is defined: That reconfigurable hardware can either be classified as Adult or as Child. A reconfigurable hardware is Adult if it's control signals are collectively all either internal or directly controlled by the top of the hierarchy (typically the human designer). A reconfigurable hardware is Child if any of its control signals are directly controlled by other hardware. This other hardware can be reconfigurable or not, Child or Adult, it matters only that the reconfigurable hardware is indirectly controlled by the human. In this way, we can classify a CPU as Adult, whereas an FPGA would be classified as Child. The FPGA's control signals are directly controlled by a CPU, which is itself reconfigurability Adult. An FPGA which sets its own control signals would be considered reconfigurably Adult.

For the purposes of reconfiguring the FPGA another distinction will need to be made. Most FPGAs have the quality that they are considered spatially reconfigurable but are considered to lack temporal reconfigurability [6]. For practical purposes these FPGAs are reconfigured by the Adult control of a CPU which is typically on a separate chip. The FPGAs are connected to the CPU through the JTAG connector, and the CPU will dump a binary file onto a

local memory bank (typically a form of SRAM) on the FPGA. From then forward the FPGA will load up its configuration from this memory bank. Following our previous discussion, it can be illuminating to consider two separate forms of control signals for the FPGA.

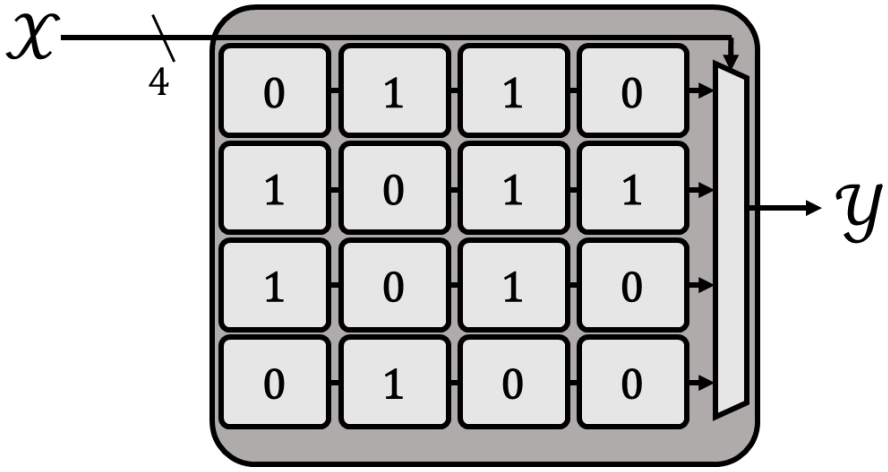


Figure 4-3: An imagined logic system which, when given 4 bits of inputs, looks up the value for a single output from a bank of memory elements.

In the canonical case, the control signals for the FPGA are direct from the CPU through the JTAG. It is through this case that the FPGA is not considered temporally reconfigurable. We have not attempted a discussion of the temporal dimension as of yet, but the primary causes of the FPGA reconfiguration stasis are design, delay, and computational cost. The FPGA's internal Look-Up Tables (LUT) are designed with the assumption that the reconfiguration will only occur with set requirements at runtime. As an example, Figure 4-3 depicts a LUT with 4 inputs. For all possible 16 elements of the input state, every element must have an associated bit of memory. In this example, 4 inputs requires a LUT to store 16 bits of memory. Memory is to be resumed in Chapter 6. The design and optimization of these LUTs is simplified by the assumption that reconfiguration occurs only at runtime, but the design is nevertheless only guaranteed to operate within these assumptions. The design of the FPGA often makes reconfiguration at a later time impractical if not impossible. When taking a step up the hierarchy the other simple

argument for lacking temporal reconfiguration is the strict delay that occurs with the transfer from the CPU to the FPGA. Lastly, the computation to map a design to an FPGA may be quite expensive. It can be convenient to think of the FPGA as being Startup Reconfigurable and as being not Runtime Reconfigurable.

That the FPGA is Startup Reconfigurable is contingent on the FPGA being reconfigurable which, as per analysis, is perspective on choice of control signals. If the signals from the memory blocks to LUTs are chosen as the control signals, rather than those through the JTAG, our results can differ. Any given LUT is given the information it needs to reconfigure from the memory block. These signals in general do not change over time. Within any LUT is a kernel which acts as a reconfigurable module without memory. This kernel takes its inputs and controls and passes the output as-is. This kernel has no information of the past, as we have excluded memory elements in our definition of the kernel. As the kernel contains no information of the past, it simply parses its current understanding of the input space. This kernel within the LUT thus satisfies our six assumptions for the chapter. This kernel is runtime reconfigurable. The control signals come from the memory block which was written to at startup, those control signals do not change for the life of the FPGA. However, that those signals are constant does not change that they are control signals for that kernel. If they changed, the kernel would reconfigure. If we consider the FPGA to be the internal state function within this kernel inside a LUT, and the control signals to be from the memory block, the FPGA is also runtime reconfigurable.

