

JOHN VON NEUMANN, EDVAC, AND THE IAS MACHINE

JULIET KEMP

The Linux Voice time machine takes us back to one of computing's eureka moments: the von Neumann architecture.

John von Neumann was born in Hungary in 1903. He was a prodigy, publishing two major mathematical papers by the age of 19. After teaching at the University of Berlin, in 1930 he was invited to Princeton University in the US, and later joined the Institute for Advanced Study there. During this time he contributed to several branches of maths, including set theory, game theory, quantum mechanics and logic and mathematical economics.

During the late 1930s, he worked on modelling explosions, which led to his involvement in the Manhattan Project. He is also credited with developing the strategy of "mutually assured destruction" which drove the Cold War. (In game theory, mutually assured destruction is an equilibrium, in which neither player has the incentive either to act or to disarm.)

Von Neumann was also heavily involved in early computing, partly because the work he was doing on the hydrogen bomb required vast and complex calculations. These were done initially by human computers – women using desk calculators to run the calculations required, on a production-line basis. During 1943 they began to use IBM punched-card machines, which worked at roughly the same speed but didn't need sleep. (A single calculation problem took three months, which Richard Feynman reduced to three weeks by running cards in parallel on the machines.) These machines, however, weren't programmable computers; they were just calculators.

Von Neumann consulted on both the ENIAC and EDVAC projects. The initial design of the ENIAC, the first programmable general-purpose computer, did not include the ability to store programs, and while it was programmable and Turing-complete, the programming was done by manipulating switches and cables. (Colossus was programmed similarly, but was not general-purpose, being dedicated to cryptanalysis.) ENIAC used an immense number of vacuum tubes to both store numbers and calculate, and punch cards for input and output. It was developed to run artillery calculations, but due to the involvement of von Neumann and Los Alamos, in the end the first calculation it ran was computations for the hydrogen bomb, using around a million punch cards.

EDVAC and the First Draft of a Report

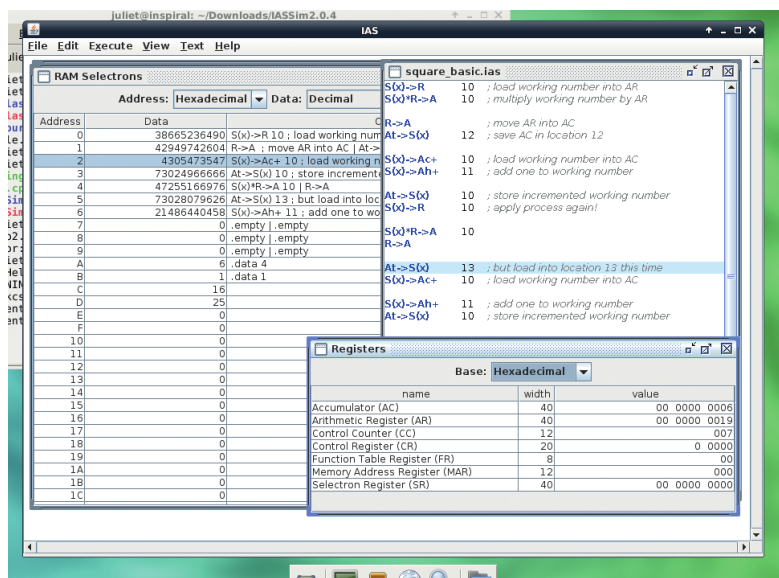
The EDVAC was proposed by the inventors of ENIAC, Mauchly and Eckert, in late 1944 – before the ENIAC was fully operational, but using improvements thought of while building it. EDVAC, like ENIAC, was built for the US Army's Ballistics Research Laboratory (at the Aberdeen Proving Ground). Although it hadn't yet been built, von Neumann's famous First Draft of a Report on the EDVAC was written (by hand while commuting to Los Alamos by train) in June 1945.

The Draft Report contains the first published description of the logical design of a stored-program computer, specifically the design that is often now known as the Von Neumann architecture and which is still widely used today. However, there is controversy over the extent to which this was solely von Neumann's work.

Some of the EDVAC team maintained that the concepts arose from discussions and work at the Moore School (where EDVAC was designed) before von Neumann began consulting there. Other documents suggest that Eckert and Mauchly had already thought of the idea of a 'stored program', but they hadn't fully outlined a design.

The First Draft of a Report on the EDVAC was, indeed, a first draft. It was intended as a summary and analysis of the logical design of the proposed EDVAC, with further extensions and suggestions from von Neumann. In it, von Neumann recommended that the computer have a central control unit to control all operations, a central processing unit to carry out operations, and a memory that could store programs and data and retrieve them from any point (ie random

To run the emulator on Linux and study von Neumann's programming methods, you will need Java version 5 or later.



access, not sequential access). He also recommended that EDVAC have a serial, rather than a parallel, processor, as he was concerned that a parallel processor would be too hard to build.

