# Game-like Visualisation of the Analytical Engine

Rainer Glaschick, Paderborn, Germany
(rainer@glaschick.de)
http://rclab.de/analyticalengine
2015-12-06

This is a quick translation (and extension) of a German text, that was the basis for an interactive program to visualise the *Analytical Engine* (AE) in the exposition *Am Anfang war Ada. Frauen in der Computergeschichte* (2. September 2015 bis 10. Juli 2016, Heinz Nixdorf MuseumsForum, Paderborn, Germany).

In the game inspired interactive program, virtual punched cards are used to represent orders, are created interactively, and can be executed as a batch.

Basic knowledge of the AE and its workings are assumed for this text (not for operating the app), in particular Menabrea's and Lovelace' *Sketch of the Analytical Engine* [Sketch].

With this kind of excercise, there is always the danger to project knowledge and assumptions backwards, and to serve today's experience in order to make the app better understandable and useable. This is mentioned when evident, but may have been overseen at other places. Note that the app as implemented may deviate even in significant aspects from what is written here.

To make it easier readable, the text presumes that an AE had existed. As common today, the application to visualise the AE is just called the *app*.

# Functional Structure

While it is clear, that punched cards were planned to control the calculations of the AE, there is not much detail on this part of the machine available.

There were three kinds of punched cards:

- operation cards, that controlled what operation the mill would perform
- variable cards, that controlled which variable was fetched from or stored into the store,
- number cards, to provide numerical constants

The AE had a second (*primed*) input and output axis. Each multiplication produced a double-precision result. The division required two numbes for the divisor, and the remainder was available together with the quotient.

The app does not support the primed axes; the division provides only the quotient. If the remainder is required, it must be calculated by subtracting the product of quotient and remainder.

Operation cards and variable cards are mixed in the same batch in the app, in particular, as no proposal for the synchronisation of two batches has be availble. This leaves the question of the position of the operation card. The *Sketch* emphasises that operation cards need only be changed if a different action in the mill is required. So the action of the mill would be started while or after the second operand was loaded. (In contrast to the Z3, this is not a stack, as the result can only be reused via a

transfert to the store). Using the operation card as a postfix operator would be naturally from today's view, but this is clearly not historically justified.

As the operations card sets the function of the mill, and controls the microprogramming sequences, it is sensible to assume that that microprogramming also controls the loading of operands, and expects variables cards in the proper order. This demands the the operation card preceeds the variable cards that load the operands. In this sense, there is no need to provide a new operation card after a calculation, the mill would just do the same operation with other operands, and always consume three variable cards.

After the operation, the result is transferred to the store by using a variable card. The *Sketch* assumes that the result can be transferred to more than one variable. As in the early versions, a 1-of-n coding of variables was used, this would be easily done by just punching more than one hole, if the variable card was to provide the variable to store the result. A general solution would be to have two kinds of cards, those loading operands and those storing operands, as the app does. However, storing a result into more than one variable is seldomly useful, and thus may not be present in the app.

A different question is whether there must always be a third (store) card in an operation cycle, or if the result of a calculation could be discarded. If not directly implemented as an operation, the result could always be stored to a dummy variable, even if this requires additional effort to clear this variable later.

While the app was programmed, the distinction of *retaining* and *erasing* (not retaining) variable fetches became apparent to me. Furthermore, I am convinced now that sending a result to the store would required that the receiving location is zero. (The machine may either stop, as Babbage planned for several *assertions*, but may also add digitwise.) This means, that — as stated in the *Sketch* — the erasing fetch would always be used if the value would not be required any longer, which is the case if in today's architectures the variable would be overwritten.

Having the app to stop with an error message if a variable is to be overwritten when non-zero, would be the correct solution. However, it would make the use for museum visitors more complicated, as starting a simple batch again would require to zero the result first (see below).

