ProgrammingParadigms-Lecture09

Instructor (Jerry Cain):Hey, everyone, we're online. I don't know have any handouts for you today. I started to introduce the next segment of the course as being the part where we actually cover computer architecture and assembly language. We're gonna spend a lot of time trying to figure out exactly what our C and C++ code snippets, or just the C and C++ code, although we'll only deal with snippets in any one example, how it actually compiles to assembly code. Talk a little bit about the memory the models, talk a little bit about how function call and return works, and also to expose you to, not a real assembly language, but at least a little mock one that we've invented for CSten7 purposes to kind of show you all of the little gears that are in place to get addition and multiplication and division and function call and return and pointers, and all those things to actually work, but at the hardware level. Okay? Now, when I left you last time I had started talking about the stack segment. And if you remember, probably about halfway through Friday's lecture I had hexagons and circles and triangles as placeholders for basically the skeletal figures that overlaid memory, and showed you how all the local variables in a particular function were actually packed together in what was called an activation record. In many ways, the assembly code that we're going to be writing, are really just these 4-byte instructions. They're ultimately 0's and 1's, but they're interpreted by the hardware to access variables within the triangles and the hexagons, okay, and pull them into registers. Maybe add one to them, or add ten to them, or pass it to some helper functions just to get the assembly code to imitate exactly what your C and C++ code was written to do. Okay?

So here's the stack segment. Let me just contrive this one little block of code. All right, int i, I have int j, and then I do i is equal to ten. And then I do j is equal to i plus seven, and then I'll do j plus plus. Let's just assume, not surprisingly, that i and j are packed together as some 8-byte activation record and they reside somewhere in the stack segment. I'm just gonna draw it randomly right here. Okay? And that address right there, I'm going to assume, is stored in one of those 32 registers that I named last time; I'm just going to go with the first one and call it R1. So R1 is a general purpose 4-byte bit pattern storer that actually happens to store the base address of the activation record that makes up this code snippet. Okay? Now, you ultimately know that a ten has to be placed there, and that an 18 will eventually go there. Okay? Does that make sense to people, I'm assuming? Okay. But I actually am more focused not on the numbers themselves, as the assembly code that actually does that relative to this address right here. So I'm going to make some assumptions that the base address of the two variables that are relative here, are stored in a special dedicated register. I'm going to call it R1 now; I'll change the name in a little bit. And I'm just gonna illustrate by example what assembly code instructions look like to actually put a ten right there, and then to pull it into a register, add seven to it and flush it back out to the space for j. Okay?

In order to get a ten into that space right there, we don't actually deal with that address specifically, we just deal with an address that's at an offset of positive four from the address stored right there. Okay? The notation for doing this, this one line in our little mock assembly language would compile to that. And that's our first assembly code instruction. Okay? That capital M, it’s like it’s the name of all of RAM. You think about all of RAM as this big array of bytes, M is the name of that thing, R1 is the base address, four is an offset. This right there identifies the base address of the 4-byte figure in all RAM that should get a ten. Does that make sense to people? Okay. So this is an example of a store operation. And it’s called a store operation because it actually updates some region in a stack segment with some new value. The very next few instructions are in place to actually take care of j is equal to i plus seven. I don’t want to bank on the fact – or rely on the fact that I know because of this small example that i is equal to ten. I wanna do the most robust thing, which is to actually go and fetch the value of i, pull it into a register where you’re allowed to do addition, added a seven to it, and then flush the result out to M of R1. Does that set well with everybody? Okay.

So don’t bank on the fact that there happens to be a ten right there. An optimizing compiler might take an advantage of that, but we’re not writing an optimizing compiler, we’re just trying to brute force without the translation process right here. I wanna do this: into R2, another general purpose register, I wanna load the 4-byte bit pattern that happens to reside right here. Now, this is an example of a load operation. Okay. Let’s make it so that that really looks like a load. R2 contains the 4-byte bit pattern for what’s residing in i right there, so this corresponds to the understanding that we’re going to operate on i’s value. Into R3, another general purpose register, I’m going to take whatever R2 has as a bit pattern and add the immediate constant seven to it; this is an ALU operation. It is actually very often the case, though, on the right hand side of some ALU operation, there are either two registers, or there’s a single register and a constant, and that wherever the result is to be stored is identified on the left hand side. Okay. That probably makes sense to people. This is a bit pattern, is the thing that has to be replicated in the space that’s really set aside for j, and we know where j is because we have R1 storing the base address of it. So right here, I have yet another store operation. So two of the registers, R2 and R3 were piecemeal updated with a ten right there, a 17 right there, and then a 17 is flushed right there. Okay. Does that make sense to people?

