Programming the ENIAC

By Brian L. Stuart

I. BACKGROUND

February 1946 saw the public unveiling of the ENIAC, an electronic computing system that had been in development at the University of Pennsylvania since 1943 under the leadership of J. Presper Eckert and John Mauchly. As with many R&D projects, what motivated the designers and what justified the funding were not exactly the same thing. The U.S. Army funded the project primarily because the teams at UPenn and at Aberdeen, MD, USA were not able to keep up with the need for computing artillery firing tables for the war effort. However, everyone involved knew that the real intent was to build a general purpose machine that could solve a wide variety of problems, not just artillery calculations [1], [2]. In fact, the first major application of the machine in late 1945 was related to bomb development at Los Alamos [3], [4]. Later in summer 1946, D. H. Lehmer used the ENIAC to carry out some number theoretic research related to prime numbers [5]–[9].

In this article, the reader is invited to take a trip back in time to experience the process of designing a program for the ENIAC, one of the earliest electronic general-purpose computers ever made.

Here, the reader is invited to take a trip back in time and to experience the process of designing a program for the ENIAC. This paper considers a simple number theoretic problem as an example of how the ENIAC could be used. It follows the task from problem statement, to the realization of the problem on the ENIAC, to running it on the simulator described in [10]. It should be noted that the problem here is not known to have been run on the actual ENIAC. Rather it serves as the type of simple tutorial problem found in many introductory programming manuals. Although most problems run on the ENIAC had the character of simulations solving differential equations, this problem is representative of those that demonstrated that the ENIAC's utility went well beyond the domain which justified its construction.

A. ENIAC Programmers

Before considering the problem, itself, it is important to understand some of the basic concepts of programming the ENIAC and how those techniques came to be. For that, we must go back to early World War II.

Rapid development of artillery guns, shells, and propellants put a huge demand on the Army's testing and computational resources at Aberdeen, MD, USA. The Moore School of Engineering at the University of Pennsylvania was contracted to assist with the computations, using differential analyzers and using large staffs of human computers with desk calculators at both locations. To meet their staffing needs, the Moore School recruited as many civilians who had studied mathematics in college as they could. Because the majority of the young men were assigned to military service, the bulk of the people recruited by the Moore School were the women who had recently graduated with mathematics degrees. LeAnn Erickson of Temple University has produced a documentary chronicling the story of these women and their contributions to the war effort [11].

As the development and construction of the ENIAC neared completion, six of the computers at the Moore School were selected to become the operational staff for the new machine. These six women were Fran Bilas (later Spence), Jean Jennings (later Bartik), Ruth Lichterman (later Teitelbaum), Kay McNulty (later Mauchly, later Antonelli), Betty Snyder (later Holberton), and Marlyn Wescoff (later Meltzer). Their anticipated role was more akin to the later notion of operator than programmer. The expectation

Digital Object Identifier 10.1109/JPROC.2018.2843998

was that a mathematician, scientist, or engineer would develop equations to solve, and these operators would assist them in developing and deploying those equations on the ENIAC. It was only after people began using the machine that it became clear that the process of mapping the problem to the ENIAC and designing the sequencing of operations was more involved than had been expected, giving birth to programming as we know it. The ENIAC Programmers Project, founded by Kathy Kleiman, has produced a documentary about these early programmers [12].

Of course, since there had never been a machine like the ENIAC before, there were no manuals or courses to teach them how to operate it. Instead, they were given access to the drawings for the machine and to the engineers who had designed it. With these resources, the mathematicians learned how the machine worked, how it could be used to solve problems, and how to use its limited resources effectively. There were a number of other important members of the ENIAC team that should be mentioned in this context. The first of these was Adele Goldstine. She wrote the first part of the ENIAC Technical Manual [13] which was part of the documentation provided to the Army by the Moore School. In it, she captured the knowledge and techniques for using the machine. The second part of the Technical Manual [14] was written by Harry Huskey, who along with John Mauchly, was one of the key engineers consulted by the new programmers. Huskey also coauthored the Operating Manual with Arthur Burkes [15].

B. ENIAC Programming Technique

Although we can recognize in the ENIAC a variety of conventional programming techniques, such as loops and conditional branching, the ENIAC is a very different kind of machine when compared to current computer architectures. As might be expected, these architectural differences lead to some differences in the approach needed for programming.

1) Dataflow: The most striking difference between the ENIAC and conventional computers is that in its original use, the ENIAC does not follow a sequential instruction flow. Instead, it operates much more like a dataflow machine. The next operation is not determined by physical proximity in the form of the next punched card, the next row on punched tape, or the next address in memory. Instead, the next operation is determined by a signal sent on the completion of the last operation. That signal activates another operation determined by the wiring of the machine. Programming a system like this is better done when thinking about the set of operations spacially in more than one dimension, rather than strictly linearly.

The dataflow character of the ENIAC is apparent not only in the way the programmer thinks about control flow, but also in the way in which programming is deployed on the system. Programming for the ENIAC is not represented in any symbolic form analogous to modern programming. Instead, it is represented in the form of wiring diagrams and 2-D tables that show the settings for each unit and the interconnections between units. Deploying a problem configuration involves setting switches on each unit to identify the operations, and connecting patch cables between units and trunks that carry signals around the whole machine.

2) Parallelism: One of the most powerful aspects of the ENIAC is that a given signal can initiate more than a single operation in a given time step. It is often quoted that the ENIAC is capable of 5000 additions per second. Because it has a 100-kHz clock and each addition takes 20 cycles, each addition does take 1/5000 of a second. However, numerous additions can be performed in parallel. In reality, the ENIAC can perform well over 5000 additions per second if the problem allows this kind of parallelism. As with the parallelism found in today's systems, making the most of it in the ENIAC is a significant challenge for the programmer.

3) Looping and Branching: From the beginning of the project, it was clear that simple straight-line sequences of operations were not sufficiently general to handle a wide variety of applications. This recognition motivated the inclusion of the master programmer in the ENIAC. This unit provided a flexible mechanism for nested loops and sequences of loops.

