[MUSIC PLAYING - "JESU, JOY OF MAN'S DESIRING" BY JOHANN SEBASTIAN BACH]

PROFESSOR: Well, up 'til now, I suppose, we've been learning about a lot of techniques for organizing big programs, symbolic manipulation a bit, some of the technology that you use for establishing languages, one in terms of another, which is used for organizing very large programs. In fact, the nicest programs I know look more like a pile of languages than like a decomposition of a problem into parts. Well, I suppose at this point, there are still, however, a few mysteries about how this sort of stuff works.

And so what we'd like to do now is diverge from the plan of telling you how to organize big programs, and rather tell you something about the mechanisms by which these things can be made to work. The main reason for this is demystification, if you will, that we have a lot of mysteries left, like exactly how it is the case that a program is controlled, how a computer knows what the next thing to do is, or something like that. And what I'd like to do now is make that clear to you, that even if you've never played with a physical computer before, the mechanism is really very simple, and that you can understand it completely with no trouble.

So I'd like to start by imagining that we-- well, the way we're going to do this, by the way, is we're going to take some very simple Lisp programs, very simple Lisp programs, and transform them into hardware. I'm not going to worry about some intermediate step of going through some existing computer machine language and then showing you how that computer works, because that's not as illuminating. So what I'm really going to show you is how a piece of machinery can be built to do a job that you have written down as a program. That program is, in fact, a description of a machine.

We're going to start with a very simple program, proceed to show you some simple mechanisms, proceed to a few more complicated programs, and then later show you a not very complicated program, how the evaluator transforms into a piece of hardware. And of course at that point, you have made the universal transition and can execute any program imaginable with a piece of well-defined hardware.

Well, let's start up now, give you a real concrete feeling for this sort of thing. Let's start with a very simple program. Here's Euclid's algorithm. It's actually a little bit more modern than Euclid's algorithm. Euclid's algorithm for computing the greatest common divisor of two numbers was invented 350 BC, I think. It's the oldest known algorithm.

But here we're going to talk about GCD of A and B, the Greatest Common Divisor or two numbers, A and B. And the algorithm is extremely simple. If B is 0, then the result is going to be A. Otherwise, the result is the GCD of B and the remainder when A is divided by B.

So this we have here is a very simple iterative process. This a simple recursive procedure, recursively defined procedure, recursive definition, which yields an iterative process. And the way it works is that every step, it determines whether B was zero. And if B is 0, we got the answer in A. Otherwise, we make another step where A is the old B, and B is the remainder of the old A divided by the old B. Very simple.

Now this, I've already told you some of the mechanism by just saying it that way. I set it in time. I said there are certain steps, and that, in fact, one of the things you can see here is that one of the reasons why this is iterative is nothing is needed of the last step to get the answer. All of the information that's needed to run this algorithm is in A and B. It has two well-defined state variables.

So I'm going to define a machine for you that can compute you GCDs. Now let's see. Every computer that's ever been made that's a single-process computer, as opposed to a multiprocessor of some sort, is made according to the same plan. The plan is the computer has two parts, a part called the datapaths, and a part called the controller.

The datapaths correspond to a calculator that you might have. It contains certain registers that remember things, and you've all used calculators. It has some buttons on it and some lights. And so by pushing the various buttons, you can cause operations to happen inside there among the registers, and some of the results to be displayed.

That's completely mechanical. You could imagine that box has no intelligence in it. Now it might be very impressive that it can produce the sine of a number, but that at least is apparently possibly mechanical. At least, I could open that up in the same way I'm about to open GCD.

So this may have a whole computer inside of it, but that's not interesting. Addition is certainly simple. That can be done without any further mechanism.

Now also, if we were to look at the other half, the controller, that's a part that's dumb, too. It pushes the buttons. It pushes them according to the sequence, which is written down on a piece of paper, and observes the lights.

And every so often, it comes to a place in a sequence that says, if light A is on, do this sequence. Otherwise, do that sequence. And thereby, there's no complexity there either. Well, let's just draw that and see what we feel about that.

So for computing GCDs, what I want you to think about is that there are these registers. A register is a place where I store a number, in this case. And this one's called a. And then there's another one for storing b.

Now we have to see what things we can do with these registers, and they're not entirely obvious what you can do with them. Well, we have to see what things we need to do with them. We're looking at the problem we're trying to solve.

One of the important things for designing a computer, which I think most designers don't do, is you study the problem you want to solve and then use what you learn from studying the problem you want to solve to put in the mechanisms needed to solve it in the computer you're building, no more no less. Now it may be that the problem you're trying to solve is everybody's problem, in which case you have to build in a universal interpreter of some language. But you shouldn't put any more in than required to build the universal interpreter of some language. We'll worry about that in a second.