The AE had a *run-up lever*, that was set as follows:

- addition: overflow, or sign of result differs from first operand
- subtraction: underflow, or sign of result differs from first operand
- multiplication: not set
- dividion: quotient too large (overflow), or divisor is zero

Instead of *run-up lever*, the term *signal* is used from now on.

The signal was set freshly by each operation, i.e. reset when the operation started. It is still unclear, how it was set when two equal negative numbers were subtracted. If a (positive) zero was created, the signal would be raised, although there was no overflow. While it is clear that the machine — with its sign and magnitude representation of numbers — had a plus and minus zero number, it is probable — considering Babbage's perfectionism — that minus zero results were automatically converted to a plus zero, and perhaps the signal set before. These aspects will not be shown by the app.

In any case, the sign of a variable's content could be determined be subtracting it from zero (a variable with zero content), and branching dependent on the signal.

As concerns the division, here the signal is not only set by an attempt to divide by

zero, but also if the remainder is not zero and discarded in the app.

Conditional branching is very sketchy described in the *Sketch*, and in particular how different batches of operation and variable cards are coordinated. This is another reason to use a single batch of cards, as was already done by [Walker].

For number cards in a third batch, there are few hints how these were operated. Babbage wanted to ensure the correct card to be used, and assumed that each number card provided a function value for an argument which was also coded on the number card. However, this still requires in the program an operation to fetch the next number card, to provide the argument, and either to input it to the mill or store it into a variable.

Number cards were considered not necessary in the app, as it allows to change the content of the variables directly. When adopting the rule that overwriting non-zero variables is not allowed, the scene changes a little. Now it would be useful to provide number cards in the batch, to set the input variables and clear all working and output variables.

Note that clearing all variables with a single command is easy in electronic computers, but likey not be provided in the AE, as it would require much power if a thousand wheels had to be turned to zero at the same time.

# Symbolic Program

John Walker ([Walker]) has used a symbolic notation for the cards, which will be followed here. In particular, variable cards are differentiated between those which transfer to the mill (*fetch*) and from the mill (*store*), which is historically not certain.

Each card has a fixed number of parameters, thus a line change between cards is not necessary to terminate the parameters. Special charcters not mentioned are ignored, so the second parameter can be prefixed by an equals sign for number cards.

A comment upto the end of the line is started with a single point (fullstop), i.e. surrounded by blanks (within numbers, it would be a decimal point, but this is not used in the examples so far.)

The cards are symbolised as follows, where capical letters are used verbatim, and small letters are parameters:

Number card: `Ni z`
: sets variable $V_i$ by the number $z$

Operand fetch and zero: `Zi`
: fetch operand $V_i$ and leave its variable store erased

Operand fetch retaining value: `Li`
: fetch operand $V_i$ and restore the number to the variable

Transfer: `Ti`
: store the current result to the variable $V_i$

Operation:
: is one of the symbols `+`, `-`, `*`, `/`.

The number cards are defined for the examples, even if not present in the app; they can be used if a configuration is to be saved.

Instead of `S` for *store*, `T` as in *transfer* is used, to avoid confusion with a subtraction.

Conditional execution as well as loops use a block notation that works well with bidirectional serial storage. Blocks are indicated with symbols `{` and `}` for begin and

end of a block. They may be properly nested.

The block begin symbol enters the loop, if the signal is inactive, and otherwise, if the symbol is active, skips forward behind the corresponding end symbol.

The block end symbol transfers control back after the corresponding begin symbol if the signal is inactive, and does nothing if the signal is active.

So you may read { as *enter unless signal is set* and } as *repeat unless signal is set*. As most operations leave the signal unset, the block is normally entered and repeated, unless a set signal prevents it. This is useful for a global loop in enumerations.

For the sake of simplicity, there is no alternative (*else*). Unless an operation to invert the signal is provided, the inverse condition has to be evaluated.

Properly nested loops are simple to implement: The mechanism just counts up begin symbols in forward direction and end symbols in backward direction, until a corresponding symbol is found when the count is zero.