For a logic system with **n** inputs, creating an input space of cardinality 2^n, **m** outputs, creating an output space of cardinality 2^m, and lacking internal memory elements, what is the number of control inputs **c** necessary to represent all possible mappings? For a system with **n** inputs and **m** outputs, no control signals, there is exactly one mapping available at any given time: the internal state function created at design time. In this trivial case, the zero control signals select between a set of one possible mappings. For a single control input, the logic

system can select among two possible mappings. By induction, **c** control signals enable the logic system to select among 2^{c} possible mappings. Note that these mappings may not be distinct or particularly differentiated in practice. It is alone sufficient to say that the **c** control signals bound the output signals to 2^{c} distinct mappings. An internal state function with **k** possible reconfigurations is depicted in Figure 4-4, note that the act of reconfiguration is solely enacted by a multiplexor at output.

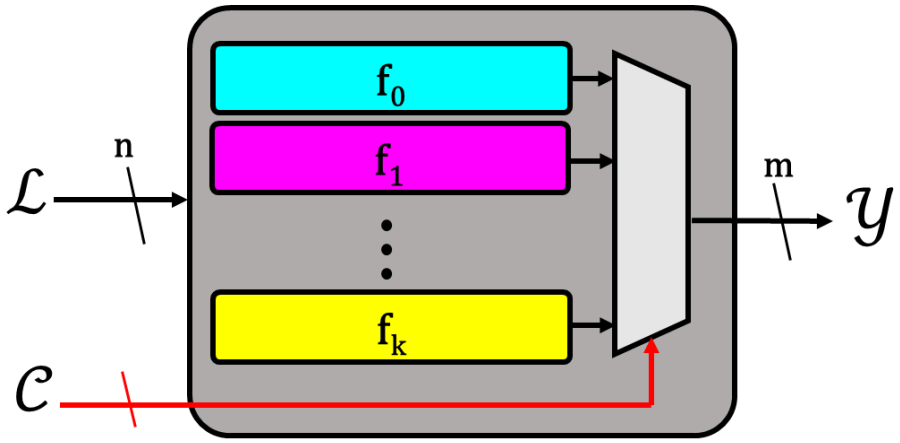


Figure 4-4: For an internal state function of **k** possible functions, the output can be modelled as a multiplexor selecting among **k** wholly separate functional blocks.

The question of how many control signals necessary to represent all possible functional mappings can be restated: What is the number of control signals whereby any input space entry can map to any output space entry. The span of possible control signals creates its own control space. Every entry in the control space uniquely corresponds to one entry in the function space. Then the number of entries in the control space must be $2^{m}{}^{\wedge}2^{n}$, where for binary signals there must be $\log(2^{m}{}^{\wedge}2^{n})$ control signals, or just **m** * 2^{n}, at minimum. This sets the minimum bound for control signals to represent all possible mappings in an **n** by **m** logic system.

In this manner we can think of control signals in a new way. In the first case we considered control signals as inputs and represented them as a named extension of the input

space. As proved in the above conjecture, we can also consider control inputs separately from the logic inputs. In this manor, the control inputs are independent from the input space, rather the control inputs select an entry from the function space. It is convenient to think of these control signals as reconfiguring the hardware, that the addition of multiple entries in the function space somehow makes the hardware “flexible”. Ultimately, the entries of the function space are defined at design time. For a memoryless logic system, there are four decisions to make at design time. The number of logic inputs to the system, the number of control inputs to the system, the number of outputs from the system, and the entries of the function space.

This analysis has centered on a memoryless logic system at a given instance in time. Real designs do not exist at an instance in time, they exist in the continuity of time. We can say that a memoryless logic system at a specific instance has a single entry in its input space, control space, and output space. For signals held constant for a duration, the entries in those spaces are held constant for the same duration. It is trivial to say that a reconfigurable memoryless logic system has a reconfiguration space at a given instance in time, but this reconfiguration space can also be defined over the continuity of time. If any of the control signals to a reconfigurable hardware are capable of changing over a duration from time zero, then the reconfigurable space is continuous in time and occupies a set of 2^c in space.

