On Noisy Additions

Technical Report

Ralf Herbrich
Hasso-Plattner Institute
Potsdam, Germany
ralf.herbrich@hpi.de

July 27, 2022

Abstract

So far, most of algorithmic development has relied on the concept of *correct computing*, in particular at the level of the basic compute unit: the arithmetic logical unit (ALU). With the rapid rise of machine learning (ML) and artificial intelligence (AI) systems, *correct computing* is no longer strictly required as almost all ML and AI algorithm are based on numerical approximations of (cost) minimization problems. On the other hand, energy-efficiency is becoming increasingly important. While current research into energy-efficient ML/AI focuses on *approximate representations* of numbers (e.g., k-bit deep neural networks), we will explore the idea of *approximate computing* resulting out of under-powering the ALU of a processor.

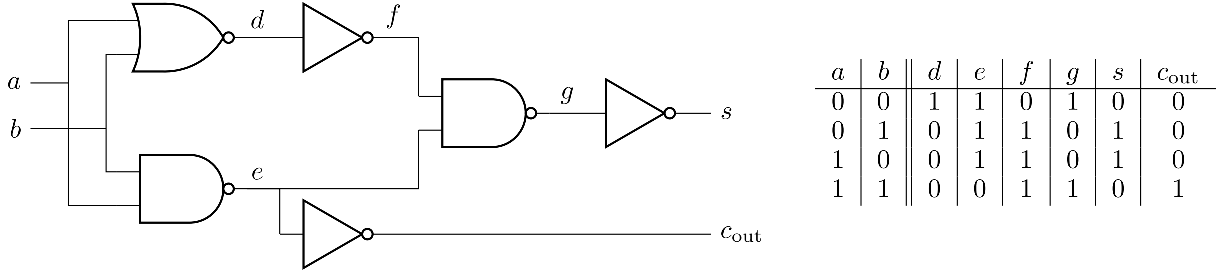
In this note, we will analytically derive the distribution over integers resulting from under-powering an ALU using message-passing to efficiently compute these distributions. One application of knowing this characterization is to use under-powered ALUs to *directly* implement probabilistic programs where the distribution parameters are directly controlled by the CMOS supply voltage.

1 Introduction

Over the past 80 years, most of algorithmic development has relied on the concept of *correct computing*, in particular at the level of the basic compute unit: the *arithmetic logic unit (ALU)*. All operations that change the state of static variables — either stored in registers or random-access memory (RAM) in a computer — rely on this special-purpose circuit. This digital circuit performs arithmetic and bitwise operations on binary numbers; it is the fundamental building block of the central processing unit (CPU) of computers, floating-point units (FPUs), and even graphics processing units (GPUs) [1]. Note that an ALU is not only used for numerical computations on program state variables but also essential for implementing the non-linear process flow in a CPU by virtue of using it to compute the next location in RAM where the program execution continues through arithmetic operations on the so called *program counter* (register).

The concept of correct computing has led to incredible advancements in the field of computational architecture as well as operating systems and databases. However, with the rapid rise of machine learning (ML) and artificial intelligence (AI) systems over the past 20 years, correctness in computing is not as essential as in operating systems or databases; almost all ML and AI algorithm are based on numerical approximations of (cost) minimization problems (see, e.g., [2, 3, 4]) and current research into energy-efficient ML/AI already focuses on *approximate representations* of numbers (e.g., k-bit deep neural networks [5]). In this paper, we will explore an orthogonal approach where rather than assuming that the representation of parameters is approximate, we assume that we reduce the voltage of a processing unit below the critical threshold where computational errors can be excluded. It is known that the switching power consumption of a CMOS transistor is directly proportional to the frequency f and also proportional to the quadratic on the supply voltage [6, Subsection 4.1.2]. However, when lowering voltage below a critical threshold [7], bit flip errors will occur.

The concept we are studying are closely related to the idea of *approximate computing* (see [8] for an overview). In approximate computing, it will be assumed that both hardware and software are only providing *estimates* of the desired behavior in exchange for energy efficiency. In this work, we will pursue this direction by studying an ALU under an assumed noise level where bit flips occur as part of the *input* to a transistor (and hence the logical gate units). In Section



a	b	d	e	f	g	s	c_{out}
0	0	1	1	0	1	0	0
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	0	0	1	1	0	1

Figure 1: **(Left)** Logic circuit design of a half-adder function in terms of NAND, NOR and NOT gates that computes the sum of two binary numbers $a \in \mathbb{B}$ and $b \in \mathbb{B}$ where s is the sum of a and b and c_{out} captures the carry-bit indicating an overflow (i.e., if a and b are both 1, then $s = 0$ and $c_{\text{out}} = 1$). **(Right)** The logical equivalent of the half-adder with the values for all intermediate results $d \in \mathbb{B}$, $e \in \mathbb{B}$, $f \in \mathbb{B}$, and $g \in \mathbb{B}$ for every possible list of (binary) input values. Note that $s = a \oplus b$ and $c_{\text{out}} = a \wedge b$.