This load ALU store sequence is actually very common, and exists more or less on behalf of any kind of assignment-oriented statement in C or C++. Okay? The very next line really expands to j is equal to j plus one, so it has a structure that’s very similar to this. I can reuse registers. I can even discard a temporary value that’s in R2 if, for whatever reason I want to. I don’t have to write to R3 right there, and then I can do M of R1 is equal to the new value that ended up in R2. Okay? I didn’t have to be that efficient about the user registers, but it’s not a problem if I don’t need the old value of j and I just want to deal with the incremented value. It’s fine to overwrite R2, and to override its old value. So this has the same exact structure as that right there, it just happens to be dealing with variable in RAM as opposed to two, and this is how this 17 goes from an 18, and R2 would also have an 18 inside of it. So a few things to say about this, by default all of the load and store and ALU operations deal with 4-byte quantities. Okay? It’s clearly the case that pointers and ints are probably by far the most common atomic type you deal with in programming, at least in C and C++, so the hardware is set up to optimize, by default, dealing with 4-byte figures. Okay. That doesn’t mean we can't deal with isolated characters and shorts, or that we can't deal with doubles and structs as a whole. But I'm more interested in being 4-byte oriented with all of my instructions here. Okay.

A lot of you may be questioning why I bother to do this. You could say, "Well, why don't I just set R3 equal to ten plus seven?" I want to – every single one of these blocks – this corresponds to that statement, these three correspond to that statement, these three correspond to that statement. I want every single block of assembly code instructions to be emitted or rendered in this context insensitive manner. And the reason I'm saying that is because this code is going to work out and do the right thing even if I change this line right here. Do you understand what I mean when I say that? Okay. If I were to hard code in an M of R1 is equal to 17, then that would stop being correct if I changed this line, which was supposed to be completely unrelated to it – to, like, a 12 or a 100. Okay? Does that set well with everybody? Okay. The same thing with this. You could argue, "Well, why don't I just go in and do M of R1 ++?" R assembly code instruction doesn't allow ALU-like operations to be performed on arbitrary memory addresses. You always have to go through the load, do an ALU operation on just registered values, okay, and then flush the result back out to memory. It more or less makes for a simpler assembly language, and it also makes for a faster clock speed when things are that simple. Okay. Make sense to people?

Let's deal with an example that actually doesn't always deal with 4-byte quantities. Let me go ahead and do int i car ch – actually, I don't want to do that – [inaudible]. Let's do short S1 short S2. Okay. The way I've declared these, we would be dealing with an activation record that looked like this: i's right there; this is where S1 is gonna be declared, this is where S2 is gonna be declared. Okay? That looks a little weird, but S2 is the last of the three declarations, and so it's gonna be at the lowest base address according to my model. Okay. I'm always gonna give you variable declarations that work out nicely and that are an even multiple of 4-bytes so that the pictures are always fully rectangular and with no, like, chunks or nips out of the corners. If I do this, that's more or less the same as that instruction up there, with a different number. This would translate to M of R1 plus four, assuming that the register R1 has been set to point to that right there. R1 plus four, memory [inaudible] reference is equal to 200. Okay. That means as a 4-byte figure, the bit pattern for 200 is laid down. Now, 200 is a pretty small number, so in a big ending world, we'd expect one byte of zero followed by another byte of zero, followed by a third byte of zero, followed by one byte that happens to have all of the 200 in it. Does that set well with everybody? Okay. This line has to somehow update S1 – let's actually write it, to be equal to, logically, 200. But it's only supposed to update two bytes of memory. Does that make sense? Well, if you wrote this, your heart would be in the right place, but there are two fairly relevant errors going on here. First of all, you can't do a load and a store in one single operation. Okay. Because that would require assembly code instructions to somehow encode in four bytes the source, memory address, and the destination memory address, and that's difficult to do. So what we wanna do is we want to evaluate i as a standalone expression, and just do this: R2 is equal to M of R1 plus four, again I'm being context insensitive about it.

