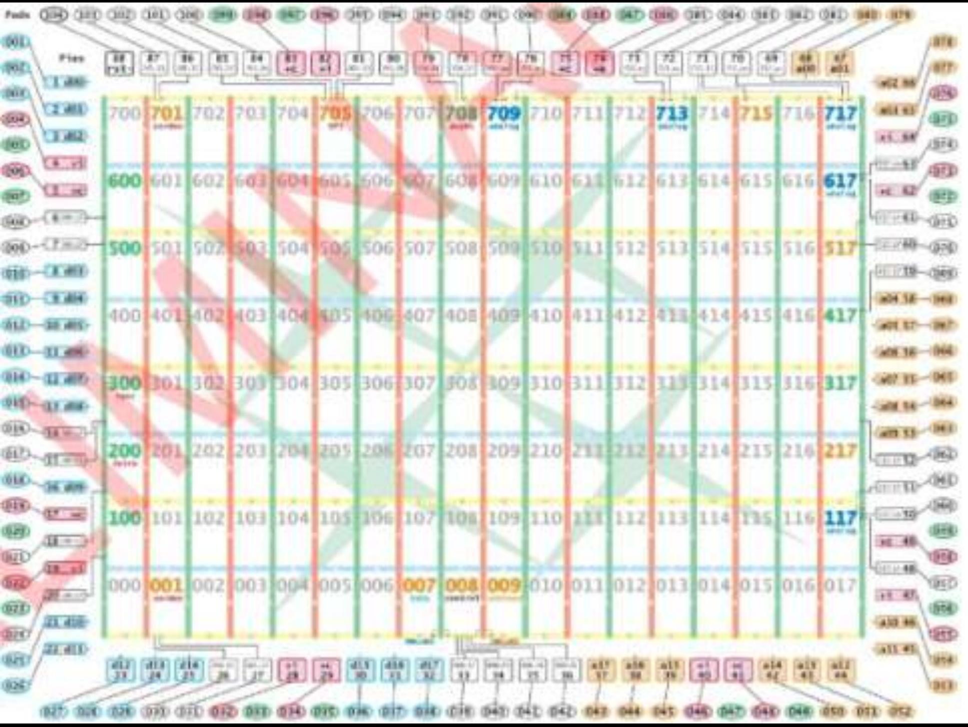
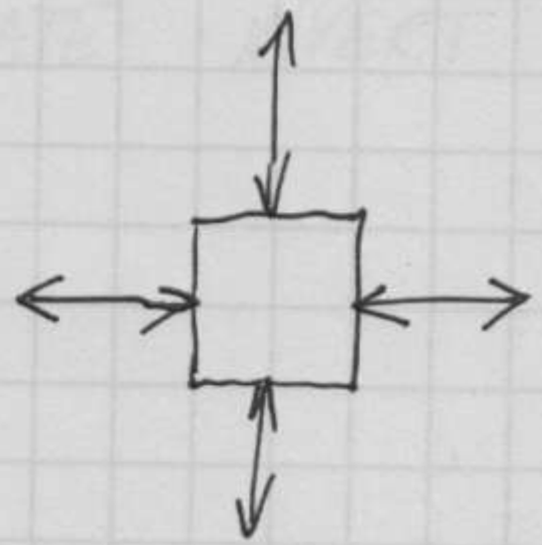
A novel execution model for GA144

