Big Idea (Week 8): Sequential, Stored-Program Processors

We have seen that if we have enough NAND gates and registers, we can build any combinational logic function or any finite-state machine. As our computations become more complicated, this requires more and more physical resources—more NAND gates. Eventually this may make our computation too big to solve with limited physical resources. At the same time, we may not even be using the NAND gates at their full capacity. Furthermore, the need to build a different circuit by wiring up a different collection of NAND gates for every different function we want to compute can limit the computations we can build and have available for use. *Stored-program processors* were developed to address these issues.

The first observation is that we can avoid needing new, unique NAND gates for every part of our computation if we are willing to sequentialize the circuit and reuse the NAND gates in time. Key to this is to virtualize, the hardware, using compact state elements to store the state in a computation. A trick here is that we can build memories that store this state very compactly. Now rather than giving each gate in the circuit its own NAND gate, we simple store its output value in a memory. Then, to evaluate the entire circuit, we read the memory values holding the inputs to a NAND gate and calculate the output, which we store back into the appropriate memory slot. This allows us to use a single NAND gate to evaluate a large number of NAND gates. In practice, we build more sophisticated hardware blocks than NAND gates, such as adders and multipliers, but the idea is the same. We can break our large computation into a sequence of small operations that can be evaluated over time using a limited amount of simple hardware and storing the intermediate results into compact memories.

In order to perform these computations, we need some control signals that specify which state values should be input to the limited hardware on each cycle and, when the hardware is more sophisticated than a NAND gate, control signals to tell the hardware which operations to perform. We call these control signals *instructions* since they instruct the hardware about what operations to perform on each cycle.

The second observation is that we can put these instructions into a memory as well. To program up a large computation, we break it down into a sequence of instructions and store the instructions into the memory. To perform the computation, we read the instructions from the memory one at a time and use them to control the behavior of the limited hardware. Since the entire computation is defined by the contents of this instruction memory, we can change the computation simply by changing the contents of the memory. That is, the resulting processor can be programmed to perform any computation simply by reprogramming its instruction memory.

Historically, the stored-program processor was the key innovation of the EDVAC, Eckert and Mauchly's successor to the ENIAC that was designed here at the Moore School. It was a more economical machine because it could reuse limited hardware in time.