At first glance, the loops provided by the master programmer would seem to be limited to simple counted loops, the limits of which are what we today would call hard coded. However, there are inputs to the master programmer that can be used to jump to the next loop in the sequence. By feeding these inputs from control signals generated as a result of computations, a number of conditional branching techniques become possible. For example, if a digit line containing a value in the interval [0, 5] is connected to the stepper direct input, then one of six possible courses of action can be selected. If the sign of a number is transformed into a single pulse using a dummy program (discussed below), then a master programmer stepper can select one of two actions according to whether the number is negative or nonnegative. Similarly, a digit passed through a dummy program will cause the stepper to select a course of action according to whether the digit was zero or nonzero. Finally, if both the sign of a value and the sign of its complement are transmitted concurrently (by way of both the accumulator's additive and subtractive outputs) to dummy programs, the control outputs of the two dummy programs can be used to select between a pair of actions without using a master programmer stepper. More details of the conditional techniques (referred to as magnitude discrimination in the technical manual) are discussed in Sections II-D and III-G.

4) Debugging: One of the interesting things included in the ENIAC

Fig. 1. *Portable control station.*

design is a portable control station, pictured in Fig. 1. It is a handheld unit that duplicates a number of the controls found on the initiating and cycling units and that has a cable long enough to walk around the whole machine. The user is able to select among continuous clocking, single addition clocking, and single cycle clocking. Four push buttons give the user access to the clear, initiate, read, and pulse controls. The first three are for starting a computation, with the clear button clearing all accumulators to zero, the initiate button triggering a pulse on the Io control output of the initiating unit, and the read button causing one punched card to be read.

The pulse button is used in conjunction with the single addition and single cycle clocking modes. When pressed in the single addition mode, the pulse button causes one addition time's worth of clocking (20 pulse times or 200 μs) to be issued by the cycling unit. In single pulse mode, only a single pulse time (10 μs) of clocking is issued. With the portable control station, the designers of the ENIAC anticipated the need for debugging facilities and provided a single stepping mechanism with two different levels of stepping granularity.

As we now know today, there are three places where errors can arise: the algorithm, the realization of the algorithm (coding), and the hardware. At least two of these were in the minds of the designers. In the Technical Manual [13], Goldstine writes: "One of the chief techniques for localizing errors in either the machine or the set-up of the machine is to operate the ENIAC in the one addition time mode or in the one pulse time mode." The extent to which the ENIAC's designers expected the design of algorithms to be error-prone and require debugging is not clear, but the prescience illustrated by the inclusion of single-stepping is striking. Furthermore, very early, the ENIAC team anticipated another of our fundamental debugging techniques [13]:

> Frequently, it is more convenient to proceed through a portion of the computation with the ENIAC operating in its normal or continuous mode and then to switch to 1 addition time or 1 pulse time operation than it is to progress through the entire computation non-continuously. This may be arranged by disconnecting the program cable which delivers the pulse used to initiate the programs which are to be examined non-continuously. We call this point where the program cable is removed a break point.

C. Conversion to an Instruction Set Processor

The capabilities of the ENIAC make it general enough to solve any computational problem within the limits of its very small working memory. In retrospect, it comes as no surprise that one computational problem it can solve is that of being a central processing unit. By deploying a setup that operates as a CPU, the ENIAC can then be used without the rewiring and switch setting necessary in its original form.

From April 1948 on, the ENIAC operated in exactly that way. The machine configuration read instructions from the function tables, interpreted them, and executed them just as any other CPU does. Neukom discusses the deployment of this configuration in [16]. Recently, Haigh *et al.* have produced a thorough account [4] of this conversion and of the use of the machine in that configuration. Because this made the machine much easier to program, the ENIAC was used in this way until it was retired in 1955. For this paper, however, we are focused on programming the ENIAC in the pre-1948 style.

II. EXAMPLE

To gain a better understanding of the process of programming the ENIAC, we focus on an example. Although a wide variety of problems were run on the ENIAC, we consider a simple number theoretic problem because it requires little explanation and it illustrates some of the more advanced ENIAC techniques.

A. Statement

The problem chosen for this example is that of finding all three-digit numbers that are equal to the sum of the cubes of the digits. Formally, this can be described as enumerating the set

$$C = \{n \,|\, n = 100x + 10y + z = x^3 + y^3 + z^3\}$$

where x, y, and z are integers such that $1 \leq x \leq 9$ and $0 \leq y, z \leq 9$. Take for example, the number 123: $1^3 + 2^3 + 3^3 = 1 + 8 + 27 = 36 \neq 123$. Therefore, 123 is not one of the numbers we are looking for. On the

for $n = 100$ to 999
  $x = \lfloor \frac{n}{100} \rfloor$
  $y = \lfloor \frac{n}{10} \rfloor \bmod 10$
  $z = n \bmod 10$
  if $n = x^3 + y^3 + z^3$ **print** n

Fig. 2. *First algorithm for finding C.*

other hand, if we consider 153: $1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$. So 153 is a member of C and one of the numbers we want to find.

B. Algorithm

There are two basic strategies that we take if programming this problem in a typical programming language. In the first approach, we let n count from 100 through 999, extract the digits of n, cube them, and test to see if the sum equals n. This approach can be summed up in the algorithm shown in Fig. 2.

A second approach to this problem is to enumerate all of the combinations of x, y, and z and test each combination to determine if $100x + 10y + z = x^3 + y^3 + z^3$. A simplistic expression of this approach is shown in Fig. 3. This algorithm is simplistic because subexpressions, such as $100x$ and y^3, are evaluated far more times than is necessary.

For solving this problem on the ENIAC, we adopt something of a hybrid approach, and attempt not to be naive about the efficiency of subexpression evaluation. Fig. 4 shows the basic algorithm.

C. Computing Cubes

With the basic structure in place, the next question to answer is how we compute the cubes of x, y, and z. Perhaps the most obvious approach would be to use the high-speed multiplier found in the ENIAC. It turns out, however, that despite the name of the unit, this is not the fastest way to compute the cubes. Each cubing requires two multiplications, each of which takes $p + 4$ addition times. By using shifting adapter cables, we can ensure that at most two places are used in the multiplier, resulting in a total of six addition times for each multiplication, or 12 addition times for the full cubing.

A second alternative would be using a function table to store a table of cubes. Table lookups take only five addition times. However, unless all three function tables are allocated, one for each digit, there will be one more addition time necessary to copy the relevant digit into the argument accumulator, resulting in six total addition times.

The approach for computing cubes in this example takes advantage of the fact that the cubes are computed in sequence. By recognizing that $(x + 1)^3 = x^3 + 3x^2 + 3x + 1$ and taking advantage of the parallelism in the ENIAC, we can perform this cubing in seven addition times.