As in the AE, there is no operation to negate a number; just subtract it from zero, for which a variable with zero contents should always be available, perhaps by using a variable which was cleared by an erasing load before.

To duplicate an number, it has to be added to zero; this is preferrably the target, which has to be zero anyhow, because overwriting non-zero variables is not allowed.

In particular with conditional blocks (see last example), it is necessary to clear a variable. Of the various methods, subtracting it from itself seems to be the most natural one.

While it was one of the primary aims of Babbage to print numbers to avoid readout errors, the way to do it in the AE is not well known. I see these options:

- a special operation card that fetches just one operand,
- use a special variable that, when a number is stored, prints that number.

The first choice would introduce a unary operation, but the AE as described in the *Sketch* has no such operation e.g. for sign inversion. So we adopt the second case and denote the special variable with an exclamtion mark, i.e. ! instead of Tx, where x would denote the printing variable. This could be an optional hole on a variable card. For simplicity, numbers are assumed to be printed in the same line, unless a block end is encounted. This may be refined later.

There is no indexed memory access as in the AE known so far. Thus, neither the automatic calculation of Bernoulli's numbers nor enumerating prime numbers by keeping a list of already found prime numbers is possible.

The preferred notation sets the operator first, then the two obtaining and the one storing variable card. If it is allowed to delay the operation card, the following two lines are equivalent:

```
+ L1 Z2 T3
L1 + Z2 = T3
```

but this, while convenient, camouflages the architecture, and thus will not be used.

# Examples

# Triangle Numbers

The sum of the first $n$ natural numbers:

$$t(n) = \sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

is called the n-th triangle number, because it is the number of tiles that could be arranged in a triangle with equal sides of length $n$.

Program code:

```
. V1: argument
. V2: triangle number
. V3: constant 1
N1=1
N2=1
N3=1
{
  + Z1 L3 T1 !      . increase argument and print
  + L1 Z2 T2 !      . calculate result and print
}
```

The second operation card + is, according to the *Sketch*, not necessary, but shown here and further on for clarity.

# Square- and Cube Numbers

Using the first binomial theorem, the square numbers could be enumerated by adding the argument twice, and adding one:

$$(x+1)^2 = x^2 + 2 \cdot x + 1$$

Program code:

```
. V1: argument
. V2: square number
. V3: constant 1
N1=1
N2=1
N3=1
N4=0              . auxiliary zero variable
{
    + Z2 L1 T2    . +x
    + Z2 L1 T2    . +x
    + Z2 L3 T2    . +1
    + Z1 L3 T1 !  . increase and print argument
    + L2 L4 !     . print square number
}
```

Using the variant:

$$x^2 = (x-1)^2 + 2 \cdot x - 1$$

the argument can be incremented before doubled:

```
. V1: argument
```

```
. V2: square number
. V3: constant 1
N1=1
N2=1
N3=1
{
    + Z1 L3 T1 !
    + Z2 L1 T2
    + Z2 L1 T2
    - Z2 L3 T2 !
}
```

Because of:

$$(x+1)^3 = x^3 + 3 \cdot x^2 + 3 \cdot x + 1$$

the cube numbers can be calulated in the same loop. To increment the argument first, the above equation is rewritten:

$$x^3 = (x-1)^3 + 3x^2 - 3x + 1$$

This results in the following program:

```
. V1: argument
. V2: square number
. V3: cube number
. V4: constant 1
N1=1
N2=1
N3=1
N4=1
{
    + Z1 L4 T1 !      . increment argument and print
    + Z2 L1 T2        . add twice instead of multiplication
    + Z2 L1 T2
    - L2 L3 T2 !      . square number, printed
    + Z3 L2 T3
    + Z3 L2 T3
    + Z3 L2 T3
    - Z3 L1 T3
    - Z3 L1 T3
    - Z3 L1 T3
    + Z3 L4 T3 !      . cube number, printed
}
```