OK, going back to here, let's see. What do we have to be able to do? Well, somehow, we have to be able to get B into A. We have to be able to get the old value of B into the value of A. So we have to have some path by which stuff can flow, whatever this information is, from b to a. I'm going to draw that with by an arrow saying that it is possible to move the contents of b into a, replacing the value of a. And there's a little button here which you push which allows that to happen. That's what the little x is here.

Now it's also the case that I have to be able to compute the remainder of a and b. Now that may be a complicated mess. On the other hand, I'm going to make it a small box. If we have to, we may open up that box and look inside and see what it is.

So here, I'm going to have a little box, which I'm going to draw this way, which we'll call the remainder. And it's going to take in a. That's going to take in b. And it's going to put out something, the remainder of a divided by b.

Another thing we have to see here is that we have to be able to test whether b is equal to 0. Well, that means somebody's got to be looking at-- a thing that's looking at the value of b. I have a light bulb here which lights up if b equals 0. That's its job.

And finally, I suppose, because of the fact that we want the new value of a to be the old value of b, and simultaneously the new value of b to be something I've done with a, and if I plan to make my machine such that everything happens one at a time, one motion at a time, and I can't put two numbers in a register, then I have to have another place to put one while I'm interchanging. OK? I can't interchange the two things in my hands, unless I either put two in one hand and then pull it back the other way, or unless I put one down, pick it up, and put the other one, like that, unless I'm a juggler, which I'm not, as you can see, in which case I have a possibility of timing errors. In fact, much of the type of computer design people do involves timing errors, of some potential timing errors, which I don't much like.

So for that reason, I have to have a place to put the second one of them down. So I have a place called t, which is a register just for temporary, t, with a button on it. And then I'll take the result of that, since I have to take that and put into b, over here, we'll take the result of that and go like this, and a button here. So that's the datapaths of a GCD machine.

Now what's the controller? Controller's a very simple thing, too. The machine has a state.

The way I like to visualize that is that I've got a maze. And the maze has a bunch of places connected by directed arrows. And what I have is a marble, which represents the state of the controller. The marble rolls around in the maze. Of course, this analogy breaks down for energy reasons. I sometimes have to pump the marble up to the top, because it's going to otherwise be a perpetual motion machine. But not worrying about that, this is not a physical analogy.

This marble rolls around. And every time it rolls around certain bumpers, like in a pinball machine, it pushes one of these buttons. And every so often, it comes to a place, which is a division, where it has to make a choice. And there's a flap, which is controlled by this. So that's a really mechanical way of thinking about it.

Of course, controllers these days, are not built that way in real computers. They're built with a little bit of ROM and a state register. But there was a time, like the DEC PDP-6, where that's how you built the controller of a machine. There was a bit that ran around the delay line, and it triggered things as it went by. And it would come back to the beginning and get fed round again.

And of course, there were all sorts of great bugs you could have like two bits going around, two marbles. And then the machine has lost its marbles. That happens, too. Oh, well.

So anyway, for this machine, what I have to do is the following. I'm going to start my maze here. And the first thing I've got to do, in a notation which many of you are familiar with, is b equal to zero, a test. And there's a possibility, either yes, in which case I'm done. Otherwise, if no, then I'm going have to roll over some bumpers.

I'm going to do it in the following order. I want to do this interchange game. Now first, since I need both a and b, but then the first-- and this is not necessary-- I want to collect this. This is the thing that's going to go into b. So I'm going to say, take this, which depends upon both a and b, and put the remainder into here. So I'm going to push this button first. Then, I'm going to transfer b to a, push that button, and then I transfer the temporary into b, push that button. So a very sequential machine, it's very inefficient. But that's fine right now.

We're going to name the buttons, t gets remainder. a gets b. And b gets t. And then I'm going to go around here and it's to go back to start.

And if you look, what are we seeing here? We're seeing the various-- what I really have is some sort of mechanical connection, where t gets r controls this thing. And I have here that a gets b controls this fellow over here, and this fellow over here.

Boy, that's absolutely pessimal, the inverse of optimal. Every line heads across every other line the way I drew it. I suppose this goes here, b gets t.

Now I'd like to run this machine. But before I run the machine, I want to write down a description of this controller, just so you can see that these things, of course, as usual, can be written down in some nice language, so that we don't have to always draw these diagrams. One of the problems with diagrams is that they take up a lot of space. And for a machine this small, it takes two blackboards. For a machine that's the evaluator machine, I have trouble putting it into this room, even though it isn't very big. So I'm going to make a little language for this that's just a description of that, saying define a machine we'll call GCD.

Of course, once we have something like this, we have a simulator for it. And the reason why we want to build a language in this form, is because all of a sudden we can manipulate these expressions that I'm writing down. And then of course I can write things that can algebraically manipulate these things, simulate them, all that sort of things that I might want to do, perhaps transform them as a layout, who knows. Once I have a nice representation of registers, it has certain registers, which we can call A, B, and T. And there's a controller.