In practice, the choice of technique for cubing depends on the problem at hand. In particular, are the function tables needed for other uses in the problem? Are there enough accumulators for the third approach, or are they needed for other parts of the problem? Ultimately, most decisions of detail on the ENIAC come down to how best to allocate the limited resources of the machine.

D. Conditional Branching

In addition to the looping structures provided by the master programmer, the ENIAC was designed with the idea of conditional branching in mind. In Goldstine's Part I of the technical manual, there are several examples of conditional techniques. In one example, the tenth digit is used to generate a control pulse that allows a computation to continue only as long as a particular value does not exceed 8,999,999,999. In another, the sign is used to select one of two computations based on whether a value is negative or nonnegative. These kind of less-than or greater-than comparisons arise naturally in physical simulations, which account for the most common types of applications run on the ENIAC. There is also an illustration of how the value of a digit can be used to select among up to ten possible actions in a way that's reminiscent of the computed-GOTO in FORTRAN or the switch statement in C.

In this problem, the comparison needed is somewhat different from those examples. Here we need to compare two variables for equality. The ENIAC does not have a mechanism to do this directly. However, we can determine if the difference between two values is negative or nonnegative. In other words, we can test if $a \geq b$ by subtracting b from a and using the sign to select one of two operations. Recognizing that $a = b$ iff $a \geq b$ and $b \geq a$, we can test for equality by subtracting both ways and determining if either result is negative. For the problem at hand, then, we assign $m \leftarrow x^3 + y^3 + z^3$ and compute both $n - m$ and $m - n$. If both results are nonnegative, then we print the value of n; otherwise, we continue with the body of the inner loop.

III. IMPLEMENTATION

Putting all the pieces together, the final algorithm used here is shown in Fig. 5. There are several things worth noting in this algorithm. First, the symbol $\overset{+}{\leftarrow}$ denotes the addition of the value on the right to that on the left. This represents the basic operation of the ENIAC accumulators. In cases where the original value of the accumulator is known to be 0 or where the accumulator is being cleared to 0, we use the $\leftarrow$ symbol instead. The second thing to note at this point is that the order of operations has moved a bit relative to that shown in Fig. 4. In particular, the calculations of the digit cubes have been moved to the bottoms of the loops. The reason for this is that by starting with the y and z values cleared, we already have the correct values for $y = 0$ and $z = 0$. Thus, it is natural to carry out the $(y+1)^3 = y^3 + 3y^2 + 3y + 1$ operation after using the existing values of the

```
for x = 1 to 9
  for y = 0 to 9
    for z = 0 to 9
      if 100x + 10y + z = x^3 + y^3 + z^3
        print 100x + 10y + z
```

Fig. 3. *Second algorithm for finding C.*

```
n ← 100
for x = 1 to 9
  x3 ← x^3
  for y = 0 to 9
    y3 ← y^3
    for z = 0 to 9
      z3 ← z^3
      if n = x3 + y3 + z3 print n
      n ← n + 1
```

Fig. 4. *First ENIAC algorithm for C.*

powers of y. Finally, note that the powers of x, y, and z are not computational operations in this algorithm. For example, z^2 does not mean squaring z; it is the name of a variable that holds the square of z. Other details are discussed in the following sections.

A. Programming Diagrams

Part 1 of the Technical Manual uses two representations of the computations performed on the ENIAC. One of these is called the setup table. Such a table is organized as a 2-D grid where each column represents a single unit of the ENIAC, and each row represents one addition time. Each cell of the table shows the configuration activated in the specific unit in the specific addition time. Because these tables tend to be large foldout sheets, we do not include one for this example.

The second representation is called the setup diagram. It schematically shows the wiring and the switch settings for a given machine configuration. Figs. 6–8 show the full configuration of the ENIAC for solving this problem.

The horizontal lines at the top of each figure are the data trunks that carry data from one unit to another. Those on the bottom of each figure are the control trunk lines, grouped into trays. Units are shown as rectangles between the data and control trunks. In each unit, the layout of terminals and switches is organized in the same way as those in the actual machine as illustrated in the Operating Manual [15].

For each accumulator, along the top of the unit, there are two groups of rectangles that represent the data terminals. The group of five to the left are the five input terminals designated α through ϵ from left to right. To the right of those are the two output terminals, A and S. Control terminals are located at the bottom of the unit in two rows. The upper row is the set of four input terminals for programs 1–4. The bottom row includes the 16 terminals which comprise the input and output controls for programs 5–12. Similarly, the switch settings are shown in boxes between the accumulator number and the control terminals, with programs 1–4 forming an upper row and programs 5–12 a lower row.

For program 5 on accumulator 3, the operation switch is set to α as shown in the upper of the three boxes. The empty circle in that box indicates that the carry/correct switch is set to 0. The middle box (found only on programs 5–12) is the repeat count and is set to 3 for program 5. The lower box for each program is available to the programmer for notation such as the computational step where this program is activated. Control terminal 5i is shown to be connected to control line 2–1 and 5o to 2–2. The effect of these connections is that a control pulse on line 2–1 causes program 5 on accumulator 3 to activate. During the next three addition times, any data received on the α input is added to the value in the accumulator. At the end of those three addition times, the accumulator sends a pulse on line 2–2.

Accumulators 1 and 2 include an unusual element in the diagram. In these two cases, the lines leaving the A output terminals are marked with a * and a † which are described above the diagram as "MP to 1–10" and "MP to 1–11." This refers to the use of an adapter cable with a data terminal connector on one end and one or more control terminal connectors on the other. Here for both accumulators, the MP (minus-plus, or sign) signal from the A terminal is connected to the specified control line, which is in turn connected to the 12i input of accumulator 3 or 4. In other words, the sign is used as a control signal to activate program 12 of an associated accumulator when the number is negative.

The remainder of this section discusses the details of the configuration in reference to the setup diagram shown in Figs. 6–8.