## Fibonacci Numbers

Fibonacci numbers are recursively defined:

$$f_{n+1} = f_{n-1} + f_n$$

Program:

```
. V1 f(n-1)
. V2 f(n)
. V3 f(n+1)
N1=1
N2=1
```

```
N3=0
{   + Z1 L2 T3 !     . calulate result and print
    . move V2 to V1, V3 to V2, and set V3 to zero
    + Z1 Z2 T1
    + Z2 Z3 T2
}
```

## Greatest Common Divisor

The greatest common divisor of two numbers is mathematically elegantly defined recursively:

```
ggt(x,y):
    ! x ≥ 0, y ≥ 0
    ? y > x
        :> ggt(y,x)
    ! x > y
    ? y > 0
        :> ggt(y, x % y)   \ % is the remainder
    :> x
```

The exclamation mark (`!`) is used for an *assertion* that the list of logical expressions that follows is always true. The question mark (`?`) stands for an *if*, and the dependent block is indented. The bigraph `:>` acts as a *return*, leaving the function with a return value. The backslash comments the rest of the line.

As the recursion is a tail recursion, it can be replaced by a loop. To make the example simpler, `y>x` is assumed (`?*` stands for *while*):

```
ggt(x,y):
    ?* y > 0
        z = x % y
        y = x
        x = z
    :> x
```

As our reduced model of an AE does not provide a remainder, it must be calculated by subtracting the product:

```
ggt(x,y):
    ! x > y, y >= 0
    ?* y > 0
        z = x // y      \ integer quotient
        z = z * y
        z = x - z       \ remainder
        y = x
        x = z
    :> x
```

Program:

```
. V1: x
. V2: y
. V3: z
. V4: 1
N1 48
N2 9
N3 0
```

```
N4 1
. assert V2>0
{
    / L1 L2 T3          . quotient in V3
    * Z3 Z2 T3
    - L1 Z3 T3          . remainder
    + Z1 L2 T1          . move (!) y to x
    + Z2 Z3 T2          . z to y, erase z
    - L2 L4             . repeat block while y>0
}
+ L1 Z3 !               . get x (z is zero) and print
```

## Collatz series

The Collatz conjecture claims that the following series always reach the number 1 and loops from that point on:

$$c(i) = \begin{cases} \frac{i}{2} & \text{if } i \text{ even} \\ 3i+1 & \text{if } i \text{ odd} \end{cases}$$

As in the second case the result is always even, a reduced version is:

$$c(i) = \begin{cases} \frac{i}{2} & \text{if } i \text{ even} \\ \frac{3i+1}{2} & \text{if } i \text{ odd} \end{cases}$$

In pseudocode:

```
collatz(i):
    ? i % 2 = 0
        :> collatz(i/2)
    :> collatz((3*i+1)/2)
```

A slight optimisation first calculates the truncated quotient `i//2`:

```
collatz(i):
    q = i // 2
    r = i % 2
    ? r = 0
        :> collatz(q)
    :> collatz(3*q+2)
```

For the last line, note that if $i$ is not a multiple of 2, then:

$$\begin{aligned} i &= 2q+1 \\ \frac{3i+1}{2} &= \frac{6q+3+1}{2} = 3q+2 \end{aligned}$$

Replacing the end recursion etc:

```
collatz(i):
    ?* i > 1
        q = i / 2
        r = q * 2
        r = i - r
        i = q
        ?  r > 0
            i = 3 * q
```