Actually, a better language, which would be more explicit, would be one which named every button also and said what it did. Like, this button causes the contents of T to go to the contents of B. Well I don't want to do that, because it's actually harder to read to do that, and it takes up more space. So I'm going to have that in the instructions written in the controller.

It's going to be implicit what the operations are. They can be deduced by reading these and collecting together all the different things that can be done. Well, let's just look at what these things are. There's a little loop that we go around which says branch, this is the representation of the little flap that decides which way you go here, if 0 fetch of B, the contents of B, and if the contents of B is 0, then go to a place called done.

Now, one thing you're seeing here, this looks very much like a traditional computer language. And what you're seeing here is things like labels that represent places in a sequence written down as a sequence. The reason why they're needed is because over here, I've written something with loops.

But if I'm writing English text, or something like that, it's hard to refer to a place. I don't have arrows. Arrows are represented by giving names to the places where the arrows terminate, and then referring to them by those names. Now this is just an encoding. There's nothing magical about things like that.

Next thing we're going to do is we're going to say, how do we do T gets R? Oh, that's easy enough, assign. We assign to T the remainder. Assign is the name of the button. That's the button-pusher. Assign to T the remainder, and here's the representation of the operation, when we divide the fetch of A by the fetch of B.

And we're also going to assign to A the fetch of B, assign to B the result of getting the contents of T. And now I have to refer to the beginning here. I see, why don't I call that loop like I have here? So that's that reference to that arrow. And when we're done, we're done. We go to here, which is the end of the thing.

So here's just a written representation of this fragment of machinery that we've drawn here.

Now the next thing I'd like to do is run this. I want us to feel it running. Never done this before, you got to do it once.

So let's take a particular problem. Suppose we want to compute the GCD of a equals 30 and b equals 42. I have no idea what that is right now. But a 30 and b is 42. So that's how I start this thing up.

Well, what's the first thing I do? I say is B equal to 0, no. Then assign to T the remainder of the fetch of A and the fetch of B. Well the remainder of 30 when divided by 42 is itself 30. Push that button.

Now the marble has rolled to here. A gets B. That pushes this button. So 42 moves into here.

B gets C. Push that button. The 30 goes here. Let met just interchange them.

Now let's see, go back to the beginning. B 0, no. T gets the remainder. I suppose the remainder when dividing 42 by 30 is 12. I push that one.

Next thing I do is allow the 30 to go to here, push this one, allow the 12 to go to here. Go around this thing. Is that done? No. How about-- so now I have to find out the remainder of 30 divided by 12. And I believe that's 6. So 6 goes here on this button push. Then the next thing I push is this one, which the 12 goes into here.

Then I push this button. The 6 gets into here. Is 6 equal to 0? No. OK.

So then at that point, the next thing to do is divide it. Ooh, this has got a remainder of 0. Looks like we're almost done.

Move the 6 over here next. 0 over here. Is the answer 0? Yes. B is 0, therefore the answer is in A.

The answer is 6. And indeed that's right, because if we look at the original problem, what we have is 30 is 2 times 3 times 5, and 42 is 2 times 3 times 7. So the greatest common divisor is 2 times 3, which is 6.

Now normally, we write one other little line here, just to make it a little bit clearer, which is that we leave in a connection saying that this light is the guy that that flap looks at. Of course, any real machine has a lot more complicated things in it than what I've just shown you. Let's look for a second at the first still store.

Wow. Well you see, for example, one thing we might want to do is worry about the operations that are of IO form. And we may have to collect something from the outside. So a state machine that we might have, the controller may have to, for example, get a value from something and put register a to load it up. I have to master load up register b with another value.

And then later, when I'm done, I might want to print the answer out. And of course, that might be either simple or complicated. I'm writing, assuming print is very simple, and read is very simple. But in fact, in the real world, those are very complicated operations, usually much, much larger and more complicated than the thing you're doing as your problem you're trying to solve.

On the other hand, I can remember a time when, I remember using IBM 7090 computer of sorts, where things like read and write of a single object, a single number, a number, is a primitive operation of the IO controller. OK?

And so we have that kind of thing in there. And in such a machine, well, what are we really doing? We're just saying that there's a source over here called "read," which is an operation which always has a value. We have to think about this as always having a value which can be gated into either register a or b. And print is some sort of thing which when you gate it appropriately, when you push the button on it, will cause a print of the value that's currently in register a. Nothing very exciting.

So that's one sort of thing you might want to have. But these are also other things that are a little bit worrisome. Like I've used here some complicated mechanisms.