B. Initialization

Step 1–1 sets up the fact that this algorithm starts at $n = 100$ because numbers in the range 0–99 are not considered three-digit numbers for the purpose of this problem. Along with setting n to 100, x, x^2, and x^3 all need to be set to 1. The constant transmitter sends the 100 necessary to load n when triggered by control signal 1–1. Both the constant transmitter output and the n accumulator (accumulator 13) α input are connected to data trunk 1 to effect the actual data transmission. At the same time, 1 is added to each of accumulators 9, 10, and 11 (containing x^3, x^2, and x, respectively) using the incrementing technique described below.

```
1-1  n ← 100; x^3 ← 1; x^2 ← 1; x ← 1
1-2  for 9 steps
1-3    for 10 steps
1-4      for 10 steps
3-1        m ← x^3
3-2        m ⟵+ y^3
3-3        m ⟵+ z^3
3-4        l ← −m
3-5        m ⟵+ −n; l ⟵+ n
3-6        transmit signs of l and m; l ← 0; m ← 0
3-7        if both signs are P print n
2-1        z^3 ⟵+ 3z^2
2-2        z^3 ⟵+ 3z; z^2 ⟵+ 2z
2-3        z^3 ⟵+ 1; z^2 ⟵+ 1; z ⟵+ 1; n ⟵+ 1
2-4      y^3 ⟵+ 3y^2
2-5      y^3 ⟵+ 3y; y^2 ⟵+ 2y
2-6      y^3 ⟵+ 1; y^2 ⟵+ 1; y ⟵+ 1; z^3 ← 0; z^2 ← 0; z ← 0
2-7    x^3 ⟵+ 3x^2
2-8    x^3 ⟵+ 3x; x^2 ⟵+ 2x
2-9    x^3 ⟵+ 1; x^2 ⟵+ 1; x ⟵+ 1; y^3 ← 0; y^2 ← 0; y ← 0
```

Fig. 5. *Full ENIAC algorithm for C.*

C. Data Flow

It turns out that although there are numerous different data transmissions from various accumulators to others, in almost all cases, there is only a single value transmitted at a time. Because of this, a single data trunk can be reused for all of these cases. The only exception occurs during the comparison operation when both n and $-n$ are transmitted simultaneously. Therefore, the only data trunks used for this problem are 1 and 2.

D. Control Signals

This algorithm is expressed in a level of detail where each line of the algorithm corresponds to a single step in the control flow. We have labeled each step with a pair of numbers. All of the statements related to initialization and looping have a 1 as the first of the pair. Those that are part of cube computations begin with 2, and those related to the conditional start with 3. This nomenclature is chosen because it directly maps onto the control trunks on the ENIAC. Consider, for example, the operation labeled 2–1. A control signal on control trunk line 2–1 initiates a program on the accumulator holding z^3 (accumulator 3) to receive three times, and one on the accumulator holding z^2 (accumulator 4) to transmit three times. The effect of this is to add $3z^2$ to z^3. At the completion of this operation, the output control pulse is fed into control trunk line 2–2 from the 5o output of accumulator 3.

This expression of the algorithm also illustrates the parallelism available in the ENIAC. Multiple operations on a single line separated by semicolons are performed simultaneously. Notice that care is taken to ensure that each accumulator has no more than one active program at a time.

E. Accumulator Settings

Returning to the statement labeled 2–1, with z^3 assigned to accumulator 3 and z^2 assigned to accumulator 4, both use program 5 for this step. As described above, control trunk line 2–1 is then connected to the 5i input of both accumulators. For accumulator 3, the operation switch for program 5 is set to α, the repeat switch is set to 3, and the clear/correct switch is set to 0. Then, for accumulator 4, the operation switch is set to A, the repeat switch is set to 3, and the clear/correct switch is set to 0. We can abbreviate these two accumulator programs as $\alpha 03$ and A03, respectively. These switch settings and the control and data connections result in the operation $z^3 \overset{+}{\leftarrow} 3z^2$ as discussed above.

The next step of the algorithm is particularly interesting. The 5o signal of accumulator 3 feeds the 2–2 control line which in turn triggers program 6 on accumulators 3, 4, and 5. (Accumulator 5 holds the value of z.) As with program 5, on accumulator 3, program 6 is set to $\alpha 03$ Similarly, on accumulator 5, program 6 is set for A03. However, on accumulator 4, program 6 is set for $\alpha 02$. The net effect of this is that accumulator 5 transmits the value of z three times, and during that same period, accumulator 3 receives the value three times, effectively adding $3z$ to z^3. However, accumulator 4 only receives during the first two of the transmissions of z, ignoring the third. This causes $2z$ to be added to z^2 during the same time period that $3z$ is being added to z^3. In steps 2–5 and 2–7, the same technique is used for y and x.

Clearing and incrementing accumulators as in steps 2–3, 2–6, and 2–9 are also worth discussing. To clear an accumulator, we make use of the clear/correct switch. When set to C on a program whose operation is one of 0, A, S, or AS, the accumulator is cleared at the end of the operation. For the clearing operations in steps 2–6 and 2–9, program 8 is used on all the relevant accumulators with the settings 0C1.

As with cubing, there are several ways in which 1 can be added to an accumulator. Coming from the perspective of modern programming, the most natural approach would be to use the constant transmitter to send 1 or to load the value 1 into an accumulator and add from one of those to all the accumulators that need to be incremented. However, there is another feature of the ENIAC accumulators that we can use for this. If the clear/correct switch is set to C on a program set for input, then an additional pulse is added to the accumulator. The primary purpose of this feature is to add the extra 1 necessary when forming a tens complement number, but we can use it any time we want to add 1. So for all of the accumulators we need to increment, we activate program 7 where we have set