That means that R2 is gonna have that as a bit pattern inside of it. And then what I want to happen is I somehow want to update this two bytes right here with the 200, but I only want those two bytes to be copied. That's consistent with what we know happens just at the C and C++ level. Okay. Now, if I do this, that won't do what we want because the assembly code that I'm writing right there has absolutely no memory as it's executing what C or C++ code was in place to generate it. And the way I've written it right there, it's just like so many other operations I've put up here. This would be taken as an instruction to update the four bytes from this address through this address, plus three, okay, with new information. And it would update that and that and that and that with four new bytes. Does that make sense to people? I don't want that to happen. This instruction right here would put a zero and a zero and a zero and a 200 right there, and then all of a sudden, S1 would be initialized to zero, and i would become some very, very large number. Okay? That's because I'm mistakenly dealing with a 4-byte byte transfer here and I don't want that. In our world, you override the 4-byte rule by actually putting a little dot-2 right there. It's as if all of these other instructions have an implicit dot-4, but we don't bother writing dot-4 because dot-4 is just for the default. But when you only wanna move around a single half-word, which is two bytes, or a single byte, you'd put dot-2 or dot-1 there. Okay? That's an instruction that when we're sourcing from a register, so just take the lower half of it and update the two bytes that they give in this address right here. Okay? And update this with zero and 200, and that's how S1 becomes 200, and that's how i is left alone as the 200 is a 4-byte quantity. Okay? Does that make sense to people? Okay. If I do this, S2 is equal to S1 plus one, it's very similar to the j++ up there, but we have a lot of dot-2's that are in place to make sure that only two bytes are moved around at any one moment.

This right here would have to load S1 into a register. This is how I would do that: R2 is equal to M – whoops – M of R1 plus two. But the way I've written it right there, it will copy four bytes as opposed to two bytes, unless I do this. What that does is forget about the old value and set R2. When it pulls two bytes into a register, it lays these two bytes in the lower two bytes of the entire register, and then it sign extends it just like it would in C and C++. So it's padded with two bytes of zero's right there. Okay. Now this plus one is just traditional plus one. R3 is equal to R2 plus one; that's what takes this to be 200 to 201. And then when I flush the result out to this other space that has just two bytes associated with it, I do it this way: M of R1 equals dot-2, the result that's stored in R3. Okay. So I understand that these examples aren't all that riveting from an algorithmic standpoint, I'm really just trying to illustrate our assembly code languages as operations on general memory where the general memory is framed as an array of bytes, but everything is taken in 4-byte chunks, by default anyway. Okay. Yep, go ahead.

Student:

Is there a reason why you used a third register in that last –

Instructor (Jerry Cain):

As opposed to there, no.

Student:

– second one?

Instructor (Jerry Cain):I actually – I'm inconsistent with my use of registers, in terms of, like, the conserving. I just conserve them if I anticipate having a lot of them in the example. I just did it there, I should have put R2. I thought of that as I was writing it. Okay. Any other questions at all? Yeah?

Student:

What is the [inaudible]?

Instructor (Jerry Cain):M of R1 identifies this space right there; that corresponds to S2. Okay. So that's receiving some 2-byte bit pattern because of that dot-2 right there. Does that make sense? R3 stores the incremented value, or the result of the plus one operation that's right there. So a full evaluation of the right-hand side ultimately made it into R3. I only have room for the bottom two bytes, which is why I have that dot-2 right there. Okay. So this is how I get zero, 201 right there. Okay.

Now there are other things that go on, but as far as – interestingly enough, except for function call and return, you've seen almost all the mechanics of the assembly code language that I want to teach you. Okay.

Let me do this, let me deal with an array of length ten – no, actually that's too big. An array of length four is big enough, and then I have a standalone integer. And I just wanna go ahead and figure out what this four loop will translate. And with each iteration, I just wanna update some value in the array to be equal to zero. After it's all over I wanna set i minus minus.