What you see here is remainder. What is that? That may not be so obvious how to compute. It may be something which when you open it up, you get a whole machine. OK? In fact, that's true.

For example, if I write down the program for remainder, the simplest program for it is by repeated subtraction. Because of course, division can be done by repeated subtraction of numbers, of integers. So the remainder of N divided by D is nothing more than if N is less than D, then the result is N. Otherwise, it's the remainder when we subtract D from N with respect to D, when divided by D. Gee, this looks just like the GCD program.

Of course, it's not a very nice way to do remainders. You'd really want to use something like binary notation and shift and things like that in a practical computer. But the point of that is that if I open this thing up, I might find inside of it a computer.

Oh, we know how to do that. We just made one. And it could be another thing just like this.

On the other hand, we might want to make a more efficient or better-structured machine, or maybe make use of some of the registers more than once, or some horrible mess like that that hardware designers like to do, and for very good reasons. So for example, here's a machine that you see, which you're not supposed to be able to read. It's a little bit complicated. But what it is is the integration of the remainder into the GCD machine. And it takes, in fact, no more registers. There are three registers in the datapaths. OK?

But now there's a subtractor. There are two things that are tested. Is b equal to 0, or is t less than b?

And then the controller, which you see over here, is not much more complicated. But it has two loops in it, one of which is the main one for doing the GCD, and one of which is the subtraction loop for doing the remainder sub-operation. And there are ways, of course, of, if you think about it, taking the remainder program. If I take remainder, as you see over there, as a lambda expression, substitute it in for remainder over here in the GCD program, then do some simplification by substituting a and b for remainder in there, then I can unwind this loop. And I can get this piece of machinery by basically, a little bit of algebraic simplification on the lambda expressions.

So I suppose you've seen your first very simple machines now. Are there any questions? Good. This looks easy, doesn't it? Thank you. I suppose, take a break.

[MUSIC PLAYING - "JESU, JOY OF MAN'S DESIRING" BY JOHANN SEBASTIAN BACH]

PROFESSOR: Well, let's see. Now you know how to make an iterative procedure, or a procedure that yields an iterative process, turn into a machine. I suppose the next thing we want to do is worry about things that reveal recursive processes. So let's play with a simple factorial procedure.

We define factorial of N to be if n is 1, the result is 1, using 1 right now to decrease the amount of work I have to do to simulate it, else it's times N factorial N minus 1. And what's different with this program, as you know, is that after I've computed factorial of N minus 1 here, I have to do something to the result. I have to multiply it by N.

So the only way I can visualize what this machine is doing, because of the fact-- think of it this way, that I have a machine out here which somehow needs a factorial machine in order to compute its answer. But this machine, the outer machine, has to exist before and after the factorial machine, which is inside. Whereas in the iterative case, the outer machine doesn't need to exist after the inner machine is running, because you never need to go back to the outer machine to do anything.

So here we have a problem where we have a machine which has the same machine inside of it, an infinitely large machine. And it's got other things inside of it, like a multiplier, which takes some inputs, and there's a minus 1 box, and things like that. You can imagine that's what it looks like.

But the important thing is that here I have something that happens before and after, in the outer machine, the execution of the inner machine. So this machine has to have a life. It has to exist on both times sides of this machine.

So somehow, I have to have a place to store the things that this thing needs to run. Infinite objects don't exist in the real world. What we have to do is arrange an illusion that we have an infinite object, we have an infinite amount of hardware somewhere.

Now of course, illusion's all that really matters. If we can arrange that every time you look at some infinite object, the part of it that you look at is there, then it's as infinite as you need it to be. And of course, one of the things we might want to do, just look at this thing over here, is the organization that we've had so far involves having a part of the machine, which is the controller, which sits right over here, which is perfectly finite and very simple. We have some datapaths, which consist of registers and operators. And what I propose to do here is decompose the machine into two parts, such that there is a part which is fundamentally finite, and some part where a certain amount of infinite stuff can be kept.

On the other hand this is very simple and really isn't infinite, but it's just very large. But it's so simple that it could be cheaply reproduced in such large amounts, we call it memory, that we can make a structure called a stack out of it which will allow us to, in fact, simulate the existence of an infinite machine which is made out of a recursive nest of many machines. And the way it's going to work is that we're going to store in this place called the stack the information required after the inner machine runs to resume the operation of the outer machine.

So it will remember the important things about the life of the outer machine that will be needed for this computation. Since, of course, these machines are nested in a recursive manner, then in fact the stack will only be accessed in a manner which is the last thing that goes in is the first thing that comes out. So we'll only need to access some little part of this stack memory.

OK, well, let's do it. I'm going to build you a datapath now, and I'm going to write the controller. And then we're going to execute this to see how you do it. So the factorial machine isn't so bad. It's going to have a register called the value, where the answer is going to be stored, and a registered called N, which is where the number I'm taking factorial will be stored, factorial of. And it will be necessary in some instances to connect VAL to N.

