Big Idea (Week 7): Combinational and Synchronous Digital Logic

At the base of our digital logic, we compute on 0's and 1's, representing false and true. We can model all our *combinational* (non-clocked) digital logic computations as functions from binary inputs to binary outputs. These are functions in the mathematical sense—for every input vector of binary values, there is a single output vector of binary values. A canonical, if not compact, representation for each such combinational logic function is a *truth table* that lists, for each possible input value, the associated output values (2^n input to output mappings for an n-input function). We can implement any such truth table by using AND, OR, and NOT gates. In fact, we can implement AND, OR, and NOT using NAND gates, so we can implement any truth table, and hence, any combinational logic function using a collection of NAND gates. We call this *NAND Universality*.

For a combinational logic function with a single output bit, we can construct a circuit to implement it by using an AND gate and a number of NOT gates to identify each input case (specific values for each of the n input bits) for which the output of the function is true and using an OR gate to combine the outputs of all such AND gates. To extend to a function with multiple output bits, we use the same approach for each output bit independently. As described, this might require n-input AND gates and 2^n-input OR gates. However, we can always implement an AND gate with many inputs as a tree of 2-input AND gates, and similarly, we can implement a multi-input OR gate with a tree of 2-input OR gates. Again, since we can replace the NOT, AND, and OR gates with NAND gate implementations, we can realize the entire function entirely from 2-input NAND gates. This construction is not the most efficient implementation (not the one that uses the fewest gates or 2-input NAND gates) but demonstrates NAND Universality. There is a rich body of knowledge and computer-aided design (CAD) tools for automatically reducing, and in many cases minimizing, the gate-level implementation of any combinational logic function.

The combinational logic model is a powerful building block, but, itself, does not allow our circuits to remember values from the past. Our synchronous digital logic model adds state elements and the notion of a clock to our combinational logic. A typical state element is a *register* (or a *flip-flop*) that has the property that, during a clock cycle, the register holds the value that was presented to its input at the end of the previous clock cycle. This *state* element gives us a way to describe and implement logic that holds on to values over time.

One standard way to formulate the resulting model is that of a *Finite-State Machine* (FSM). An FSM combines a combinational logic function with a number of registers holding state bits. Simply put, the combinational logic in the FSM takes as input both the circuit inputs and the outputs of the registers (the state bits) and produces as outputs the circuit outputs and the inputs to the registers (the values of the state bits on the next clock cycle). This way, all circuit behavior can be expressed as a combinational logic function operating on inputs and state. The state is "finite" because any practical circuit only has a finite number of registers and hence a finite number of potential values the registers can represent (for s register bits, 2^s potential states).