2, we will start with describing the most basic building block of a (bitwise) *half-adder* before progressing to describe a (bitwise) *full-adder* and then a 4-bit ripple-carry adder (RCA). This will be followed by a description of our noise model and the resulting distributions of the outputs of a half-adder, full-adder as well as the 4-bit RCA in Section 3.

In the rest of the paper, we use $\mathbb{B} := \{0, 1\}$ to denote the set of binary numbers. Also, we will use the symbols $\wedge$, $\vee$, $\oplus$ $\neg$ to denote the two-valued functions of AND, OR and XOR as well as the one-valued function of NOT; we will use $\wedge$, $\vee$ and $\oplus$ in infix notation (e.g., $a \wedge b$) and assume that they map from $\mathbb{B} \times \mathbb{B}$ to $\mathbb{B}$.

2 Arithmetic Logic Units and Adders

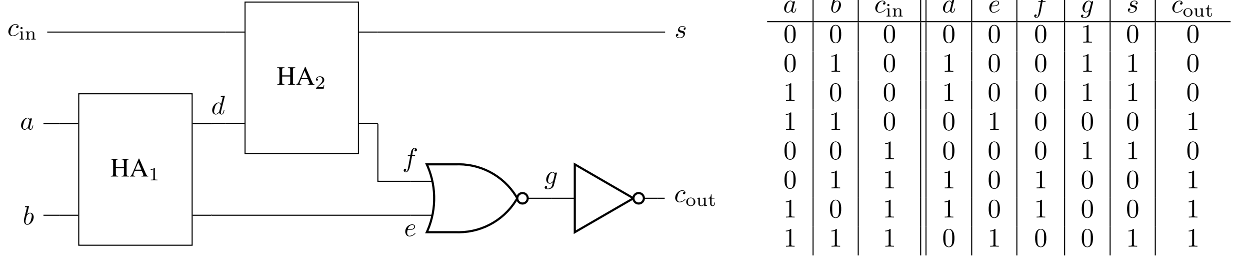
Every computer is an electric device which changes the state of its memory (i.e., randomly-accessible memory (RAM) or on-chip memory also referred to as *registers*) in response to the execution of instructions provided to the computer through RAM. The change of the state is performed by a unit called the *arithmetic logic unit* (ALU) which implements at the very least n-bit addition, bit-wise logic operations (i.e., AND, OR, NOT) as well as bit-wise shift operations[1]. Note that the instructions themselves are part of the state of the computer and that the program flow through this state is handled by using the ALU on a special register called the *program counter* (always pointing to the location in RAM where the current instruction is contained). Thus, the ALU is the most central component that allows any computer to perform all its tasks.

We will focus on one operation of the ALU that is central to all operations namely *addition*. Depending on the native bandwidth of the CPU, this includes the addition of two 4-, 8-, 16-, 32- or even 64-bit numbers provided in registers. As each number is represented by bits, we will denote the two inputs to addition $A = a_0a_1a_2a_3 \cdots a_n$ and $B = b_0b_1b_2b_3 \cdots b_n$, respectively. The addition of A and B is performed in a pipeline of n steps:

1. **Half-Adder**. First, a_0 and b_0 get added according to the rules of binary addition resulting in a sum bit s_0 as well as a carry-out bit c^0_{out}.
2. **Full-Adder**. Second, a_1, b_1 and c^0_{out} get added according to the rules of binary addition with a carry-in bit $c^1_{\text{in}} = c^0_{\text{out}}$ resulting in the sum bit s_1 as well as the carry-out bit c^1_{out}.
3. **Ripple-Carry Adder**. This sequence is repeated until the nth position given the bit string $s_0s_1s_2s_3 \cdots s_nc^n_{\text{out}}$ which is the $(n+1)$-bit answer of the addition of A and B.

