A Java Simulation of the ENIAC

Peter Hansen
Universität Osnabrück

February 16, 2004

Abstract

This thesis describes the ENIAC simulator, written in Java by the author. A short overview of the original machine is given, followed by instructions on how to operate the simulator. Implementation details together with an example of how to use the simulator to run an ENIAC program complete the paper. The simulator as Java applet can be found online [1].

Contents

1 Introduction **4**

2 The Original ENIAC **5**
2.1 General Features 5
2.2 A 100 kHz Clock 5
2.3 The Accumulators 6
2.4 The Constant Transmitter 6
2.5 The Multiplier 6

3 The Graphical User Interface **7**
3.1 The Main Screen 7
3.2 The Accumulator Window 11
3.3 The Initiating Unit Window 11
3.4 The Printer Window 13
3.5 The Constant Transmitter Window 13

4 Java Implementation **14**
4.1 Organization of Source Code 14
4.1.1 The Makefile 15
4.2 Package `eniac.model` 15
4.2.1 Class `eniac.model.Eniac` 15
4.2.2 Class `eniac.model.InitUnit` 16
4.2.3 Class `eniac.model.PrintUnit` 16
4.2.4 Class `eniac.model.ConstUnit` 16
4.2.5 Class `eniac.model.AccuUnit` 16
4.3 Package `eniac.view` 17
4.3.1 Class `eniac.view.EniacApplet` 17
4.3.2 Unit Windows 17
4.3.3 Class `eniac.view.EditWindow` 17
4.3.4 Class `eniac.view.OpenWindow` 18
4.4 Package `eniac.control` 18
4.4.1 Class `eniac.control.Controller` 18
4.4.2 Class `eniac.control.CPPException` 18
4.4.3 Class `eniac.control.Display` 18
4.4.4 Class `eniac.control.Loader` 18

5 The Simulator's Programming Language **19**
5.1 The Syntax 19
5.1.1 The `content` Block 19
5.1.2 The `connect` Block 20
5.1.3 The `action` Block 20
5.2 A Real-life Example: `modulo` 21

5.2.1 Division 25

6 Comparison between Simulator and Original **25**

7 Room for Further Research **26**

8 Summary **28**

1 Introduction

During World War II, the Moore School of Electrical Engineering at the University of Pennsylvania developed a general purpose computing machine for the US Army, the ENIAC—Electronic Numerical Integrator and Computer. Its primary task was to perform ballistical calculations. The ENIAC was the first American digital computer, and interestingly it was based on the decimal system. It consisted of 40 panels, standing in the shape of a big U in a space of 170 m^2, contained around 18 000 vacuum tubes, consumed 140–174 kW of electrical power, and weighed 30 tons [2, p. 178].

Figure 1: ENIAC, J. P. Eckert (left) , J. Mauchly (right) in the foreground (from [2]).

The ENIAC was no stored-program computer, so programming was very different from what we know of modern computer systems. There was no ENIAC programming language; all instructions were given to the machine by connecting wires to special ports on the panels, in a way to allow goal-directed information flow to occur, and by setting switches on the various units. Examples of how to do this are given later in this paper.

The original hardware of the ENIAC is not complete anymore, there are only some surviving panels. To celebrate the ENIAC's 50th birthday, a team around Jan Van der Spiegel of the University of Pennsylvania reconstructed the ENIAC machine in modern hardware [2], which resulted in a single silicon

chip [3]. For this project, extensive research on the original hardware details was done.

In contrast to other ancient computers, such as the German Zuse Z3 [4], or the EDSAC [5][6], there has been no software simulation of the ENIAC. The aim of this paper is to document the author's writing of a Java simulation of the basic parts of the ENIAC, and to provide a user manual. With the simulator, old programs for the ENIAC can be tried out and traced back in every detail. Also, the simulator offers a comfortable environment for the development of new programs. All sources, this documentation in various formats, and the running simulator as a Java applet can be found on the web [1]. For detailed specification of what parts of the original machine are not yet implemented in this version of the simulator, see Section 7.

After a short account on the original machine, this paper continues by describing the graphical user interface of the simulator, and then goes into the Java implementation details, showing also an example of how to program the simulator (and the original) to calculate a modulo division.

2 The Original ENIAC

2.1 General Features

The 40 panels of the ENIAC are grouped into 30 units with specialized tasks. There are 20 accumulators, which perform simple additions and subtractions, and can store intermediate results. There is a dedicated hardware multiplier to speed up multiplications, and a divider/square rooter unit. A constant transmitter is used to feed constant data from punch cards into the system, but it can also be used to input data via manual switches. The printer units produces output on punch cards. Three function tables can be used for providing read-only tabular data. Additionally, there are control units as the initiating unit, the cycling unit, and the master programmer. All units can be connected to each other using trunks, which transfer digit data and control signals, and provide for synchronization of the units.

2.2 A 100 kHz Clock

The ENIAC ran on a system clock of 100 kHz (as a comparison: a modern computer is clocked in the range of some GHz). A cycle of twenty clock times is called an *addition time*, the atomic time unit of calculations. An addition time takes up 200 μs, and the cycling unit generates several pulse trains during that period. These pulse trains have specialized functions. The first half of the addition time is used to transmit digit information from one unit to another, while program control information flows during the second half. An important pulse is the *Central Programming Pulse*, CPP. It is sent by the initiating unit to start a calculation, and by other units to signal

their completion of a calculation task. Reception of a Cpp triggers a unit to start action.

2.3 The Accumulators

There are twenty accumulators in the Eniac design. The accumulator is one of the most important units, as it is the only one to provide read/write memory. It can store a ten-digit decimal number with sign. Communication with other units is made possible by several input and output ports.

