Contents

Preface

This tutorial describes how to program the ENIAC Java simulator developed by Till Zoppke available at the Free University (FU) of Berlin. With the help of the modulo algorithm elementary function groups of the ENIAC are briefly described and wired up for proper usage. The tutorial has been initiated by Prof. Dr. Andreas Polze in response to his Origins of Operating Systems Course (additional slides available) at the Hasso-Plattner-Insitute for IT-Systems Engineering at the University of Potsdam in the summer term 2006. Additional Powerpoint slides about the ENIAC and other pre-OS computer systems are here available.

The final modulo algorithm code of this tutorial for the ENIAC simulator can be downloaded. To load it, run the simulator and open the load menu, select "Load from local file system" and select the downloaded file. Additionally, a video (MPEG, 5m31s, approx. 27.5 MB) of the solution running in the ENIAC simulator is also available for download.

Matthieu-P. Schapranow, June 2006.

Overview

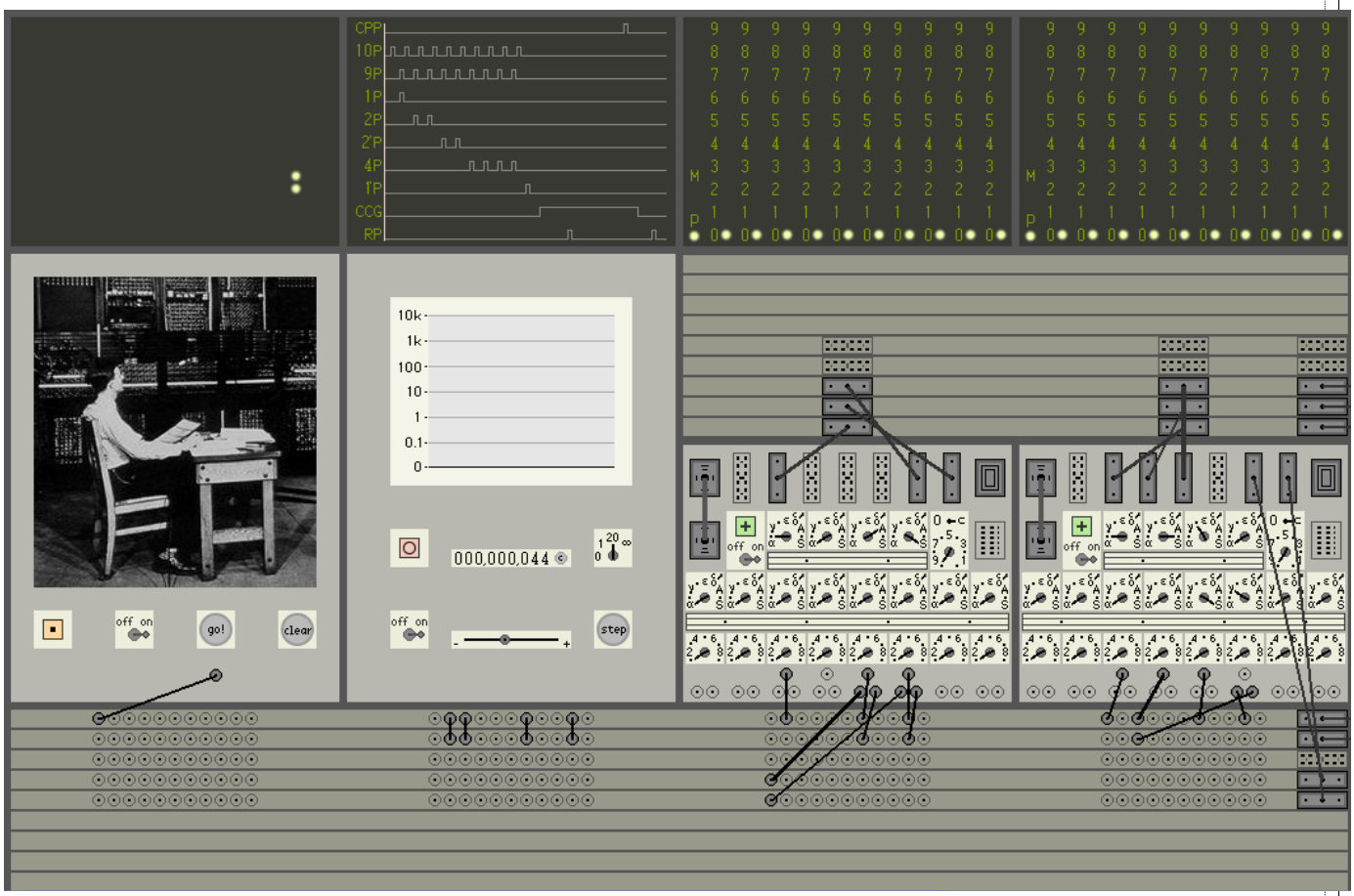

Figure 1: Final result showing Initializing Unit, Cycling Unit, Accumulator 1, and Accumulator 2.

program cycle	Accumulator 1	Accumulator 2	Comments
0	0	0	initial state
1	0	m	A2 := m
2	n	m	A1 := n
3	n	m	A2 < 0 goto 4; a2 > 0 goto 5
4	n	m - n	goto 3
5	n	m + n	end

Table 1: The work of the desired algorithm: m modulo n

Figure 1 shows the overview screenshot containing (from left to right) the Initializing Unit, Cycling Unit, Accumulator 1 (A1), and Accumulator 2 (A2). The work of the desired algorithm is described in Table 1.

The Constant Transmitter

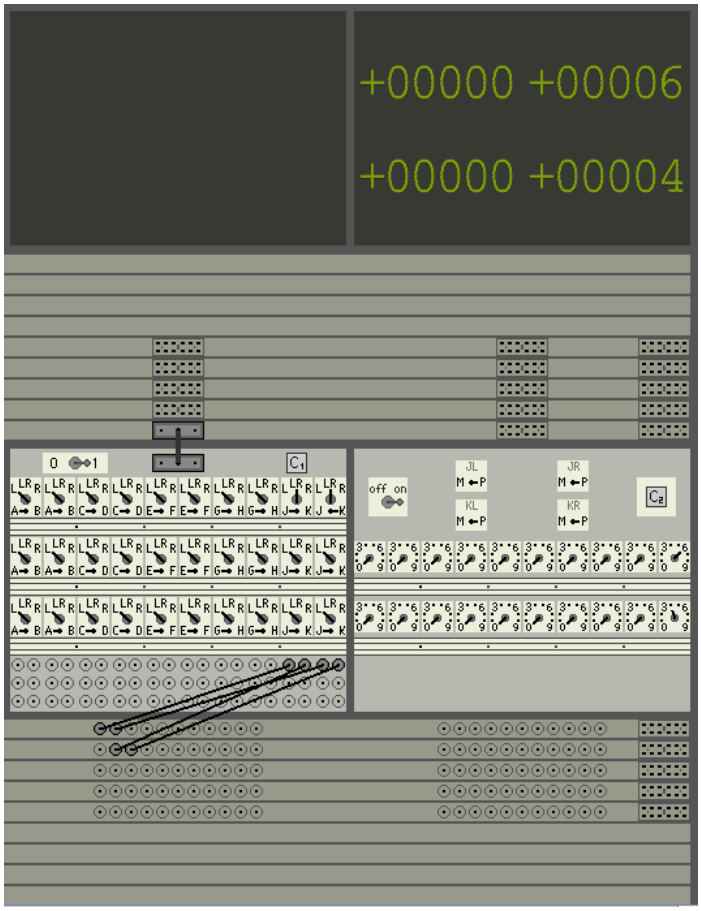

Figure 2: The Constant Transmitter wired up.

The Constant Transmitter consists of two directly interconnected module parts. The module part on the right side provides input mechanisms to set constants in the simulation. The associated number display shows the current state of the constants. It is possible to set up to four five digit constants or two 10 digit constants with the help of this module.