Michael Schuldt, Mark Flamer, James Bowman

August 28, 2017



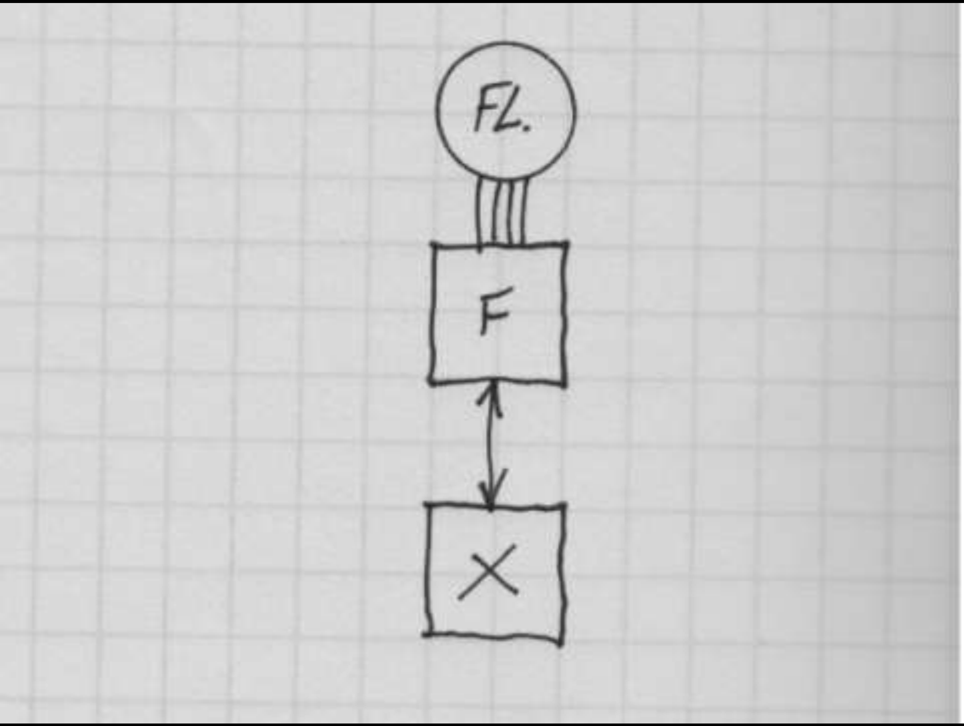
http://www.greenarraychips.com/



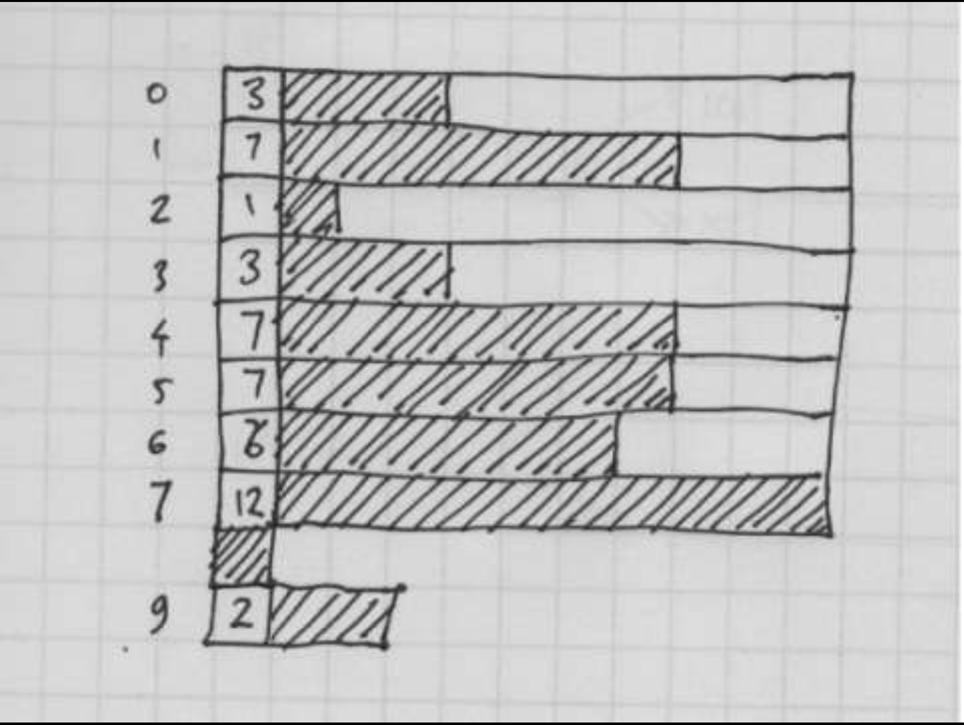
Instructions are grouped into polyads, 1-4 micro-ops packed into an 18-bit word. These 18-bit polyads are the basic unit of execution. Unused slots are NOPs.