There are five inputs for digital data, named α through ε. Trunks bearing numerical output from other units have to be connected to one of these digit input ports in order for the accumulator to receive the transmitted number. Reception is equivalent to addition of this number to the content already in memory.

Two digit output ports, A and S, are used to send the accumulator's content (through port A) or the complement of the content (through port S) to a digit trunk. Addition of complementary numbers in the receiving accumulator is in fact a subtraction.

The number stored by an accumulator can have one to ten digits precision, according to a *significant digits* switch. By means of special cables, two accumulators can be connected to yield 20-digit precision.

Program pulses, i. e. Cpps, reach an accumulator through twelve program input ports. Every port is connected to a switch, and the position of this switch controls which action is to be performed on Cpp reception. Possible actions are digit reception on ports $\alpha \ldots \varepsilon$, sending of content through ports A, S, or both, and no action at all, the well-known Nop of modern microprocessors.

Eight of these twelve program switches can be programmed to perform their action repeatedly up to nine times, before they send a Cpp. An accumulator's content can be reset to zero after an action by means of a clear switch.

2.4 The Constant Transmitter

The constant transmitter features thirty program controls, of which 24 make available data read from punch cards. Six controls are connected to manual switches, which can be used to input data before the start of a calculation.

2.5 The Multiplier

Apart from the trunks mentioned so far, there are so-called *static interconnections* between several units. The multiplier is constantly fed with the contents of two special accumulators, regardless whether they send or not. If triggered, it can calculate a multiplication of a 10-digit number with a

second number of p digits in $p + 4$ addition times. Static interconnections are also used to feed the printer.

3 The Graphical User Interface

The simulator's user interface is implemented as a Java applet embedded in a web page, so it can be used online [1]. On loading this page, `EniacApplet` is started, the main interface class. By clicking the variuos unit buttons, the user can take a closer look at the contents of the variuos units of the ENIAC simulator, load example programs, write new ones (see Section 5), and start a calculation.

Figure 2: The main screen.

3.1 The Main Screen

The main screen looks more or less like the window shown in fig. 2. The layout of the unit buttons (to the left, top, and right of the window), mirrors the floor plan of the original machine. White and light grey buttons refer to implemented units, dark grey buttons are place holders for future extensions:

In the center of the window is a text output area, which is very unhistorical, but also very handy (fig. 3). The simulator can give feedback to all actions it performs, which makes debugging own programs easier. Also, output in the course of a running program that on the original machine would be sent to the printer, appears in text form in this area.

```
Loading file http://127.0.0.1/eniac/programs/n_square_cube...
:  Example program file
:
:       `n n^2 n^3'
:
Loading finished.

1-5     +0000000000   +0000000000   +0000000000   +0000000000   +0000000000
6-10    +0000000000   +0000000000   +0000000000   +0000000000   +0000000000
11-15   +0000000000   +0000000000   +0000000000   +0000000000   +0000000000
16-20   +0000000000   +0000000000   +0000000002   +0000000004   +0000000008

VM came to a halt in cycle 8

1-5     +0000000000   +0000000000   +0000000000   +0000000000   +0000000000
6-10    +0000000000   +0000000000   +0000000000   +0000000000   +0000000000
11-15   +0000000000   +0000000000   +0000000000   +0000000000   +0000000000
16-20   +0000000000   +0000000000   +0000000003   +0000000009   +0000000027

VM came to a halt in cycle 8
```

Figure 3: The central text area.

Open program

The *Open Program* button can be used to load an example program stored in the directory `webcontent/programs` (see Section 5). A window appears showing the contents of this directory. Note that the Makefile must have been executed for own files to appear in this list. Upon selection of a file, it is loaded into the virtual machine and ready to run (fig. 4).

Edit program

The *Edit Program* button lets one take a look at the source code of the program currently loaded into the virtual machine. The text area in the edit window is editable, so it can be used to both load into and save from the virtual machine: To save a program created by setting switches on the units and connecting the trunks, as described below, one opens the edit window and copies its contents into the favourite text editor. This text can now be saved to local disc. To load it again, open the edit window and paste the text from the editor into it. This may somewhat differ from the normal course of action of other programs, but it makes possible working with the simulator via a web interface, storing only one's own source code files on local disc (fig. 5).

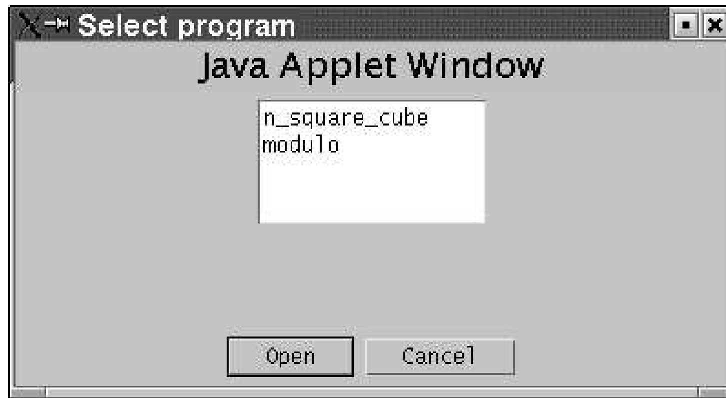

Figure 4: The file selection window.