Initialization of A1 and A2

Setup of Constant Transmitter (1)

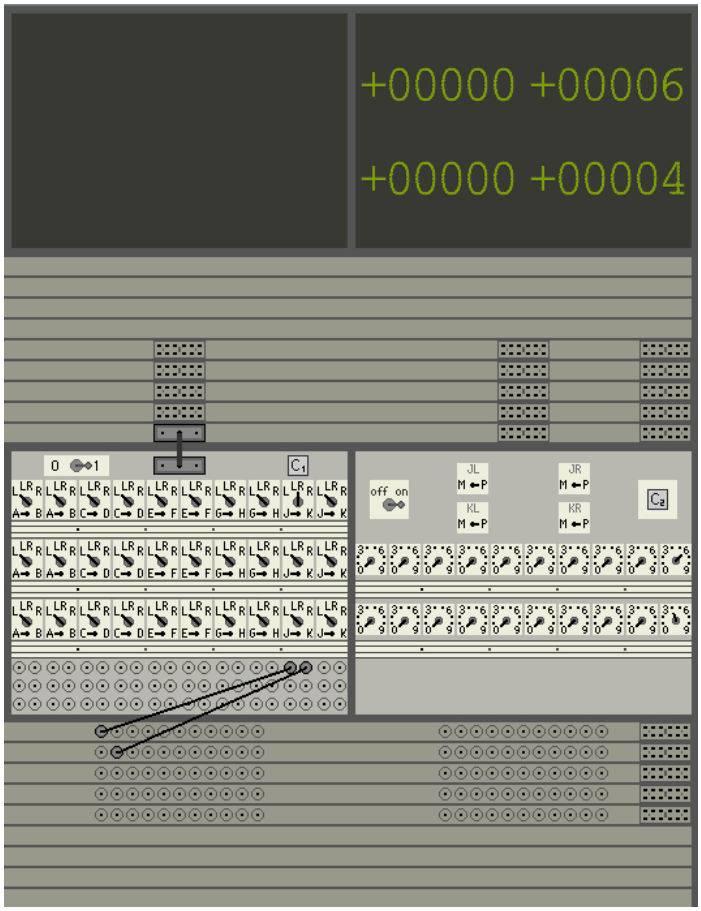

Figure 3: Setup of Constant Transmitter module.

In Figure 1 the Initializing Unit (at the most left of the picture) is connected to the program tray A (lower tray). That means after initializing the ENIAC system the first pin of tray A will be stimulated. This input pulse is carried via the whole tray. Figure 3 shows that the available input pulse (at program tray A, pin 1) is connected to input 9, tray 1 of the Constant Transmitter. With the help of the associated three-positional switch (first row, second from the right) the interpretation of the given constant from the constant storage is set. Three possible interpretations for the given number can be distinguished.

- L - interprets the left five digits as a ten digits number, ignores the right five digits and shifts the given number five digits to the right (fills up with zeros)
- R - interprets the right five digits as a ten digits number, ignores the left five digits (fills up with zeros)
- LR - interprets the whole digits as a ten digits number

With the help of the lower bi-positional switch it is possible to select either the upper digit (position J) or the lower digit (position K) storage to read from. In Figure 3 the associated three-positional switch is set to position LR, the bi-positional switch is set to J. Thus it offers a ten digits number which is used as initial input for A1. Additionally, the data output plug of the Transmitter module part is connected to the data tray I (upper tray). So, keep in mind, that data tray I is the data bus which contains the transmitted constants from the constant provider. After finishing the output cycle of the transmitter the next part functionality can be triggered. Therefore, output 9, tray 1 of the Constant Transmitter is connected to the program tray B, pin 2.

