ProgrammingParadigms-Lecture10

Instructor (Jerry Cain):Everyone, welcome. I have a good slew of handouts for you today. Sorry about the lines on the photocopies. It's the just the photocopier, it's not me putting this really annoying background image behind all the text. It's Chapters 3 and 4 of this little computer architecture series of handouts that I'm giving you. I'm gonna spend today and easily the rest of Friday talking about all this stuff and I'll go through plenty of examples.

You also get your fourth assignment today. I'm gonna make it due next Thursday evening. I will tell you now that this is the one that surprises everybody a little bit. It's certainly doable, but there are aspects of using the vector and hash set that always take a few people by surprise, particularly how you store dynamically allocated C strings in these vectors and hash sets. If you've just made one key mistake, then you can actually waste an hour or two trying to figure out why it's not working even though it's compiling.

So be sensitive to the fact that you might just want to read through the handout, and maybe get the first 25 percent of the assignment done because immediately you have to start dealing with these C strings, and once you figure out how to store C strings in these things, you do much, much better and it goes much more smoothly after that. Now I'm gonna do something this week that I won't do very often, but I'm gonna have another discussion section this Friday at 2:15.

Now I'm just inventing the time and I know I can't check with everybody in the class as to whether it's convenient or not, so I just had to schedule it. We're gonna videotape it. We're gonna put it online. The discussion section just happens to come at a kind of crappy time in the assignment cycle. It's like two days before the assignment's due. It's not a disaster for the assignments that we've had so far, but I really want to show you some examples using the vector and the hash set in a structured discussion section, and I want to do it more than two days before the assignment deadline.

So this Friday at 2:15 if you wanna attend live by all means do it. Skilling 191. It's just this week. We will have discussion section next Tuesday as well. I just have this one I wanna insert into the sequence just so you have a little more practice before you really tackle the assignment this weekend, which is when I most of you will start.

Okay. I wanna continue with this cogeneration thing. I wanna get the piece of chalk that I'm gonna use, and I wanna talk a little bit more realistically about what activation records look like. I have kind of blown off the parameters, and where they reside in these activation records. I've only dealt with local variables, but all the functions I've invented have had no parameters passed in. I did that because I just wanted to simplify things.

So there's a couple of confessions I have to make about how I slightly misled you on Monday just to make things easier. If you have this as a function prototype, and you'll be able to infer structure based on this small example, I think. I'll just call it foo, and I'll pass in an int l bar and an int star r called baz, and internally I'll declare some variables. A char

array, I'll it snink. I have no idea what these words are; I'm just making them up. And then let's say a short star called Y.

And I don't care about the code at the moment because I just want to show you what the activation record will look like. Now obviously it shouldn't surprise you that this and this and this and this are all packed somewhere close to each other in memory. And in the example I gave you on Monday, I only had stuff like this. The way these are laid out relative to one another does not change. I would have a character array of 4 bytes, this would be this thing called snink, which is a word I've never used before, but here it is.

The four characters would be packed in a static array that resides inside the activation record. Below that, I don't have a short, I have an address to a short that is called Y. Now you may ask what about these things? These things are certainly close. Let me just draw the picture and explain why it looks the way it does. There is this reserved four byte figure that sits right there that has nothing to do with parameters or local variables.

But on top of that there's gonna be bar and on top of that there's gonna be this thing called baz. Okay. And this right here is the full 20 byte activation record that makes up – or that accompanies any call to the foo function right here. Now I put bar below baz. Is that arbitrary? The answer is it is not arbitrary. Now I can't really gracefully explain why this goes below that and why – basically parameters are laid down from high to low address from right to left.

In other words, the zero parameter here is always below all of the other ones. And the first is stacked on top of that and the second is stacked on top of that. Give me 40 minutes of more lecture and I'll be able to explain why that has to be the case in a language like C and C++. These right here are actually stacked in the order they appear. All of these appear at lower addresses than these. This right here is something I'll be able to discuss a little bit more in about 15 minutes.

