Welcome to Part 2 of 6.004x!

In this part of the course, we turn our attention to the design and implementation of digital systems that can perform useful computations on different types of binary data.

We'll come up with a general-purpose design for these systems, we which we call "computers", so that they can serve as useful tools in many diverse application areas.

Computers were first used to perform numeric calculations in science and engineering, but today they are used as the central control element in any system where complex behavior is required.

We have a lot to do in this chapter, so let's get started!

Suppose we want to design a system to compute the factorial function on some numeric argument N. N! is defined as the product of N times N-1 times N-2, and so on down to 1.

We can use a programming language like C to describe the sequence of operations necessary to perform the factorial computation.

In this program there are two variables, "a" and "b".

"a" is used to accumulate the answer as we compute it step-by-step.

"b" is used to hold the next value we need to multiply.

"b" starts with the value of the numeric argument N.

The DO loop is where the work gets done: on each loop iteration we perform one of the multiplies from the factorial formula, updating the value of the accumulator "a" with the result, then decrementing "b" in preparation for the next loop iteration.

If we want to implement a digital system that performs this sequence of operations, it makes sense to use sequential logic!

Here's the state transition diagram for a high-level finite-state machine designed to perform the necessary computations in the desired order.

We call this a high-level FSM since the "outputs" of each state are more than simple logic levels.

They are formulas indicating operations to be performed on source variables, storing the result in a destination variable.

The sequence of states visited while the FSM is running mirrors the steps performed by the execution of the C program.

The FSM repeats the LOOP state until the new value to be stored in "b" is equal to 0, at which point the FSM transitions into the final DONE state.

The high-level FSM is useful when designing the circuitry necessary to implement the desired computation using our digital logic building blocks.

We'll use 32-bit D-registers to hold the "a" and "b" values.

And we'll need a 2-bit D-register to hold the 2-bit encoding of the current state, i.e., the encoding for either START, LOOP or DONE.

We'll include logic to compute the inputs required to implement the correct state transitions.

In this case, we need to know if the new value for "b" is zero or not.

And, finally, we'll need logic to perform multiply and decrement, and to select which value should be loaded into the "a" and "b" registers at the end of each FSM cycle.

Let's start by designing the logic that implements the desired computations - we call this part of the logic the "datapath".

First we'll need two 32-bit D-registers to hold the "a" and "b" values.

Then we'll draw the combinational logic blocks needed to compute the values to be stored in those registers.

In the START state , we need the constant 1 to load into the "a" register and the constant N to load into the "b" register.

In the LOOP state, we need to compute a*b for the "a" register and b-1 for the "b" register.

Finally, in the DONE state , we need to be able to reload each register with its current value.

We'll use multiplexers to select the appropriate value to load into each of the data registers.

These multiplexers are controlled by 2-bit select signals that choose which of the three 32-bit input values will be the 32-bit value to be loaded into the register.

So by choosing the appropriate values for WASEL and WBSEL, we can make the datapath compute the desired values at each step in the FSM's operation.

Next we'll add the combinational logic needed to control the FSM's state transitions.

In this case, we need to test if the new value to be loaded into the "b" register is zero.

The Z signal from the datapath will be 1 if that's the case and 0 otherwise.

Now we're all set to add the hardware for the control FSM, which has one input (Z) from the datapath and generates two 2-bit outputs (WASEL and WBSEL) to control the datapath.

Here's the truth table for the FSM's combinational logic.

S is the current state, encoded as a 2-bit value, and S' is the next state.

Using our skills from Part 1 of the course, we're ready to draw a schematic for the system!

We know how to design the appropriate multiplier and decrement circuitry.

And we can use our standard register-and-ROM implementation for the control FSM.

The Z signal from the datapath is combined with the 2 bits of current state to form the 3 inputs to the combinational logic, in this case realized by a read-only memory with 2^3=8 locations.

Each ROM location has the appropriate values for the 6 output bits: 2 bits each for WASEL, WBSEL, and next state.

The table on the right shows the ROM contents, which are easily determined from the table on the previous slide.