Setup of A2 input

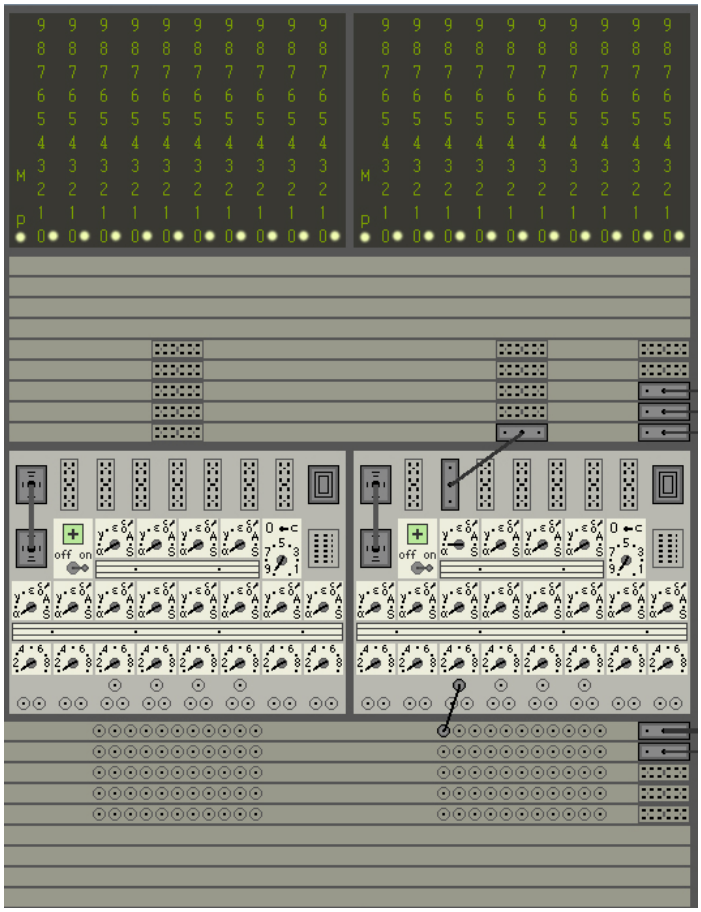

Figure 4: Settings for A2 to listen for input from data tray I on its input port beta.

The input pulse available at program tray A, pin 1 is also connected to input 1 of A2. Furthermore, the data tray I is connected to the data input socket beta of A2. With these settings A2 is now able to gather data from the data tray I via its input beta if its input 1 is stimulated. As aforementioned, that works, because data tray I will hold the data transmitted by the Constant Transmitter.

In case of the presence of a program pulse at tray A, slot 1 the constant transmitter as well as the A2 will receive this pulse via attached cables. A2 is set to store data from data tray I and the constant transmitter is set to transmit the upper ten digits via data tray I in response to the initial pulse.

Wire jumpers to branch triggering impulse

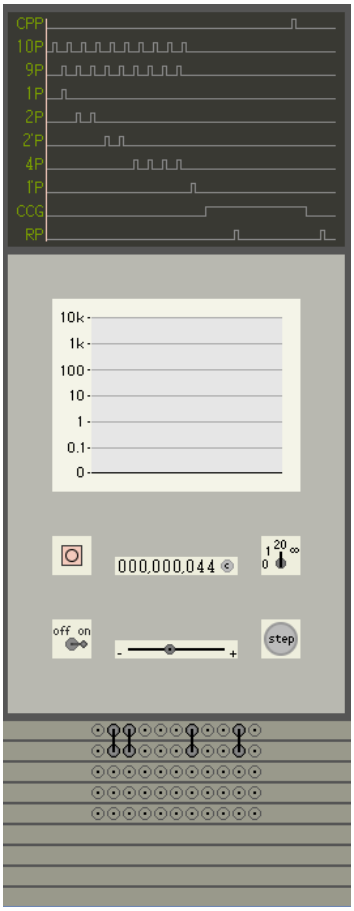

Figure 5: Wire jumpers to branch triggering impulse.

You may have mentioned that it is only possible to connect one cable to one pin. Thus, using one and the same impulse for two events in one component will result in a long wire gathering the pulse from a far away component. To reduce the necessary width of the program slot (and the resulting complexity of wired chaos), the usage of wired jumpers or links to stimulate two or more slots in different trays simultaneously is desired.

At the current programming position it is important to mention the link between program tray B, pin 2 and program tray A, pin 2. Program tray B, pin 2 is used by the output pulse of the constant transmitter, as mentioned before. With the help of the described jumper it is additionally possible to get the input pulse for the successor component from program tray A, slot 2.