That is the space that sits in between parameters and local variables. It actually has information about the function that called us. Obviously, foo is invoked from the main function or from some other function or maybe even foo itself if it turns out to be recursive. We're gonna need to lay down a little piece of popcorn right there about where in the code base we actually found this call to foo. Okay. And when the function exits, it relies on this value right there, which I am also going to call a safe PC.

It's what the safe PC value would have been had some function call not interrupted the stream of instructions. Do you understand what I mean when I say that? Okay. But don't worry about how that's manipulated yet. Just understand that it's there and I'll be more sensitive to using it in a few minutes. So that is the activation record layout for an arbitrary function. What I want to do now is talk about how something like that is constructed because a function like foo is called.

Now I think foo is kind of a weird looking function, but I'm gonna go with it. Int main – I'm not gonna concern myself with the parameters – actually I lied. I will. Int arg c char

star star arg v and I'm gonna declare one local variable I = 4 and I'm gonna call foo of I and & of I. And then I'll just return zero at the end. Recall back to last Friday when I termed everything in terms of triangles and hexagons. I want to be a little bit more scientific than that.

And I wanna explain how we go from this, where this is arg c and this is arg v and this is set to point to something like 2, and this is set up to point to an array of char stars. Okay. I wanna figure out how we go from that, actually to technically this – I'll call it the safe PC here. When I generate code for this right here, eventually I have to generate code for the function call, you actually generate code to basically allocate space for all the local variables.

Only main's implementation knows how many local variables are needed in order to accomplish what it's trying to accomplish. So by protocol, the very first thing a function does – a C function does, is it makes space for its local variables. Main is called with a partial activation record. It has to complete the full activation record by doing something like this. That's P = SP - 4. Now remember on Monday I was using R1 to always track the base address of the activation record? Okay.

Well, R1 is actually supposed to be a general-purpose register, but there's actually a dedicated register. We just call it SP. It's short for stack pointer. Okay? And that is always the thing that is truly pointing to the lowest address in the stack that's irrelevant to execution. When main gets called SP is pointing right there. The allocation or the creation of this variable right here actually compiles to an instruction to demote this value by four more bytes.

Why four? Because I only have one four-byte figure right here. Okay. And then this becomes the boundary between what's being used in the stack segment and what's not in use. Does that sit well with everybody? Okay. Then it carries on and it does this one initialization right here. The next thing that would happen is that it would just do M. It was R1 on Monday, today it's SP = 4. That takes care of the assignment right there. Okay.

And then I have to actually prepare to call the foo function and wait for it to return before I go ahead and return zero. So the instructions that are executed on behalf of this foo function has to do what somebody did for the main function. It has to set aside space for the parameters. So it has to build a partial activation record for that thing right there. Okay.

It can tell because it sees the prototype how many bytes are gonna be above that safe PC or that return link. Because I have four bytes, and four bytes making up the upper half of the activation record, the first thing that would happen is that you would do something like SP = SP - 8. That would bring this down to there. Okay. This part right there is still the activation record for main, but I've just built 40 percent of the activation record that needs to be in place for foo to run. Does that make sense to people? Okay.

I have to pass in I and I also have to pass in the address of I. I have to make sure that the current value of I is place right there. The address of I is placed right there, and then I have to transfer control over to the code that emulates whatever foo is supposed to be doing. Okay? Does that make sense to people? Yes? No? Okay.

So in R1 I'm gonna do M of SP + 8. In R2, I'm gonna put the actual result of SP + 8. This address right there is relevant to both of those lines. What placed in R2 is the actual value of this thing. What placed in R1 is the contents of this thing right here. Okay. The reason I want to do that is because I want to lay down an M of SP and M of SP + 4 the actual values that need to be communicated to the foo function.

SP is the lower of the parameters. That's the thing that's supposed to get R1. This is supposed to get what I've laid down in R2. Okay. So what happens there is I have effectively done this. I've copied a 4 – actually, a 4 went there in response to that line right there. I copy a 4 right there. I effectively laid down that address in the second of the two boxes. Does that make sense?