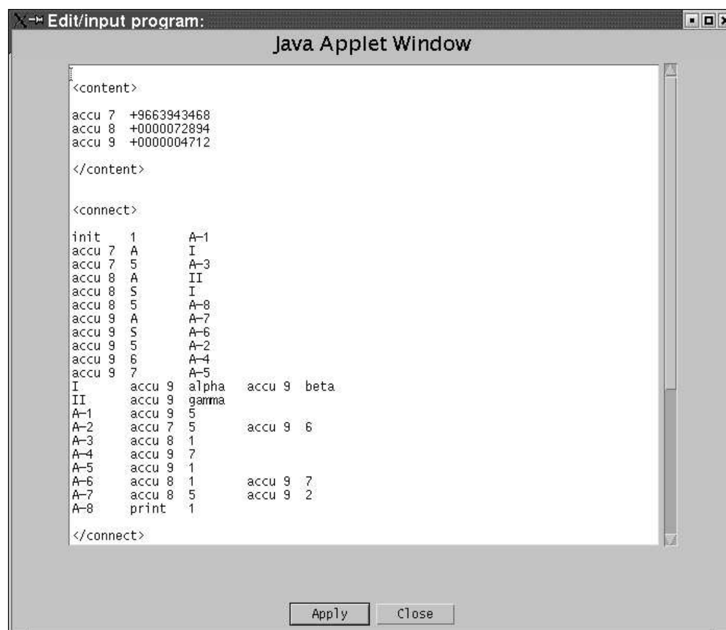

Figure 5: The source code edit window.

`core dump`

The *core dump* switch can be used to get very detailed debugging information during a program run. Every half cycle (after `run1()` and `run2()` methods, respectively, see Section 4.2.1), information is printed into the central text area about

- which units send messages,
- what messages are on the way,
- the contents of all units,
- accumulators' switch settings,
- reception of messages,
- the actions these messages trigger,
- the connections of input and output ports to trunks.

`clear VM`

The *clear VM* button is used to reset the whole virtual machine into virginal state. There are no connections between units and trunks, all memory is reset to zero, and all switches are set to NOP on the accumulators.

`initiating pulse`

The *initiating pulse* button makes the initiating unit send its first CPP to the connected units. This is the way to start a calculation on the ENIAC.

`cycle limit: 0`

To avoid rebooting the simulator during development of ENIAC "software" because of infinite loops, one can set a *cycle limit*. On reach of a given number of cycles, the virtual machine stops. This limit must of course be higher than the normal number of cycles needed for successful execution of the program, otherwise the machine will stop without generating the desired result.

If the special number 0 (zero) is chosen, the simulator runs virtually without limits (see Section 4.2.1).

`dump now`

The *dump now* button makes the virtual machine perform a dump, similar to those obtained by selecting *core dump*.

3.2 The Accumulator Window

The graphical interface of a simulated accumulator mimics the appearance of the original ENIAC accumulators (fig. 6, fig. 7).

The neon lamps of the original, which were a visual representation of the number the unit stored, are simple yellow dots in a dark area at the top of the window. The content itself, a signed 10-digit number, can be seen and edited in the text field next to it. When the *Apply* button is pressed, the "neon lamps" display the number.

Then there are pull-down menus for selecting the digit trunk to connect with each digit input and output port.

The *significant digits* switch is implemented as a menu as well. To the function of this switch: When another number n of significant digits than ten is selected, the programmer can clear this accumulator using a program switch set on NOP. Then it clears to all zeroes, except for digit $n + 1$ (counting from left), which is set to 5. Now a subsequent addition to this accumulator gets rounded correctly to n digits. This procedure is adopted from the original machine.

Left of the *significant digits* switch are four program control switches, *1–4*. With the pull-down menu the switch's action can be selected (receiving, NOP, sending). A little switch labelled *cc* is the clear-correct switch, which, on the simulator, has the sole function of clearing the accumulator's content according to the *significant digits* switch after completion of an action. The program ports to program switches *1–4* can be found at the bottom of the window, a rows of four light grey pull-down menus.

Program switches *5–12* are repetitive program switches, which can repeat their action up to nine times, as set by the pull-down menu directly below the action menu. The program input and output ports can be found as the last two rows of eight light grey menus. Input is top, output is bottom.

3.3 The Initiating Unit Window

This window is very easy, it contains only one pull-down menu to select the trunk the first CPP will be sent to. The *initiating pulse* button originally located on this unit is now on the main interface (fig. 8).

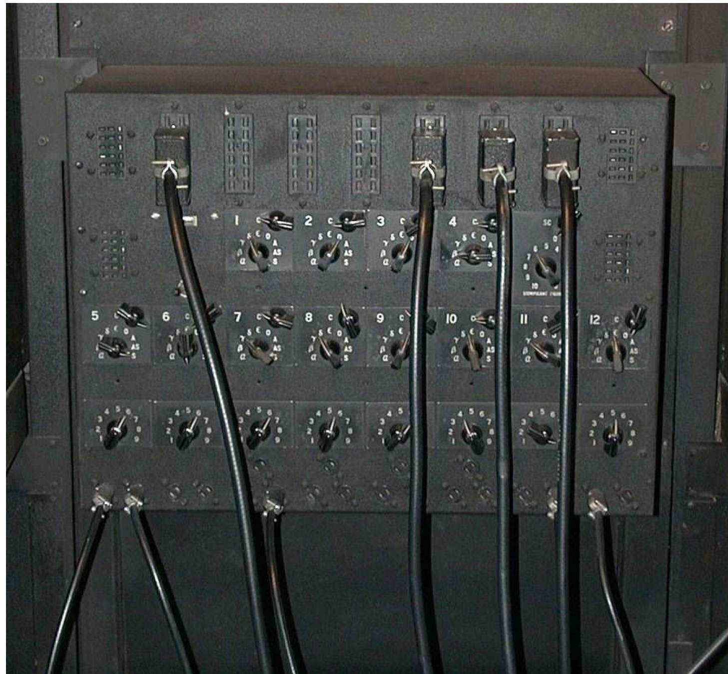

Figure 6: Front panel of an accumulator (from [2]).