In fact, one nice case of this is if I just said over here, N, because that would be right for N equal 1N. And I could just move the answer over there if that's important. I'm not worried about that right now.

And there are things I have to be able to do. Like I have to be able to, as we see here, multiply N by something in VAL, because VAL is the result of computing factorial. And I have to put the result back into VAL.

So here we can see that the result of computing a factorial is N times the result of computing a factorial. VAL will be the representation of the answer of the inner factorial. And so I'm going to have to have a multiplier here, which is going to sample the value of N and the value of VAL and put the result back into VAL like that.

I'm also going to have to be able to see if N is 1. So I need a light bulb. And I suppose the other thing I'm going to need to have is a way of decrementing N. So I'm going to have a decrementer, which takes N and is going to put back the result into N. That's pretty much what I need in my machine.

Now, there's a little bit else I need. It's a little bit more complicated, because I'm also going to need a way to store, to save away, the things that are going to be needed for resuming the computation of a factorial after I've done a sub-factorial. What's that? One thing I need is N.

So I'm going to build here a thing called a stack. The stack is a bunch of stuff that I'm going to write in sequentially. I don't how long it is. The longer it is, the better my illusion of infinity. And I'm going to have to have a way of getting stuff out of N and into the stack and vice versa. So I'm going to need a connection like this, which is two-way, whereby I can save the value of N and then restore it some other time through that connection. This is the stack.

I also need a way of remembering where I was in the computation of factorial in the outer program. Now in the case of this machine, it isn't very much a problem. Factorial always returns, has to go back to the place where we multiply by N, except for the last time, when it has to return to whatever needs the factorial or go to done or stop. However, in general, I'm going to have to remember where I have been, because I might have computed factorial from somewhere else. I have to go back to that place and continue there.

So I'm going to have to have some way of taking the place where the marble is in the finite state controller, the state of the controller, and storing that in the stack as well. And I'm going to have to have ways of restoring that back to the state of the-- the marble. So I have to have something that moves the marble to the right place.

Well, we're going to have a place which is the marble now. And it's called the continue register, called continue, which is the place to put the marble next time I go to continue. That's what that's for. And so there's got to be some path from that into the controller.

I also have to have some way of saving that on the stack. And I have to have some way of setting that up to have various constants, a certain fixed number of constants. And that's very easy to arrange. So let's have some constants here. We'll call this one after-fact. And that's a constant which we'll get into the continue register, and also another one called fact-done.

So this is the machine I want to build. That's its datapaths, at least. And it mixes a little with the controller here, because of the fact that I have to remember where I was and restore myself to that place.

But let's write the program now which represents the controller. I'm not going to write the define machine thing and the register list, because that's not very interesting. I'm just going to write down the sequence of instructions that constitute the controller.

So we have assign, to set up, continue to done. We have a loop which says branch if equal 1 fetch N, if N is 1, then go to the base step of the induction, the simple case.

Otherwise, I have to remember the things that are necessary to perform a sub-factorial. I'm going to go over here, and I have to perform a sub-factorial. So I have to remember what's needed after I will be done with that.

See, I'm about to do something terrible. I'm about to change the value of N. But this guy has to know the old value of N. But in order to make the sub-factorial work, I have to change the value of N. So I have to remember the old value. And I also have to remember where I've been. So I save up continue.

And this is an instruction that says, put something in the stack. Save the contents of the continuation register, which in this case is done, because later I'm going to change that, too, because I need to go back to after-fact, as well. We'll see that.

We save N, because I'm going to need that for later. Assign to N the decrement of fetch N. Assign continue, we're going to look at this now, to after, we'll call it. That's a good name for this, a little bit easier and shorter, and fits in here.

Now look what I'm doing here. I'm saying, if the answer is 1, I'm done. I'm going to have to just get the answer. Otherwise, I'm going to save the continuation, save N, make N one less than N, remember I'm going to come back to someplace else, and go back and start doing another factorial.

However, I've got a different machine [? in me ?] now. N is 1, and continue is something else. N is N minus 1.

Now after I'm done with that, I can go there. I will restore the old value of N, which is the opposite of this save over here. I will restore the continuation.

I will then go to here. I will assign to the VAL register the product of N and fetch VAL. VAL fetch product assign.

And then I will be done. I will have my answer to the sub-factorial in VAL. At that point, I'm going to return by going to the place where the continuation is pointing. That says, go to fetch continue.

And then I have finally a base step, which is the immediate answer. Assign to VAL fetch N, and go to fetch continue. And then I'm done.

Now let's see how this executes on a very simple case, because then we'll see the use of this stack to do the job we need. This is statically what it's doing, but we have look dynamically at this. So let's see.