Unfortunately (and apparently without von Neumann's knowledge), Goldstine distributed the First Draft with just his and von Neumann's names on it, and without any credit given to Eckert and Mauchly. (From the gaps in the report, it is likely that von Neumann intended to insert further credits before 'proper' publication.) Goldstine likely only meant to share the ideas as quickly as possible, but it had the unfortunate effect of linking this architecture with von Neumann alone, rather than with the whole group of people who had been working on it.

When EDVAC was in due course built, it had a computational unit that operated on two numbers at a time then returned the results to memory, a dispatcher unit which connected this to the memory, three temporary operational tanks, nearly 6,000 vacuum tubes, and a mercury delay line memory of 1,000 words (later 1,024 words). It read in magnetic tape. It finally began operation in 1951, by which time von Neumann had moved back to IAS; not only that, but the Manchester Mark I team in the UK (who were later joined by Turing) had beaten them to the post of developing the first stored-program computer, running their machine for the first time in June 1948.

The IAS Machine

Meanwhile, in 1946, von Neumann wrote another paper, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument", which further developed his ideas. The IAS machine was the embodiment of those ideas.

One of the big differences between the IAS machine and EDVAC was that the IAS machine had a parallel processor. Words were processed in series, but the bits in each word were stored and operated on in parallel. This shows how fast the technology was moving – in the report on EDVAC in 1945, von Neumann thought that a parallel processor would be too difficult to build, so recommended a serial processor. By the time IAS machine project started in May 1946 (or possibly soon after, while they were working on the design), von Neumann had become convinced that parallel processing could work.

The IAS machine itself used a 40-bit word (with two 20-bit instructions per word), with a 1024-word memory and two general-purpose registers. Unlike many other early computers, it was asynchronous, with no clock regulating instruction timing. Instructions were executed one after the other. It used vacuum tubes for its logic, and Williams tubes (cathode ray tubes) for its memory, known as the Selectron.

Cathode ray memory relies on the fact that when a dot is drawn on a cathode ray tube, the dot becomes positively charged and the area around it negatively charged. When the beam is next pointed at that location, a voltage pulse is generated, which will differ



Von Neumann invented cellular automata. Turing invented parts of mathematical biology. These days, cellular automata are at the forefront of our investigations into mathematical biology – and those investigations rely on the computers that Turing and von Neumann put so much work into.

depending on whether there was a 'dot' or a 'dash' stored there. A metal pickup plate over the tube detects the voltage pulse and passes the data out to the next part of the memory system and ultimately to the control unit. The act of reading the memory also wipes it, so it must immediately be rewritten; and as the charge well fades quickly, the whole thing must also be frequently rewritten. The advantage, though, over mercury delay lines was that as the beam could point at any location immediately, memory was entirely random-access. With mercury lines, you had to wait until your data word came around to the output of the line before you could read it.

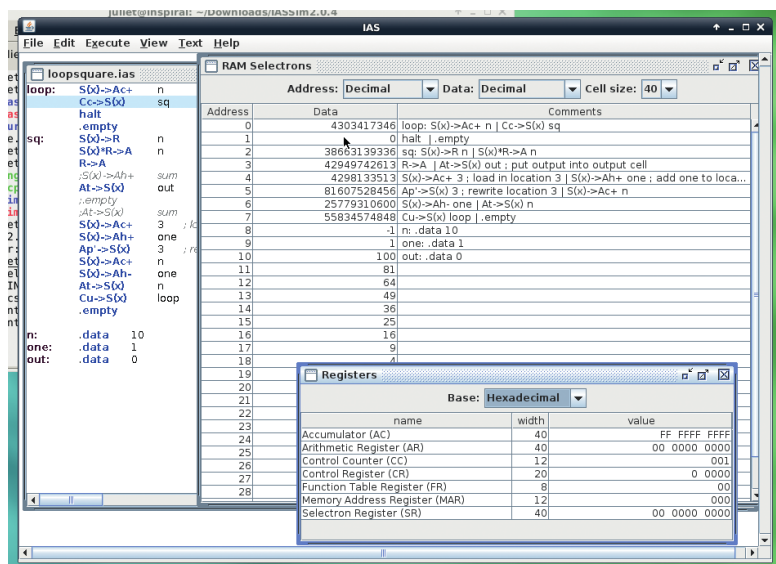
Von Neumann architecture

The crucial point about the 'von Neumann' architecture was that it combined both instructions and data in a single memory. This meant that you could, for example, implement a loop by modifying stored instructions. Unfortunately this also has the effect that all operations are using the same memory, so the machine cannot fetch an instruction and act on data at the same time. This came to be known as the von Neumann Bottleneck. The alternative, the Harvard Architecture (originating with the Harvard Mark I), separates data and instruction storage. Most modern computers use von Neumann architecture for main memory, but a modified Harvard architecture is used for some caches and in some other situations.