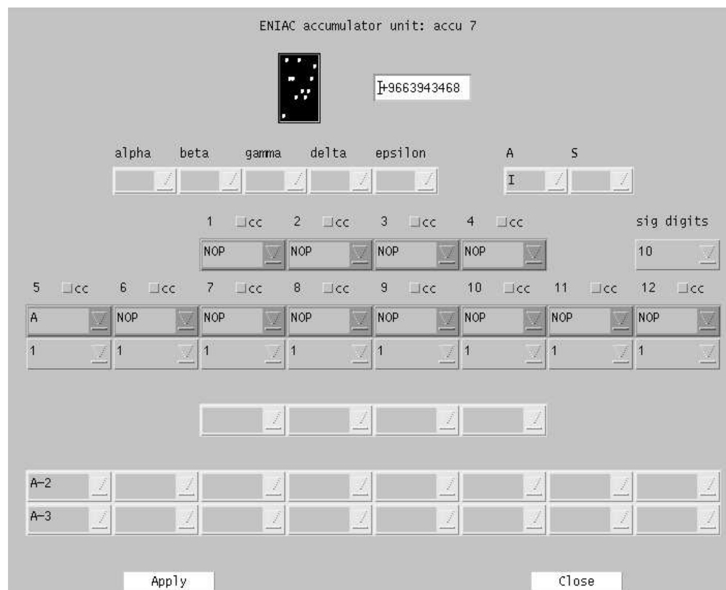

Figure 7: The simulator's accumulator.

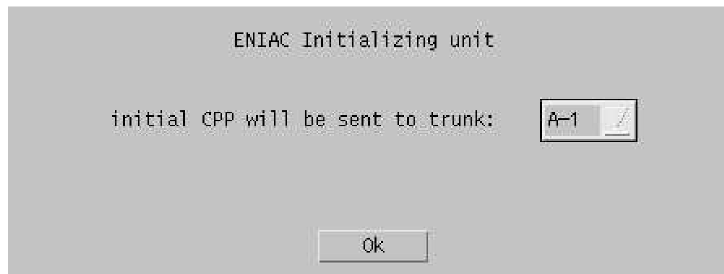

Figure 8: The initiating unit window.

3.4 The Printer Window

The printer window is pretty straightforward, there is one menu to select the trunk signalling to print, and another one to select a trunk to send a final CPP to (fig. 9).

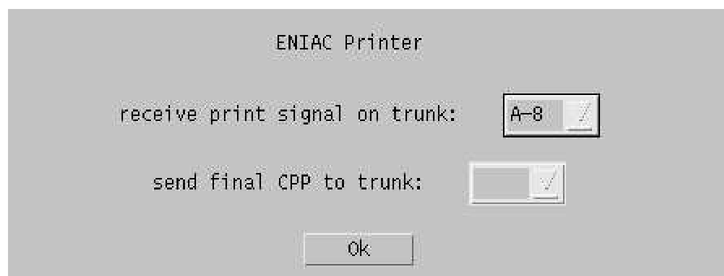

Figure 9: The printer window.

3.5 The Constant Transmitter Window

The simulator's constant transmitter lets the user set 30 numbers which can be used as read-only memory in calculations. Every number field has two accompanying menus for selection of program input and output ports. On reception of a CPP, e. g. on port *7*, the numerical content of memory field 7 is sent both via digit output ports A and S, as positive and negative value (see Section 2.3). These digit ports can be connected to digit trunks $I \ldots V$ (fig. 10).

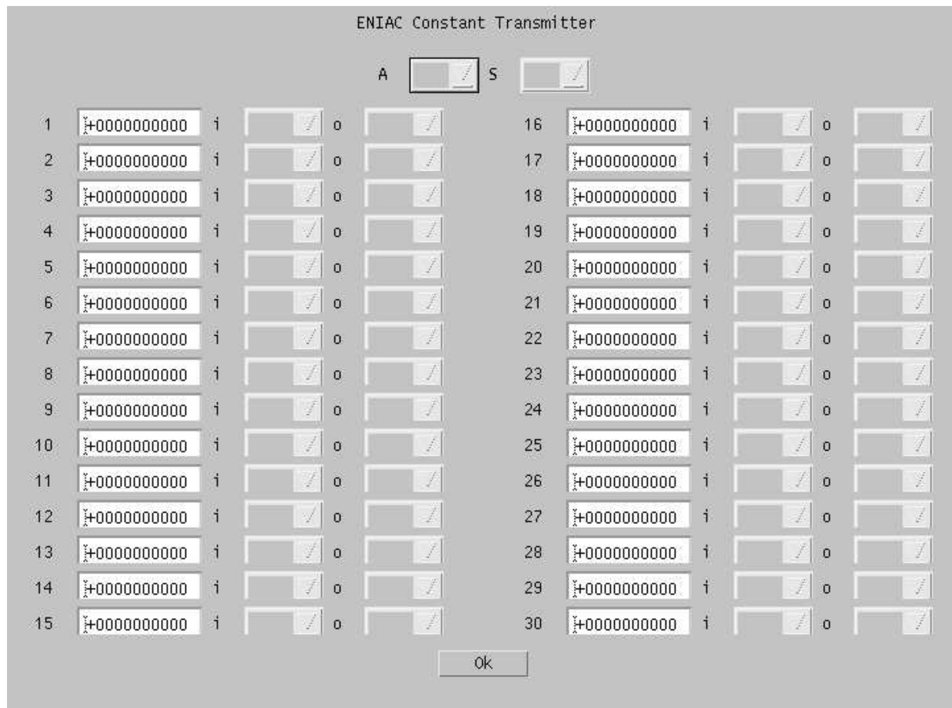

Figure 10: The constant transmitter window.