Just to say something silly right here, do you understand how this code is simple enough that it's just executed sequentially? The assembly code is executed in sequence as well. First clock cycle it updates memory according to that rule right there; next clock cycle it does that and then that and then that and that and that and that. And over the accumulation of six clock cycles, we've effectively realized these three C statements. Memory is updated in a way that's consistent with the way these three lines have been written. Okay.

There's some looping going on there, clearly. If there's looping in a language, then you shouldn't be surprised that there are at least directives in it in the assembly code language to jump back an arbitrary distance to start over some loop again. Okay. Or with the case of, like, if statements and switch statements; they're based on the result of some test, you actually jump forward four instructions or 12 instructions to the point it starts executing the else clause or some particular case statement. Okay. Does that make sense to everybody? Okay.

So as I write this, I should write assembly code, and you're familiar with how that and that would be executed, maybe to some degree that, as well, although you haven't seen a raise in the context of assembly language, but the test is new to us. The i minus minus is not, but the actual looping certainly is.

You know how there are six different relational operators that can be set in between any two integers? Less than, less than or equal to, greater than, greater than or equal to, double equals, not equals to. It's typically the case in most assembly languages that they have branch instructions that are guided by the same six types of relational operators. Okay.

Let me just write the code for this. Let me draw the picture; the way this would be laid out is that I'd have 16 bytes with four more bytes below it, that's I, this is array of zero through three, but I'll emphasize that this is the zero, the oneth, the twoth, and the threeth. Okay. And assume that R1 points to the base address of the entire figure. The actual hardware will make sure that the base address of the currently relevant activation record – that that address is stored in some register; I just happen to be calling it R1 at the moment. Okay.

So we come here. We assume that R1, as a register, stores the address of that picture up there. And the first thing that happens, is that this thing gets executed exactly once because it's in the [inaudible] portion of the four-loop, obviously. So what happens is that right up front, M of R1 is set equal to zero. And that gets a zero right there. Okay.

We next execute code on behalf of this to decide whether we're going inside the body of the four loop, or we're circumventing it because the test failed. Okay. These are the assembly code instructions that would be expanded on behalf of this test right there. I'll put a double line right there to mean that there's some new C statement that we're dealing with.

I would have to load into R2 the value that's at M of R1. Okay. Again, I'm starting to generate code in a context insensitive manner, I don't want to assume that there's a zero in there. In fact, you'll see in a little bit that there won't always be a zero in M of R1. So I wanna load that into a register, and then I have this branch instruction, b, and I'm gonna leave a blank and a blank right there. We're gonna fill this in with some abbreviation for not equals or greater than or equal to or whatever. We'll decide what it is in a second. It takes three arguments. It takes – whoops – the first register in the comparison or constant, the second register in the comparison or a constant, in this case it's four; and then it takes as a third argument what's called a target address, the place to jump to if the branch instruction passes. Okay. Now we jump forward several instructions – I'm sorry, let's say one C instruction here if this test fails. But if I take the logical and inversion of this, and I jump forward when this test passes, okay, then I'm circumventing the four loop. Does that make sense to people? This is where code for the array of i equals zero will be placed. When this test fails, this test needs to pass so that I jump forward a certain number of assembly code instructions to the part that actually executes that right there. Okay. That means that I want to branch, I want to circumvent the normal pattern of advancing to the next assembly code instruction if this as a number is greater than or equal to four. Okay? And so the abbreviations for these branch instructions shouldn't surprise you; they are: branch on equal, branch on not equal, branch on less than, branch on less than or equal to, branch on greater than, branch on greater than or equal to. Okay. So if R2, which stores the current value of i is greater than or equal to four, we know the loop is over, so we wanna jump forward as if the loop never existed before. Okay. We have to fill this part in. All I can tell you right now is that the address here is gonna be framed as some offset relative to a special register I call PC. Now, PC is really gonna be like the 27th or the 29th or the 31st of the 32 registers, we just give a better name for it. PC stands for program counter, and it stores the assembly code instruction of the currently executing instruction. Okay. I'm sorry, it stores the address of the currently executing instruction. We never know what PC really is, but if this is PC, this is PC plus four, PC plus eight, PC plus 12, PC plus 16, et cetera. Does that set well with everybody? And with each clock cycle, as part of each clock cycle, it by default just updates PC to be four larger than what it was before because all of our assembly code instructions are 4-bytes wide. Okay. And by default, it always just advances to the next one unless some jump instruction or some branch instruction like that tells us to do otherwise. Okay. I'm leaving a question mark right here because I just don't know how many lines array of i equals zero is gonna translate to. Okay. All I can tell you right now is that the offset is gonna be positive, and that it's going to be some multiple of four. So something like plus 16 or plus 12 or plus 24, we have no idea what yet. Okay.