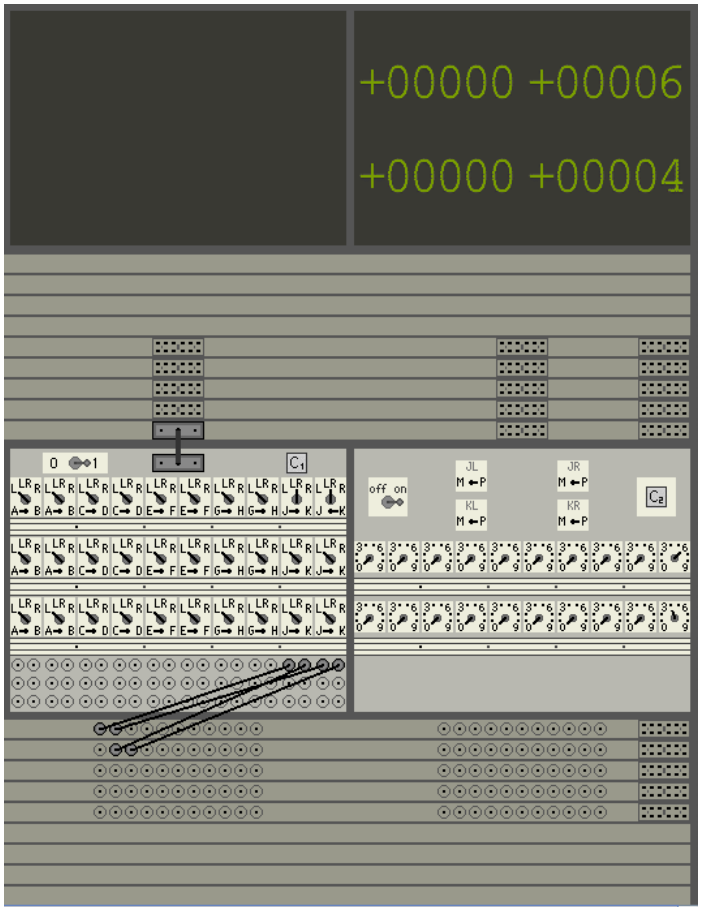

Figure 6: Setup of Constant Transmitter module for initializing A1.

Accumulator A1 has to be initialized in a similar way as described for accumulator A2. Now, the Constant Transmitter has to populate the lower ten digit constant from its storage via data tray I. With the help of the aforementioned jumper it is possible to connect program tray B, pin 2 to the input 10, tray 1 of the Constant Transmitter module. So, the output 9, tray 1 of the Constant Transmitter is directly connected to its input 10, tray 1. The according switch is also set to position LR to read one ten digits constant from the constant storage. The constant selector is set to K, so that the lower constant is used during this calculation cycle.

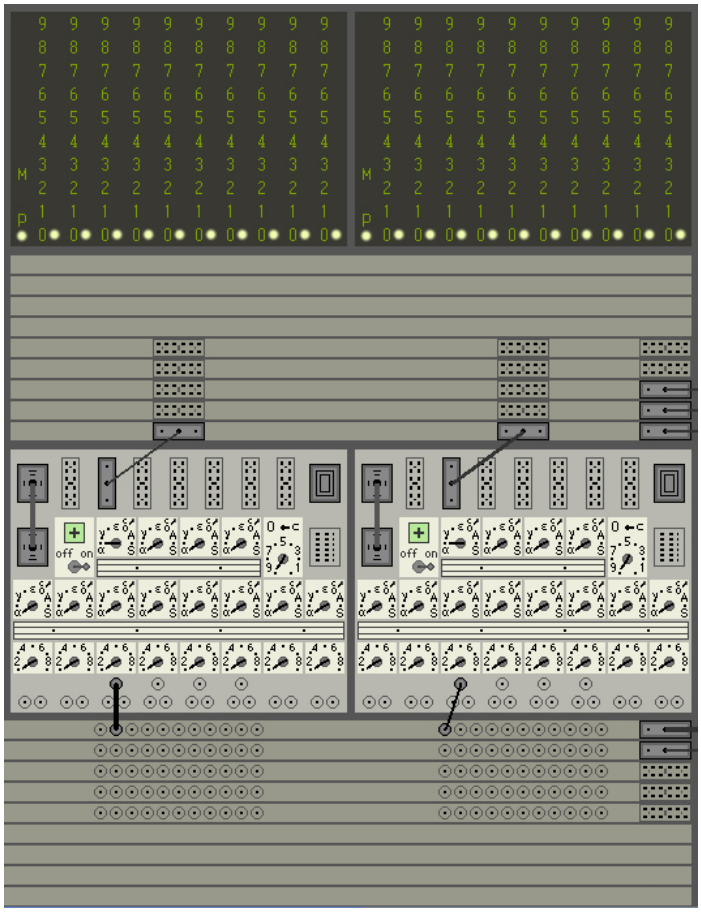

Figure 7: Settings for A1 to listen for input from data tray I on its input port beta.

A1 has to be wired in a similar way like A2. In this situation, the input pulse arrives from program tray A, pin 2. Obviously, with the described wiring scheme A1 has to read data from the data tray I in the second program cycle after A2 has been initialized.