Do you understand this part below the arc? It is basically 40 percent of that thing right up there. Yes? No? Okay. After I've done this, what I do in response to after I've set up the parameters, I actually transfer control to the foo function by using this assembly code instruction. That's basically a jump instruction that says jump to the very first assembly code instruction that's associated with the foo function, execute that, and then somehow figure out once you're done executing that to jump back to this right here.

Whatever this is right here, it's gonna actually do this – actually it's gonna do SP = SP + 8. I'll explain why it's that in a second. But do you understand that if I weren't jumping to the foo function that this would be the next thing that gets executed. Does that make sense to people? Okay. This address is actually, what gets laid down in this little safe PC thing. Okay. And it's automatically placed there by the call instruction.

At the time that the call instruction executes, it has a clear idea of what PC is so it knows what PC + 4 is. It actually on our behalf decrements SP by four more bytes and lays down that safe PC right there. So when foo is done executing it has information in its activation record about where to jump back to. Okay. This is basically at the electronic level or the hardware level, basically a piece of popcorn to remember where you were walking before you took a turn. Okay.

Does that make sense to everybody? Okay. This transfers control over to the foo function. Well all I'm gonna do as part of foo, is I'm gonna do something like let's say Y = short star snink + 2 and then I will do *Y = 50. And then I will return. Okay. What foo has to do is it has to complete its activation record by decrementing the stack pointer even further to accommodate and make space for those two variables right there.

So what happens is that this – let me actually write this somewhere else. What foo does, it makes space for these right here and that right there. What happens is SP is set equal to SP – 8. Why is it minus 8? Because it can tell while it's being compiled that that's how

many bytes are needed to extend the activation record distance enough just to complete the full activation record. So it does this, brings the stack pointer down to there, leaves it unitialized. Okay. You know that because C isn't an initialized local variables so the assembly code won't either.

And then it carries on and compiles code for this, and it accesses this variable and that variable relative to the value that it's the SP register. So the first thing that happens here is the value of Y, the contents of this thing right here, has to be updated to include the address of that thing right there. Okay. Does that make sense to people. Forget about the cast, I have to evaluate what snink + 2 is. That's really & of snink + 2. Okay.

So what I can do, in R1, I can set that equal to SP + 6. Why that? SP points to the base of the full activation record at this point, has to go four bytes beyond to basically circumvent Y, and go two bytes forward because it knows that snink is a char star so pointer [inaudible] the same thing, and it wants to store that address at this moment into a register so that it can be assigned to M of SP.

So R1 stores that value, M of SP identifies this space right there. That's what we want to get this value because that's the space that overlays the variable called Y. Okay. Make sense? As far as this is concerned, what has to happen is I have to get a 50 somewhere in memory. Where in memory? Well it's at whatever address happens to be stored in the Y variable.

So what would happen here – that separates the variable declaration. There's that. Into a local register, I would actually reload Y, R1 stores the address where a 50 should be written, but I only want to write it as a 2-byte figure because it's typed to point to a 2-byte figure. Does that make sense? Okay. So I would do M of R1 = .2, 50. Okay. Does that make sense to people? This is done. Okay. So now, it has to exit and basically return somehow to the main function that's right here. Okay.

You understand why that SP = SP - 8? Sits right there. That was to make space for the two local variables to basically make use of eight more bytes in the local stack segment for its local variable set. Well it has to promote SP back to where it was before it entered this function. This is basically the equivalent of – I don't want to say it's the equivalent of malloc; it's the equivalent of allocating space for variables, we have to deallocate that space. Okay.

So for SP=SP + 8 would be the last thing that's done right here. That leaves SP to be pointing right there. Do you understand that SP is addressing the very 4-byte figure that has the little piece of popcorn in it? Okay. Does that make sense to people? So this final instruction, RET for return, is understood to be an assembly code instruction that pulls this value out, places it – basically populates the true PC register with what this is, brings the SP register up four bytes and then carries on as if it were executing at PC + 4 all along. Does that sit well with everybody? Okay.