If this branch instruction fails, it's because this test passed which means I'd have to just fall right to the next line of C code, which means I'd have to fall to the next line of assembly code, which implements this right here. Okay. Now you know enough about pointer math, this is easy pointer math. But this has to translate at the assembly code level to something that finds the address of the ith figure in the array and writes a zero there. There's implicit scaling of i times size of integer here; does that make sense? The scaling over here has to be explicit. Okay. If you just write assembly code by hand, which you can do, and you're just really thinking in C and C++ terms while you're doing it, you have to make sure that you assign to an offset of plus zero or plus four or plus eight relative to the base address of the array when you're writing a zero. If you're trying to emulate a four loop inside assembly code, then you have to make sure you take care of the pointer math explicitly. So what I wanna do is I want to reload the value of I because I am being context insensitive about how I use variable values. It's a zero, a one, a two, or a three; we know that. But then what I wanna do, is I wanna take R3 and I wanna multiply it by four. Now that four is there because ints are actually four bytes. I can't write size of int here like I encourage you to, not in this example but whenever you have to manually deal with type sizes in C and C++ code because this has nothing to do ultimately with C or C++. Okay. It happens to be imitating the execution of a C++ program, but all size and type information at the assembly code level is completely gone. It just has to be the case that the code is written in a way that's consistent with this is intended to do. Okay. R3 has the value of i, R4 has the value of i scaled by four, so now R4 has the distance in terms of bytes from the base address of array to where the zero has to be placed. Does that make sense? What is the value of array? We know that it's ampersand of array of zero, right? Array of zero resides right here. Okay. So the ampersand of array of zero, which is synonymous with just plain old array, it is synonymous with R1 plus 4. So I'm gonna do this. This stores the offset; this stores the base address of the entire array. R6 is equal to R4 plus R5 – whoops – R6 has the address within the array that should get a zero on this particular iteration. Okay.

Every single iteration – R5 gets the same value every single time, but R4 certainly does not. So that means that M of R6 is equal to – I'm sorry, M of R1 – oh, no, I'm sorry, that's right. R6 is equal to zero; that's the base address of the place in the stack activation record, that should get the zero and that's why a zero was there. These five lines right there, it's complicated, but they actually are in place to emulate that line right there. If we were writing a CS107 assembly language compiler, this line would translate to these five lines or something that's equivalent to it. Okay. Then what happens next unconditionally, is that we jump back up here to i plus plus, and we execute this. That just does this – R2 is equal to M of R1; R2 is equal to R2 plus 1; M of R1 is equal to R2. And then we know that we execute the test again. So what happens here? Rather than actually writing the code for the test again, you go back and you reuse the same code you wrote for it the first time. So this is an unconditional branch. Okay. We don't use the word branch, we just use jmp because they didn't have room for the u – jmp, and you actually jump to a hard-coded address, but we always frame it in terms of the current PC value. So it's not PC itself, it's PC minus some value. I wanna jump back to this line right here; the line that loads i, compares it to four and decides what it's gonna do. Okay? This minus has to be scaled by four, but I wanna jump back one, two, three, four, five, six, seven, eight, nine, ten instructions. Does that make sense to people? So this would be a PC minus 40. Okay. Right here, I'm out of room, but this is where I would continue. This is where the code for i minus minus would go, and it would be assembly code that's emitted for i minus minus that has nothing to do with this four loop. And it doesn't bank on the fact or the understanding that I would be equal to four at the time it gets there, it would just do the R2 is equal to M of R1; R2 is equal to R2 minus one, and then flush it back out to i. The reason I'm writing that there and the reason I have this here is because I wanna make it clear that this is the place where we should be jumping forward to at the assembly code level when that test passes. Okay? When this test fails, I just do the implicit update of PC to PC plus four. You don't have to write anything for that that just happens by default. If you wanna override what gets used as the new PC value, you have to set PC right here to be PC plus one, two, three, four, five, six, seven, eight, nine, ten instructions. Okay. Times four, so this question mark would become a 40. Okay? Do you guys understand why those numbers are 40?

