Software Lab 1 : Analytical Engine

The aim of this lab is to introduce the concept of an instruction and how these can be combined to produce a simple program. To illustrate these ideas a software simulation of Babbage's Analytical Engine will be used. This simulator is written in the Java programming language and runs within a web browser as a Java applet allowing both a Windows or Linux platforms to be used. Note, there may be some Java 'issues' under Linux.

The original website which contains this application and additional information on this computing machine can be found at:

`http://www.fourmilab.ch/babbage/applet.html`

However, for the purposes of this lab we shall be running the simulator locally. To start the simulator left click on the Windows icon in the bottom left of the screen and type :

```
-> The Analytical Engine
```

Note, A zip file containing all the required files can also be downloaded from the module web page (link under lab script). This has the advantage that you can continue these exercises at home on your own computer without having to be connected to a network.

This will launch the two simulator windows as shown in figure 1. This may take a few seconds to appear (approx 30s). The main window contains the user interface allowing you to select the "programming cards" you wish to execute, displays output results and error messages. It also allows you to initialise the computer `[reset]` execute one instruction at a time `[step]`, or to select continuous execution `[start]`, terminating when all programming cards have been executed.

The second window opened by this simulator is the Annunciator panel as shown in figure 2. This windows displays the current state of the three main functional blocks within the Analytical engine i.e. `Card-reader`, `Mill` and `Store`, as shown in figure 3.

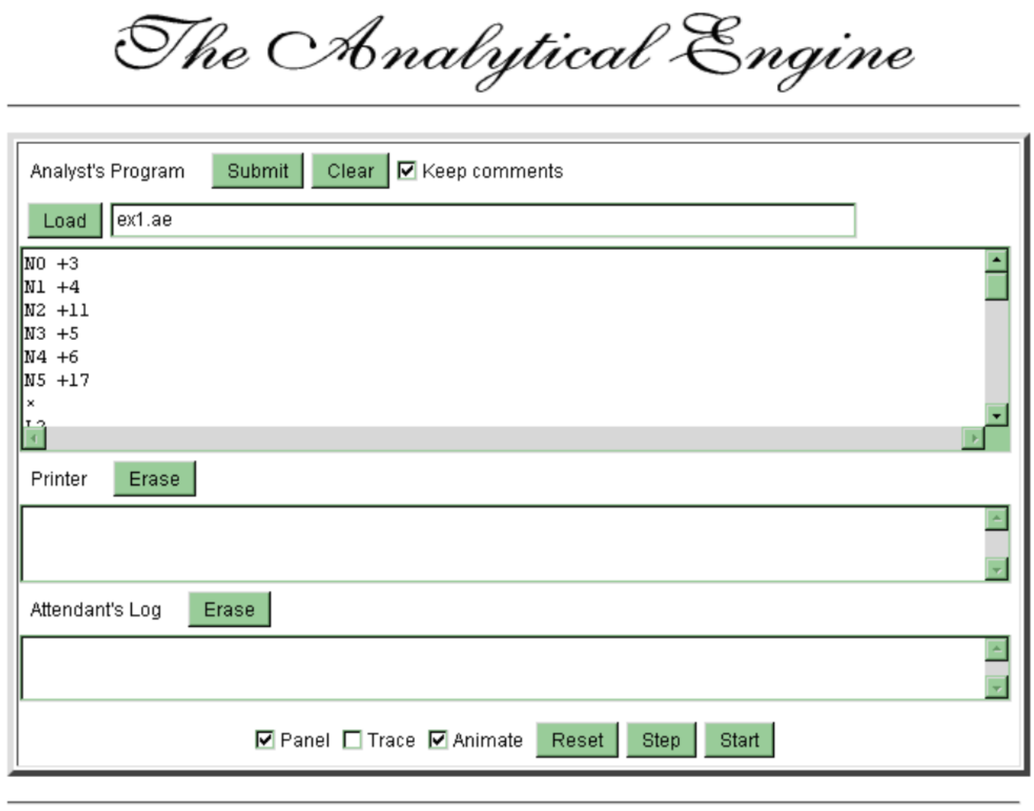
Task 1

By default the simulator will load the programming example shown in table 1 from the text file `ex1.ae`.