Modulo algorithm

After initializing both accumulators A1 and A2 the conceptual modulo algorithm has to be wired up. The implementation of the modulo algorithm is reduced for simplicity to work only with positive arguments. The algorithm can be described as the following pseudo code which transforms the storage content of A2 to A2 mod A1 for non-negative numbers A1 and A2.

```
while(A2 >= 0) A2 =- A1;
A2 += A1;
```

The given pseudo code is based on a high level of abstraction. But, it is necessary to reduce the level of abstraction and introduce more atomic operations to understand the programming decisions described in the following paragraphs.
The used while-loop is reduced to some goto-statements, thus it is possible to implement them as wire loops to a former calculation cycle.
Additionally, the if-condition used within the while-loop has to be transformed to two if-conditions using strong relations, i.e. *smaller than* and *larger than*. This has to be done, because of the limited branching capabilities of the ENIAC. If the content of A2 equals 0 in the aforementioned code A1 is subtracted, the while-loop exits and afterwards A1 is again added to the value. However, these operation do not change the value of A2, if it is already 0. Obviously, it is reasonable to stop the code if A2 already contains 0. Therefore, the equals-statement can be ignored in the following code. It is not possible to test the identity of two values using the ENIAC in a similar fast way. To test for identity one way would be to subtract both values from each other and test for < 0 AND > 0 which would be more is cost-expensive.

```
010 if (a2 > 0) goto 030
020 if (a2 < 0) goto 050
030 a2 = a2 - a1
040 goto 010
050 a2 = a2 + a1
```

Magnitude Discrimination

Obviously, it is necessary to check whether the temporary result in A2 is larger than zero (cf. line 010). Therefore, a trick is used on the ENIAC, which makes it possible to use branching. This is an interesting fact, because the initial ENIAC was not engineered with branching capabilities. By mixing data information and program information it is possible to interpret data as control information. This data information can be used for branching decisions - the so called **magnitude discrimination**.
Why does this work? Each signed ten digits number is transferred with its sign via the eleven pins of a data tray plug. The most left pin carries the signing information for the current number. For instances, how is the number - 00000 01832 transferred via the data tray? First of all, numbers below zero are stored as their 10's complement value, because numbers are stored by their decimal representation. The complement of P 00000 01832 is calculated by subtracting the given value from M 99999 99999 and adding the value 1 to it.

M 99999 99999

-	P	00000 01832
	M	99999 98167
+	P	00000 00001
	M	99999 98168

Table 2: Calculation of the 10's complement.

The resulting stimulation of the eleven pins of a data tray is described in the following pulse sequences.
Pin 1 (sign pin): The sign information is only set, if the transferred number is below zero, thus ten pulses on pin 1 occurs.
Pin 2..Pin7: The digit zero has to be transferred, also no pulses occur here.
Pin 8: One pulse occur here to transfer the digit 1.
Pin 9: Eight pulses occur here to transfer the digit 8.
Pin 10: Three pulses occur here to transfer the digit 3.
Pin 11: Two pulses occur here to transfer the digit 2.

Pulse	M	9	9	9	9	9	8	1	6	8	P	0	0	0	0	0	1	8	3	2
1.	x	x	x	x	x	x	x										x	x	x	x
2.	x	x	x	x	x	x	x	x		x								x	x	
3.	x	x	x	x	x	x	x	x		x								x	x	
4.	x	x	x	x	x	x	x	x	x	x								x		
5.	x	x	x	x	x	x	x	x	x	x								x		
6.	x	x	x	x	x	x	x	x	x	x								x		
7.	x	x	x	x	x	x	x	x	x	x								x		
8.	x	x	x	x	x	x	x	x	x	x								x		
9.	x	x	x	x	x	x	x	x	x	x										
10.																				x

Table 3: Decimal number representation and pulse sequences.

Now, the mixing of data and program information can be used for branching. Table 3 shows, that the sign pin carries either always a signal (number is smaller than zero) or no signal (number is larger than zero). It is possible to connect this sign pin with the input of another accumulator. However, it is necessary to trigger an event, if the transferred number is larger than zero. This is done by using the complement's output via output S of A2.

