Okay, we've figured out a way to design hardware to perform a particular computation: Draw the state transition diagram for an FSM that describes the sequence of operations needed to complete the computation.

Then construct the appropriate datapath, using registers to store values and combinational logic to implement the needed operations.

Finally build an FSM to generate the control signals required by the datapath.

Is the datapath plus control logic itself an FSM?

Well, it has registers and some combinational logic, so, yes, it is an FSM.

Can we draw the truth table?

In theory, yes.

In practice, there are 66 bits of registers and hence 66 bits of state, so our truth table would need 2^66 rows!

Hmm, not very likely that we'd be able to draw the truth table!

The difficulty comes from thinking of the registers in the datapath as part of the state of our super-FSM.

That's why we think about the datapath as being separate from the control FSM.

So how do we generalize this approach so we can use one computer circuit to solve many different problems.

Well, most problems would probably require more storage for operands and results.

And a larger list of allowable operations would be handy.

This is actually a bit tricky: what's the minimum set of operations we can get away with?

As we'll see later, surprisingly simple hardware is sufficient to perform any realizable computation.

At the other extreme, many complex operations (e.g., fast fourier transform) are best implemented as sequences of simpler operations (e.g., add and multiply) rather than as a single massive combinational circuit.

These sorts of design tradeoffs are what makes computer architecture fun!

We'd then combine our larger storage with logic for our chosen set of operations into a general purpose datapath that could be reused to solve many different problems.

Let's see how that would work… Here's a datapath with 4 data registers to hold results.

The ASEL and BSEL multiplexers allow any of the data registers to be selected as either operand for our repertoire of arithmetic and boolean operations.

The result is selected by the OPSEL MUX and can be written back into any of the data registers by setting the WEN control signal to 1 and using the 2-bit WSEL signal to select which data register will be loaded at the next rising clock edge.

Note that the data registers have a load-enable control input.

When this signal is 1, the register will load a new value from its D input, otherwise it ignores the D input and simply reloads its previous value.

And, of course, we'll add a control FSM to generate the appropriate sequence of control signals for the datapath.

The Z input from the datapath allows the system to perform data-dependent operations, where the sequence of operations can be influenced by the actual values in the data registers.

Here's the state transition diagram for the control FSM we'd use if we wanted to use this datapath to compute factorial assuming the initial contents of the data registers are as shown.

We need a few more states than in our initial implementation since this datapath can only perform one operation at each step.

So we need three steps for each iteration: one for the multiply, one for the decrement, and one for the test to see if we're done.

As seen here, it's often the case that general-purpose computer hardware will need more cycles and perhaps involve more hardware than an optimized single-purpose circuit.

You can solve many different problems with this system: exponentiation, division, square root, and so on, so long as you don't need more than four data registers to hold input data, intermediate results, or the final answer.

By designing a control FSM, we are in effect "programming" our digital system, specifying the sequence of operations it will perform.

This is exactly how the early digital computers worked!

Here's a picture of the ENIAC computer built in 1943 at the University of Pennsylvania.

The Wikipedia article on the ENIAC tells us that "ENIAC could be programmed to perform complex sequences of operations, including loops, branches, and subroutines.

The task of taking a problem and mapping it onto the machine was complex, and usually took weeks.

After the program was figured out on paper, the process of getting the program into ENIAC by manipulating its switches and cables could take days.

This was followed by a period of verification and debugging, aided by the ability to execute the program step by step." It's clear that we need a less cumbersome way to program our computer!