You may think that this minus value right there and that plus value there have to be exactly the same, that's not always the case. Okay. It just depends on how complicated the test is, but sometimes this right here, which is the part that evaluates the i, this could be something that's arbitrarily complicated, like i plus 24 plus j or something like that. So it might actually be a lot of code that would have to be accounted for in this jump right here. Okay. But this plus 40 deals with an offset from this to the line after the unconditional jump back; it may be a smaller value. Okay. So they're not always the same number. Okay. I think you understand, even if the assembly code is weird for you at the moment, you understand that it really is this brute force translation of this right here to code in a language that just thinks about moving 4-byte quantities around by default. Okay. Short action branch instructions, it has some unconditional branches and some conditional branches. There are gonna be a few more things in place, but really you've seen like 70 percent of the language already. Okay. You guys get what's going on here? Okay. Very good. Questions? Yeah, right there.

Student: If you wanted to, instead of jumping back ten lines, could you just jump back nine because the top line is R2 equals MR1 and you [inaudible] right before the jump set MR1 equals R2?

Instructor (Jerry Cain):Yu could. It actually – it would certainly be correct in the sense that it would do the right thing. That would just be taking advantage of the fact that the register happens to have the right value, but I just wanna be consistent. I'm not going to enforce this – I'm not gonna police the matter to the point where I actually yell at you about it, but I'd actually like you to just get in the habit of generating code in this context insensitive manner. And this load right here, this one happened to do with the fact that there's an i present in the test; does that make sense? This i – I'm sorry, this R2 being set to that right there, okay, just happens to be associated with this i plus plus right here. And as it turns out, actually – I'm sorry, this right here has the right value. It's – I don't want to say it's a coincidence, because it's not a coincidence, it's a four loop and it's the traditional idiom with the four loop. But I'd rather you just be fastidious about just generating it and, like, basically going brain dead about what you've generated code for before because then you know you're always right, okay, regardless of whether or not this changes.

You want the code that's emitted on behalf of this right here and that right there to be the same every single time, or at least allow it to work if it's the same every single time, regardless of what you do right here. Okay. Yeah?

Student: If all of the translations of every single translation translates into 32 bits –

Instructor (Jerry Cain):That's right. We're dealing with assembly code language where all instructions are four bytes wide. Okay. Thirty-two bits, all of them, yeah. Okay.?

Does that make sense to people? Okay. Let me explain a little bit how 32 bits are usually just enough for you to encode an instruction, just to understand what encoding is like. When a 4-byte figure is understood to be an assembly code instruction, it's typically subdivided into little packets or little sub-packets. Here is a 4-byte instruction, and I'll just draw very loose boundaries here for the bytes. We're used to instructions like this: R1 is equal to M of R2 – I'm sorry, M of R2 plus four. Something like R1 is equal to a constant is not unusual either. Maybe you see something like this: R3 is equal to R6 times R10. And then something like M of R1 minus 20 is equal to let's say R19. A load, a direct immediate constant load, an ALU operation, and a store; these are little bit more elaborate than you'd see in practice, but nonetheless we have to be able to support them.

This type of instruction and that and that, they're certainly different from one another. This is a load, this is an ALU operation, this is a store. You understand what I mean when I say that. I even argue that this right here is technically a different type of instruction than this one because this is framed in terms of a register and an arbitrary memory address, and this is framed in terms of the constant. Does that make sense?

Let's say that I've decided that my assembly code language has let's say – let me – 59 different types of instructions. I have to somehow encode in this four bytes right here, which of the 59 instructions we're actually dealing with. Does that make sense? Well, 59, unfortunately, isn't a perfect power of two, so if I really want to be able to distinguish between 59 patterns, I have to be able to distinguish between, it turns out, 64 different patterns, okay? So I might set aside the first six bits of all 32 bits of – of all 32 of them to be the part that the hardware looks at to figure out what type of instructions should be executing. Maybe it's the case that this corresponds to – in this space right here would be called an operation code, or an op code; maybe it's the case that all zeros means this type of load instruction. Maybe this right here is the op code for an immediate constant load into a register. Maybe this right here, being a multiplication, corresponds to that type right there, and then that right there might be let's say all ones. Does that make sense to people?