To ask what kind of problems a piece of hardware can compute, is to ask what exists within the reconfiguration space. The reconfiguration space is dependent on choice of control signals, so encompassing all inputs as control signals would maximize the reconfiguration space. It may be inappropriate to consider all input signals as controls given that control signals are most useful when an end-user has access. The simplest of problems to compute are memoryless and commonly found in the lexicon of human use, the simplest examples being the AND gate or the ripple-carry add operation. More complicated computational problems may have memory elements (towers of Hanoi), layered computation (Fourier Transform), or elements of both (CPU simulation). Any mapping of a hardware, indeed any software run on the machine, must

exist within its reconfiguration space. This space is defined by choice of control signals and internal state function operation, both of which can be determined at design time.

The following is a naïve approach to permute all possible entries within the reconfiguration space. At the completion of logic design, the designer specifies a subset of the input signals as control signals and permutes through the control space entries. For a memoryless system where the inputs have no delay constraints and are independent and uniformly distributed, then an entry in the reconfiguration space at time tau is equivalent to this entry at any time. For two functions that exist within a closed set it is possible in general to determine equivalence. For two logic systems, one larger or of equal size to the other, it is possible to determine the equivalence of the reconfiguration space of the larger system to the complete logic system of the other.

The naïve approach to this is to permute all possible reconfigurations of the larger logic system and determine equivalence to the complete logic system of the smaller. A more sophisticated method would be to attempt to data flow graph node reduce both systems before permuting through the simplified reconfiguration space. A final means is to establish large parts of the smaller system as subsets of the larger systems graph, and then attempt through reconfiguration to establish the complete subset of the smaller system within the larger.

CHAPTER 5

Adding the Continuity of Time

Two assumptions about delays in the logic system will be assumed for simplicity in proofs and examples. The first assumption is that delays once created will remain the same delay for all time. That is to say, a delay instantiated with a wallclock delay of 1 picosecond will remain a delay of 1 picosecond for the duration of operation. The second assumption is that two or more delays will not interact nonlinearly, that two delays in series are equivalent to one delay with its duration as the sum of the two. In reverse, any one delay can be composed as one or more delays whose sum is equal to the initial delay. We will adjust our existing 5 assumptions to this new set:

Assumptions

1. All systems will be purely digital
2. The internal state function is well defined for all time
3. Maximum entropy for the input set
4. Delays are stationary in time
5. Delays do not interact nonlinearly
6. Memoryless state function

A delay within a data flow graph is represented as a node which takes a set of inputs and passes the inputs along after its prescribed delay to the outputs. A delay within a data flow graph has no logic of its own and passes the data along unchanged. The simplest example of a delay within a data flow graph is a single delay element with a single input and single output. This

simplest case is depicted in Figure 5-1. The prescribed delay of the delay element, tau, is constant for all time, per assumption 4. This delay element passes its single input along to its single output after a delay of tau. The single delay element can also be separated into two or more smaller delays whose delay sum is tau, a trivial case can exist of a smaller delay element with a delay of 0. For a larger delay element of multiple inputs and multiple outputs, the convention is chosen: there can only exist as many inputs as outputs, one input maps exclusively to one output, and the delay from any one given input to its output is identical across all input-output pairs.

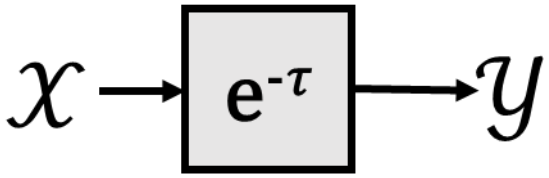
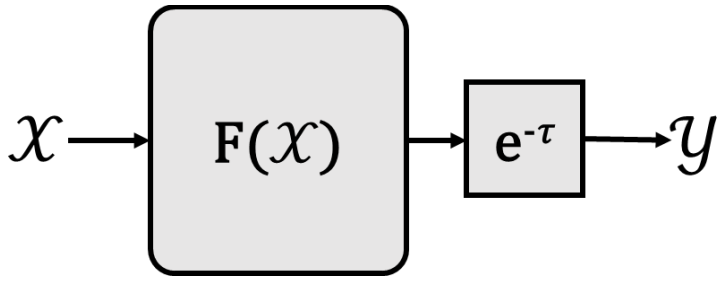


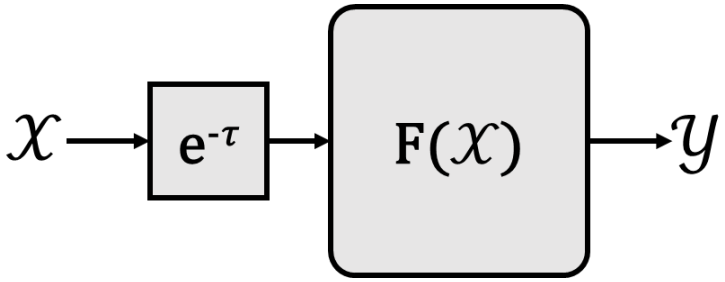
Figure 5-1: An isolated delay element.