In Figure 1 and 2 we have shown both the circuit design as well as the logic table for a 1-bit half-adder and full-adder. In terms of logic circuits, the sum bit of the half-adder is nothing more than and XOR of both inputs a and b and the carry-out bit is the logical conjunction of the two inputs. For the full-adder, the sum bit is the XOR of the carry-in bit, c_{in}, and the XOR of both inputs a and b; that is, if the carry-in bit is zero, the sum is the same as that of a half-adder but if the carry-in bit is one, the sum is the negation of the sum bit of the half-adder. Similarly, for the full-adder, the carry-out bit, c_{out} is one if either both inputs a and b are one or if the carry-in bit, c_{in} is one and at least one of a and b

[1]Computer architecture differs in the range of instructions is supports in hardware—the so-called instruction set architecture (ISA)—as well as the way that some operations are supported on RAM or only on registers.



a	b	c_{in}	d	e	f	g	s	c_{out}
0	0	0	0	0	0	1	0	0
0	1	0	1	0	0	1	1	0
1	0	0	1	0	0	1	1	0
1	1	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1	0
0	1	1	1	0	1	0	0	1
1	0	1	1	0	1	0	0	1
1	1	1	0	1	0	0	1	1

Figure 2: **(Left)** Logic circuit design of a full-adder function that computes the sum of two binary numbers $a \in \mathbb{B}$ and $b \in \mathbb{B}$ when a carry-in bit $c_{\text{in}} \in \mathbb{B}$ is also passed on. **(Right)** The logical equivalent of the full-adder with the values for all intermediate results $d \in \mathbb{B}$, $e \in \mathbb{B}$, $f \in \mathbb{B}$ and $g \in \mathbb{B}$ for every possible list of the eight (binary) input values.

are one. More formally,

$$s = a \oplus b \qquad c_{\text{out}} = a \wedge b \tag{1}$$

$$s = (a \oplus b) \oplus c_{\text{in}} \qquad c_{\text{out}} = (a \wedge b) \vee ((a \vee b) \wedge c_{\text{in}})\,, \tag{2}$$

where the first line is for the half-adder and the second line is for the full-adder, respectively.

Regarding the ripply-carry adder, looking at the rule for c_{out} in (2) we can see that $(a \wedge b) = 1$ is *forcing* the carry-out bit c_{out} to be one—irrespective of the value of c_{in}— and that $(a \vee b) = 1$ is *propagating* the carry-out bit from left to right (i.e., if $c_{\text{out}} = c_{\text{in}}$). One can use these two properties when parallelizing additions over many bits because the individual conjunctions $a_i \wedge b_i$ do not need to wait for the carry-in bit from the prior positions (which would be required in the ripple-carry adder) and can be computed in parallel; similarly, the individual disjunctions $a_i \vee b_i$ can be computed ahead of time all that needs to happen in sequence is a correction of carry-out flags where the disjunctions were one. This circuit design is faster and is known as a *carry-lookahead* adder; however, we will not consider this design here.

Finally, it is worth noticing that we expressed the half-adder, full-adder and ripple-carry adder in terms of the basic logic circuits of NAND, NOR and NOT (rather than AND, OR and NOT). The reason for this choice is that we will present a realistic noise model of the adders for an *actual* implementation of the logic circuits on *complementary metal-oxide-semiconductor (CMOS)*, the most commonly used fabrication process which uses complementary and symmetrical pairs of p-type and n-type *metal-oxide-semiconductor field-effect transistors (MOSFETs)* for logic functions. In CMOS, the smallest circuit is the NOT-gate which needs exactly one one p-type and n-type MOSFET in series: if the input voltage is zero (i.e., shorted w.r.t. ground), the n-type MOSFET is blocking and the current flows from the source to the output and therefore the output is non-zero (i.e., not shorted w.r.t. ground). Similarly, if the input voltage is above the threshold (i.e., not shorted w.r.t. ground), the p-type MOSFET is blocking and the current flows from input to ground resulting in a voltage of zero at the output. Similarly, in CMOS, the next smallest (elementary) circuit are the NAND- and NOR-gate which require two p-type and two n-type MOSFETs either in series or parallel (but always in opposite ways for a single circuit). Every other logical function is a composition of these three elementary logic gates and we therefore present all our analysis of adders in terms of these basic logic gates[2].

3 A Noise Model for Adders

In this section, we are describing a noise model based on the assumption that the input voltage to the actual circuit implementing the logic NAND, NOR or NOT gates are no longer kept constant resulting in variation of the *actual value* of the input to the NAND, NOR or NOT unit (see [9] for related work on the NOT-gate). Looking at the electrical circuit diagram of a CMOS NOT-, NAND- or NOR-gate in Figure 5, we see that such a variation would alter the input voltage V_i thereby leading to switching behavior of the corresponding MOSFET transistors.