4 Java Implementation

4.1 Organization of Source Code

The whole simulator is built upon the *model-view-controller* design pattern [7]. One-to-one modelling of the implemented ENIAC hardware building blocks is done in the package `eniac.model`. Package `eniac.view` provides graphical user interface classes, which often stand in a one-to-one relationship to the `eniac.model` classes. Classes in the package `eniac.control` are finally responsible for global interactions of all classes (fig. 11).

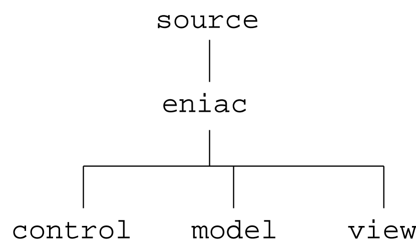

Figure 11: Source tree.

4.1.1 The Makefile

To provide fast development cycles, a Makefile was created. `make` is a tool to automatically compile only those files in a large source tree that have changed [8]. The following targets are supported (Table 1).

Table 1: Makefile targets.

`all`	compile all known `.java` files in the source tree using `javac`, and run a script called `publish`, which copies all `.class` files to a seperate directory tree and applies the `jar` command to generate a web-ready `.jar` archive. Furthermore, the script provides for local (`http://localhost/eniac`) access to the ENIAC web page using the `/var/www/eniac` directory, if available.
`archives`	execute target `clean`, and produce `.zip`, `.tar.gz`, and `.tar.bz2` archives for the download section. This target runs a script called `archives`.
`clean`	remove all `.class` files from the source tree.
`doc`	create Javadoc documentation of source tree in directory `doc`.
`docclean`	remove contents of `doc` tree.

4.2 Package `eniac.model`

4.2.1 Class `eniac.model.Eniac`

This is the central class in the `model` directory. It begins with a definition of constants such as the names of the parts of the machine and the numerical value of a CPP (-1). On creation of an instance of this class, 23 units are created, which are the initiating unit, 20 accumulators, the constant transmitter, and the printing unit. The simulator also creates five digit trunks (`I` ... `V`) and ten program trunks (`A-1` ... `A-10`). The number of program trunks can easily be changed to suit your needs.

Like most of the classes, `eniac.model.Eniac` provides data access methods for its units and trunks, some of which are able to find units by name.

`eniac.model.Eniac` keeps track of all sending action of units, which have to use the method `send()`. The machine is stopped if there is no more sending activity, and no repetitive program switch is doing repeat cycles.

The central method is `start()`, which sends `init` a CPP when a calculation is to be started. It then goes through the predefined number of cycles, which is `Long.MAX_VALUE` ($9.223 \cdot 10^{18}$) if no limit is set. Every machine cycle consists of two half-cycles, as on the original ENIAC. In the first

half of the cycle, digit transmission takes place through call of the `run1()` method of all units. CPP sending is done in the second half-cycle, when all `run2()` methods are invoked. As mentioned above, activity in the machine is obligatory to keep this main loop going.

4.2.2 Class `eniac.model.InitUnit`

On invocation of its `run2()` method, this class send the first CPP to all connected units. `init` has only one program output port, which is called `1`.

4.2.3 Class `eniac.model.PrintUnit`

The original ENIAC printer was a little different from this implementation. `print` accepts a CPP on its only input port `1`, prints the contents of all accumulators (method `dumpAccus()`), and sends a final CPP on its output port `1`.

In addition, it provides the usual data access methods, and a method called `dump()`, which can print a core dump of the whole machine if called from anywhere. It is invoked by pressing the *dump now* button, or automatically during computation processes if *core dump* is set.

4.2.4 Class `eniac.model.ConstUnit`

Another deviation from the original: All 30 fields of the constant transmitter are directly settable, as there is no equivalent to the punched cards in the simulator.

The constant transmitter consists of two output ports, A and S, and 30 `CFields`, constant fields with program input and output connections.

On reception of a CPP, which is detected in the first half-cycle, the content of the appropriate constant field is sent on both digit ports, `A` and Ŝ. The process is completed by a CPP through the program output port of the same field during the second half-cycle in method `run2()`.

4.2.5 Class `eniac.model.AccuUnit`

Finally, the simulation of the ENIAC's accumulators. An accumulator consists of a `Content`, i. e. the number it stores. There are the five digit input ports, `alpha` ... `epsilon`, and the two digit output ports, `A` and `S`. Furthermore, there are twelve program switches, `1` ... `12`, of which the last eight are repetitive ones, and a significant digits switch.

On call of method `run1()`, the accumulator unit first checks if it is in repetition mode. If yes, the virtual machine is instructed not to stop execution, even if there is no momentary CPP flow—after the last repetition, the repeating switch will send a signal, which is able to continue calculations.

The action stated by the active program switch is executed, if it is a sending task (`A, AS, S`).

If we are not in repetition mode, the unit checks for multiple CPP reception, which would be a “bug in the wiring” [2, p. 142]. Otherwise, it executes `A, AS,` or `S` commands.

Method `run2()` begins with the same check for repetition mode, but now it is the turn for receive commands (`alpha` ... `epsilon`) in both modes. On reach of the last repetition, the switch can send a CPP on its output port. The input register (arrival point of CPPs) is cleared, and the content as well according to the significant digits switch, if the clear-correct switch is set.

4.3 Package `eniac.view`

4.3.1 Class `eniac.view.EniacApplet`

This is the first class to be run during a session with the ENIAC simulator. It initializes the field `eniac.control.Controller.myEniacApplet` with the applet itself (`this`), and creates a new instance of an `eniac.model.Eniac`. As described there, the constructor of `Eniac` creates all virtual machine parts, units and trunks.