When the hardware looks at an assembly code instruction, it doesn't actually see these. This is an assembly language instruction that makes sense to us, but it's actually expressed in the hardware as a machine code, which is 32 zeros and ones. It would actually have to look at the first six during the first few percents of the clock cycle, okay, to figure out how to interpret the remaining 26 bits. Does that make sense to people? Okay. Maybe this is the type of instruction that allows any one of 32 registers to be updated; it allows any one of 32 registers to be the base right there, and then it allows all of the other bits to express this as a signed constant. Do you understand what I mean when I say that?

Okay. There are 32 possibilities for this, there are 32 possibilities for that, and there's however many possibilities are allowed based on how much room we have left to encode that, we would need five bits to encode which register gets the update, five bits which determines the base address and the memory offset. And then maybe it's the case that all the remaining bits, in theory, hardware – people who actually still believe would laugh at this, but in theory, the remaining whatever 16 bits could be used to express a signed offset from this right here. Okay. And I draw this subdivision of five and five and 16 right there. That subdivision's only relevant when the first six bits happen to contain all zeros. Does that make sense to people?

For this right here, all I would need to do is set aside five bits. If it read zero, zero, zero, zero, one right there, then it would say, "Oh, you know what? The hardware is implemented in such a way that when there's all zeros followed by a one right here, that the first five bits tell me which of 32 registers I'm assigning to, and all of the remaining bits, all 21 of them, can be expressed a signed integer that actually gets put into that space.” Okay. So the subdivision scheme from bit seven forward actually depends and how it’s interpreted depends on what the op code says.

Now ultimately what happens at the EE level is that these are all taken as instructions as to how to propagate signals through the hardware so that the signals at the beginning of the clock cycle look like the ones and the 17’s and the ten’s and the 200’s that we’ve dealt with in the prior examples. Okay. That’s the extent of my understanding of EE, the way I just said that. Okay. But you get the principles of what I’m trying to say, right?

Okay. What else did I wanna say? There’s absolutely no requirement that all op codes be exactly the same size. You all did – a lot of you did the – I’m sorry, there’s this one assignment that we use in 106X called the Huffman encoding assignment. Those who have heard about it know about it. 106B doesn’t do Huffman encoding do they? Okay. Well, that’s an example where they actually have variable length encodings. Here we have a constant length encoding for all op codes. It could be the case that one of the op codes could just be this right there. Okay. And then some other op codes would have to be longer, so maybe this is another op code. It would only require that the first three bits don’t happen to be coincidence if it’s something that could be interpreted as a 3-bit op code. Does that make sense to people? That may seem like a silly thing to do, except that this might be the type of instruction that benefits from having lots and lots of bits set aside for some unsigned integer offset. Okay. Or it might be the one that’s most popular so it wants the most flexibility in how it expresses its arguments. Okay. I don’t wanna say that this is not common; I think it actually is common. It certainly comes up in Nips which is the assembly code instruction that EE’s study in EE 108B, I think it is now, and CS majors currently have to as well. Okay. I’m just gonna assume for all of our examples that this is the case, that we have constant op code lengths just because it’s simpler to just rely on that type of information. Okay? Does that make sense to everybody? Okay. So there are a couple things I can do in the final six minutes. Before I start talking about function call and return, which I’ll get to on Wednesday, I should talk a little bit about structs, but more importantly, I should talk about pointers and casting. That’s the part that makes C hard, but somehow compiles. And when it compiles, it means it compiles to something like this, but in – not in CS107 assembly language, but like in X86 or whatever, or, like, Nips or whatever the target language happens to be.