We assume that the random variation is fully described through two parameters $\alpha \in \left[0, \frac{1}{2}\right]$ and $\beta \in \left[0, \frac{1}{2}\right]$ which denote the probability that a low-voltage input (i.e., bit state of 0) switches the n-type MOSFET on (i.e., bit state of 1) and vice versa for the p-type MOSFET[3]. More formally, we assume that

$$P(a_{\text{obs}} = 1 | a = 0) = \alpha\,, \tag{3}$$

$$P(a_{\text{obs}} = 0 | a = 1) = \beta\,. \tag{4}$$

[2]Note that in terms of Boolean algebra, NAND or NOR are the *smallest* set of logic functions that can express any Boolean expression because $\neg a = \neg(a \wedge a)$ but this type of analysis would not yield a realistic noise model and does not correspond to the mapping of a logic equation to CMOS.

[3]Note that a flip probability of more than 50% means that the inverted gate would make less mistakes.

With this noise model, it is possible that an NAND, NOR or NOT gate make mistakes in their computation. In the following table, we have listed the respective probability of each outcome 0 and 1 for the full list of bit-wise inputs a and b.

		$P(a \wedge b = 1)$	$P(a \wedge b = 0)$	$P(a \vee b = 1)$	$P(a \vee b = 0)$	$P(\neg a = 0)$	$P(\neg a = 1)$
a	b	$q_{\text{nand}}(a, b, 0)$	$q_{\text{nand}}(a, b, 1)$	$q_{\text{nor}}(a, b, 0)$	$q_{\text{nor}}(a, b, 1)$	$q_{\text{not}}(a, 0)$	$q_{\text{not}}(a, 1)$
0	0	α^2	$1 - \alpha^2$	$1 - (1-\alpha)^2$	$(1-\alpha)^2$	α	$1 - \alpha$
0	1	$\alpha(1-\beta)$	$1 - \alpha(1-\beta)$	$1 - (1-\alpha)\beta$	$(1-\alpha)\beta$	α	$1 - \alpha$
1	0	$\alpha(1-\beta)$	$1 - \alpha(1-\beta)$	$1 - (1-\alpha)\beta$	$(1-\alpha)\beta$	$1 - \beta$	β
1	1	$(1-\beta)^2$	$1 - (1-\beta)^2$	$1 - \beta^2$	β^2	$1 - \beta$	β

Note that this probability distribution reduces to point functions for $\alpha = \beta = 0$. Also, for $\alpha = \beta = \frac{1}{2}$, there is *still* information in the computation as the resulting probability distributions for NAND and NOR are *not* uniform: for NAND there are three bit patterns where $\neg(a \wedge b)$ equals one and only one bit pattern where $\neg(a \wedge b)$ equals zero and thus, for truly random bit flips there is a three times larger probability that the value of the NAND computation is one than zero; $P(\neg(a \wedge b) = 1) = \frac{3}{4} = 3 \cdot P(\neg(a \wedge b) = 0) = \frac{1}{4}$. A similar argument applies to NOR but with the event probabilities reversed. Only the NOT function is truly uniform if $\alpha = \beta = \frac{1}{2}$.

Noise Half Adders Given the logic circuit design in the top row of Figure 1, we can now compute the marginal distribution over the sum bit s and the carry-out bit c_{out} of a half-adder by summing over all possible values of $d \in \mathbb{B}$, $e \in \mathbb{B}$, $f \in \mathbb{B}$ and $g \in \mathbb{B}$ using the probability distributions given above. More formally, we have