First thing we do is continue gets done. The way that happened is I pushed this. Let's call that done the way I have it. I push that button. Done goes into there.

Now I also have to set this thing up to have an initial value. Let's consider a factorial of three, a simple case. And we're going to start out with our stack growing over here. Stacks have their own little internal state saying where they are, where the next place I'm going to write is.

So now we say, is N 1? The answer is no. So now I'm going to save continue, bang. Now that done goes in here. And this moves to here, the next place I'm going to write.

Save N 3. OK? Assign to N the decrement of N. That means I've pushed this button. This becomes 2.

Assign to continue aft. So I've pushed that button. Aft goes in here.

OK, now go to loop, bang, so up to here. Is N 1? No.

So I have to save continue. What's continue? Continue is aft. Push this button. So this moves to here.

I have to save N. N is over here. I got to 2. Push that button. So a 2 gets written there. And then this thing moves down here.

OK, save N. Assign N to the decrement of N. This becomes a 1. Assign continue to aft. A-F-T gets written there again.

Go to loop. Is N equal to 1? Oh, yes, the answer is 1.

OK, go to base step. Assign to VAL fetch of N. Bang, 1 gets put in there.

Go to fetch continue. So we look in continue. Basically, I'm pushing a button over here that goes to the controller. The continue becomes aft, and all of a sudden, the program's running here.

I now have to restore the outer version of factorial. So we go here. We say, restore N. So restore N means take the contents that's here. Push this button, and it goes into here, 2, and the pointer moves up.

Restore continue, pretty easy. Go push this button. And then aft gets written in here again. That means this thing moves up. I've gotten rid of something else on my stack.

Right, then I go to here, which says, assign to VAL the product of N an VAL. So I push this button over here, bang. 2 times 1 gives me a 2, get written there.

Go to fetch continue. Continue is aft. I go to aft. Aft says restore N. Do your restore N, means I take the value over here, which is 3, push this up to here, and move it into here, N. Now it's pushing that button.

The next thing I do is restore continue. Continue is now going to become done. So this moves up here when I push this button. Done may or may be there anymore, I'm not interested, but it certainly is here.

Next thing I do is assign to VAL the product of the fetch of N and the fetch of VAL. That's pushing this button over here, bang. 2 times 3 is 6. So I get a 6 over here.

And go to fetch continue, whoops, I go to done, and I'm done. And my answer is 6, as you can see in the VAL register. And in fact, the stack is in the state it originally was in.

Now there's a bit of discipline in using these things like stacks that we have to be careful of. And we'll see that in the next segment. But first I want to ask if there are any questions for this. Are there any questions? Yes, Ron.

AUDIENCE: What happens when you roll off the end of the stack with--

PROFESSOR: What do you mean, roll off of?

AUDIENCE: Well, the largest number-- a larger starting point of N requires more memory, correct?

PROFESSOR: Oh, yes. Well, I need to have a long enough stack. You say, what if I violate my illusion?

AUDIENCE: Yes.

PROFESSOR: Well, then the magic doesn't work. The truth of the matter is that every machine is finite. And for a procedure like this, there's a limit to the number of sub-factorials I could have.

Remember when we were doing the y-operator a while ago, we pointed out that there was a sequence of exponentiation procedures, each of which was a little better than the previous one. Well, we're now seeing how we implement that mathematical idea. The limiting process is only so good as as far as you take the limit.

If you think about it, what am I using here? I'm using about two pieces of memory for every recursion of this process. If we try to compute factorial of 10,000, that's not a lot of memory. On the other hand, it's an awful big number.

So the question is, is that a valuable thing in this case. But it really turns out not to be a terrible limit, because memory is el cheapo, and people are pretty expensive. OK, thank you, let's take a break.

[MUSIC PLAYING - "JESU, JOY OF MAN'S DESIRING" BY JOHANN SEBASTIAN BACH]

PROFESSOR: Well, let's see. What I've shown you now is how to do a simple iterative process and a simple recursive process. I just want to summarize the design of simple machines for specific applications by showing you a little bit more complicated design, that of a thing that does doubly recursive Fibonacci, because it will indicate to us, and we'll understand, a bit about the conventions required for making stacks operate correctly.

So let's see. I'm just going to write down, first of all, the program I'm going to translate. I need a Fibonacci procedure, it's very simple, which says, if N is less than 2, the result is N, otherwise it's the sum of Fib of N minus 1 and Fib of N minus 2. That's the plan I have here.

And we're just going to write down the controller for such a machine. We're going to assume that there are registers, N, which holds the number we're taking Fibonacci of, VAL, which is where the answer is going to get put, and continue, which is the thing that's linked to the controller, like before. But I'm not going to draw another physical datapath, because it's pretty much the same as the last one you've seen.