Establishing more complicated networks of delays within data flow graphs is possible by composing graphs of multiple delay elements. But data flow graphs require distinct logic nodes alongside delay nodes for meaningful computation. Starting with the simplest case of a data flow graph that contains no delays, the data flow graph has 1 input arrow and 1 output arrow. The data flow graph contains a single logic node which maps the 1 input to the 1 output. For a binary logic node, the node can only meaningfully PASS the input, NOT the input, HIGH the input, or LOW the input. We assume that the delay node must be distinct from the logic node, so the delay node can only be introduced at the input of the graph or at the output of the graph. This is equivalent to saying that the delay node can be introduced before the logic node or after the logic node. It is easy to prove that the introduction of a delay node on one side of the logic node is equivalent to introducing an identical delay on the other side. Based on our assumption of

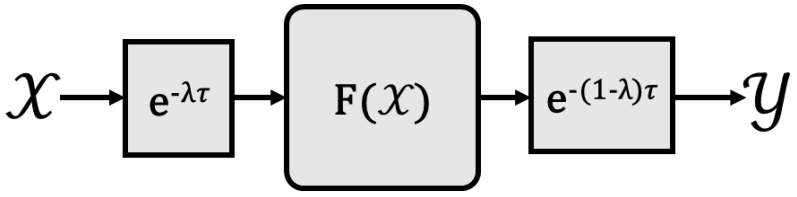
nonlinear combination of delays the delay element being added to the graph can be completely mixed between one side of the logic node and the other. In the extreme case, all of tau is delayed on one side. Whereas, in the median case, half of tau can be delayed before the logic node and half can be delayed after, this is also easy to prove. Depictions for all of these examples can be seen in Figure 5-2.



a)



b)



c)

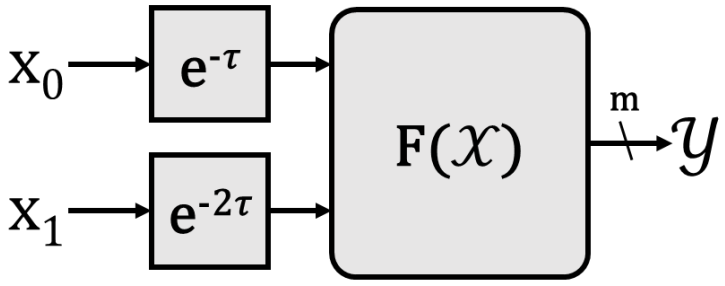
Figure 5-2: A delay element being moved through a logic element.

In the more generalized case we can extrapolate to a graph with **n** inputs and **m** outputs; this is equivalent to the logic system of **n** inputs, **m** outputs, and an internal state function. Where in this case the data flow graph solves to the internal state function. In this more general case we can introduce delay to the data flow graph, or logic system, such that the delay from any input to any output is always a specified tau for any input and output paring for all time. Given our initial assumptions of constant delays for all time, the latter is easy to assume. Forcing consistent delays across all input-output pairings can be trivially solved with the introduction of **n** delay elements, one at each input, or **m** delay elements, one at each output. This is equivalent to partitioning the logic from the delay and solving each separately. This example is not far removed from the single input, single output case, differing only in that the input and output sets have increased in size.