$$\begin{aligned} P(s, c_{\text{out}}|a, b) &= \sum_{d\in\mathbb{B}}\sum_{e\in\mathbb{B}}\sum_{f\in\mathbb{B}}\sum_{g\in\mathbb{B}} P(s, c_{\text{out}}, d, e, f, g|a, b) \\ &= \sum_{d\in\mathbb{B}}\sum_{e\in\mathbb{B}}\sum_{f\in\mathbb{B}}\sum_{g\in\mathbb{B}} P(d|a, b) \cdot P(e|a, b) \cdot P(f|d) \cdot (g|e, f) \cdot P(s|g) \cdot P(c_{\text{out}}|e) \\ &= \sum_{d\in\mathbb{B}} P(d|a, b) \cdot \sum_{e\in\mathbb{B}} P(e|a, b) \cdot P(c_{\text{out}}|e) \cdot \sum_{f\in\mathbb{B}} P(f|d) \cdot \sum_{g\in\mathbb{B}} (g|e, f) \cdot P(s|g) \\ &= \sum_{d\in\mathbb{B}} q_{\text{nor}}(a, b, d) \cdot \sum_{e\in\mathbb{B}} q_{\text{nand}}(a, b, e) \cdot q_{\text{not}}(e, c_{\text{out}}) \cdot \sum_{f\in\mathbb{B}} q_{\text{not}}(d, f) \cdot \sum_{g\in\mathbb{B}} q_{\text{nand}}(e, f, g) \cdot q_{\text{not}}(g, s) \\ &=: q_{\text{ha}}(a, b, s, c_{\text{out}}), \end{aligned} \tag{5}$$

where the first line follows from the law of total probability, the second line follows from the directed graphical network structure of the half-adder, the third line follows from pulling out each of the terms as far as possible from the inner sums, and the last line follows from the definition of the probabilistic logic gates. Note that in an efficient implementation, the partial sums after summing out $g \in \mathbb{B}$, $f \in \mathbb{B}$, and $e \in \mathbb{B}$ are the so-called *messages* in the factor graph of logic circuit functions (see [10] for a full overview of message passing).

Noisy Full Adders Using (5) and the logic circuit design of the full-adder in Figure 2, we can now compute the marginal distribution over the sum bit s and the carry-out bit c_{out} of a full-adder by summing over all possible values of $d \in \mathbb{B}$, $e \in \mathbb{B}$, $f \in \mathbb{B}$ and $g \in \mathbb{B}$. We have

$$\begin{aligned} P_{\text{fa}}(s, c_{\text{out}}|a, b, c_{\text{in}}) &= \sum_{d\in\mathbb{B}}\sum_{e\in\mathbb{B}}\sum_{f\in\mathbb{B}}\sum_{g\in\mathbb{B}} P(s, c_{\text{out}}, d, e, f, g|a, b, c_{\text{in}}) \\ &= \sum_{d\in\mathbb{B}}\sum_{e\in\mathbb{B}}\sum_{f\in\mathbb{B}}\sum_{g\in\mathbb{B}} P(d, e|a, b) \cdot P(s, f|c_{\text{in}}, d) \cdot P(g|e, f) \cdot P(c_{\text{out}}|g) \\ &= \sum_{d\in\mathbb{B}}\sum_{e\in\mathbb{B}} P(d, e|a, b) \cdot \sum_{f\in\mathbb{B}} P(s, f|c_{\text{in}}, d) \cdot \sum_{g\in\mathbb{B}} P(g|e, f) \cdot P(c_{\text{out}}|g) \\ &= \sum_{d\in\mathbb{B}}\sum_{e\in\mathbb{B}} q_{\text{ha}}(a, b, d, e) \cdot \sum_{f\in\mathbb{B}} q_{\text{ha}}(c_{\text{in}}, d, s, f) \cdot \sum_{g\in\mathbb{B}} q_{\text{nor}}(e, f, g) \cdot q_{\text{not}}(g, c_{\text{out}}) \\ &=: q_{\text{fa}}(a, b, c_{\text{in}}, s, c_{\text{out}}), \end{aligned} \tag{6}$$

where the first line follows from the law of total probability, the second line follows from the directed graphical network structure of the full-adder, the third line follows again from pulling out each of the terms as far as possible from the inner sums, and the last line follows from the definition of the probabilistic logic gates. Again, in an efficient implementation the probability distribution table $q_{\text{ha}} \in \mathbb{R}^{\mathbb{B}\times\mathbb{B}\times\mathbb{B}\times\mathbb{B}}$ of the half-adder is computed once and re-used in all the inner sums.