A crucial feature of the GA144 is that each node can execute instructions from its neighbor. This is "port execution" - you just jump to an IO port, and the node executes whatever it receives.

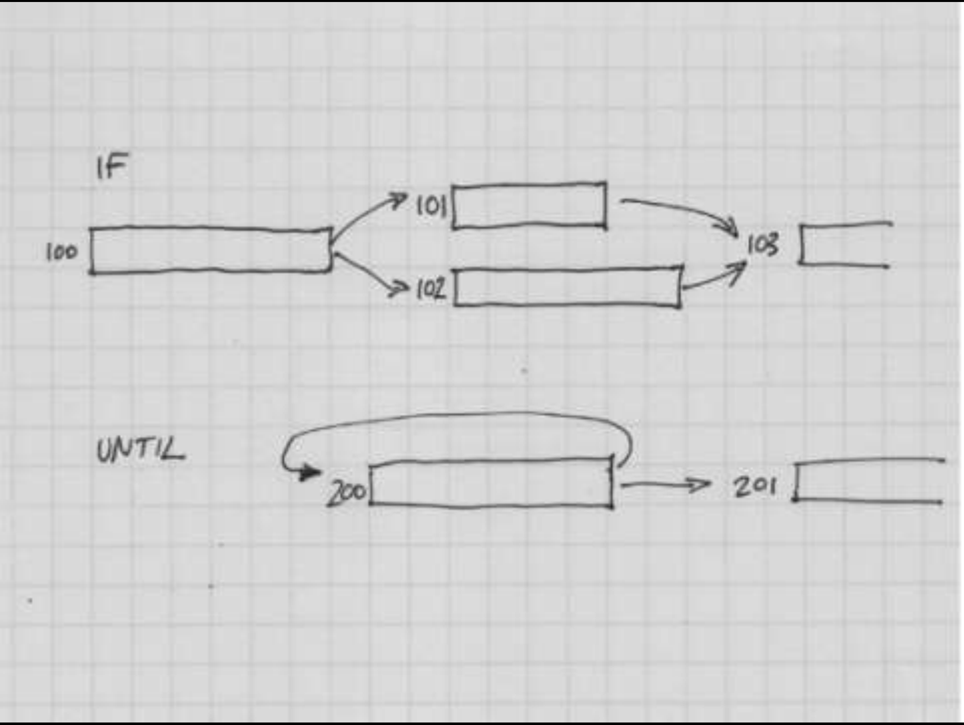
That's how the GA144 boots. ROM in the SPI node (for example) reads out the SPI contents and sends it to a neighbor. These instructions can make the neighbor do anything, such as load RAM or set up registers, or even send some instructions to *its* neighbor. This can be extended indefinitely, potentially making a boot chain that covers the whole chip.



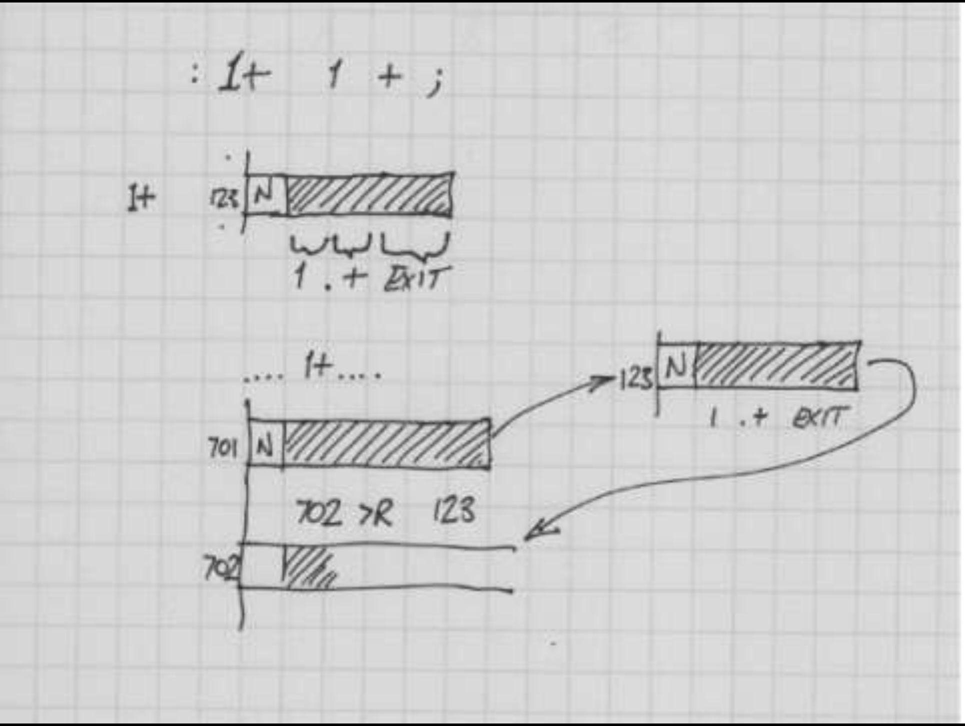
With one modification, this bootstrap procedure is the basis for the new execution model. The spi node boots by reading a stream from flash, but instead of loading from flash at address zero, it reads an address from the running node.