To initialise the simulator left click on the Load button within the main user interface window. The programming 'cards' used to implement this program are displayed in the user interface within the Analyst's Program panel. You can move up and down this chain to view these cards using the side scroll bar. The programming card to be executed next is highlighted in yellow on the Annunciator panel, within the blue card reader section. All calculations are performed in the `Mill`, with operand data stored in `Ingress` registers 0 and 1. Results from these calculations are stored in the `Egress` register 0.

Note, the `Mill` needs storage (memory, registers) to perform its calculations. Babbage called these locations `Ingress` for input data (operands, numbers to be processed) and `Egress` for output data (results, data produced by instructions).

The default storage size of the `Ingress` and `Egress` registers is 50 decimal digits. For larger operands and result data the prime registers `Ingress 0′` and `Egress 0′` are used to represent 100 decimal digit values i.e. the result from a multiplication, or division (quotient and remainder).

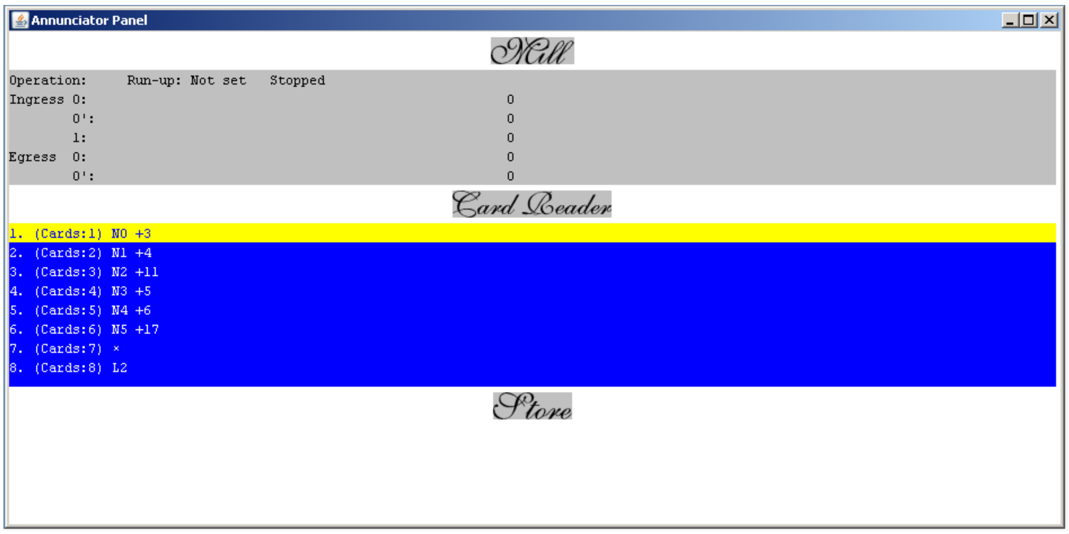


ex1.ae

```
N0 +3
N1 +4
N2 +11
N3 +5
N4 +6
N5 +17
×
```

Keep comments [✓], Panel [✓], Trace [ ], Animate [✓]

Figure 1 : Main user interface



```
Operation:      Run-up: Not set    Stopped
Ingress 0:                                      0
        0':                                     0
        1:                                      0
Egress  0:                                      0
        0':                                     0
```

```
1. (Cards:1) N0 +3
2. (Cards:2) N1 +4
3. (Cards:3) N2 +11
4. (Cards:4) N3 +5
5. (Cards:5) N4 +6
6. (Cards:6) N5 +17
7. (Cards:7) ×
8. (Cards:8) L2
```