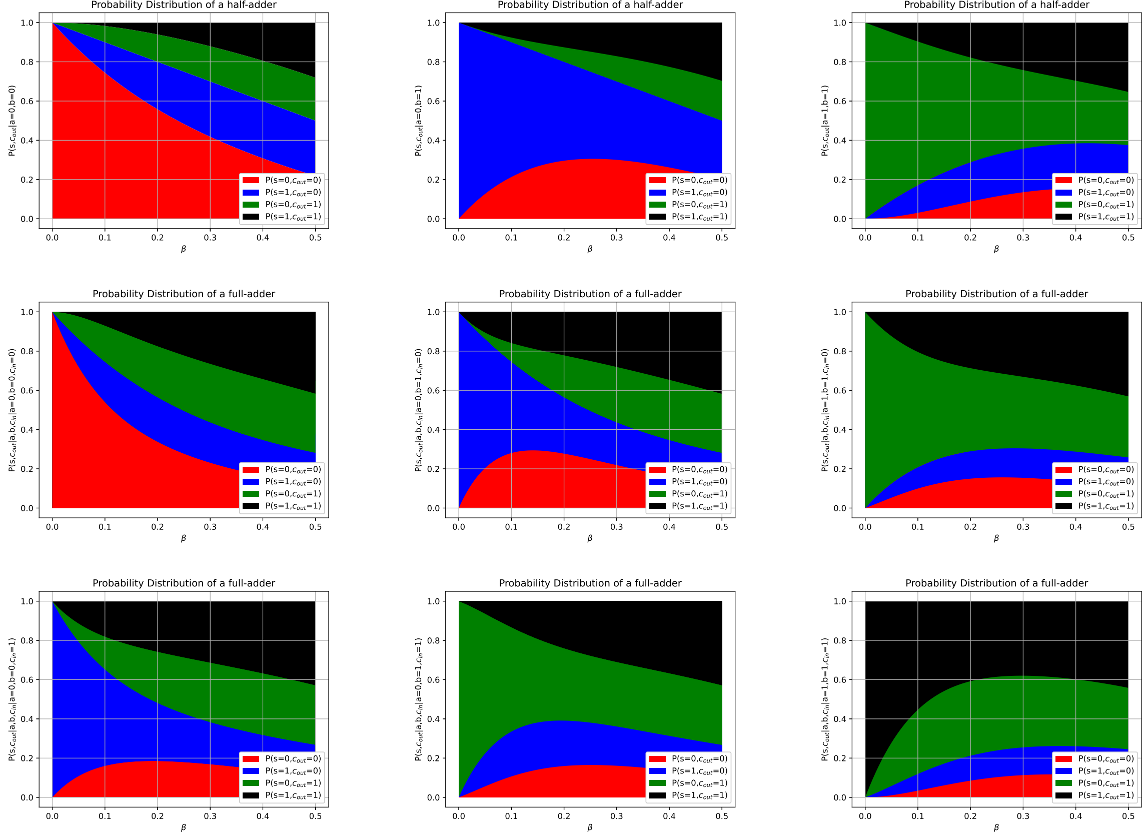


Figure 3: Probability distribution for $P(s, c_{\text{out}})$ for all distinct inputs for the noisy half-adder (top row) and the full-adder (middle & bottom row). The graphs show the change of the probability distributions over the four possible outcomes as stack-plots as β in (4) varies from 0 to $\frac{1}{2}$ and $\alpha = 0$; this case is equivalent to assuming there is only leakage of information due to low-voltage. In all graphs, red denotes $P(s = 0, c_{\text{out}} = 0)$, blue denotes $P(s = 1, c_{\text{out}} = 1)$, green denote $P(s = 0, c_{\text{out}} = 1)$ and black denotes $P(s = 1, c_{\text{out}} = 1)$. Note that longer computation paths bias towards more errors, for example the plots in the top and middle row should be identical because they have the same truth table.

Noisy 4-bit Adders Finally, using (5) and (6), we can compute the marginal distribution over the output $S := s_0 s_1 s_2 s_3 c^3_{\text{out}}$ of a 4-bit adder of A and B by summing over all possible values of c^0_{out}, c^1_{out} and c^2_{out}. We have

$$
\begin{aligned}
P(S|A, B) &= \sum_{c^0_{\text{out}} \in \mathbb{B}} \sum_{c^1_{\text{out}} \in \mathbb{B}} \sum_{c^2_{\text{out}} \in \mathbb{B}} P(s_0, s_1, s_2, s_3, c^3_{\text{out}}, c^0_{\text{out}}, c^1_{\text{out}}, c^2_{\text{out}} | A, B) \\
&= \sum_{c^0_{\text{out}} \in \mathbb{B}} \sum_{c^1_{\text{out}} \in \mathbb{B}} \sum_{c^2_{\text{out}} \in \mathbb{B}} P(s_0, c^0_{\text{out}} | a_0, b_0) P(s_1, c^1_{\text{out}} | a_1, b_1, c^0_{\text{out}}) \cdots P(s_3, c^3_{\text{out}} | a_3, b_3, c^2_{\text{out}}) \\
&= \sum_{c^0_{\text{out}} \in \mathbb{B}} \sum_{c^1_{\text{out}} \in \mathbb{B}} \sum_{c^2_{\text{out}} \in \mathbb{B}} q_{\text{ha}}(a_0, b_0, s_0, c^0_{\text{out}}) \cdot q_{\text{fa}}(a_1, b_1, c^0_{\text{out}}, s_1, c^1_{\text{out}}) \cdots q_{\text{fa}}(a_3, b_3, c^2_{\text{out}}, s_3, c^3_{\text{out}}) \,,
\end{aligned}
$$