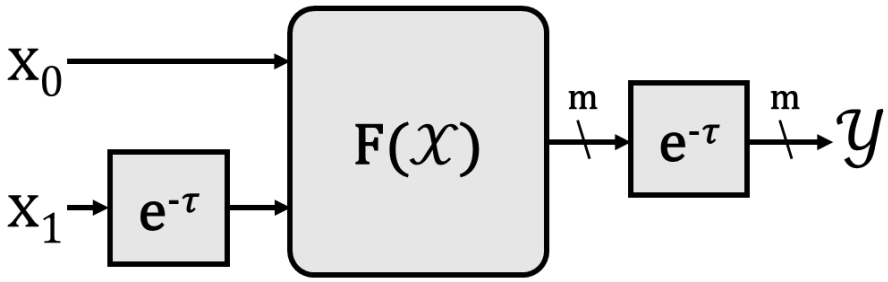
For any data flow graph that consists solely of arrows, logic nodes, and delay nodes the movement of delay nodes among logic nodes can be simply defined. In the simple case, where all inputs have equal delays, movement is simple. The delays can all collectively be moved through a logic node, just as depicted in Figure 5-2. Moving a delay element through a logic node will violate equivalence if a select number of conditions are not met. The direction that the delay node moves through the logic node does not matter, whether from input delay to output delay or vice versa, the process is reversible. But regardless of direction, all delay elements moved into the logic element must have the same delay subtracted, this delay must then appear as delay elements for every edge extending from the other side of the logic node.

Taking the more sophisticated example of a logic node with two inputs and **m** outputs, where input 1 has a delay of 1 picosecond from the previous logic node and input 2 has a delay of 2 picosecond from the previous logic node. Delay nodes can be arbitrarily separated into smaller delay nodes, so the delay in input 1 is separated into two 1 picosecond delays. One of these 1 picosecond delays is combined with the 1 picosecond delay from input 2 and moved through the logic node. In total, the final graph has a delay element of 1 picosecond on input 1, no delay on

input 2, and a 1 picosecond delay following both outputs. Look to Figure 5-3 for a clearer depiction.



a)



b)

Figure 5-3: a delay of one tau is moved through a logical element. In a) there are delays on both input arrows of differing delay. After a delay of tau is moved through in b), a delay of tau is also left on one of the input arrows.

The analysis of Chapter 3 establishes that any internal state function of **n** inputs and **m** outputs can be represented as **m** parallel submodules each taking a subset of the **n** inputs, this form can be constructed in a manner logically equivalent to any internal state function. We can extend this generalized internal state function idea, to an idea of generalized data flow graphs, only now including the addition of delay nodes. Starting with the intuition that an internal state function can be perfectly modeled as data flow graphs, delay node linear interaction, and the movement of delays nodes within a data flow graph. Delay nodes placed arbitrarily within the graph can always be pushed toward the input and output of the entire graph. From any one

input traced to any one output there will be a delay that corresponds to the duration in time it takes for an update at the input to drive a change at the output.

A simulation of this logic system is actually simplified by the nonideality of real circuits. The delays in real logic gates violate both assumptions for delay elements. The delay for a logic gate can vary with temperature, flicker noise, and physical deformation, so the assumption that delay is constant for all time may not be a reasonable for specific operations. Secondly the delays do not add nonlinearly because the rise time of one logic gate can affect the delay of the proceeding gates, and a collection of other effects. Fortunately, the best and worst times for these gates have been found to add up approximately linearly. And, fortuitously, in the case of logic simulation, we may not particularly care about the intermediate results of computation. The computation is only registered when the computation has completed, and the result is final.

In this matter, the nonlinear delays through the logic gates is not necessarily a concern, provided we can guarantee the best and worst timing delays. Once an input has been changed we can be reasonably sure that any corresponding output will update within a window of best and worst delays. For circuit simulation on a computer, for the duration that the logic gate is solving toward the correct result, we can leave the output as-is, drive it to a specified logic value, or drive to “don’t cares” for the duration of the computation. This simplifies our simulation, as for each logic gate (or logic node) we drive the output however we choose, and after the input-output pair delay update the output to the genuine value. Given the assumptions, in combination with real-world nonideality, it is perfectly reasonable to model this graph, and therefore logic systems, as a set of **nm** delays (for **n** inputs and **m** outputs) unique for each input-output pair and a data flow graph in the center lacking any delay nodes.

At any given instant in time, the input to delay nodes can be modelled as outputs of the entire data flow graph, likewise for the outputs of the delay nodes: They can be modelled as inputs to the entire graph. If we constrain the data flow graph as if it were a single instant in time, and only allow the spatial dimension to exist, these delay nodes act very similarly to global

inputs and outputs. Starting at a trivial case, where there are no delay nodes within the graph, but the inputs change over time. Then it can be conveniently to think of the function space as being able to change over time. For a function space with **k** entries, then at any given moment in time the functional space is capable of representing **k** functions (this is a trival result). But if an internal state function is represented as a given function at a specific time, the rate at which that function can change is determined by the entropy rate of the control signals. This is, in effect, the bandwidth, owing to the assumption of independent input signals. To the logic input set the internal state function would appear simply as the **i**th function of the set and appear over time and shifting through the various elements (or colors) of the entire function space. This is depicted in Figure 5-4. The span of the function space across both the space and (continuous) time dimension is dependent on the cardinality of the control space (and therefore function space) and the bandwidth of the control signals selecting among the control space.