Figure 2 : Annunciator panel

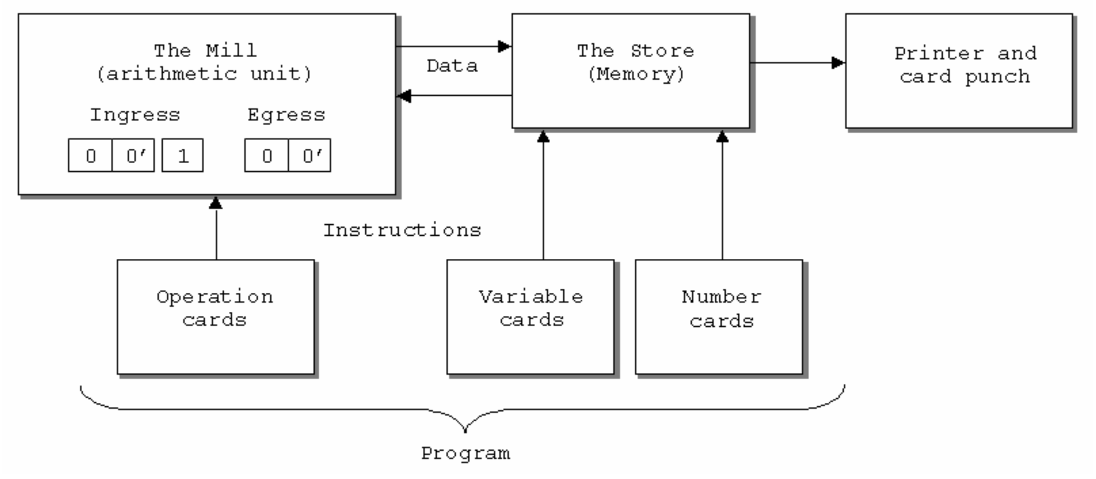
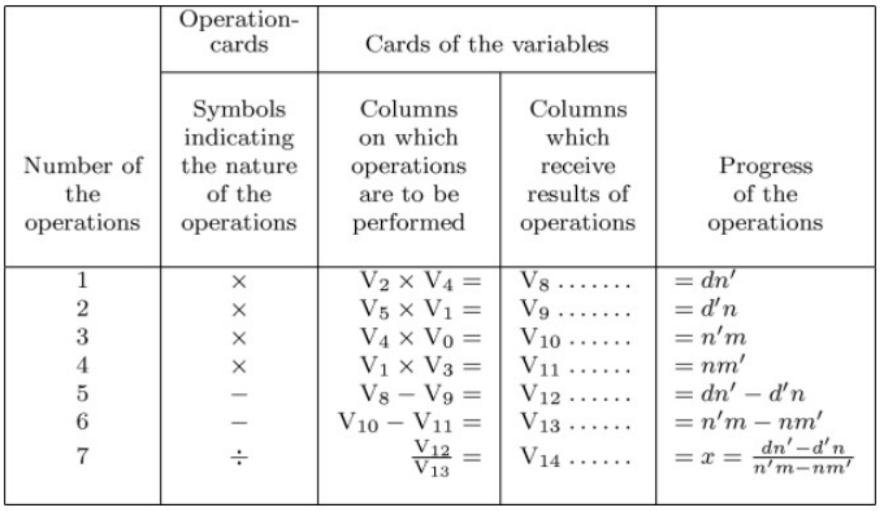


Figure 3 : computer architecture block diagram



Number of the operations	Operation-cards	Cards of the variables		Progress of the operations
	Symbols indicating the nature of the operations	Columns on which operations are to be performed	Columns which receive results of operations	
1	$\times$	$V_2 \times V_4 =$	V_8	$= dn'$
2	$\times$	$V_5 \times V_1 =$	V_9	$= d'n$
3	$\times$	$V_4 \times V_0 =$	V_{10}	$= n'm$
4	$\times$	$V_1 \times V_3 =$	V_{11}	$= nm'$
5	$-$	$V_8 - V_9 =$	V_{12}	$= dn' - d'n$
6	$-$	$V_{10} - V_{11} =$	V_{13}	$= n'm - nm'$
7	$\div$	$\frac{V_{12}}{V_{13}} =$	V_{14}	$= x = \frac{dn'-d'n}{n'm-nm'}$

Table 1 : programming example

Group	Symbol	Operation
Load	NX V	Initialise location X with value V X : 000 to 999 V : signed 32 digit decimal number
	LX	Transfer Store location X to Ingress (store value unchanged) X : 000 to 999
	ZX	Transfer Store location X to Ingress (store value zeroed) X : 000 to 999
Store	SX	Transfer Egress to Store location X X : 000 to 999
Arithmetic	+	Add, Ingress 0 + Ingress 1, result stored in Egress 0 If a carry is generated the run-up flag is set
	–	Subtract, Ingress 0 – Ingress 1, result stored in Egress 0 If a borrow is generated the run-up flag is set
	× (*)	Multiply, Ingress 0 × Ingress 1, result stored in Egress 0 and 0' Lower digits stores in Egress 0, upper digits in Egress 0'
	÷ (/)	Divide, Ingress 0, 0' ÷ Ingress 1, result stored in Egress 0 and 0' Quotient stored in Egress 0' remainder in Egress 0
Action	B	Ring bell

Table 2 : instruction set

When the current `Mill` arithmetic operation has been set (+,-,*,/) the `Ingress` registers can be loaded. This is achieve using the `LX` instruction, transferring data from the `Store` to the `Mill`. The first `LX` command transfers its data to `Ingress` register 0, the second to `Ingress` register 1. When both registers are loaded the arithmetic function is triggered, updating the specified `Egress` registers and reinitialising the `Mill` ready for the next `LX` data pair.