where the second line follows from the (chain) directed graphical network structure of the 4-bit adder. Again, we use the pre-computed functions q_{ha} and q_{fa} for a fast computation of the output distribution of the 4-bit adder.

In Figure 3 we have shown how the distributions of the half-adder and full-adder vary as functions of β when $\alpha = 0$ (i.e., it is not possible that any of the input bits ever flip erroneously from 0 to 1). Similarly, in Figure 4 we have shown the distribution of the sum for the 4-bit adder for different values of β. One thing to note about this induced probability distribution is that, the longer the computation path, the more errors get introduced: for example, the top and middle row of Figure 3 should have identical distributions as their truth tables are identical but the middle row uses more logic gates as an additional half-adder is used. Also, while the variance of the noisy 4-bit adder is somewhat symmetric, but for more than 10% of bit-flip noise the right output is not always the most likely anymore.

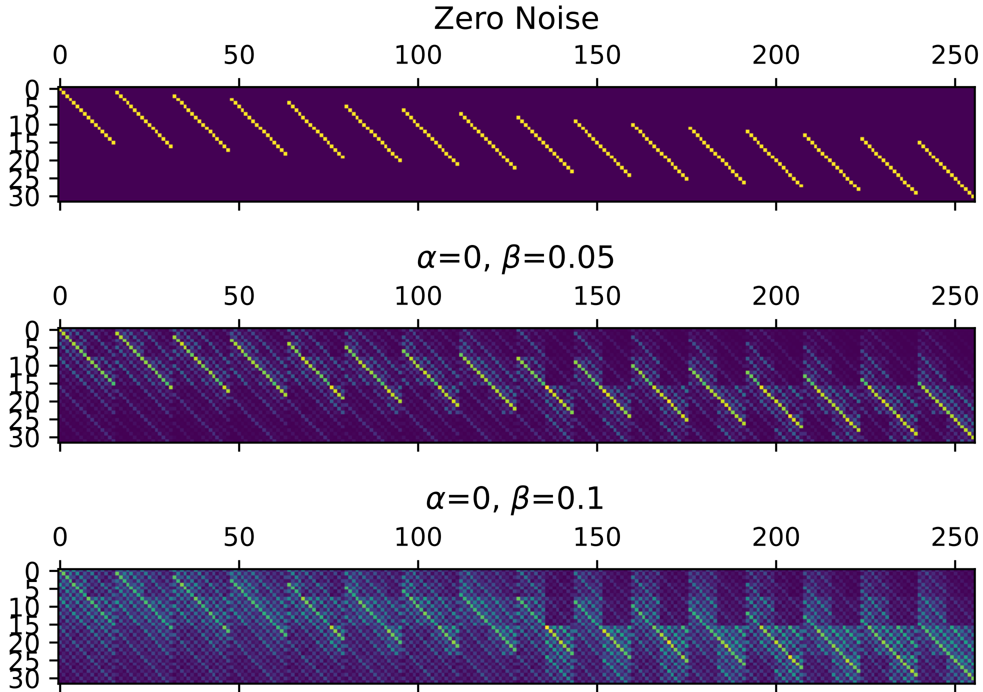


Figure 4: Probability distribution for $P(S|A,B)$ for all distinct 256 inputs for the noisy 4-bit adder for increasing values of the noise β in (4). In each of the 3 plots, S takes up to 31 values indexing the y-axis and (A,B) take up to 256 values (i.e., the bit-string $a_0a_1a_2a_3b_0b_1b_2b_3$ interpreted as an 8-byte number).

A CMOS Gates

References

[1] Carl Hamacher, Zvonko Vranesic, Safwat Zaky, and Naraig Manjikian. *Computer Organization and Embedded Systems*. McGraw Hill, sixth edition, 2012.

[2] Tom M. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[3] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[4] Kevin Patrick Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.

[5] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In Corinna Cortes, Neil D. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 3123—3131, 2015.