So there's that. As far as main is concerned, this is the next instruction that gets executed after this return executes. Okay. It jumps back, and it puts the address of that in the PC register. This SP is equal to SP + 8. Actually deallocates the space that's set aside for these two parameters that it put down there in preparation for the foo call. Does that make sense? Okay. So that's what that is, and then what I do, is I use another register. There's not too many dedicated registers with special names, but this is one more of them.

We have PC, we have SP, we also have this thing called RV which is this four byte register dedicated to communicating return values between caller and callee functions. Okay. Does that make sense to people? Okay. I'm returning a zero to whoever called main, so you have to think of RV as this little cubbyhole where return information is placed so that once it jumps back to whatever function called main, it knows to look in RV immediately, and pull that value out to take it as a return value.

The metaphor I usually use here is that think about you're trying to leave money in a locker at an airport. Okay. You wanna put the money in the locker, and immediately walk away when you know the person who's supposed to get the money is the next person looking at it. Does that make sense? Okay. We didn't have a return value for this foo function. I'll write another function that's a little bit simpler than this that really just makes use of a meaningful return value. This is also a meaningful return value, we usually just blow it off because we're not concerned about who's calling main.

Let me just draw a general activation record. Here are all the params. Here's the safe PC where the return link, and here are all the locals. These are allocated and admitted by the person calling the function. These are allocated and initialized by the actual function that's being called, the callee. So the entire picture is the activation record that has to be built in order for the code inside a function to execute and have access to all the variables.

It may seem a little weird that this part, and technically everything through that right there, but that it's set up and initialized by the person calling the function, and the rest of it is set up and initialized by the function itself. Why is there this separation of responsibility between actually building the entire thing? Why can't the caller build the entire thing? Do you understand what I mean when I say that? Why couldn't main set aside space for all the variables?

Why couldn't foo actually set aside space for all the variables? The reason is that the caller has to be involved in setting at least this portion up because only it knows how to actually put meaningful parameter values in there. Does that make sense? Who else is going to put the four and the ampersand of I in there other than the caller. Make sense to people? This right here can't be set up by the caller because the caller has no idea how many local variables are involved in the implementation of, in this case, foo. Okay. Does that make sense?

So the separation of responsibility really is the most practical and sensible thing to do. The caller knows exactly how many parameters there are, can look at the prototype to tell, and it knows how to initialize it. That's why the top half is set up by the caller. The

bottom half, the caller doesn't even know how many local variables there are much less how to manipulate them.

So you have to rely on the callee function, this is foo in this case, to go ahead and complete the picture by deallocating SP to make space for the local variables. Now I put this right here. When I say call foo right there, it's really a jump instruction that's associated with the address of this first instruction right here. RET is an instruction to basically jump back to whatever address happens to occupy the saved link. Okay. Sit well with everybody? Okay, very good.

Let me write a little bit more of a practical function just so you understand how this return register and call and return all work in a case of a recursive function. Okay. So let's write a function that's a little more familiar to us. I want to write this function called factorial which is framed in terms of a single parameter called N, and I'm not gonna have any local variables. If it's the case that N double equals zero, I just want to go ahead and return 1.

Otherwise, I want to go ahead and return m times whatever the recursive call to factorial of M – 1 returns. I'm not concerning myself with M being negative. I don't care about the error checking. I want to concern myself with how this translates to assembly code in our little language. Okay. Let me erase this. Yep?

Student: Okay, in the first [inaudible] of foo, you substituted [inaudible], I mean are you insinuating that you can substitute [inaudible]?

Instructor (Jerry Cain):In other words, do one variable at a time?

Student: Yes.

Instructor (Jerry Cain):Yeah, you could do that. It wouldn't be incorrect. Compilers when they're generating this code, they can look at the full set of local variables that are declared at the top, and it can compute the sum of all the sizes, and reduce the allocation of all of them to one assembly code instruction instead of several.

Student: So it's more efficient, right?

Instructor (Jerry Cain):Well, yes, technically. I don't want to say that it's – that's not a top priority, but real compilers would just use that right there because they could very quickly add up all the variable sizes, and just know that they're gonna be packed together, and just do this on one little swoop. Okay. Now as far as this factorial function is concerned, I want this to label the first assembly code instruction that has anything to do with this as a function.