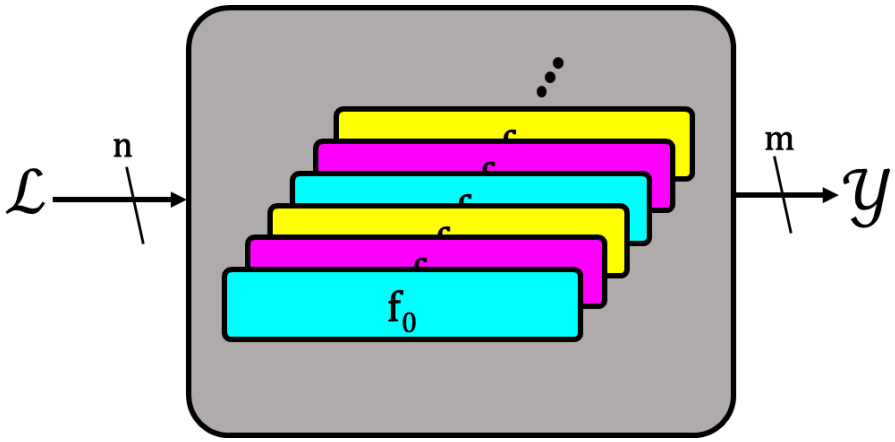


Figure 5-4: An internal state function switching between different elements of a function space.

Finalizing the analysis of Startup versus Runtime reconfigurable, an alternative classification is provided. A logic system can be Statically reconfigurable if the entropy rate (bandwidth) of the control signals tends to zero over operation. Or a logic system can be Dynamically reconfigurable if it has a nonzero entropy rate for its control signals.

CHAPTER 6

Discretized Memory Elements

The combination of logic nodes and delay nodes is insufficient to encompass all digital logic, the inclusion of memory nodes is also necessary. At any given instance in time, the output of the memory element appears as an input to the data flow graph, and the input of the memory element appears as an output of the data flow graph. If a memory element is unchanged over a duration, the memory element appears as a constant input over the duration. In a way, the logic inputs can be thought of as inputs over the spatial dimension, and memory elements can be thought of as inputs over the temporal dimension.

Assumptions

1. All systems will be purely digital
2. The internal state function is well defined for all time
3. Maximum entropy for the input set
4. Delays are stationary in time
5. Delays do not interact nonlinearly

If we consider for a moment, an idealized form of memory. This idealized memory begins sampling the inputs to the memory block at the instant of turn on. For the purposes of simplicity, we assume the inputs are updated at discretized events, a clock, and that there are **d** binary inputs. At every tick of the clock the memory needs to store an additional **d** bits. Before device turn on, the memory stores no information, 0 bits, so the space of possible values before

turn on is a set of cardinality zero. At the first tick of the clock the memory has **d** binary inputs and has stored **d** bits. The space has expanded to cardinality of 2^d. For every tick of the clock for all time, the cardinality of the space of possible values increases by a product of 2^d. Such that at a given time **j** the memory has stored **jd** bits of information and the set of possible values has expanded to 2^{jd} elements.

At a certain point this idealized memory element needs to be constructed in practice. If we accept at face value that this memory element both has the space to store and the internal logic to correctly store all **jd** inputs for a very large **jd**, then we still face the problem of doing something practical with this stored memory. The stored information must be passed along to some logic farther down the system. For any given time **j** the internal memory can have up to **jd** bits of stored information, in order for a logic system downstream from the memory to receive all this information, the memory element must also have **jd** output signals. But these output signals in hardware present themselves in hardware as wires, and the number of wires is determined during design time, meaning that the maximum information of **jd** sent is determined before runtime (i.e. at a **j** of zero). The output bandwidth from this stored memory presents as a practical limit to the amount of information that can be useful downstream at any given point in time.

As an example of the peculiarities of this effect, we return to the LUT example of Chapter 4. The LUT returns a single bit of information at output for every 4 inputs which are presently connected. But, the analysis showed that this LUT necessitated 16 bits of internal information, one for each element of the input state. Guaranteeing a complete span of the function space over the space dimension required 16 bits of information. In practice, the values of the LUTs are not determined dynamically, but are set at the design time or upon device startup. If you consider that all 16 bits of the information present in the LUTs is provided by an external control signal, or a set of control signals, then the complete span of the functional space would require that the bandwidth of the control signal was 16 bits for every 4 bits of logical input (4 logical signals), see

Figure 6-1. Which means that your control network would represent four times as much area as your logic distribution network. This is probably unrealistic in practice, so the compromise forces the entropy rate of the control network to zero, forcing the total FPGA so be statically reconfigurable (or startup reconfigurable).