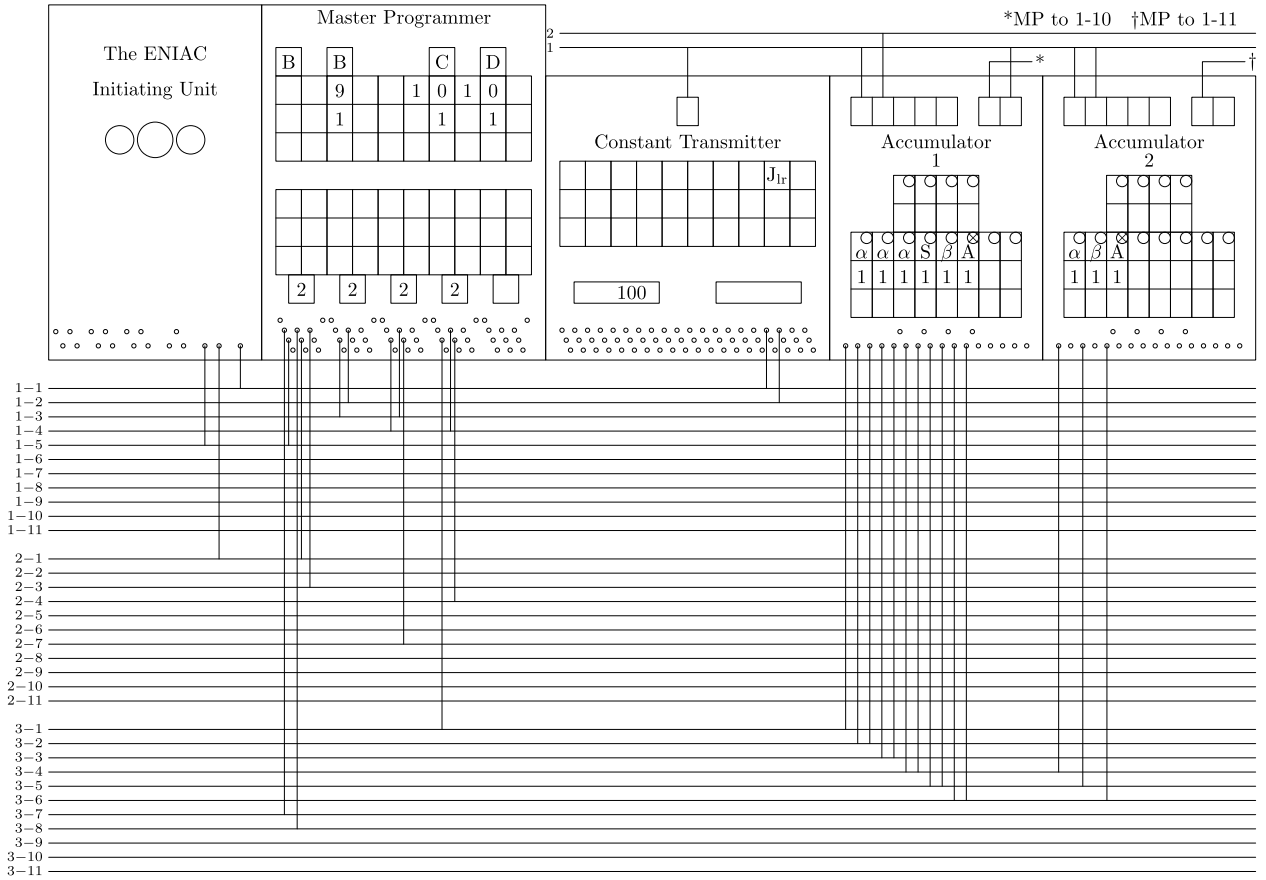

Fig. 6. *Initiating unit, master programmer, constant transmitter, and accumulators 1 and 2.*

the operation to ϵ, the count to 1, and the clear/correct switch to C (ϵC1). Because there is nothing connected to the ϵ input, the effect of this program is to add 0+1 to the accumulator.

F. Master Programmer Settings

The master programmer in the ENIAC has ten six-position steppers and 20 decade counters. Each stepper can be configured to implement a sequence of up to six loops. In the example here, there are three nested loops, however. To realize this structure on the ENIAC, we use three of the steppers, each configured to perform a single loop. In the implementation given here, we use the steppers, B, C, and D. In all three cases, the stage 1 output is used to trigger the body of the loop, and the stage 2 output is used to transfer control out of the loop.

Because the values of the decade counters in the master programmer are not directly accessible, it is best to think of the loop configurations in terms of simply the number of iterations, rather than in terms of starting and ending points. Here, stepper B is used to cycle through the hundreds digits, and since they go from 1 through 9, this stepper is configured for nine iterations. Steppers C and D cycle through the tens and ones digits, respectively, and each is set for ten iterations. The actual starting and ending values for x, y, and z are determined by the initialization and clearing mechanisms described above.

Looking closely at the inner-most loop, the looping mechanism is triggered by a signal on line 1–4. The body of the loop is initiated by control line 3–1, and the last operation of the body is initiated by control line 2–3. Therefore, there are two different outputs that drive control line 1–4. The first of these is the stage 1 output of the C stepper (the signal labeled C_1o in the operating manual). This signal reflects the fact that the inner-most loop is the first step in the body of the middle loop. The second signal is the output indicating the end of the operation triggered by signal 2–3. This is the transfer of control from the end of the loop body back to the increment and test of the loop. The stage 1 output of this stepper (D_1o) drives control signal 3–1 to initiate the body of the loop, and the stage 2 output (D_2o) drives line 2–4 to transfer control to the statement after the loop upon loop completion. The other two loop structures are configured similarly, with one exception. Because there is nothing left to do at the end of the outer-most loop, the B_2o line is not connected to anything. When stepper B reaches stage 2, the control signal does not drive any other operation and the computation stops.

G. Conditional Branching

A fourth stepper (A) is used to implement conditional branching. In Fig. 6, the left-most B in the master programmer indicates that decade