It has to assume this as a picture. It has to assume that some value of N has already been passed in. Okay. That whoever called it, laid down a call to the fact function, and as a result of that, laid down some safe PC so that it knows where to jump back to after

factorial computes its answer. Does that sit well with everybody? We can momentarily forget about the fact that function, call, and return is involved because the first few statements right here are just normal little C code. Okay.

What I want to do up front is I want to load the value of N into a register, SP + 4 is the address of M in this case. Okay. And I want to pull the 4 into a register called R1 because I want to conditionally branch around this return statement if some test passes. Okay. Well, I want to branch – I'll leave those open for the moment, depending on whether R1 and zero basically mismatch. If it's equal to zero, I just want to fall to the return statement and scram, but if they're not equal, which is why I will put M and E right there.

If they're not equal, then I'm doing a little transition of this, then I want this to not execute and it to come down here. So all I'll do is I'll put PC +, and I'll leave this open because I don't know how many assembly code instructions I'm jumping forward yet until I execute the code for that. Okay. If this test fails, it's because this test passes and I want to basically populate RV with a 1 and then call return. So I would do this return value is equal to 1 and then I would return. Okay. Does that make sense to people?

If this doesn't happen – I'm sorry, if this test fails, I better jump beyond the return statement, which means that this should be a 12. Okay. And the next instruction I'm gonna draw, has to start on the computation of this thing right here. Now I have no reason to load M into a register because I'm only going to potentially clobber that register when I call this function recursively.

So what I need to do is I have to prepare the value of M – 1, push that onto the stack frame in preparation for a recursive call to factorial, let the recursive call do what it needs to do, assume it puts an answer in the RV cubbyhole, and then use that answer immediately to multiply it by what my local value of N is to figure out what RV should not be populated with. So I will do this, R1 = M of SP + 4, that loads in into R1. I will compute what

N 1 is.

And now I have all the information I need to make this recursive call. So I'm going to set up the partial activation record. I'm gonna do SP = SP - 4. Okay. I'm gonna write to SP, this value of R1. So what just happened here? I decremented – this is where SP is pointing right now, I decremented to a point right there. That right there is the original activation record. I'm in the process of building the activation record for the next call. I lay down a 3 right there, because that's what R1 has at the moment. It had a 4, but I just pulled it down to a 3. Okay. Make sense?

And then I call on the factorial function. In response to that, this is decremented four more bytes, the address of this instruction is placed in the safe PC, but execution jumps back to follow this recipe all over again, but on behalf of a second activation record. Does that sit well with everybody? Yes? No? Okay.

So it's as if the primary call here – you just can think of it as suspending, it's not really what's happening, but access to this activation record is temporarily suspended while the recursive call deals with these addresses downward. You just have to assume by protocol, that the recursive call is the person you're in cooperation with and the recursive call is placing money in the RV locker, okay, so that when you pick up execution right here, you know that the RV register has something meaningful. Okay.

Well, when you get right here, you have to clean up space for the local parameters. The return statement that brings us back here, gets rid of this, and then that SP is equal to SP + 4, brings this arrow back up here, and that was the picture that was in place before we involved any type of recursive function call or any function call at all. Okay.

We know that RV has – it would be six if factorial was really doing its job. So I'll actually put that. It has a six in it, if we just take the leap of faith, but at the assembly code level that the recursion is working, and then once I clean these up right here, I have to reload M, so R1 now has a 4 in it. I can do this. That emulates that multiplication right there. Okay.

RV has the return value of this. R1 has the value of that. I have no local parameters to clean up. RV has the register that I need to communicate back to whoever called me, so I will just return. Okay. Does that make sense to people? Question right there?

Student:[Inaudible]

Instructor (Jerry Cain):The PC + 12, where did I put that? I'm drawing a blank. Oh, this right here? This is normally, if there are no branch instructions, you know how PC is the address of the currently executing instruction, and each instruction is 4 bytes wide. So by default with each clock cycle, the PC value was updated to be 4 bytes larger than it was before. So normally it executes things in order.