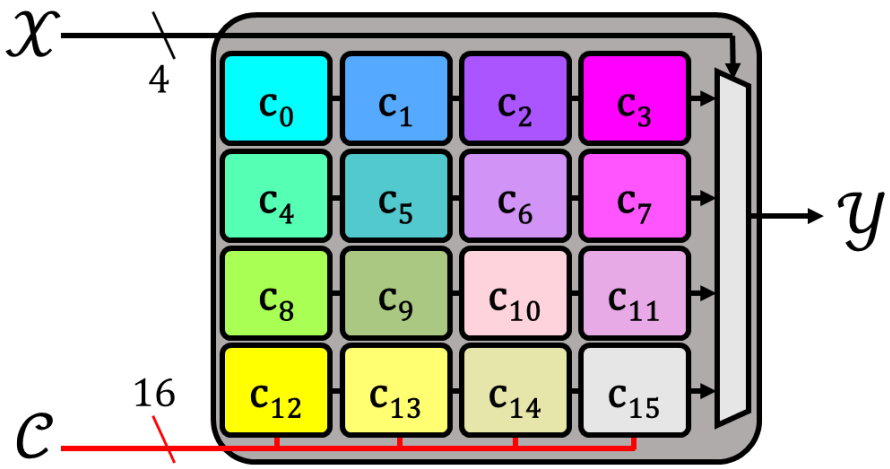


Figure 6-1: The function space of a LUT is determined at runtime by the control signals provided externally, alternatively, the values of the LUT are determined at startup, in which case, the span of the function space is severally reduced.

Indeed, memory elements that exist in hardware are not designed according to the idealities listed above. A memory element is defined by the number of bits that it can internally store and its control signals. Memory elements are not designed to store information for all time, but to controllably store information for a single instance in time. This information is then controllably passed as output for all time after the recording instance. For a memory element which stores a single bit of information, the information as to which instance in time the memory was recorded was not also stored. This single bit stored and passed as output by the memory is only a single bit of information.

The memory elements that exist within modern digital systems consist of ROMs, DRAM cells, SRAM cells, registers, and latches. For all intents, the ROMs can be abstracted as tying

particular lines to high and low, which do not present as new information since this was defined during design time. DRAM cells and SRAM cells are typically sufficiently far from the logic of interest that they are neglected. Registers are a single bit of information and controlled by a clock and an input. The memory stored in the register for a simple D-Q register is always the input from the previous discrete time instance. One bit of information is all that is stored, all that is passed from a register. Likewise can be said for a hardware latch, only one bit is ever stored or passed from a latch.

For a logic system that contains a set of inputs, a set of outputs, an internal state function, and a single bit of stored memory, the memory output presents itself as an additional input to the internal state machine. As stated, the output from this memory is only ever presented as a bit of information. The inputs to the internal state function are only ever presented as a single bit of information. For a logic system with **n** inputs and a single bit of memory, the input to the internal state machine is equivalent to **n**+1 inputs, and the input space to the internal state machine has a cardinality of 2^{n+1}.

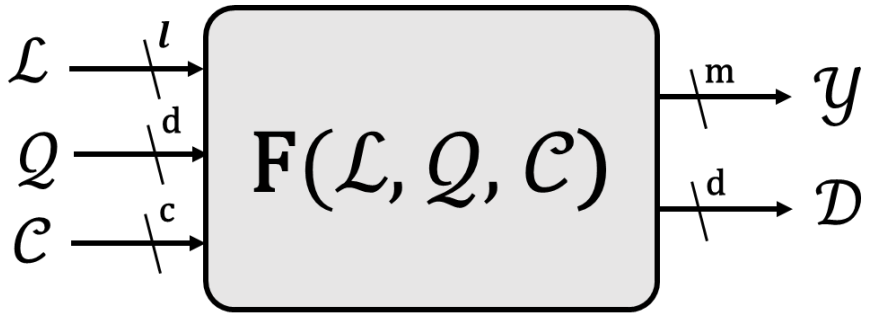


Figure 6-2: The Von Neumann Architecture reduced to its functional equivalent.