Note, a **variable** is a symbolic name (identifier) representing an unknown quantity of information (value) e.g. `A=B`. Within a computer variables have to be stored somewhere otherwise this information is lost, typically memory or registers.

Question, if the code fragment below performs the calculation `A=B+C`, where are `A`,`B` and `C` are stored in memory? What registers are used to store these values during the calculation?

```
+
L 300
L 101
S 002
```

Question, the `Mill's` arithmetic operation will remain the same until a new arithmetic programming card is issued. What is the advantages of this approach?

At this stage you should have a basic understanding of the computer's architecture (figure 3) i.e. what's connected to what, its instruction set (table 2) and the instructions (functions) it can perform.

The instructions currently loaded into the simulator (file `ex1.ae.`) implement the program described in table 1 and are used to solve the following simultaneous equations.

```
3x + 4y = 11
5x + 6y = 17
```

The algorithm used to solve these equations for `x` takes the form of

```
Ax + By = C
Dx + Ey = F
```

$$x = \frac{C \times E - F \times B}{E \times A - B \times D}$$

The operations required to solve this equation are broken down into a number of steps each implemented by an instruction shown in table 1. A full listing of this program is shown in Appendix A, figures A1 and A2.

Question, examine this code (figure A1) can you see how these instructions implement the above algorithm? Think how you would do this using pen & paper.

Task 2

Using the Step button, single step through the current program file `ex1.ae`. After each step examine the state of the `Ingress`, `Egress` and `Store`. Check that the intermediate results generated are as defined by the program shown in table 1.

Tip, within the main user interface tick the `Trace` box, this will create a list of all instructions executed and their results within the Attendant's Log panel.

Question, how is the result of the final division stored in the `Mill` i.e. what registers are used? Hint, refer to table 2, `+-` are different to `*/`, what are `Egress 0` & `0'` used for? A full description of each programming card can be found on the VLE.

Task 3

Edit the existing program `ex1.ae` (described below) to calculate the following equation

```
A = B + C − D × E
```

where variables `A`, `B`, `C`, `D` and `E` are stored in memory. You can choose any unused memory location i.e. 0 to 1000, using the following initial conditions:

```
A = 0   B = 1   C = 2   D = 3   E = 4
```

Note, variable `A` will be overwritten with the final result, so strictly speaking you do not need to set it to zero. However, its considered good practice to initialise all variables.

Editing a program : within the main user interface edit the text within the Analyst's Program panel directly. To update the machine with these modification left click on the Submit button. These changes can be saved to a text file by left clicking in this panel, pressing `CNTRL-A` to select all the text, `CNTRL-C` to copy. Then open Notepad by selecting:

```
Start -> All Programs -> Accessories -> Notepad
```

Then within Notepad press `CNTRL-V` to paste this text into the editor. You can now save these updates by clicking on the pull down menu:

```
File -> Save As
```

Note, you will need to update the 'File name:' text box to `*.*` to avoid adding an `.txt` file extension. To load a pre-saved program revise the previously described options i.e. select and copy the text from Notepad, then paste into the Analyst's Program panel.

Again single step through the program. After each step examine the state of the `Ingress`, `Egress` and `Store`. Check that the intermediate results generated are as defined by your program. Is the final value correct?

Task 4

In addition to using the multiplication programming card '×' an alternative implementation could be performed using repeated addition. Using this technique write a program to perform the following calculation, you may not use the '×' card:

```
A = B × 3
```

where variables `A` and `B` are stored in memory. You can choose any unused memory location i.e. 0 to 1000, using the following initial conditions:

```
A = 0   B = 10
```

Again single step through the program and confirm that the correct result is produced.