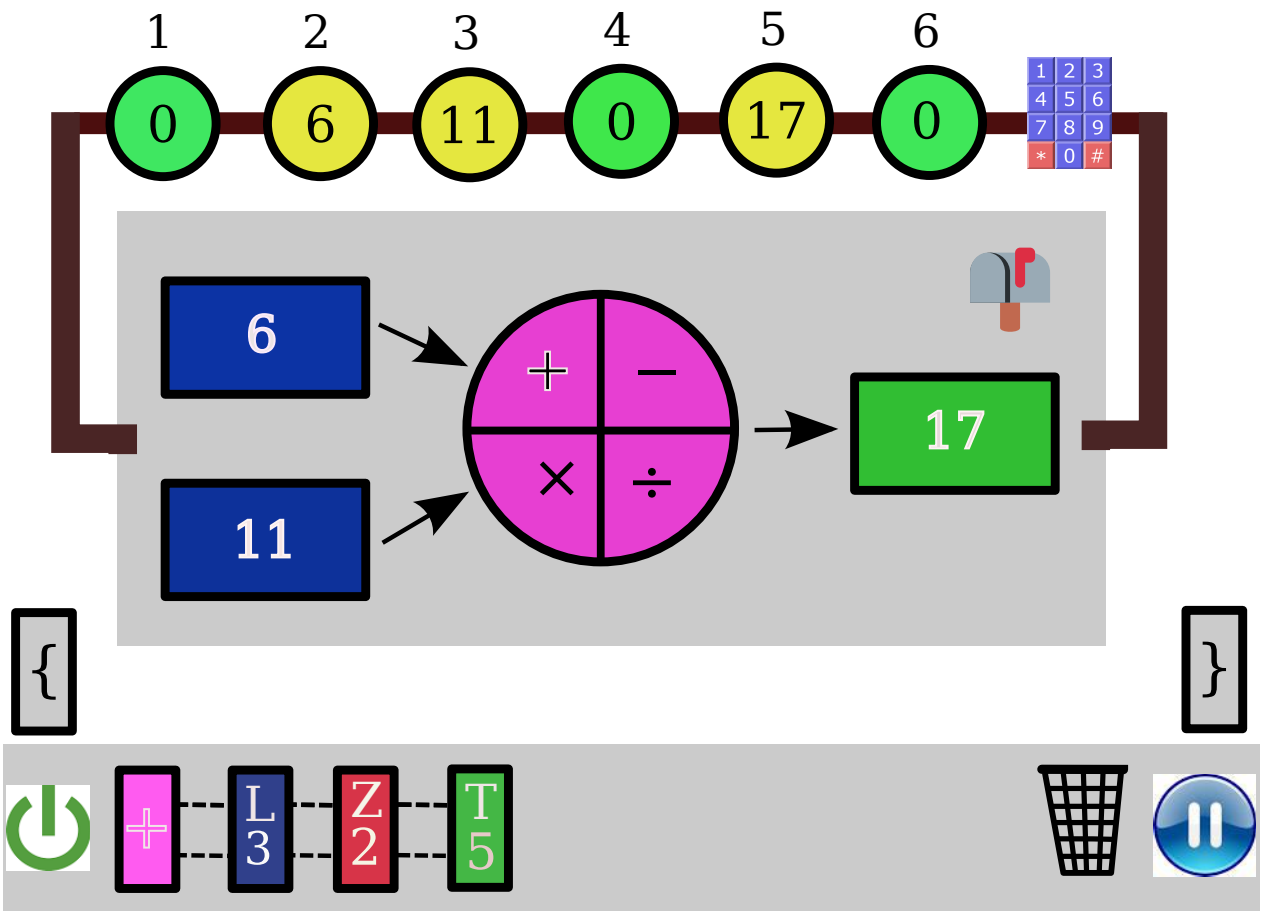
```
                i = i + 2
```

This produces the following program:

```
. V1: 1
. V2: 2
. V3: 3
. V4: i
. V5: q
. V6: r
N1 1
N2 2
N3 3
N4 55               . input value
N5 0
N6 0
{   . outer loop
    / L4 L2 T5      . integer truncted quotient
    * L5 L2 T6
    - Z4 Z6 T6      . remainder, erase V4
    + Z4 L5 T4      . V5 to V4
    - L6 L1         . enter block if r>0
    {
        - Z4 Z4 T4  . clear V4
        * L3 L5 T4
        + Z4 L2 T4
        - L1 L2     . raise flag to end condition
    }
    - L4 L2         . repeat block while i>1
}
```

# Graphical User Interface

The game-like user interface could be approximately like this:




As touch screens for casual users might give problems if a widget is dropped prematurely, no drag-and-drop onto another widget is used. Dragging a widget may be used if only the direction is relevant, or the dropped widget could be picked dragged again. All other interaction uses single touch (or click, if a mouse is used) except noted.