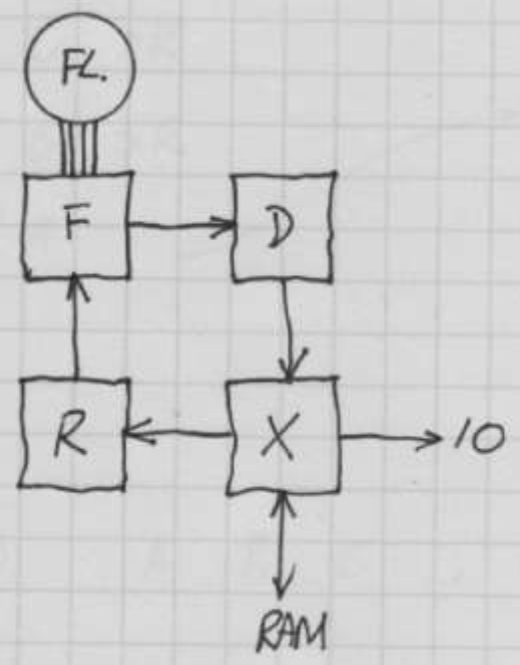
Each code sequence is a "fragment", accessed by a fragment number – a scaled byte offset into flash. Fragments are sized sequences of instruction polyads – these execute on the X node. Every fragment must end with the instructions to tell F which fragment is next. One important restriction is that fragments can only execute from the beginning.



Some examples of code flow with fragments like this.
In fact a program can be broken up into fragments like this.
Every time there is a branch or label, there is a fragment boundary. In other words, a fragment is a basic block: a sequence of non-branch instructions.



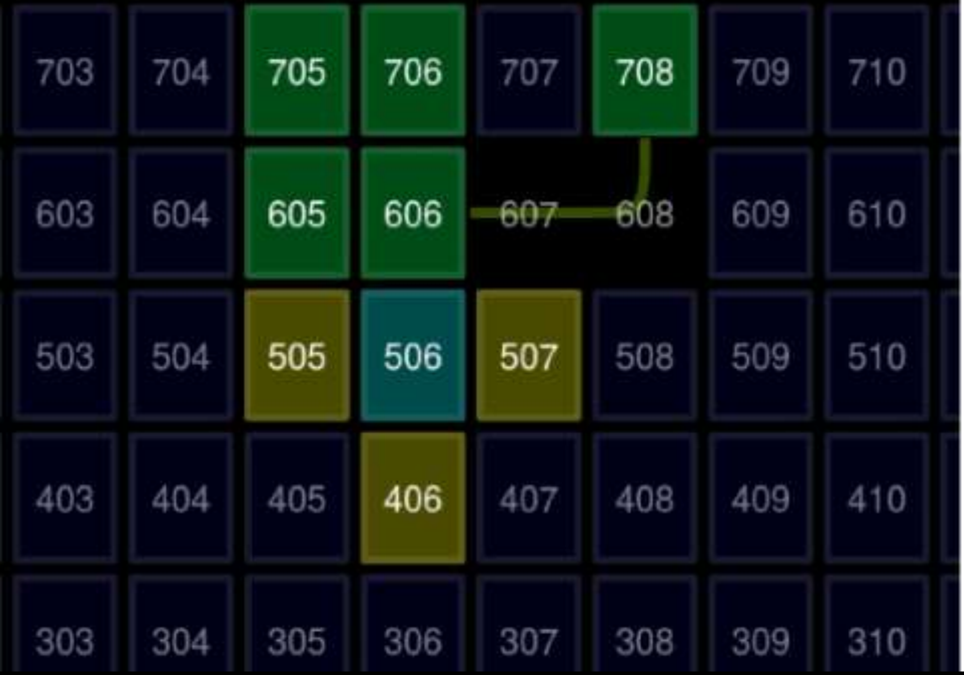
But consider subroutine calls – in other words, Forth words. To call a word FOO, a fragment needs to request that FOO executes next, but also must specify another fragment to be executed when FOO completes. So there must be a call-return stack