Every logic system contains an understanding of the present: The inputs, an understanding of the past: The memory, an internal logic to solve: the internal state function, and its outcome: The outputs. This is summed up in Figure 6-2. Notice the similarities of this

model to that of the Von Neumann Architecture. The memory can be thought of as containing all of the information of the past available to the logic system. The input can be thought of as all the information of the present available to the logic system. The input information can be partitioned into the controls input which are codified with the information of being controls, and the logic inputs which are codified with the information as being strictly for logic. The **c** control input signals, **l** logic input signals, and **d** memory input signals collectively provide an input space of cardinality 2^{c+l+d}. This input space must be mapped to an output space of logic outputs, but also of memory inputs, so an output cardinality of 2^{m+d}. This collectively leads to a total of 2^{m+d}^2^{c+l+d} meaningfully unique internal state functions. And a total of $\min(2^{m+d}, 2^{c+l+d})$ meaningfully unique outputs.

CHAPTER 7

Conclusion

Any logic system is defined at first by its number of inputs, its number of outputs, and the internal state function which defines the logic of the system's operation. The rise of parallelism and so-called reconfigurability of modern hardware devices has not fundamentally altered this. Even with the end of Moore's Law, and the increasing reliance on parallelism, the logic fundamentals have not changed. For any given set of inputs, the value across all inputs combine to establish an element in the space of possible input sets. The same can be said of the output sets and the space that they combine to. For any logical system, even those which are not digital, the output entropy of the system is constrained by the input entropy of the system. Such that, even with the bifurcation of the inputs into a set of logic inputs and a set of control inputs, that entropy constraint is withheld.

Ultimately, for any given system with multiple control signals, it can be thought that the logical system is capable of selecting among a collection of different functional possibilities. Designers have a new set of mathematical tools to aid in the design of future parallel or reconfigurable devices. Starting with an initial understanding of the acceptable network bandwidth of the design, designers can solve for the tradeoffs between high levels of controllably (i.e. reconfigurably) versus the necessary bandwidth that is necessitated for that reconfigurability. For an environment where being able to reconfigure the system on startup is an acceptable tradeoff, the bandwidth of the control network can be reduced with memory elements stored adjacent to logical elements. This idea can be extended such that for an

environment that demands reconfiguration, but only infrequently, the control bandwidth network can be simplified for an arbitrarily low control entropy rate.

For a compute environment which requires high control bandwidth for all time the only real capable systems are CPUs, which have exceptionally low logic compute for the complexities and power draw of the whole system [6]. Indeed, the tradeoff between the control compute and the logic compute of the entire design determines not only how much logical compute a given device is capable of, it also directly determines the set of problems that the device can solve. The span of all compute problems that a design can solve are those that exist within the span of the design's functional space over the continuity of time.

Any logic system, in a sense, is capable of reconfiguration when such a system is given the proper set of control inputs and designed with a large functional space in mind. But this notion obscures the fundamental truth of that particular logic system, that the functional capabilities of the system are determined at design time. And that any expansion in the functional space of the logical system necessitates an expansion in the logic necessary to solve the functional space. In total, meaningful reconfigurability always comes at the expense of efficiency.

REFERENCES

[1] G.E. Moore, “Cramming More Components onto Integrated Circuits”, Electronics, vol. 38, no. 8, April 19, 1965

[2] G. E. Moore, “Progress in Digital Integrated Circuit,” Proceedings of the IEEE International Electron Devices Meeting , pp. 11-14, December 1975.

[3] N.S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J.S. Hu, M.J. Irwin, M. Kandemir, and V. Narayanan, “Leakage current: Moore’s law meets static power,” Computer, vol. 36, no. 12, pp. 68–75, Dec. 2003.

[4] D. Geer, “Chip makers turn to multicore processors,” IEEE Computer,vol. 38, no. 5, pp. 11–13, 2005

[5] J. von Neumann, “First Draft of a Report on the EDVAC,” Moore School of Electrical Engineering, Univ. of Pennsylvania, Philadelphia, June 30, 1945

[6] F.-L. Yuan, C. C. Wang, T.-H. Yu, and D. Markovic, “A multi-granularity FPGA with hierarchical interconnects for efficient and flexible mobile computing,” IEEE Int. Solid State Circuit, vol. 50, no. 1, pp. 137–495, Jan. 2015.

[7] N. P. Jouppi, C. Young, and e. a. Patil. In-datacenter performance analysis of a tensor processing unit. In Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA ’17, pages 1–12, New York, NY, USA, 2017. ACM