On the upper side, the variables are denoted as circles, with their name (address) above and the contents inside. Zero numbers are green to indicate that they can be overwritten. A nicer version would use vertially stacked numbers like in the AE. The *bus*, i.e. the shaft though which all send and receive numbers, connects them to the mill; only the distiction between various *ingrss* and *egress* shafts is not shown.

The variables can be set directly by touching the keypad first and then the variable, opening a box where to enter the number. If the variables are vertically stacked wheels, dragging them left or right might be used to set a digit.

Number cards are not used; all initial values must be entered directly.

In the middle is the mill, with a circle having four quadrants for the four operations possible, on the left the *ingeress*, on the right the *egress*.

At the bottom is an area for the assembled cards in a chain, a start and pause button to the left and right, and the symbols for block begin, block end and delete a card. The letters in the cards denote the type, L for retaining load, Z for erasing load, and T for transfer to the storage.

Cards can only be added (or removed) to (from) the (right) end of the chain:

- Touching an operation symbol generates an operation card (unless the same card is already at the end) and highlights the operation symbol.
- Touching a variable creates a variable card, where first the upper input field of

the mill is highlighted, then the lower one, and finally the output field, generating a transfer card.

During programming, the actions are performed as cards are added; i.e. the variable loading sets the input fields, and the transfer sets the variable. This also could issue a warning if the destination of a transfer card is not zero. On the other hand, after programming the initial values have to be re-entered again. So it might be useful to allow number cards, that are generated when a variable is set, but added at the front (replacing older ones for the same variable).

Immediate execution is somewhat misleading, as clearly program assembly would have been done without the machine as a desk work, but will help the casual user to understand what is happening.

If more cards are to be uses as there is space, scrolling must be used (not shown).

To handle the difference between retaining and erasing load, the erasing load is the default: touching a variable while a loading variable card is expected, moves the number to the next input field to be used and erases the source. Touching that input field sends back the number to the variable where it came from.

# Appendix

## Z3 Operation codes

The Z3 made by Konrad Zuse has a very small instruction set and a certain similarity to the AE, which was tempting to use for the AE. According to his son, Konrad Zuse learned about the AE afterwards, when he filed a patent for the Z3.

However, the Z3 is a stack machine with a postfix operation code that processes the arguments. Moreover, it is a floating point computer without loops and jumps.

The Z3 uses 8-channel punched tape, where two bits encode the instruction group:

```
01  Operation, details in the other bits
10  Store, adress in the remaining 6 bits
11  Load, adress in the remaining 6 bits
00  no action
```

The operations use only 4 bits to encode the operation:

```
0001    input from the console
1110    output to the console
0010    add
0011    subtract
0100    multiply
0101    square
0110    divide
0111    reciprocal
1000    square root
1001    negate
1010    double
1011    half
1100    multiply by 10
1101    divide by 10
1111    end of sequence (?)
```

# References

Ceruzzi1981:
Paul Ceruzzi: *The Early Computers of Konrad Zuse*. Annals of the History of Computing, Vol. 3, No. 3, July 1981, pp.241-262

Walker:
John Walker: *The Analytical Engine*. Online at http://www.fourmilab.ch/babbage/

Babbage:
H.P. Babbage: *The Analytical Engine*. Proceedings of the British Association (1888) Digital version online on http://www.fourmilab.ch/babbage/hpb.html

Sketch:
L.F. Menabrea: *Sketch of the Analytical Engine invented by Charles Babbage*, with Notes from A.A. Lovelace; Scientific Memoirs, Vol. III Part XII, London 1843, pp.666-731.
Facsimile online: https://archive.org/stream/scientificmemoir03memo#page/666