If this is PC, this is PC + 4, PC + 8, PC + 12. If this branch instruction actually passes then I don't want to fall to this, you specify as the third argument the actual address of the instruction you should jump to relative to the current PC value. Okay. Now in some real systems, PC's already been promoted by four, so this could be PC + 12 on some other systems, but in our little world I'm just assuming that PC retains its value during the full execution of this thing right here. Okay. And it's just identifying that right there. Were there other questions? Yeah?

Student: What it be alright to do something less arbitrary –

Instructor (Jerry Cain):Yeah, then it's a little different. I'm gonna just constrain all our examples that deal with things that are exactly 4 bytes. I'm sorry, 4, 2 or 1 bytes; things that can fit into an RV register. MIPS as an architecture, I know that one best of all of them, there are two return value registers and it usually only uses one of them unless you're returning like a double or a long, long, which is an 8 byte integer, and it would use both of them.

If you're returning a struct with 12 or more bytes in it, then what it'll do, is it'll actually place the address of a temporary struct somewhere in memory, and just assume that the caller knows that a struct is being returned and will take the RV register and dereference it to go to the thing that's actually a struct. Does that make sense? So they figured it out. I'm just not gonna deal with that particular stuff because it's more minutia to worry about. Yeah?

Student:[Inaudible] I didn't quite get that.

Instructor (Jerry Cain):That's emulating that multiplication right there. At the C level you know you have to spin on the return value from the cursive call, and multiply it by the current local value of M. Right?

Student: Right.

Instructor (Jerry Cain):So this right here, just stores the current value of M and RV right there, stores the result of the recursive call. Okay. Does that make sense? Now I have the first piece of evidence as to why I always want you to reload variables. Do you understand that right there I absolutely had, absolutely had to reload the value of M. Okay. You may say, well I did it right there. Okay.

And even if I didn't do that right there, I did this with R2, I'd absolutely have to reload the value of M because I have no idea how complex and how motivated this as a recursive function, or any function at all, is to actually clobber all of the register values. All of the function calls are using the same exact register set. Does that make sense? Okay. So that's why I want you to get in the habit of reloading all your variables as you advance from one C statement to the next one. Okay. Does this make sense to people? Yeah. Question right there? Student:

Why do [inaudible] right after the return statement above?

Instructor (Jerry Cain):Because – this right here, for the same reason basically. I want all code emission to be context insensitive and I don't want it to leverage off of things that happened in prior C statements. Real compilers would do that. I just don't want us to do it. I'm not adamant about it, but I just think that there's a nice clean formula about always generating things from scratch because if this code changes because the C code statement prior it changed, this code still does the right thing for the statement that it's being admitted for. Does that make sense? Okay.

So I don't want to kill this example yet. I want to run through an animation, which is why I have my computer here just so you see how the stack grows and shrinks in response to assembly code instructions actually running. Yep?

Student:[Inaudible] why, for example, isn't there a PC - 12? Or not 12, PC – [inaudible]

Instructor (Jerry Cain):Oh, I see what you're saying. When the program is actually loaded, this actually would be replaced by PC - 28 or whatever it is. These are just placeholders at the moment because these are easier to deal with. Think of these as like go-to labels. That's really what they function as at the assembly code level. A real assembler or linker would in fact go through all the dot O files, and replace these things right here with the actually PC-relevant addresses. It usually will defer it until link time because it wants to be able to do the same replacement of these things with PC-relevant addresses between functions that are actually in different dot O files. Okay.

And I'll talk a little bit more about that on Monday, and probably Wednesday about how all the dot O's are basically assembled into an A dot out file or a 6 dash degrees or what have you. Okay. Okay. So give me two seconds to do this. Okay. We should be good to go. Create some mood lighting. Let's see how well this translates to the screen. Okay. So this is more or less – it's an exact replica except for spacing of the function I just wrote on the board. Okay. And, oh, that's not showing up. That's not good.