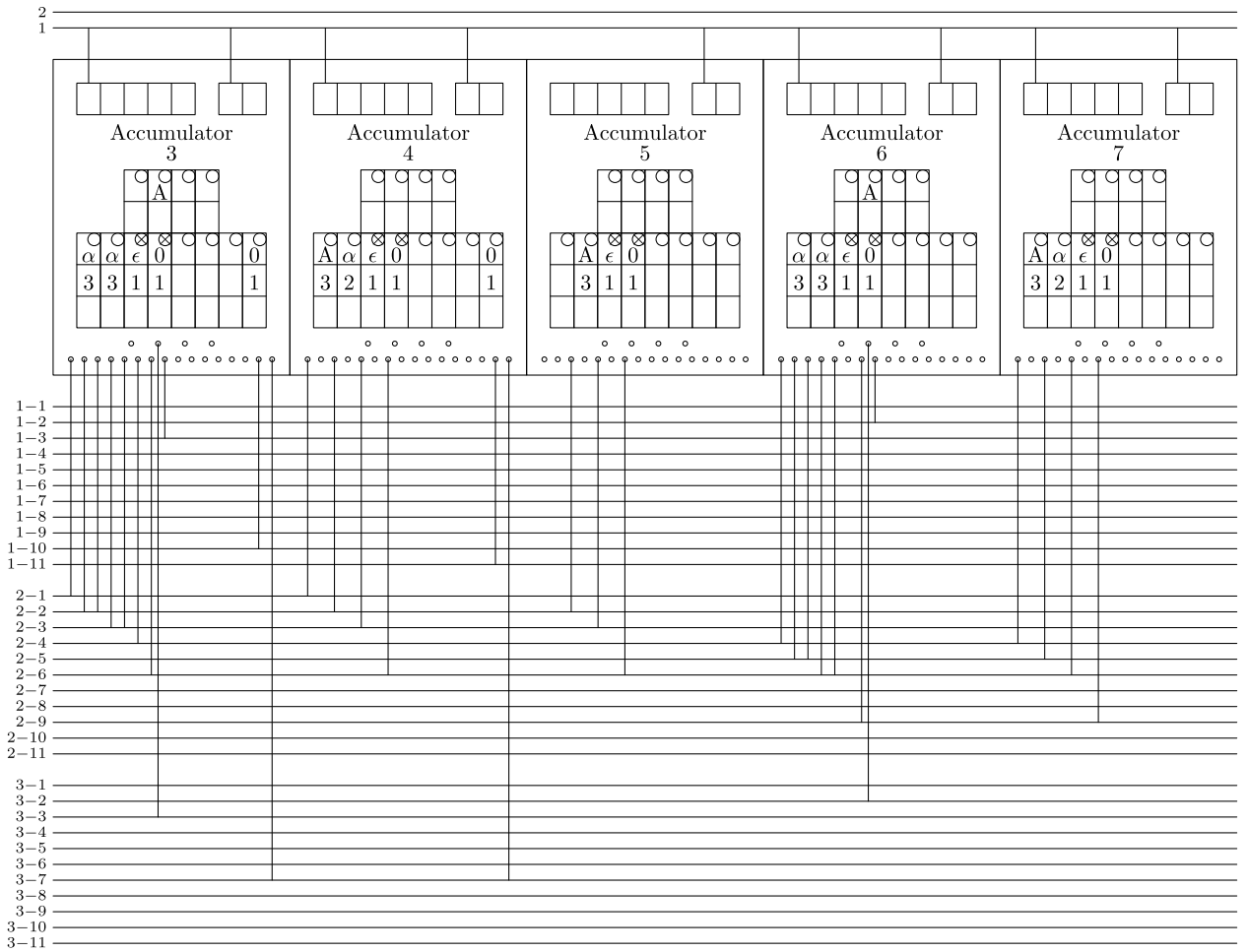

Fig. 7. *Accumulators 3–7.*

20 is assigned to stepper B, leaving no decades assigned to stepper A. By configuring this stepper to have no decade counters, the stepper direct input can be used to select among up to six possible control paths. Here we use only two of the possible stages to handle the **if** in step 3–7 of the algorithm. After computing $m-n$ and $n-m$, the values of the two accumulators are then transmitted through adapters that extract the signs. These adapters are indicated in Fig. 6 by the asterisk and dagger on the lines connected to the A outputs of accumulators 1 and 2. They are connected to control lines 1–10 and 1–11, respectively. The pulses from the sign "digits" are then fed into the 12i program control inputs on accumulators 3 and 4. (Accumulators 3 and 4 are used here for clarity as they are otherwise idle during the addition time when the differences are being transmitted.) Each of these programs is set so that the operation is 0, the repeat count is 1, and the clear/correct switch is 0 (001). This configuration is known as a dummy program and in this case is used to convert data pulses into a properly timed control pulse. The effect of this arrangement is that if $m-n<0$, then accumulator 3 will emit a control pulse on its 12o line. Similarly if $n-m<0$, then accumulator 4 will emit a control pulse on its 12o line. Both of these 12o lines drive the control trunk line 3–7, creating a wired-OR. (A similar wired-OR technique can be seen in [8] and [9].) Summing up, if $m\neq n$, then either $m-n<0$ or $n-m<0$, and a control pulse appears on line 3–7. If, on the other hand, $m=n$, then neither will be negative, and no pulse will appear on line 3–7.

Control line 3–7 drives the stepper direct input on stepper A. A pulse on this input causes the stepper to immediately advance one stage, regardless of the values on the decade counters. At the same time, line 3–8 delivers a pulse to the stepper input. If there is no stepper direct input when the stepper input arrives, then a control pulse is emitted on the A_1o line. However, if both pulses arrive, then the output pulse is transmitted on the A_2o line. A_1o is connected to the Pi line of the initiating unit via control trunk line 1–5. Thus, when there is no stepper direct input pulse on line 3–7, then a punch operation is initiated. If there is a stepper direct pulse or when the punch operation is completed, a pulse is sent on the 2–1 control line to start the cubing operation on z.

Before leaving the details of conditional branching, the timing of control signal 3–8 is worth noting. Suppose that the control signal to initiate transmission of n for computing $n-m$ and