[6] Bernd Steinbach. *Recent Progress in the Boolean Domain*. Cambridge Scholars Publishing, 2014.

[7] Erhan Baturay Onural, Ismail Emir Yuksel, and Behzad Salami. Demonstrating reduced-voltage FPGA-based neural network acceleration for power-efficiency. In *International Conference on Field-Programmable Logic and Applications*, pages 371–371, 2020.

[8] Qiang Xu, Todd Mytkowicz, and Nam Sung Kim. Approximate computing: A survey. *IEEE Design and Test*, 33 (1):8–22, 2016.

[9] Hua Li, J. Mundy, W. Patterson, D. Kazazis, A. Zaslavsky, and R.I. Bahar. A model for soft errors in the subthreshold CMOS inverter. In *Proceedings of Workshop on System Effects of Logic Soft Errors*, 2006.

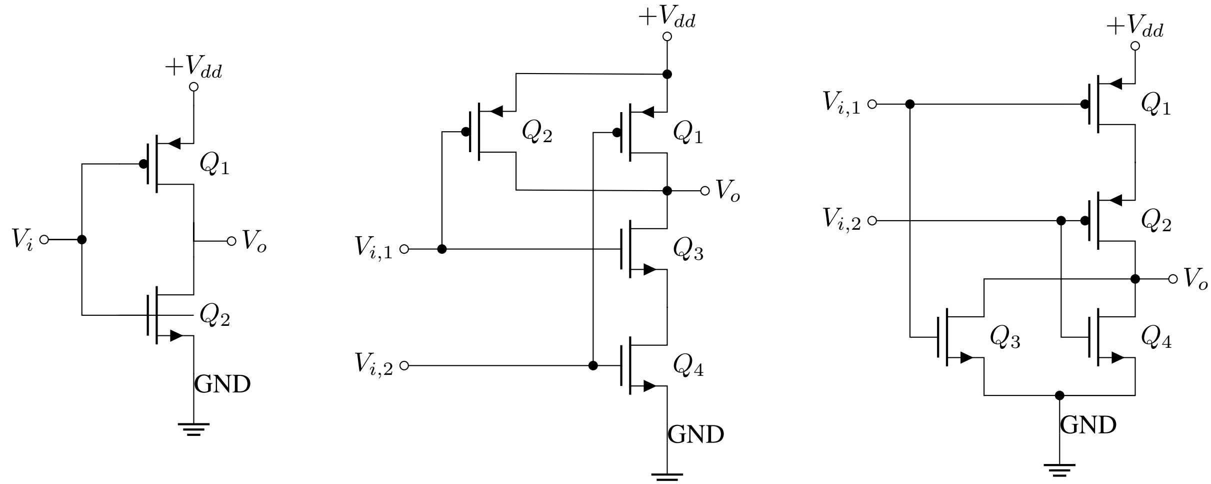


Figure 5: MOSFET circuit diagram for the basic logic functions NOT (left), NAND (middle) and NOR (right). The input voltage is at V_i (or $V_{i,1}$ and $V_{i,2}$) and the output voltage (corresponding to the output of the logic gate) is at the V_o node. The supply voltage that is provided to the MOSFETs is at node V_{dd} (typically $5V$); ground is indicated by GND. **(Left)** A NOT-gate reverses the input logic state by employing two series-connected MOSFETS, one n-type and one p-type. Applying $+V$ (logic 1) to the input (V_i), transistor Q_2 is "on," and transistor Q_1 remains "off" resulting in an output voltage (V_o) close to $0V$ (logic 0). **(Middle)** A two-input NAND gate where p-type MOSFETs Q_1 and Q_2 are connected in parallel and n-type MOSFETs Q_3 and Q_4 are connected in series between $+V$ and the output terminal V_o. Only when both Q_3 and Q_4 are "on" (logic 1 on both V_{i_1} and V_{i_2}), the output voltage V_o is close to $0V$ (logic 0) as it is shorted with GND. **(Right)** A two-input NOR gate where p-type MOSFETs Q_1 and Q_2 are connected in series and n-type MOSFETs Q_3 and Q_4 are connected in parallel between $+V$ and the output terminal V_o. Here, only when both Q_3 and Q_4 are "off" (logic 0 on both V_{i_1} and V_{i_2}), the output voltage V_o is close to $+V$ (logic 1) as there is no connection to GND.

[10] Frank R. Kschischang, Brendan J. Frey, and Hans-Andrea Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transaction on Information Theory*, 47(2):498–519, 2001.