The top of the parameter stack (TOS) is the hardware top-of-stack (T) in X. The rest of the stack is in RAM in X. The X node's A register points to the top of the stack, which grows down. There are about 24 cells for the parameter stack.

The return stack is in node R. So to words **>R**, **R@** and **R>** X has to make a request to R. Like the parameter stack, the return stack grows down. The return stack is about 50 cells deep.

The IP is a little bit subtle. It is really the current location that F is fetching from flash. It's not tracked directly, except as the fragment number that flows from R to F.



actual layout.

The RAM controller is cyan, and its storage nodes are yellow.

192 words total, about 384 bytes.

```
CODE +                     #returns #inline
    @+ . +
```

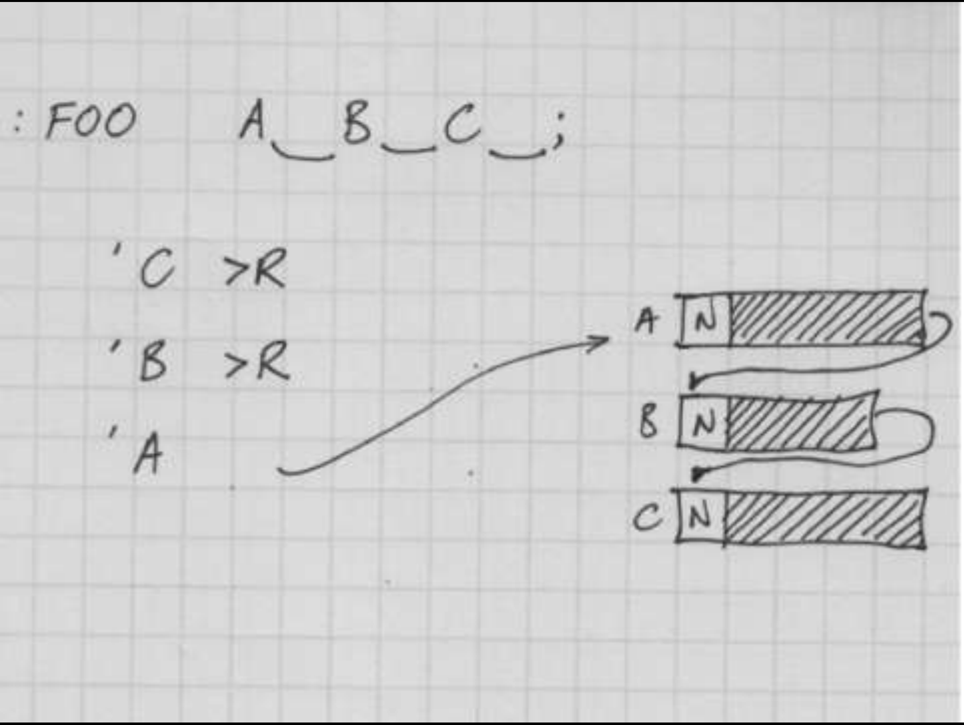
+ fetches the value from the parameter stack, then adds them, leaving the result in T

```
CODE DROP            #returns #inline
    @+
```