And of course, one of the most amazing things about computation is that after a while, you build up a little more features and a few more features, and all of the sudden, you've got everything you need. So it's remarkable that it just gets there so fast. I don't need much more to make a universal computer.

But in any case, let's look at the controller for the Fibonacci thing. First thing I want to do is start the thing up by assign to continue a place called done, called Fib-done here. So that means that somewhere over here, I'm going to have a label, Fib-done, which is the place where I go when I want the machine to stop. That's what that is.

And I'm going to make up a loop. It's a place I'm going to go to in order to start up computing a Fib. Whatever is in N at this point, Fibonacci will be computed of, and we will return to the place specified by continue.

So what you're going to see here at this place, what I want here is the contract that says, I'm going to write this with a comment syntax, the contract is N contains arg, the argument. Continue is the recipient. And that's where it is. At this point, if I ever go to this place, I'm expecting this to be true, the argument for computing the Fibonacci.

Now the next thing I want to do is to branch. And if N is less than 2-- by the way, I'm using what looks like Lisp syntax. This is not Lisp. This does not run. What I'm writing here does not run as a simple Lisp program. This is a representation of another language.

The reason I'm using the syntax of parentheses and so on is because I tend to use a Lisp system to write an interpreter for this which allows me to simulate the machine I'm trying to build. I don't want to confuse this to think that this is Lisp code. It's just I'm using a lot of the pieces of Lisp. I'm embedding a language in Lisp, using Lisp as pieces to make my process of making my simulator easy. So I'm inheriting from Lisp all of its properties.

Fetch of N 2, I want to go to a place called immediate answer. It's the base step. Now, that's somewhere over here, just above done. And we'll see it later.

Now, in the general case, which is the part I'm going to write down now, let's just do it. Well, first of all, I'm going to have to call Fibonacci twice. In each case-- well, in one case at least, I'm going to have to know what to do to come back and do the next one. I have to remember, have I done the first Fib, or have I done the second one? Do I have to come back to the place where I do the second Fib, or do I have to come back to the place where I do the add?

In the first case, over the first Fibonacci, I'm going to need the value of N for computing for the second one. So I have to store some of these things up. So first I'm going to save continue. That's who needs the answer. And the reason I'm doing that is because I'm about to assign continue to the place which is the place I want to go to after.

Let's call it Fib-N-minus-1, big long name, classic Lisp name. Because I'm going to compute the first Fib of N minus 1, and then after that, I want to come back and do something else. That's the place I want to go to after I've done the first Fibonacci calculation. And I want to do a save of N, because I'm going to need it later, after that.

Now I'm going to, at this point, get ready to do the Fibonacci of N minus 1. So assign to N the difference of the fetch of N and 1. Now I'm ready to go back to doing the Fib loop.

Have I satisfied my contract? And the answer is yes. N contains N minus 1, which is what I need. Continue contains a place I want to go to when I'm done with calculating N minus 1. So I've satisfied the contract. And therefore, I can write down here a label, after-Fib-N-minus-1.

Now what am I going to do here? Here's a place where I now have to get ready to do Fib of N minus 2. But in order to do a Fib of N minus 2, look, I don't know. I've clobbered my N over here. And presumably my N is counted down all the way to 1 or 0 or something at this point. So I don't know what the value of N in the N register is.

I want the value of N that was on the stack that I saved over here so that could restore it over here. I saved up the value of N, which is this value of N at this point, so that I could restore it after computing Fib of N minus 1, so that I could count that down to N minus 2 and then compute Fib of N minus 2. So let's restore that. Restore of N.

Now I'm about to do something which is superstitious, and we will remove it shortly. I am about to finish the sequence of doing the subroutine call, if you will. I'm going to say, well, I also saved up the continuation, since I'm going to restore it now.

But actually, I don't have to, because I'm not going to need it. We'll fix that in a second. So we'll do a restore of continue, which is what I would in general need to do. And we're just going to see what you would call in the compiler world a peephole optimization, which says, whoops, you didn't have to do that.

OK, so the next thing I see here is that I have to get ready now to do Fibonacci of N minus 2. But I don't have to save N anymore. The reason why I don't have to save N anymore is because I don't need N after I've done Fib of N minus 2, because the next thing I do is add. So I'm just going to set up my N that way. Assign N minus difference of fetch N and 2.

Now I have to finish the setup for calling Fibonacci of N minus 2. Well, I have to save up continue and assign continue, continue, to the place which is after Fib N 2, that place over here somewhere. However, I've got to be very careful. The old value, the value of Fib of N minus 1, I'm going to need later.

The value of Fibonacci of N minus 1, I'm going to need. And I can't clobber it, because I'm going to have to add it to the value of Fib of N minus 2. That's in the value register, so I'm going to save it. So I have to save this right now, save up VAL. And now I can go off to my subroutine, go to Fib loop.