The rest of the class creates the visual appearance of the applet, the buttons for all units, the text area, and the control buttons. Active buttons are provided with an `ActionListener`, which opens an appropriate window, or runs a method of `eniac.control.Controller`, if the button is pressed.

4.3.2 Unit Windows

`EniacApplet` buttons for accumulators, and the other units, are instances of `eniac.view.components.UnitButton`. If pressed, the `ActionListener` of this class calls a method `getUnitWindow()` of class `UnitWindowFactory`. We have here an application of the Factory design pattern; dependent on the type of unit button, the appropriate window is opened.

For the different types of units, there are special classes (`AccuWindow, ConstWindow,` etc.), that contain the GUI elements needed to control the various units.

4.3.3 Class `eniac.view.EditWindow`

This class contains the code to show the internal state of the virtual machine in ENIAC simulator source code, using the method `eniac.control.Controller.show()`. User changes of the code are acknowledged after pressing the *Apply* button: The virtual machine is reset, and then fed with the code, using method `eniac.control.Controller.input()`.

4.3.4 Class eniac.view.OpenWindow

This class lets the user select one of the example programs located in the programs folder. Fully qualified name is given as an argument to Controller.load, which in turn calls Loader.load().

4.4 Package eniac.control

4.4.1 Class eniac.control.Controller

This central class is used whenever there is communication between classes in different packages, to have a unified interface. Furthermore, it provides methods for resetting the virtual machine (initialClear()), and printing in the central GUI text area (print(), println()).

4.4.2 Class eniac.control.CPPException

An instance of this exception is thrown if there is more than one Cpp reception on a program input port at a time. This means that there is an error in the connections. eniac.control.CPPException is the only custom exception defined so far, but pre-defined exceptions, such as java.lang.ParseException, and java.io.IOException, are for example used in eniac.control.Loader to signal problems.

4.4.3 Class eniac.control.Display

Here we find the method show(), which forms a long String, which contains the complete description of the current virtual machine state. This value is used in eniac.view.EditWindow (eniac.control.Controller.show() calls eniac.control.Display.show()).

4.4.4 Class eniac.control.Loader

This class contains two similar parts, one for reading virtual machine code from a file, the other for reading an argument string. Parsing is done by the methods load(), or input(), which in turn call methods parseContent(), parseConnect(), and parseAction(), to process the three blocks of the machine code source files.

5 The Simulator's Programming Language

As a tribute to modern technology, programming the simulator is not restricted to building different wirings using the GUI, but it can also be programmed with a special language that was developed during the design process. In the following paragraphes, syntax is shown using EBNF [9], [10]. An example program for calculation of modulo divisions brings the language to life.

A local copy of the simulator can be equipped with new programs in the `webcontent/programs` directory. They have to be plain ASCII files, and should bear sensible names. Execution of the Makefile is needed for the simulator to recognize new programs in the `programs` directory.

5.1 The Syntax

Every program for the ENIAC simulator consists of three major parts, the `content`, `connect`, and `action` blocks, which are enclosed in HTML-like tags. Comment lines have to start with a hash sign (`#`) or a double hash (`##`) for printing comments. Printing comment lines will be output on the ENIAC output screen during compilation. They can be used for short explanations of what the program does, and which accumulators it uses (try the example program files).

The reserved words `content`, `connect`, and `action` can easily be changed to suit different moods of the programmer (cf. class `eniac.control.Loader`).

5.1.1 The `content` Block

The `content` block states initial values for the accumulators and the fields of the constant transmitter. Its syntax is

```
content_block ::=
  "<content>" ↩
  { content_line }
  "</content>" ↩

content_line ::=
  accu_name → long_value ↩

content_line ::=
  const_name → const_field
  → long_value ↩
```

(→ means a literal tabulator, ↩ means a literal newline character. For unit and trunk names, see Table 2.)

Table 2: Unit names.

Unit	unit_name
initializing unit	init
accumulators	accu 1 ... accu 20
constant transmitter	const
printer	print
digit trunks	I, II, III, IV, V
program trunks	A-1 ... A-10

5.1.2 The connect Block

Wired connections between units or between different ports on the same unit can be expressed in the connect block. As on the original machine, a trunk must be used for every interconnection. Thus, we have to specify a sender (unit and output port, or trunk) and a receiver (trunk, or unit and input port) to establish a connection.

```
connect_block ::=
  "<connect>" ↩
  { connect_line }
  "</connect>" ↩

connect_line ::=
  unit_name → port_name
  → trunk_name ↩

connect_line ::=
  trunk_name → unit_name
  → port_name
  { → unit_name → port_name } ↩
```

5.1.3 The action Block

The action block is exclusively for programming switch settings on the accumulator units. As there are repetetive (5–8) and non-repetitive program switches (1–4), the syntax is varying. Every switch has a name (its number), and states an action (expressed by alpha ... epsilon, NOP, A, AS, or S). Repetitive program switches take another two optional arguments, the repetition count, and clear-correct switch setting ("cc" means the switch is set).

```
action_block ::=
```

```
  "<action>" ↩
  { action_line }
  "</action>" ↩

action_line ::=
  accu_name → switch_name
  → action_name
  [ → repeat_count ] [ → "cc" ] ↩
```

5.2 A Real-life Example: `modulo`

To illustrate the task of programming the simulator using the language described above, we want to develop an ENIAC program that calculates the result of a modulo division. It takes two values $a_7 > a_8 > 0$ as arguments (to be given as initial values of accumulators 7 and 8, respectively), and stores the result $a_7 \bmod a_8$ in accumulator 9.