Let me do one example here that’s framed in terms of the struck fraction we dealt with lecture three or four. Struct traction int, num, int, denom, and that’s it. And I declare a struct fraction called pi, and I do this: pi dot nu is equal to 22. The way I told you that structs were laid out in the third or fourth lecture, that’s actually true on virtually – in any architecture that I know of. That the structure packed – that the first field is at the lowest address and everything is stacked on top of it. And if I declare this one variable so that this is logically functioning as my pi variable, then R1, in our world at the moment, stores the base address of the one variable that’s there. But the one variable actually knows how it’s decomposed into smaller atomic types. Okay. When I do this, not surprisingly, this actually translates to M of R1 is equal to 22. If I do this, pi dot denom is equal to seven, I get M of R1 plus four is equal to seven. So that’s easy, except for the fact that you’re dealing with structs and you have to understand that the assignments have sort have forgotten about the structs and just are really updating individual integers inside the struct. The part that's interesting, transition to slightly more scary stuff, is if I do this ampersand of pi dot – let me rewrite this. Ampersand of pi dot denom, if I write it that way and then I cast it to be a struct fraction star, I'll abbreviate there, and then I do this. Forget about the assembly code for a second, you know how memory is supposed to be updated; it's a little weird to see this type of thing, but it's irrelevant that it's weird because it's a legal C code, and it's supposed to compile to something. This says, "Identify the l-value of pi dot denom." Where is it located?" Okay. Stop pretending it's a standalone int and think that it's a base address of an entire fraction, go to its ghost denom field and put a 451 there, okay?

The order at which things are kind of realized here, is it discovers this, it evaluates that address, it casts it to think that that right there is the base address of not a standalone int inside a struct, but the base address of an entire struct fraction, and a 451 needs to be placed there. Okay. What assembly code instruction, there's only one of them that's needed, what's the assembly code instruction that would need to be in place in order for that 451 to be placed there? It just translates to this. Now you may think I'm cheating there and I'm actually doing work, but this right here is understood to be an offset of four from R1; the address is four beyond to that base address. Just because I cast it to be a struct fraction star doesn't change the value of the address, it just has a different idea as to what is at that address. It's like the compiler puts on a different set of glasses all of the sudden when it's looking at this one address right here, and it knows that at the denom field that you'd get by de-referencing this, "Oh, it's a struct fracture because it says so," it's an offset of four beyond what it was already an offset from, R1, and then it puts a 451 there. So all the energy that's in place to compile this, either by hand or by code if you wanna write your own compiler, it actually can discover that this is really referring to an integer that's presumably at an offset of eight from R1, which is where pi originally lives. Okay. Does that set well with everybody? So the cast – when you put a cast in place, at least in pure C code, there's no assembly code instruction that gets generated as a result of the cast operation. All the cast does, is it allows the compiler to generate code that it otherwise wouldn't have been able to generate code for because it's like taking a little permission slip to behave differently, okay? Does that make sense? It wants to trust that there really is some legal interpretation associated with the code that it will emit by seeing that struct traction cast right there. Okay?

These are the types of things that exist in the assembly code that's generated by your IMDB get methods and get cast class. The C++ turns out it's not that much more than C in terms of compilation. Okay? You're doing all these void star and car star casts inside assignment three; either you have or you're going to very shortly. The same type of thing happens as it is compiled to either X86 or spark assembly, which is one of the two things you're dealing with if you're on either of the pods or the [inaudible], okay? Does that make sense to people? Okay. I can get arbitrarily complicated with all of these casts. You know I'm the type of person that will be arbitrarily complicated with them, so you'll see some examples of these in a section handout for next Tuesday, and also in the problem set assignment that will go out not this Wednesday, but next Wednesday. It'll be your final assignment before the mid-term, okay? Or I want you to just master this pointer stuff. And believe it or not, you get a lot of mastery by actually coding with it, but if you're forced to draw pictures and generate code and make sure that the code you write down is assembly code, logically matches what mine and the answer key does, it actually resolves any remaining mysteries that might be in place in spite of the fact you get assignment two and assignment three working. Okay? There's still some mysteries that are in place for a lot of people and this usually resolves those mysteries. Okay. Come Wednesday I'll do one more intense example with a little bit more casts; I'll probably bring in an old exam problem and show you how fun they can be. And then we'll move on to starting to understand how function call and return works at the assembly code level, how we introduce the hexagons and the circles to the stack frame and how we get rid of them. Okay. I will see you all on Wednesday.

End of Audio

Duration: 52 minutes