1 Forwword

While teaching a Discrete Structures class in Winter 2016 at Clark College, I was leturing on the universal nature of a NAND gate: the fact that other gates can be realized by using combinations of NANDs. This led to the offhand comment that one can design an a computer whose only native instruction is a NAND. In the spirit of provoking their curiosity and interest, I suggested that this could be considered, in some sense, a 0-bit computer ("ZBC"): meaning that the instruction word is 0 bits in length (since your set of instructions is comprised of a single instruction, no bits are required to specify the choice of instruction).

I described how I had tried to do this once years before, writing programs that consisted of triples of memory locations: src_1, src_2 and dst, where the operation was $NAND(src_1, src_2) \rightarrow dst$. I also commented that, at the time, the question of shifting the contents of a memory location has puzzled me: I couldn't figure out how to achieve a shift without effectively coding a very large truth table.

A student asked me after class about this comment, and it got me rethinking the issue later that day. I decided I would simply memory-map a shift register into the address space of the ZBC, and not worry about it any further. This led to the idea of memory mapping various hardware devices, such as ALUs, I/O units, etcc. If I memory-mapped logic circuits, then I no longer needed a NAND to be a native instruction. The instruction set was suddenly reduced to a single MOV instruction: just move from a source to a destination. By memory-mapping the PC into the address space, jumps could be effected by simply loading the target address into the PC's location in memory. Self-modifying code could be used to implement conditional jumps, thereby completing the trilogy of sequential-execution, looping and conditionals.

I wrote a small simulator for this architecture, and hand-assembled a program to count from 1 to 10; that was pretty easy. Then I tried to do this ffor a prime number generator. This took longer, but was still relatively easy to write (but almost impossible to read!). I now had a very reduced transfer-based machine that could be used to implement algorithms similar to what can be implemented on today's computers.

The reduced architecture felt good: it removed some of the complications of traditional machine code (addressing modes, status register, opcodes!), but it still felt somehow bloated and artificial. The discrete nature felt wrong, like there was still an extra layer somewhere that was forcing this artificial construct onto the system. So I started thinking about how to restore continuity to the system. This led to the next document.