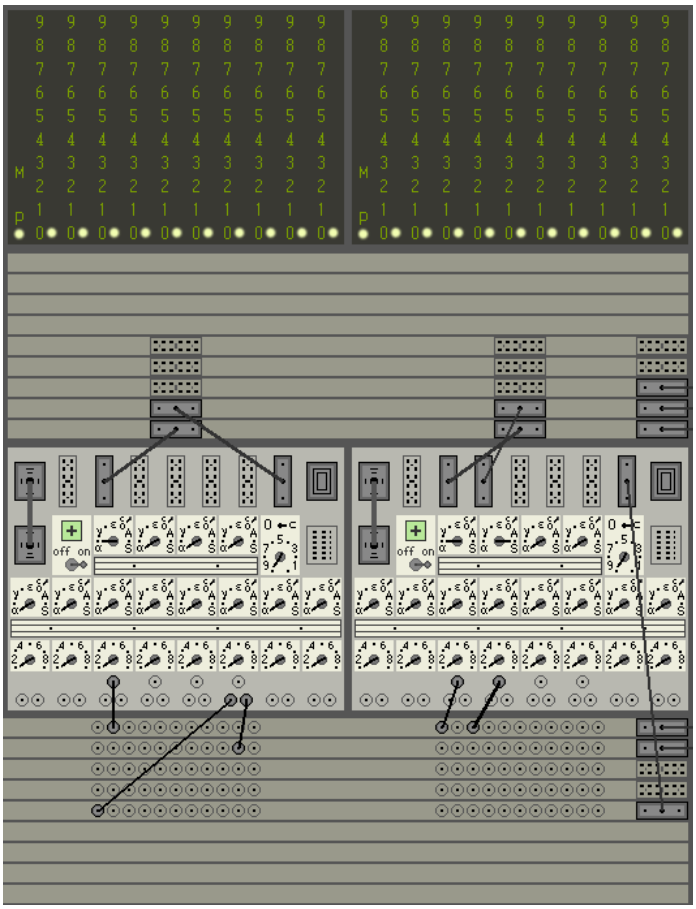

Figure 8: Magnitude discrimination used for if-branching with A2.

In the given example the complement output of A2 using output S is connected to the program tray E on the right side of the picture. The most sign pin 1 of tray E is connected to input 10 of A1. This is a must to delay the pulse spread for the current calculation cycle. Otherwise the pulse will result in ten immediate triggering actions which are not intended here. By delaying those triggering events, one single output event can be produced within the next calculation cycle to trigger the real event. The output 10 of A1 is connected to program tray B, pin 10. This pin will be stimulated within the next calculation time. Please mention the existence of the link between program tray B, pin 10 and program tray A, pin 10 for further reading.

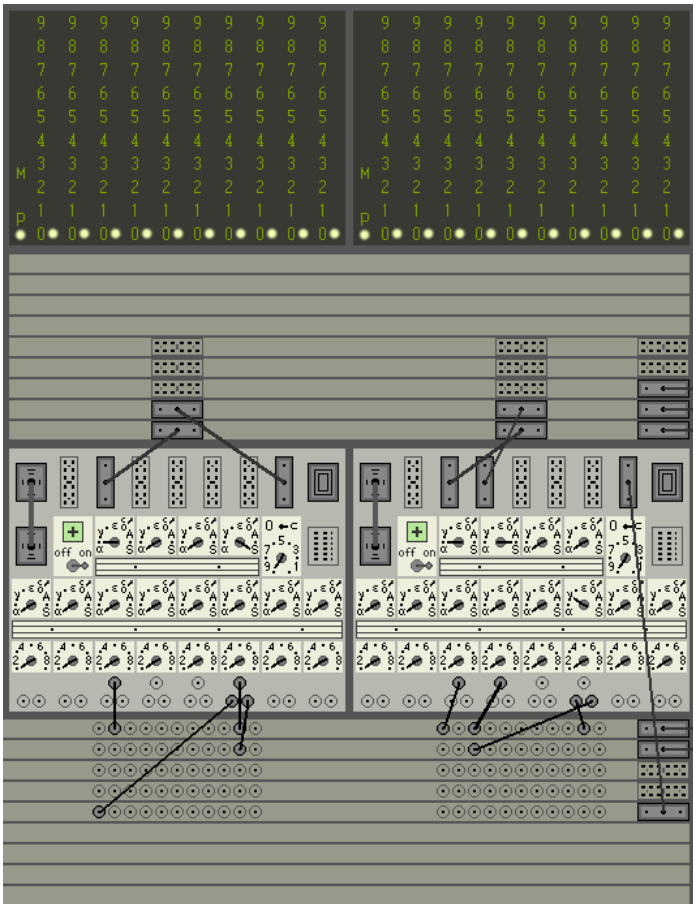

Figure 9: Subtracting value of A1 from value of A2.