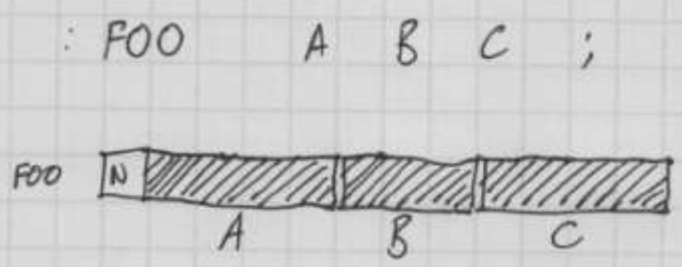
DROP fetches from the stack into T. Note that it doesn't need to balance X's hardware stack::

```
CODE SWAP               #returns #inline
    @ over !
```

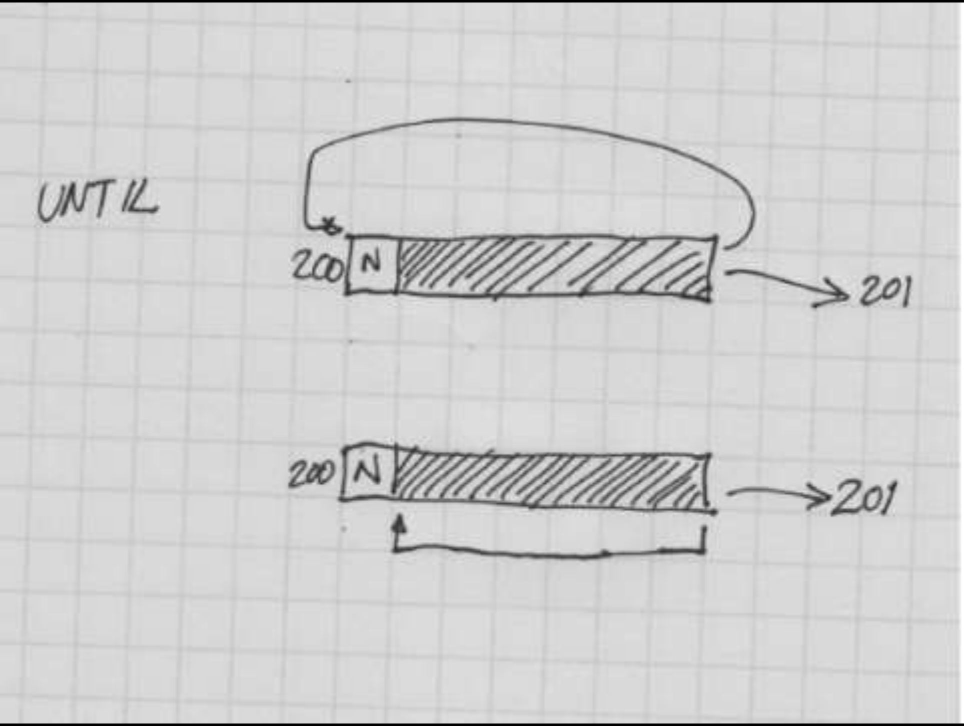
SWAP again does not need to balance the hardware stack



As described, there needs to be 'glue' fragments between consecutive calls. So FOO needs three glue fragments after calls to A, B and C.
Of course the tail call optimization can remove one of these.
But actually can do better than this.
The first fragment of FOO pushes the fragment numbers of B and C on the R stack, then jumps to A. When A returns, it executes B. When B returns it executes C.
This is not new - it's sometimes used by exploit code where it's called "return oriented programming"
`https://en.wikipedia.org/wiki/Return-oriented_programming`
So FOO compiles to a single setup fragment. The return stack needs to be deep enough to hold a chain of calls like this – so the implementation has a 50 cell R stack.



And of course we can go even further. If the words are all inlinable, then FOO doesn't need any calls at all. Compilation in this case is simple concatenation of the previous words.
The fixed overhead of call/return is avoided.
Uses flash space. The price of flash is about 1 cent for every 100 Kbytes.



One more thing.

Some fragments jumps back to thmself - like a FOR, DO or UNTIL.

If the fragment is straight-line code, it's an easy transformation to compile a native jump back to the start of the code instead of re-requesting the same fragment.

This optimization makes things about 200X faster.