The IAS had five main parts: Central Arithmetic (which performed arithmetic operations), Central Control (which passed information between CA and memory), Memory, Output, Input, and the recording medium (magnetic tape, initially). It had seven registers, three in the CA and four in CC:

Central Arithmetic

- **AC** Accumulator.
- **MQ/AR** Multiplier/Quotient register (aka Arithmetic Register).



Using a little more assembly language makes coding loops straightforward.

- MDR Memory Data Register.
- Central Control
- IBR Instruction Buffer Register.
- IR Instruction Register.
- PC Program Counter.
- MAR Memory Address Register.

The IAS instructions took the form of “8-bit operation code” + “12-bit memory address” (the memory address was ignored if the instruction did not need it). So the instruction **S(x)->R 010** meant “load the number at Selectron location **x** into the Arithmetic Register; location **x** is **010**”. The available instruction set had 21 operations (plus Halt making 22), which copied numbers into and out of the AC and AR, subtracted, added, multiplied, or divided them, and controlled execution. The execution control enabled the programmer to jump to a particular memory address, or to check whether a given value was greater than or equal to 0, or to rewrite a given instruction; it was these abilities that enabled loops.

The IAS architecture and plans were implemented in several machines across the world, as the plans were freely distributed. However, all of these machines, although IAS derivatives, were slightly different; you couldn't just run software written for one machine on another machine without rewriting it for the quirks of that individual machine. Some of the famous IAS machines include MANIAC (at the Los Alamos National Laboratory; von Neumann was involved with this one too and was responsible for the name), the IBM 701 (IBM's first commercial scientific computer, with 17 installations), and ORACLE (in Oak Ridge National Laboratory). Other IAS machines existed in Copenhagen, Moscow, Stockholm, and Sydney, among others.

IAS emulator

There's a Java-based emulator, IASSim, available from the Princeton IAS machine, from www.cs.colby.edu/djskrien/IASSim/ -- so you can try out IAS machine

coding for yourself. Download the Zip file, unpack it, and **cd** into the folder. This command will launch the emulator in its own window:

```
java -cp IASSim2.0.4.jar:jhall.jar:IASSimHelp2.0.jar iassim.Main -m IAS.cpu
```

You can load in an assembly language text file from the File menu, and there is a tutorial and online help available from the Help menu.

The folk who wrote the emulator have also written a basic assembler for it, to make life a little easier. For this first example I'll use the assembler as little as possible, to give you the best flavour of the IAS machine's language. Open up a new text file in the emulator and enter this (without the line numbers):

```
0. S(x)->R 10 ; load working number into AR
   S(x)*R->A 10 ; multiply working number by AR
1. R->A ; move AR into AC
   At->S(x) 12 ; save AC in location 12
2. S(x)->Ac+ 10 ; load working number into AC
   S(x)->Ah+ 11 ; add one to working number
3. At->S(x) 10 ; store incremented working number
   S(x)->R 10 ; and start again!
4. S(x)*R->A 10
   R->A
5. At->S(x) 13 ; but save in location 13 this time
   S(x)->Ac+ 10
6. S(x)->Ah+ 11
   At->S(x) 10
7. .empty
   .empty
8. .empty
   .empty
9. .empty
   .empty
10. .data 4
11. .data 1
```

Let's take a look at that. First of all, each 'line' (which is in fact the register address where the instruction is stored) has two instructions, since the IAS machine had two instructions per 'word' on its tapes.

Line 0: The first half of our first pair of instructions loads the number in location 10 into **AR**, the Arithmetic Register. **S(x)** refers to Selectron (memory) location **x**, and 10 is given for **x** at the end of the line. The second half, **S(x)*R->A 10** multiplies **S(10)** by **R**, and stores the result in **A**. Multiplication on the IAS gave rise to a result stored in two halves: the left half of the number in **AC** and the right half in **AR**. Since we are only multiplying small numbers, only the right half is useful.

Line 1: The next instruction, **R->A**, therefore moves the right half of the result from **AR** into **AC**. We can then save it to location 12 with **At->S(x) 12**. That gives us the first answer, the square of the working number, stored in location 12.