Can I get it to show up? It shows up over there. That's great. Oh, it's really up there. That stinks. That's never happened before. Hold on a second. Maybe it'll just readjust to the screen size. Stalling, stalling, stalling, stalling. It has no input so something's weird going on. Okay. Actually, I have an idea. Yeah, but that's still not really very good. Let me do – let me just run it in place. Okay, you guys can all see this right here. Let's make sure the full picture can be seen. What's up?

Okay, there's that. It's actually pretty close. I don't think it zooms very well when it's driving both this computer and that monitor. Okay. That's actually not bad. The return type of fact is int. Okay. It's being clipped up by that mushroom that's on the screen up there. So this right here is the code – the C code obviously. This right here is the assembly code I more or less just wrote on the board. Okay. This is the animation, and I'll try and do this this way. Let me do next slide. Yeah, this'll work. There's that. Okay.

So the original call is the factorial of 3 and it's just gonna follow these instructions one by one and use the accumulation of the stack to just compute what 3 factorial is supposed to be. Now it checks immediately. It loads in. It checks to see whether or not N = 0. It's not, so it actually does branch forward and it prepares for the recursive call. Does this make sense to people? Okay. It has to assemble the recursive value, has to make space on the stack frame for that new – the recursive value of M, it has to write to there.

So you how we're basically in the process of building the activation record for the recursive call and then we have to call factorial and some things are updated. The little circle that's been populated in the safe PC portion, it actually has as a piece of popcorn, a pointer to the next instruction so it knows where to carry forward when the recursive call returns. Does that make sense? Okay. Question?

Student: So we aren't required to store PC?

Instructor (Jerry Cain):No, it actually happens – the call instruction does that for us. Okay. Do this – comes back up here – the original call doesn't forget where it should carry forward when the recursive call returns because we have that piece of popcorn in the stack frame. Okay. So there is this is exactly the same thing where N = 2 and N = 3. Notice that both recursive calls have as their safe PC the instruction after the recursive call. That should make sense. They all need to know where to carry forward from after the call returns.

It just happens to be the same place both times. It does go through one more recursive call, lays down a zero, one more, and then finally it comes to an invocation. It's following the recipe for a fourth time. The first three times it's suspended execution, but this time this branch instruction is going to fail, so it's going to go and actually populate RV – actually I can't – this right here has been updated with a 1, and so it's gonna go ahead and return these second to last recursive call is like, oh, I'm active again, I'm going to continue carrying on from where I was left off before.

I immediately look into the RV register and spin RV 1 x 1 stays the one, but as things unwind, the all carry off from here, the 1 in RV becomes a 2 – this isn't working very well – the RV becomes a 6, and finally I return to whoever made the original call which is probably – or which may not be a factorial. Does that make sense to people? Yep?

Student: Those arrows that were pointing down onto the screen, where those –

Instructor (Jerry Cain):They were the safe PC registers. The safe PC blocks. You remember the 4 bytes that existed between parameters and locals? Well there is a real number that's placed there. It happens to be the address of the instruction that comes after the call statement so it knows where to jump back to. It's almost as if it pretends that it wishes the call thing were just one assembly code instruction, but since it isn't and it jumps away only to come back later on, it needs to know how to come back. Okay. Okay. So does that sit well with everybody?

So that was a breeze through that, but I have a little bit of stuff to talk about come Friday. The one thing I will tell you is that C++, if you code in pure C++, you're probably programming in the object-oriented paradigm so your object and data focused as opposed to function and verb focused. In C, which we're dealing with right now, you always think about the function names, the data is incidental, you always pass the data in its parameters as opposed to having the data being messaged by method calls.

We're gonna see on Friday, and this will be an aside, I won't actually test you on this part on the mid-term, but you'll see that C++ method calls and C function calls really translate to the same type of assembly code that everything is emulated by a function call and return with call and return thing at the assembly code level, and it's really just an adaptation of either object orientation or imperative procedural orientation to assembly code to get it to run, emulate either C code or C++ code. Okay. And that's what we'll focus on on Friday. Okay. Have a good night and I will see you on Friday.

[End of Audio]

Duration: 47 minutes