An interesting question for computer architects is what capabilities must be included in the ISA?

When we studied Boolean gates in Part 1 of the course, we were able to prove that NAND gates were universal, i.e., that we could implement any Boolean function using only circuits constructed from NAND gates.

We can ask the corresponding question of our ISA: is it universal, i.e., can it be used to perform any computation?

What problems can we solve with a von Neumann computer?

Can the Beta solve any problem FSMs can solve?

Are there problems FSMs can't solve?

If so, can the Beta solve those problems?

Do the answers to these questions depend on the particular ISA?

To provide some answers, we need a mathematical model of computation.

Reasoning about the model, we should be able to prove what can be computed and what can't.

And hopefully we can ensure that the Beta ISA has the functionality needed to perform any computation.

The roots of computer science stem from the evaluation of many alternative mathematical models of computation to determine the classes of computation each could represent.

An elusive goal was to find a universal model, capable of representing *all* realizable computations.

In other words if a computation could be described using some other well-formed model, we should also be able to describe the same computation using the universal model.

One candidate model might be finite state machines (FSMs), which can be built using sequential logic.

Using Boolean logic and state transition diagrams we can reason about how an FSM will operate on any given input, predicting the output with 100% certainty.

Are FSMs the universal digital computing device?

In other words, can we come up with FSM implementations that implement all computations that can be solved by any digital device?

Despite their usefulness and flexibility, there are common problems that cannot be solved by any FSM.

For example, can we build an FSM to determine if a string of parentheses (properly encoded into a binary sequence) is well-formed?

A parenthesis string is well-formed if the parentheses balance, i.e., for every open parenthesis there is a matching close parenthesis later in the string.

In the example shown here, the input string on the top is well-formed, but the input string on the bottom is not.

After processing the input string, the FSM would output a 1 if the string is well-formed, 0 otherwise.

Can this problem be solved using an FSM?

No, it can't.

The difficulty is that the FSM uses its internal state to encode what it knows about the history of the inputs.

In the paren checker, the FSM would need to count the number of unbalanced open parens seen so far, so it can determine if future input contains the required number of close parens.

But in a finite state machine there are only a fixed number of states, so a particular FSM has a maximum count it can reach.

If we feed the FSM an input with more open parens than it has the states to count, it won't be able to check if the input string is well-formed.

The "finite-ness" of FSMs limits their ability to solve problems that require unbounded counting.

Hmm, what other models of computation might we consider?

Mathematics to the rescue, in this case in the form of a British mathematician named Alan Turing.

In the early 1930's Alan Turing was one of many mathematicians studying the limits of proof and computation.

He proposed a conceptual model consisting of an FSM combined with a infinite digital tape that could read and written under the control of the FSM.

The inputs to some computation would be encoded as symbols on the tape, then the FSM would read the tape, changing its state as it performed the computation, then write the answer onto the tape and finally halting.

Nowadays, this model is called a Turing Machine (TM).

Turing Machines, like other models of the time, solved the "finite" problem of FSMs.

So how does all this relate to computation?

Assuming the non-blank input on the tape occupies a finite number of adjacent cells, it can be expressed as a large integer.

Just construct a binary number using the bit encoding of the symbols from the tape, alternating between symbols to the left of the tape head and symbols to the right of the tape head.

Eventually all the symbols will be incorporated into the (very large) integer representation.

So both the input and output of the TM can be thought of as large integers, and the TM itself as implementing an integer function that maps input integers to output integers.

The FSM brain of the Turing Machine can be characterized by its truth table.

And we can systematically enumerate all the possible FSM truth tables, assigning an index to each truth table as it appears in the enumeration.

Note that indices get very large very quickly since they essentially incorporate all the information in the truth table.

Fortunately we have a very large supply of integers!

We'll use the index for a TM's FSM to identify the TM as well.

So we can talk about TM 347 running on input 51, producing the answer 42.