Line 2: Load the working number itself into **AC**, then add the contents of location 11 to it (**S(x)->Ah+ 11**). As you'll see in a moment, location 11 contains 1, so this just increments our working number by 1.

Line 3: We store the incremented working number back in location 10, and start the process again.

Lines 4–6: As above, but this time around our new result is stored in location 13. We increment the working number one more time before stopping.

Lines 7–9: These are empty just for ease of setup. The empty lines mean that we can store our working number in location 10 and leave a bit of room to add more instructions if desired. (If you remember coding in BASIC with line numbers, you may recall numbering in tens to give yourself wiggle room; same thing!).

Lines 10–11: The assembler instruction `.data` is used to put the numbers 4 (our working number) and 1 (for use when incrementing) into locations 10 and 11. The original programmers would have just been able to write numbers (whether all zeros for an empty line, or data numbers) straight to tape.

To run this, go to the Execute menu and choose Clear, Assemble, Load, and Run. Check out the RAM Selectrons window to see the contents of the registers – you should see 16 in location C (hexadecimal) and 25 in location D. (You might need to change the Data view to Decimal.) You can also step through the program one instruction at a time using Debug mode, and watch the registers change in the Registers window, if you prefer.

In fact, if you change the Data view of the RAM Selectrons window to Hexadecimal, you can code your instructions directly into the Selectron locations. Each location has two sets of one 2-place and one 3-place hex number, corresponding to an instruction+location, twice. So, for example, the hex representation of **S(x)-R** is 09, and the first instruction of our first line is **09 00A** (A being 10 in hex).

Here's a loop version of our squares code using assembly language (with thanks to the writers of the IAS Sim software for the loop control code):

```
loop: S(x)->Ac+ n ; load n into AC
```

```
    Cc->S(x) sq ; if n >= 0, go to sq
```

```
    halt
```

```
    .empty
```

```
sq: S(x)->R n
```

```
    S(x)*R->A n
```

```
    R->A
```

```
    At->S(x) out
```

```
    S(x)->Ac+ 3
```

```
    S(x)->Ah+ one
```

```
    Ap'->S(x) 3
```

```
    S(x)->Ac+ n
```

```
    S(x)->Ah- one
```

```
    At->S(x) n
```

```
    Cu->S(x) loop
```

```
    .empty
```

```
n: .data 10
```

```
one: .data 1
```

```
out: .data 0
```

The labels here are part of the assembly language, to make looping easier (but it could be done by hand if

you prefer – feel free to try it out!). We start off with **n** (see the data labels at the bottom), load it into the AC, and check that it is still non-negative. If so, we jump to the **sq** subroutine.

The first four lines of **sq** are familiar – load up **n**, square it, and store the square in the out location. Next is the interesting part.

S(x)->Ac+ 3 loads the instruction at location 3 into the AC register. Location 3, if you count lines (remember that the locations start at 0) contains the instruction **R->A** on its left side and **At->S(x)** out on its right side. So we now have a number representing those instructions in the AC.

The next instruction, **S(x)->Ah+** adds one to that. This effectively alters **At->S(x)** out to **At->S(x) out+1**.

We then write this altered instruction back to location 3, with

Ap'->S(x) 3 (specifically, **Ap'** alters the right-hand side of the instruction at location 3, and **Ap** alters the left-hand side). So the next time

we loop around this code, instead of writing the output to the location labelled out, we'll write it to the next location along. To watch this happen, you can use Debug mode, step through the code, and keep a close eye on the Registers window.

The next three lines load **n** up again, decrease it by one, and save it. We then jump back to loop with the **Cu** instruction, and go round the loop again.

If you load and run this, you'll see that you get the squares from 100–0 output in locations 10–20. This is the behaviour that the von Neumann architecture makes possible: altering the program's stored instructions as you go along.

Final years

Von Neumann carried on working on computing, alongside his other areas of interest, for the rest of his life. In 1949, he designed a self-reproducing computer program, which is considered to be the first ever computer virus, and he worked on cellular automata and other aspects of self-replicating machines. He also introduced the idea of stochastic computing (which, broadly, uses probability rather than arithmetic) in a 1953 paper, although the computers of the time weren't able to implement the theory.

Sadly, he died in 1955 from bone or pancreatic cancer. (A biographer has speculated that this might have been due to his presence at the Bikini Atoll nuclear tests in 1946.) His contribution across his fields of interest was truly immense and he might well have contributed still further had he lived longer. 📖

“Von Neumann designed a self-replicating computer program, which is considered to be the first computer virus.”

Juliet Kemp is a scary polymath, and is the author of O'Reilly's Linux System Administration Recipes.