Question, why couldn't this program be used to perform the following calculation:

```
A = B × C
```

where `A`, `B` and `C` are variables stored in memory?

A full list of the programming cards used in the Analytical engine can be found on the module web page. Using the 'Stepping Up / Down' card modify your program to perform the following calculation, again you may not use the '×' card:

```
A = B × 1000
```

Hint, this is a base-10 machine, you will need to use the '+' card to save the result back to memory.

Task 5

Using the 'Combinatorial' card described in the online documentation rewrite your multiplication program to allow it to perform multiplication through repeated addition were both operands are stored in memory i.e. using only the + and – arithmetic functions implement the following equation:

```
A = B × C
```

where variables `A`, `B` and `C` are stored in memory.

```
N1 0
+
L1
L0
CF?1
CF+4
-
L1
L0
S0
B
```

"Jump" Examples

CF?1 : jump forward 1 instruction **if** the run-up lever is set

CF+4: jump forwards 4 instructions

Note, the `F` character may be replaced with `B` to jump backwards to previous instruction.

Figure 4: 'Combinatorial' card example

Hint, the 'combinatorial card' allows you to 'jump' to different instructions within your program i.e. both forwards (`F`) and backwards (`B`). This jump is either unconditional i.e. will always be taken, or conditional on the result of the last arithmetic operation, which will set or clear the run-up lever. The run-up lever state is shown in the Annunciator panel (top left), figure 2. Conditional jumps are taken if the run-up lever is set, otherwise they are ignored and the next sequential card executed.

Consider the operation of the example shown in figure 4. This sequence of cards replaces the number in column 0 of the Store (memory location 0) with its absolute value:

```
MEM[0] = ABS( MEM[0] )
```

where column 0 stores the users input value e.g. N0 +100, or N0 -100 (not shown in figure 4). In this example the run-up lever controls the flow of instructions, from the online documentation its operations is described as follows:

"Addition sets the run-up lever if the sum of two quantities differs in sign from the first argument; since we are adding the unknown to zero, the run-up lever will be set only when the number in column 0 is negative."

One possible implementation of `A=B×C` could be:

```
A = 0
LOOP
     C = C - 1
     IF C < 0 EXIT LOOP
     A = A + B
END LOOP
STOP
```

Using this algorithm implement a program to perform multiplication through repeated addition.

Summary

Modern computers are normally base 2 machines i.e. use a binary representation. However, this is only because the technology used i.e. transistors, naturally support two stable states (on and off). The key thing to realise is you do not need to use a binary representation, you need to match the number base used to the technology. For Babbage he selected base 10, but he could of used a different base, one matched to the 40 teeth technology (cogs) used in his machine e.g. base 40, or base 20, as shown in figures 5 and 6.

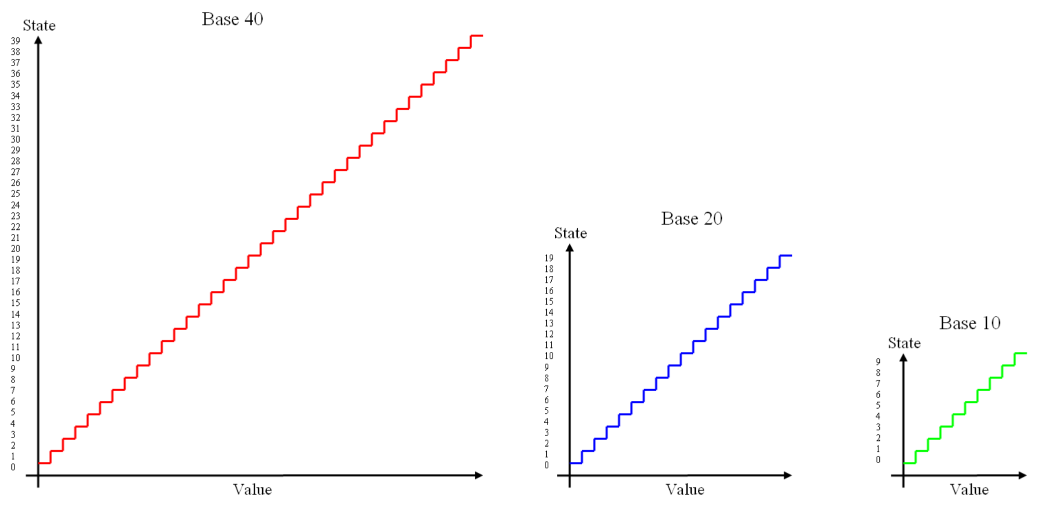


Figure 5: base selection, possible range of numbers per cog

Base 40 : { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39 }

$$427_{10} = (40^2 \times 0) + (40^1 \times 10) + (40^0 \times 27) = (10)(27)_{40}$$