The next action is to subtract the content of A1 from the working result in A2 (cf. line 030). A1 has to transfer the complement of its current value via data tray II. Therefore, program tray A, pin 10 is connected to A2 and A1 to trigger an event. On the one hand, program tray A, pin 10 is connected to input 4 of A1 and the associated operation switch of A1 is set to S. On the other hand, program try A, pin 10 is also connected to input 10 of A2 and the associated operation switch of A2 is set to gamma. Data input port gamma is linked to data tray II. Thus, A2 is listening for incoming data from data tray II.
Finally, the output 10 of A2 is connection to program tray B, pin 3 which is also linked to program tray A, pin 3 by a jumper. You may have mentioned, that program tray A, pin 3 gets already stimulated when by a program pulse when the modulo algorithm starts. So, this link realizes the goto-statement in line 040 of the pseudo code.

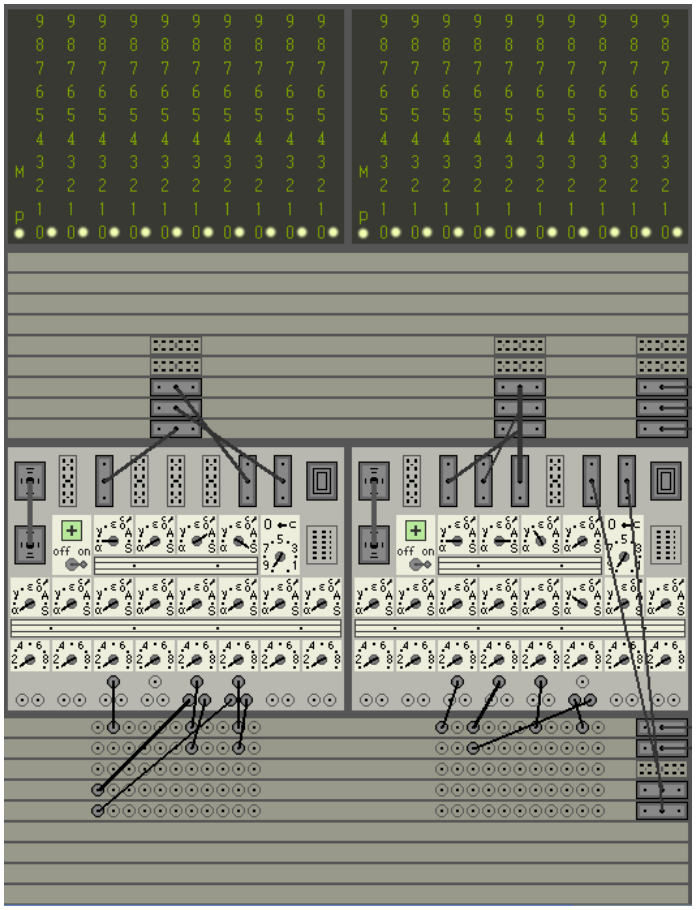

Figure 10: Adding value of A1 to value of A2.

The only thing left to do now is to finish the algorithm with one final addition of A1 to A2 as described in the pseudo code line 050. Therefore, the if-branch of line 020 is realized by connecting the non-complement output (A) of A2 to the program tray D. The most left pin of this plug is connected to input 9 of A1 to delay the action as aforementioned. The output 9 of A1 is connected to program tray B, pin 7, which is also linked to tray A, pin 7 by a jumper. Program Tray A, pin 7 is connected to input 3 of A1 as well as to input 3 of A2. The output A of A1 is connected to the data tray III and the data input delta of A2 is connected to the same data tray. If input 3 of A1 is triggered it transmits its current value via data tray III. Meanwhile A2 is listening for data from data tray III. This last action adds the current value of A1 to the current value of A2 which finalizes the given algorithm. Therefore no more output pins are connected, the program pulse disappears at this moment and the ENIAC reaches its final stop state.