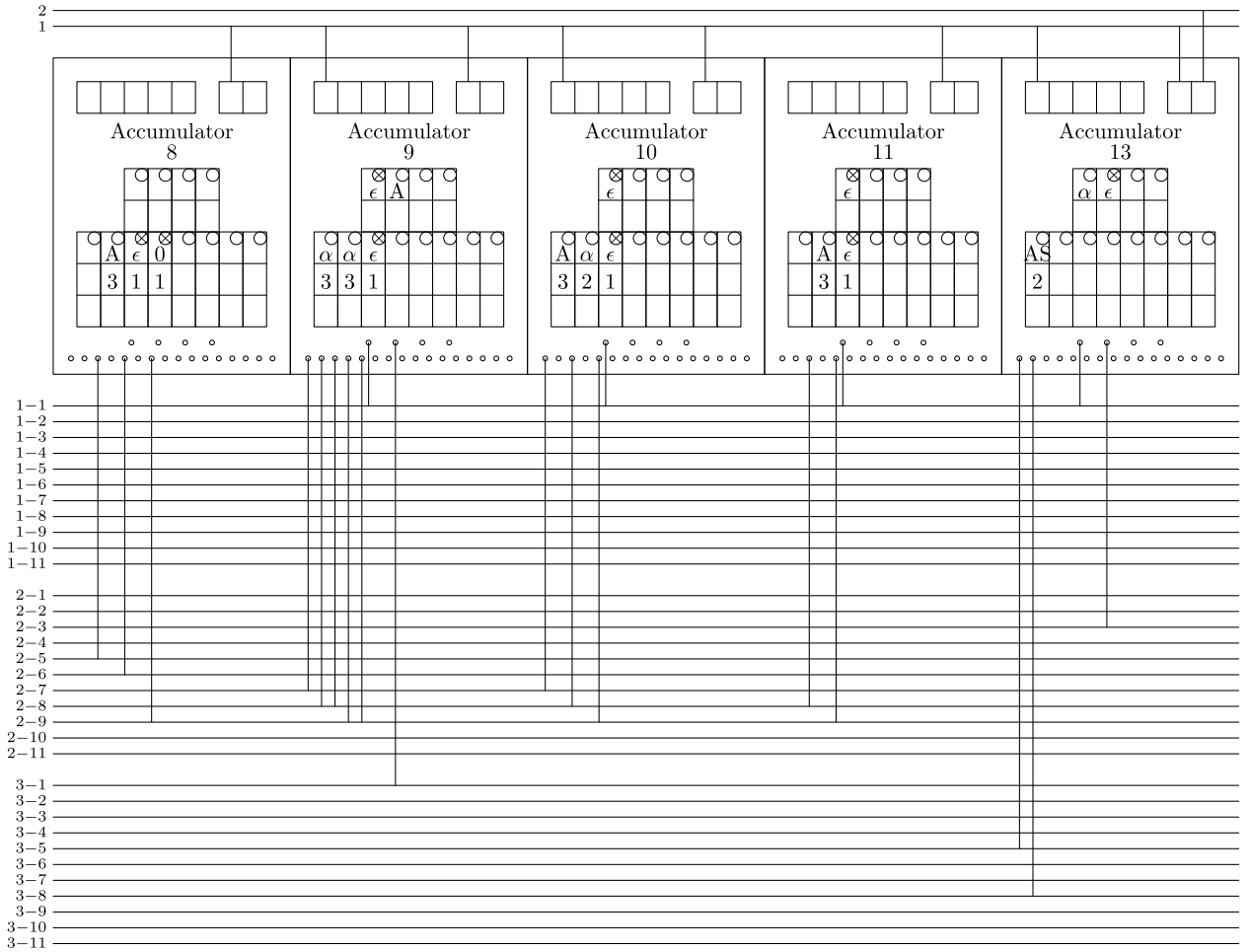

Fig. 8. *Accumulators 8–11 and 13.*

$m - n$ occurs at central programming pulse (CPP) time of addition time t. The differences are then calculated as n is being transmitted during addition time $t + 1$ and at the end of that addition time, the control signal is sent to cause the differences to be transmitted through the adapters. This means the signs of the differences are presented to the 12i inputs on both accumulators during addition time $t + 2$, which then implies that the potential signal on 3–7 occurs at CPP time of $t + 2$. Since the stepper direct input and stepper input can occur simultaneously, the earliest we can transmit the signal on 3–8 is $t+2$. To accomplish that, we set a repeat count of two on the transmission of n and let the output control signal from that program drive 3–8. It is natural to ask whether there is any problem with transmitting n two times when it is only needed once, and it turns out the answer is no. Because the accumulators receiving n only receive once, just n, rather than $2n$, is involved in the differences. Furthermore, neither data trunks 1 nor 2 are used for anything else during the following addition time, meaning that the extra transmission will not interfere with other transmissions. Although it seems a waste of effort to transmit n more times than needed just to get a delay, it saves the need for an extra dummy program. We find that this technique is quite common in ENIAC programming.

IV. SIMULATION CONFIGURATION FILE

Each of the switch settings and each of the vertical lines in the diagrams are specified by a line of text in the configuration file. The portion of the configuration file that correspond to the switch settings and control signals for program 5 on accumulator 3 are as follows:

```
p 2-1 a3.5i
p a3.5o 2-2
s a3.op5 α
s a3.rp5 3
s a3.cc5 0
```

This format can be thought of as a sort of ENIAC object code, where the "p" instruction specifies plugging a patch cable, and the "s" instruction specifies setting a switch. Each of these instructions takes two operands. In the case of the switching instruction, the two operands are a switch and a value. In the case of the plugging instruction, the two operands are two terminals. Unlike real patch cables, the plugging operation in the simulator is directional. In this example, signals travel from terminal `2-1` to terminal `a3.5i`. This is not a significant limitation since each terminal on a unit of the ENIAC is itself directional.

Fig. 9. *Simulation run snapshot.*

All together (including comments and blank lines for organization), the configuration file for this program contains 313 lines. Of that, 113 lines are patch cable specifications and 163 lines are switch settings.

V. SIMULATION RESULTS

Fig. 9 shows a snapshot of the simulator graphical interface during the simulation run. That screen capture was taken after all four of the values were printed on the output. From the image, it can be seen that the four three-digit numbers that are equal to the sum of the cubes of the digits are 153, 370, 371, and 407. On the author's laptop, the full simulation run takes approximately 22 s.

VI. ALTERNATIVE TECHNIQUES

The presentation here is focused on its role as a tutorial example. However, realistic applications often require more complex design decisions, especially with regard to the limited resources of the machine. The remainder of this section considers some possible alternative design choices that could be made.

A. Conserving Accumulators

The approach presented in this paper uses 12 of the 20 available accumulators. Because the accumulators are the only volatile memory on the machine, most applications require especially careful use of the accumulators. In the configuration shown here, we can easily save three accumulators, because accumulators 5, 8, and 11 all contain a digit that duplicates one in accumulator 13. By transmitting the contents of accumulator 13 when we would otherwise transmit from the other three and arranging that the receiving accumulators receive only the relevant digit, we can free accumulators 5, 8, and 11 for other purposes.

The first step in modifying the configuration in this way is to use three additional programs on accumulator 13, each configured to the operation A03, triggered by control lines 2–2, 2–5, and 2–8, respectively. Next, program 6 on each of accumulators 3, 4, 6, 7, 9, and 10 is changed from α to β. The final step is to connect data trunk 1 to the β input of each of those same accumulators through adapters that cause only the relevant digit to be received. The simplest adapter to consider is the one for x (the hundreds digit). Shifting the value two digits to the right will result in the desired digit in the correct position. Such a shifter is standardly available on the ENIAC and is referred to as a -2 shifter. Therefore, the β inputs of accumulators 10 and 11 should be connected to data trunk 1 through -2 shifters.

The ones and tens digits are a bit more tricky as they require the deletion of digits to the left of the desired ones. Although standard deleters do exist, they are a little problematic for this application. In particular, they are oriented around deleting less significant, rather than more significant, digits. Additionally, they must be used on the output terminals, rather than input terminals. A second possibility is to construct custom adapters. This is quite frequently done, as needed, because the construction of a custom adapter can be done quite quickly, and then it becomes available if needed again in the future. For the purposes of simulation, we have tested a third alternative. Because shifters and deleters are constructed with a socket and plug, they are designed to

be inserted between a data cable and a data terminal. This also allows them to be connected in series. Cascading a left shift of nine digits followed by a right shift of nine digits results in the ones digit being the only digit transmitted. This pair of shifters should be used on the β inputs of accumulators 3 and 4. Similarly, a left shift by eight followed by a right shift by nine will result in the tens digit being transmitted in the position of the least significant digit and should be used on the β inputs of accumulators 6 and 7.