Now before I go any further and finish this program, I just want to look at this segment so far and see, oh yes, there's a sequence of instructions here, if you will, that I can do something about. Here I have a restore of continue, a save of continue, and then an assign of continue, with no other references to continue in between. The restore followed by the save leaves the stack unchanged.

The only difference is that I set the continue register to a value, which is the value that was on the stack. Since I now clobber that value, as in it was never referenced, these instructions are unnecessary. So we will remove these.

But I couldn't have seen that unless I had written them down. Was that really true? Well, I don't know.

OK, so we've now gone off to compute Fibonacci of N minus 2. So after that, what are we going to do? Well, I suppose the first thing we have to do-- we've got two things. We've got a thing in the value register which is now valuable. We also have a thing on the stack that can be restored into the value register. And what I have to be careful with now is I want to shuffle this right so I can do the multiply.

Now there are various conventions I might use, but I'm going to be very picky and say, I'm only going to restore into a register I've saved from. If that's the case, I have to do a shuffle here. It's the same problem with how many hands I have. So I'm going to assign to N, because I'm not going to need N anymore, N is useless, the current value of VAL, which was the value of Fib of N minus 2.

And I'm going to restore the value register now. This restore matches this save. And if you're very careful and examine very carefully what goes on, restores and saves are always matched. Now there's an outstanding save, of course, that we have to get rid of soon.

And so I restored the value register. Now I restore the continue one, which matches this one, dot, dot, dot, dot, dot, dot, dot, down to here, restoring that continuation. That continuation is a continuation of Fib of N, which is the problem I was trying to solve, a major problem I'm trying to solve. So that's the guy I have to go back to who wants Fib of N. I saved them all the way up here when I realized N was not less than 2. And so I had to do a complicated operation.

Now I've got everything I need to do it. So I'm going to restore that, assign to VAL the sum of fetch VAL and fetch of N, and go to continue. So now I've returned from computing Fibonacci of N, the general case. Now what's left is we have to fix up a few details, like there's the base case of this induction, immediate answer, which is nothing more than assign to VAL fetch of N, because N was less than 2, and therefore, the answer is N in our original program, and return continue-- bobble, bobble almost-- and finally Fib done.

So that's a fairly complicated program. And the reason I wanted you see to that is because I want you to see the particular flavors of stack discipline that I was obeying. It was first of all, I don't want to take anything that I'm not going to need later. I was being very careful. And it's very important. And there are all sorts of other disciplines people make with frames and things like that of some sort, where you save all sorts of junk you're not going to need later and restore it because, in some sense, it's easier to do that. That's going to lead to various disasters, which we'll see a little later.

It's crucial to say exactly what you're going to need later. It's an important idea. And the responsibility of that is whoever saves something is the guy who restores it, because he needs it. And in such discipline, you can see what things are unnecessary, operations that are unimportant.

Now, one other thing I want to tell you about that's very simple is that, of course, the picture you see is not the whole picture. Supposing I had systems that had things like other operations, CAR, CDR, cons, building a vector and referencing the nth element of it, or things like that. Well, at this level of detail, whatever it is, we can conceptualize those as primitive operations in the datapath. In other words, we could say that some machine that, for example, has the append machine, which has to do cons of the CAR of x with the append of the CDR of x and y, well, gee, that's exactly the same as the factorial structure. Well, it's got about the same structure.

And what do we have? We have some sort of things in it which may be registers, x and y, and then x has to somehow move to y sometimes, x has to get the value of y. And then we may have to be able to do something which is a cons. I don't remember if I need to like this is in this system, but cons is sort of like subtract or add or something. It combines two things, producing a thing which is the cons, which we may then think goes into there. And then maybe a thing called the CAR, which will produce-- I can get the CAR or something. And maybe I can get the CDR of something, and so on.

But we shouldn't be too afraid of saying things this way, because the worst that could happen is if we open up cons, what we're going to find is some machine. And cons may in fact overlap with CAR and CDR, and it always does, in the same way that plus and minus overlap, and really the same business. Cons, CAR, and CDR are going to overlap, and we're going to find a little controller, a little datapath, which may have some registers in it, some stuff like that. And maybe inside it, there may also be an infinite part, a part that's semi-infinite or something, which is a lot of very uniform stuff, which we'll call memory.

And I wouldn't be so horrified if that were the way it works. In fact, it does, and we'll talk about that later. So are there any questions?

Gee, what an unquestioning audience. Suppose I tell you a horrible pile of lies. OK. Well, thank you. Let's take our break.

[MUSIC PLAYING - "JESU, JOY OF MAN'S DESIRING" BY JOHANN SEBASTIAN BACH]