$$= AR_{40} \quad (0-9,\ A-Z,\ \ldots)$$

Base 5 : { 0, 1, 2, 3, 4 }

$$427_{10} = (5^3 \times 3) + (5^2 \times 2) + (5^1 \times 0) + (5^0 \times 2)$$

$$= 3202_5$$

Base 2 : { 0, 1 }

$$427_{10} = (2^8\times1) + (2^7\times1) + (2^6\times0) + (2^5\times1) + (2^4\times0) + (2^3\times1) + (2^2\times0) + (2^1\times1) + (2^0\times1)$$

$$= 110101011_2$$

Figure 6 : representing the value 427 in base 40, 5 and 2

How we encode instructions and data is a important consideration when designing a computer. As discussed in lectures moving to a higher base e.g. from base 10 to base 40, can have some advantages:

- Less digits (figure wheels) needed to store a value as each cog represents more symbols (teeth).
    - Require less hardware to represent a number.
    - However, makes figure wheels more complex to manufacture.
    - Quicker, smaller rotation needed to transfer data.

However, there are also disadvantages:

- Cost - require more complex hardware to represent a digit, need to machine more teeth per cog.

Moving to a lower base e.g. from base 10 to base 2, can also have some advantages:

- Less symbols (teeth) per cog, simpler, cheaper hardware.
-

Disadvantage

- More digits (figure wheels) needed to represent a number.
    - Internal components have to move faster transfer the same information i.e. less teeth per revolution.

This raises the question what is the best number base to use? Answer: application dependent. For Babbage the main purpose of his machine was to print tables of base 10 values. Therefore, if he did use an internal base 40 representation to maximise the range of values stored on each cog he would of needed additional hardware in the printing sub-system to convert from base 40 into base 10. There is also the question of the user interface i.e. the operator would need to enter base 40 values, which would be a little tricky for the time. Note, this was one reason why early machines used a base 10 representation even when internally they used switch based technology e.g. ENIAC.

Matching a systems hardware to the processing task required is still a relevant question today, what is better:

- NISC, RISC, CISC, VLIW, Super-scalar, Multi-core ...

Answer : very dependent on the software being executed.

Appendix A : test code

```
N0 +3
N1 +4
N2 +11
N3 +5
N4 +6
N5 +17

×
L2
L4
S8
L5
L1
S9
L4
L0
S10
L1
L3
S11

-
L8
L9
S12
L10
L11
S13

÷
L12
L13
S14'
B
```

Figure A1 : Program listing

```
     ** Load simultaneous equation constants **

N0 +3     load memory 000 with the value 0
N1 +4     load memory 001 with the value 4
N2 +11    load memory 002 with the value 11
N3 +5     load memory 003 with the value 5
N4 +6     load memory 004 with the value 6
N5 +17    load memory 005 with the value 17

     ** Calculate intermediate multiplication results **

×         set arithmetic operations to multiply
L2        calculate intermediate result C × E
L4
S8        store result in memory location 8
L5        calculate intermediate result F × B
L1
S9        store result in memory location 8
L4        calculate intermediate result E × A
L0
S10       store result in memory location 8
L1        calculate intermediate result B × D
L3
S11       store result in memory location 8

     ** Calculate intermediate subtraction results **

-         set arithmetic operations to subtract
L8        transfer memory 008 to Ingress 0 (C × E)
L9        transfer memory 009 to Ingress 1 (F × B)
S12       store result of C × E - F × B to memory 012
L10       transfer memory 008 to Ingress 0 (E × A)
L11       transfer memory 009 to Ingress 1 (B × D)
S13       store result of E × A - B × D to memory 012

     ** Calculate final result

÷         set arithmetic operations to divide
L12       transfer memory 012 to Ingress 0
L13       transfer memory 012 to Ingress 0
S14'      store quotient to memory 014

B         ring bell to indicate program complete
```

Figure A2 : Annotated Program listing