The calculation is done by repeatedly subtracting a_8 from a_7, until the result gets negative. Then a final addition of a_8 is done, which yields the positive result (Table 3). In case of the result being zero, the virtual machine just stops and prints the number of cycles it had to run.

Table 3: Pseudo code of modulo algorithm.

```
a7 ← 9663943468
a8 ← 72894
a9 ← 4712

a9 ← 0
a9 ← a9 + a7

label sub:
a9 ← a9 − a8

if (a9 > 0)
   goto label sub
if (a9 < 0)
   a9 ← a9 + a8
   dump accumulators' contents
   halt
if (a9 = 0)
   halt
```

Now we can easily start with the `content` block:

```
<content>
accu 7   9663943468
accu 8   72894
accu 9   4712
</content>
```

Every ENIAC calculation starts with the initiating unit, `init`, which sends the CPP when its button is pressed. We begin by connecting `init` to trunk `A-1`.

```
<connect>
init    1       A-1
</connect>
```

This first CPP is used to clear the contents of accumulator 9 (4712) to zero before starting the calculation.

```
<connect>
A-1     accu 9  5
</connect>

<action>
accu 9  5       NOP     cc
</action>
```

CPP leaves accumulator 9's program switch after the clearing action and is fed via trunk `A-2` into accumulators 7 and 9 to transfer a_7 to a_9: $a_9 \leftarrow a_7$.

```
<connect>
accu 9  5       A-2
A-2     accu 7  5       accu 9  6
accu 7  A       I
I       accu 9  alpha
</connect>

<action>
accu 7  5       A
accu 9  6       alpha
</action>
```

Execution on accumulator 7 triggers negative transmission of a_8 to a_9: $a_9 \leftarrow a_9 - a_8$. Reception is done on accumulator 9 through switch 7 and input port β, which in turn is triggered by the execution of switch 6 via trunk `A-4`.

```
<connect>
accu 7  5       A-3
A-3     accu 8  1
accu 9  6       A-4
A-4     accu 9  7
accu 8  S       I
I       accu 9  beta
</connect>

<action>
accu 8  1       S
accu 9  7       beta
</action>
```

Accumulator 9/switch 7 execution leads to a CPP to switch 1 via trunk `A-5`. Here is the central point of the algorithm: Accumulator 9 sends its contents a_9 through A and S ports, i. e. a_9 and $-a_9$, via trunks `A-7` and `A-6` to switch 2 and accumulator 8/switch 5, and switch 7 and accumulator 8/switch 1, respectively. This is "creative" use of digit signals instead of a CPP to do conditional branching: A negative number as input on a program input port will act exactly as a CPP would, whereas a positive number is simply ignored. Now, depending on a_9 being positive or negative, the flow of action takes different ways.

Case $a_9 > 0$: the S port signal triggers another subtraction ($a_9 \leftarrow a_9 - a_8$).

Case $a_9 < 0$: the A port signal forces a final addition of a_8 through port γ, to return to positive result values ($a_9 \leftarrow a_9 + a_8$).

Case $a_9 = 0$: the virtual machine stops, because there is no more CPP flow.

```
<connect>
accu 9  7       A-5
A-5     accu 9  1
accu 9  S       A-6
A-6     accu 8  1       accu 9 7
accu 9  A       A-7
A-7     accu 8  5       accu 9 2
</connect>
```

```
<action>
accu 9  1       AS
accu 8  5       A
accu 9  2       gamma
</action>
```

Finally, accumulator 8 sends a Cpp to the printer via trunk `A-8`, which triggers a dump of all accumulators' contents.

```
<connect>
accu 8  5       A-8
A-8     print   1
</connect>
```

The complete listing might look like this:

```
<content>
accu 7  9663943468
accu 8  72894
accu 9  4712
</content>

<connect>
init    1       A-1
accu 7  A       I
accu 7  5       A-3
accu 8  A       II
accu 8  S       I
accu 8  5       A-8
accu 9  A       A-7
accu 9  S       A-6
accu 9  5       A-2
accu 9  6       A-4
accu 9  7       A-5
I       accu 9  alpha   accu 9  beta
II      accu 9  gamma
A-1     accu 9  5
A-2     accu 7  5       accu 9  6
A-3     accu 8  1
A-4     accu 9  7
A-5     accu 9  1
A-6     accu 8  1       accu 9  7
A-7     accu 8  5       accu 9  2
```

```
A-8     print   1
</connect>

<action>
accu 7  5       A
accu 8  1       S
accu 8  5       A
accu 9  1       AS
accu 9  2       gamma
accu 9  5       NOP     1       cc
accu 9  6       alpha
accu 9  7       beta
</action>
```

5.2.1 Division

The `modulo` algorithm can be used to calculate divisions, as the simulator always prints the number of cycles it ran. If we are looking for $\frac{a_7}{a_8}$, we can calculate it as following (let n be the number of cycles):

Case $(a_7 \bmod a_8) = 0$: $\frac{a_7}{a_8} = \frac{n-3}{2}$

Case $(a_7 \bmod a_8) \neq 0$: $\frac{a_7}{a_8} = \frac{n-6}{2}$

6 Comparison between Simulator and Original

The original ENIAC ran on a clock rate of 100 kHz. An addition time consisted of 20 pulse times, so it was able to perform 5 000 additions per second. The example program `modulo` needs 265 156 addition times for its execution, which would lead to an overall execution time of around 53 s (fig. 12).

The simulator, running on a Duron 1300 MHz, calculates the result in 2.6 s. The 13 000-fold higher clock rate of the modern computer leads to a performance gain of factor 20.