One final adjustment must be made to the algorithm for these changes to work. In the algorithm shown in Fig. 5, n is incremented earlier in the sequence than either y or x. This is problematic because we are now extracting the values of y and x from n. To correct this, we can change the initialization of n to 99 and move its increment from step 2–3 to step 3–1. With these changes in place, the same computation is accomplished using nine rather than 12 of the 20 available accumulators. Furthermore, the timing is exactly the same, as no steps of the algorithm were added or removed.

B. Using the Function Tables

Because the majority of the accumulators (nine of 12 in the original formulation) are used for cubing, a substantial number of accumulators can be saved by using a function table to hold a table of cubes. One straightforward approach frees accumulators 4–11 by using accumulator 3 to hold the value of a digit that will be looked up in the table. This accumulator can be loaded using the same technique as above for extracting the digits from n. For each of x, y, and z, the digit is loaded into accumulator 3 and the function table is activated. The output from the function table is then taken by accumulator 1 (holding m) and added to the current value. Steps 3–4 through 3–7 of the algorithm in Fig. 5 continue to operate in the same way. However, steps 2–1 through 2–9 are no longer needed with this approach.

It should be noted that this approach is slower than the one given originally. The reason for this is that the table lookups for y^3 and x^3 are done in each iteration of the inner loop, whereas in the original approach, the computations of those were done outside of the innermost loop. The loss of performance can, however, be recovered by allocating two more accumulators: one to hold the current value of x^3 and one to hold the current value of y^3 (or $x^3 + y^3$ to save one more addition time per iteration of the inner-most loop).

VII. CONCLUSION

From its public unveiling in 1946, the ENIAC stood for several years as the most powerful computational instrument in the world. In its use prior to the deployment of an instruction set processor configuration, the ENIAC had a reputation of being hard to program. Although many of the programming concepts for the ENIAC differ significantly from those that have become familiar since, the example presented here shows that it is actually quite an interesting machine to program.

Source code, executables, programming examples, and documentation for the simulator used here can be found at http://cs.drexel.edu/~bls96/eniac/. ■

REFERENCES

[1] J. P. Eckert, Jr., "The ENIAC," in *A History of Computing in the Twentieth Century*, N. Metropolis, J. Howlett, and G.-C. Rota, Eds. New York, NY, USA: Academic, 1980, pp. 525–539.

[2] J. W. Mauchly, "The ENIAC," in *A History of Computing in the Twentieth Century*, N. Metropolis, J. Howlett, and G.-C. Rota, Eds. New York, NY, USA: Academic, 1980, pp. 541–550.

[3] A. W. Burks, "From ENIAC to the stored-program computer: Two revolutions in computers," in *A History of Computing in the Twentieth Century*, N. Metropolis, J. Howlett, and G.-C. Rota, Eds. New York, NY, USA: Academic, 1980, pp. 311–344.

[4] T. Haigh, P. M. Priestley, M. Priestley, and C. Rope, *ENIAC in Action: Making and Remaking the Modern Computer*. Cambridge, MA, USA: MIT Press, 2016.

[5] D. H. Lehmer, "A history of the sieve process," in *A History of Computing in the Twentieth Century*, N. Metropolis, J. Howlett, and G.-C. Rota, Eds. New York, NY, USA: Academic, 1980, pp. 445–456.

[6] D. H. Lehmer, "On the converse of Fermat's theorem II," *Amer. Math. Monthly*, vol. 56, no. 5, pp. 300–309, May 1949. [Online]. Available: http://www.jstor.org/stable/2306041

[7] L. D. Mol and M. Bullynck, "A week-end off: The first extensive number-theoretical computation on the ENIAC," in *Proc. 4th Conf. Computab. Eur. (CiE)*, Athens, Greece, Jun. 2008, pp. 158–167. [Online]. Available: https://doi.org/10.1007/978-3-540-69407-6_19

[8] M. Bullynck and L. De. Mol, "Setting-up early computer programs: D. H. Lehmer's ENIAC computation," *Arch. Math. Logic*, vol. 49, no. 2, pp. 123–146, 2010. [Online]. Available: https://doi.org/10.1007/s00153-009-0169-8

[9] M. Bullynck, "Computing primes (1929-1949): Transformations in the early days of digital computing," *IEEE Ann. Hist. Comput.*, vol. 37, no. 3, pp. 44–54, Jul. 2015. [Online]. Available: https://doi.org/10.1109/MAHC.2015.46

[10] B. L. Stuart, "Simulating the ENIAC," *Proc. IEEE*, vol. 106, no. 4, pp. 761–772, Apr. 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8326772/

[11] L. Erickson, "Top secret Rosies," Documentary film, 2010.

[12] K. Kleiman and J. Palfreman, "The programmers," Documentary film, 2013.

[13] A. K. Goldstine, "A report on the ENIAC Part I: Technical description of the ENIAC," Moore School, Univ. Pennsylvania, Philadelphia, PA, USA, Tech. Rep., 1946. [Online]. Available: http://archive.org/details/ReportonENIACEl00MoorB

[14] H. D. Huskey, "A report on the ENIAC Part II: Technical description of the ENIAC," Moore School, Univ. Pennsylvania, Philadelphia, PA, USA, Tech. Rep., 1946. [Online]. Available: http://archive.org/details/reportoneniacele05moor

[15] A. W. Burks and H. D. Huskey, "Report on the ENIAC, operating manual," Moore School, Univ. Pennsylvania, Philadelphia, PA, USA, Tech. Rep., 1946. [Online]. Available: http://archive.org/details/ReportonENIACEl00Moor

[16] H. Neukom, "The second life of ENIAC," *IEEE Ann. Hist. Comput.*, vol. 28, no. 2, pp. 4–16, Apr. 2006.

ABOUT THE AUTHOR

Brian L. Stuart received the B.S. degree in computer science and electrical engineering from the Rose-Hulman Institute of Technology, Terre Haute, IN, USA, the M.S. degree in electrical engineering from Notre Dame University, Notre Dame, IN, USA, and the Ph.D. degree in computer science from Purdue University, West Lafayette, IN, USA.

Currently, he is an Associate Teaching Professor of Computer Science at Drexel University, Philadelphia, PA, USA.