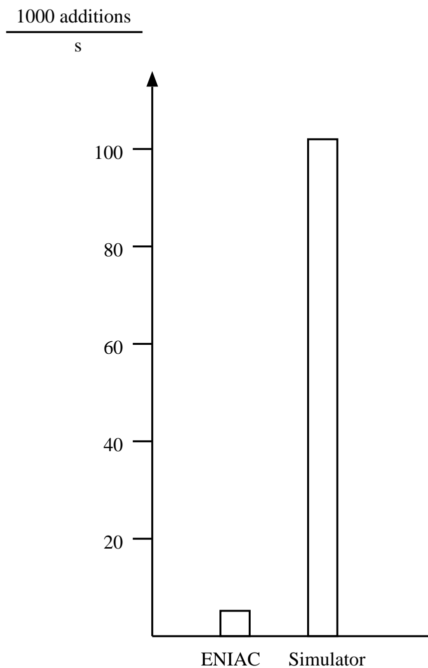

Figure 12: Comparison: Number of additions per second.

7 Room for Further Research

As we have seen, the simulator provides fairly good representations of the basic features of the ENIAC, so that new programs can be developed, and old ones tested. Using the new `modulo` program, one could for example implement the famous *Collatz* algorithm on the ENIAC simulator (see Table 4 and [11]). I will now name some areas where software extensions could be made.

- Low-level hardware properties of the ENIAC are not represented yet. There is no real cycling unit, and therefore no digit and program control pulses (1P, 2P, 4P, etc.) – in the course of a calculation, numbers are transmitted as `long` values in the simulation.
- A master programmer, i. e. a high-level program control has to be written.
- Other units are not implemented, as the function tables, the divider/square rooter, and the multiplier.
- The IBM punch card reading and printing system is not implemented.
- A possibility to connect two accumulators to yield results with a precision of 20 digits could be provided.

Table 4: The *Collatz* algorithm.

```
z ← 0
read n > 0

while n ≠ 1 do
   if n mod 2 = 0: n ← n/2
   else: n ← 3n + 1
   z ← z + 1
done

print z (number of iterations before n gets 1)
halt
```

- The loading of accumulators with initial values is a simplification, as the only possibility to input data was to use either the constant transmitter or the funtion tables.
- The initializing unit provided step-by-step execution for debugging purposes.
- The ENIAC's static interconnection lines are not present in the simulator, as data transfer to the multiplier unit is not needed. Also, the printer can access the contents of all the accumulators, and not only those connected via static lines.
- There are no adapter cables to influence the flow of digit signals, as would be required for shifting or deletion of digits.
- Live "neon lamp" displays could be programmed to show the contents of the accumulators during a running calculation.

The parallel processing capabilities of the ENIAC were not used extensively the original machine, partly because of the poor reliability of vacuum tubes: Massively parallel calculations would have greatly increased the probability of system failures. With the simulator, more complex and parallel programs can be developed and run, which can lead to a more profound understanding of the inherent parallel power of the ENIAC architecture.

8 Summary

Many of the complex features of the original ENIAC computer are now available to the public in the form of a Java simulation. It is now possible to gain a decently deep insight into the laborious task of developing programs for this machine by using the simulation, which provides a pretty faithful representation of the implemented units.

This paper can serve as a user manual for the simulator, and it can be a starting point to deeper exploration of the ancient hardware. Several possible extensions to the current Java program have been proposed, and the inner structure of the source code is open for new features.

```
1-5     +0000000000   +0000000000   +0000000000   +0000000000   +0000000000
6-10    +0000000000   +9663943468   +0000072894   +0000021418   +0000000000
11-15   +0000000000   +0000000000   +0000000000   +0000000000   +0000000000
16-20   +0000000000   +0000000000   +0000000000   +0000000000   +0000000000

VM came to a halt in cycle 265156
```

References

[1] P. Hansen, *A Java Simulation of the ENIAC*, (The simulator source code, this paper, and the running Java applet itself)
`http://home.arcor.de/-ph/eniac/`

[2] J. Van der Spiegel, J. F. Tau, T. F. Ala'ilima, and L. P. Ang (2000). The ENIAC: History, Operation and Reconstruction in VLSI. In R. Rojas (Eds.), *The First Computers – History and Architectures*, MIT Press.

[3] J. Van der Spiegel, *ENIAC-on-a-Chip.*
`http://www.ee.upenn.edu/~jan/eniacproj.html`

[4] *Konrad Zuse Internet Archive*
`http://www.zib.de/zuse/English_Version/`

[5] M. Campbell-Kelly, *The Edsac Simulator.*
`http://www.dcs.warwick.ac.uk/~edsac/`

[6] M. Campbell-Kelly. Past into Present: The EDSAC Simulator. In R. Rojas (Eds.), *The First Computers – History and Architectures*, MIT Press.

[7] Computer Science Department at University of Oberlin on MVC Design Pattern.
`http://exciton.cs.oberlin.edu/JavaResources/DesignPatterns/MVC.htm`

[8] The GNU Make home page.
`http://www.gnu.org/software/make/`

[9] L. M. Garshol, *BNF and EBNF: What are they and how do they work?*
`http://www.garshol.priv.no/download/text/bnf.html`

[10] M. Kuhn, *A Summary of the ISO EBNF Notation.*
`http://www.cl.cam.ac.uk/~mgk25/iso-ebnf.html`
featuring R. Scowen, *ISO 14977.*
`http://www.cl.cam.ac.uk/~mgk25/ISO-14977.pdf`

[11] E. W. Weisstein (1999). *The Collatz Problem*
`http://mathworld.wolfram.com/CollatzProblem.html`

[12] H. H. Goldstine and A. Goldstine (1946). The Electronic Numerical Integrator and Computer (ENIAC). In B. Randell (Eds.), *The Origins of Digital Computers*, Springer-Verlag (1982).