00:00:00 assembly language is a low level programming  language for a computer or other programmable device that is closest to the machine language.  It is often specific to a particular computer architecture. So there are multiple types  of assembly languages arm is an increasingly popular assembly language, Scott Cosentino teaches  this course about assembly programming with arm. Scott is a popular technical instructor  and author. It is estimated that over 200 billion devices contain an ARM chip, making  the arm language valuable to understand.

00:00:36 By understanding the arm assembly language,  programmers can work at a lower level, allowing them to write code that interacts  with hardware in an efficient manner. By the end of this course, you will have a  fundamental understanding of the ARM processor, as well as assembly programming in general, you will  be able to write basic arm assembly programs using various instructions available in the processor. I  encourage you to share the most interesting thing you learn in the comments for the benefit of other  campers who are watching this course. Hello, and

00:01:10 welcome. I'm excited today to be presenting to you  an introduction to the arm assembly language. The main goal of this course is really to give you an  introduction to the ideas of low level programming through assembly languages. As well as working  with the arm programming language. In general. Arm is a programming language that's used in  a variety of different applications. You see it in a lot of different embedded devices. And  really popular devices such as different phones, a lot of Android devices and some iPhone devices  are running ARM based chips. And we're also

00:01:42 starting to see them become a lot more used.  And a lot of laptops like the new MacBooks are actually using ARM based processors. And because  of this, it's become a really important skill to actually understand how ARM processors work and  how you're actually able to program directly with them. This is because working at a lower level  lets you be closer to the hardware. So lets you write better interactions between the hardware  and the software. Now let's see more efficiently interact between hardware in any situations  where you might need to do something like this.

00:02:12 Generally, this course is going to give you  a good familiarity with the ideas of arm, including some of the basic instructions that arm  has the different ways that arithmetic works and different operations such as logical operators  and shift operators that are commonly used in multiple assembly languages. We'll take a look  at different ways that we can branch as well as the loop inside of assembly languages. And then  finally, we'll finish off by taking a look at some hardware interactions, as well as the different  ways that we can use and troubleshoot arm assembly

00:02:40 in a Linux based system. Now I'm going to  primarily be using an emulator known as CPU later, as it's very easy for you to follow along with  this on the actual web browser that you have. So for the most part, you want to install any sort  of special equipment in order to make this work. I mean approach here is to make sure that you can be  able to interact with this course and follow along easily without any sort of special prerequisites.  Now the last few videos here are going to be Linux based. So you will need some Linux based machine  or virtual device to be able to follow along with

00:03:12 those. But overall, even if you just follow along  with the emulator ones, you'll come away from this course with a very strong understanding of arm  as well as assembly languages. So I hope that this course is valuable for you. And I hope that  you enjoy it. If you enjoy it, please feel free to leave a comment and let me know. And if you  have any sort of questions, also let me know and I'm happy to answer anything that may come up. So  in this video, I'm going to spend some time just discussing the general idea of our architecture of  our ARM processor, as well as getting familiarized

00:03:43 a little bit with the development environment that  I'm going to be using in this set of videos, which is actually the CPU later emulator. This emulator  is available online for free, so you can access it from pretty much any computer. And instead  of this emulator, we have a lot of different different architectures that exist for us.  The one that we're mainly interested in is arm V seven. If you're familiar with MIPS  or any other assembly language like that, you'll see that those emulators are also available  for you. But generally we're going to be working

00:04:12 in an arm V seven architecture. And  specifically in this set of videos, I'm going to demonstrate the arm V seven D one  soc. So this will be the one that will utilize throughout this series of videos. So I'll  press go and we'll head into the emulator. Just to note if you're following along, you  can follow along through the emulator here or you can follow along on your own computer.  If you have an ARM processor of some sort. Any V seven processor you'll be able to follow  along fine with even earlier ones you'll probably

00:04:40 be able to follow along relatively well. The main  goal of this video is not just to teach you arm based assembly, but also to teach you the general  principles of assembly so that you can easily adapt to any other assembly language things like  x86, other versions of arm and so on like that. So in general, the principles that you learn  And throughout this series of videos are going to apply to really any assembly language, with  the syntax specifically being targeted towards ARM processors. So when we get into our emulator,  there are a few things that we have available to

00:05:13 us. But there's actually like a lot of things on  the screen that we have available to us. And I'm going to walk you through some of the essential  pieces that we need, just to get started. So the first idea that I'm going to introduce to  you is the idea of registers, which we see over here on the left hand side. Registers are  areas in memory that are very close to the CPU, so they can be accessed quickly, and they can be  written too quickly. So the general storage is going to be relatively fast. However, we have  a limited amount of things that we can store

00:05:42 inside of these registers. You'll notice that  there's eight zeros inside of this register. Each zero represents a single hexadecimal value.  Now, if you know about hexadecimal, you'll know that hexadecimal is going to represent four bits.  So each hex value represents four bits of data. Since there's eight hex values, each one  represents four bits, we have a total of 32 bits that we can work with. This is because this gen  processor is a 32 bit processor. So we're going to be working in 32 bits. And that's the constraint  of how much we can store in a single register is

00:06:19 32 bits worth of data will often refer to 32  bits as a word in terms of the size. So if you ever hear me refer to as you know, changing a word  worth of data, I'm referring to 32 bits of data, the idea of a word transitions to other assembly  languages as well. And it generally represents the total like max size of data that can be stored  in a register. So for instance, if you were in a 64 bit processor, a word would be 64 bits in size.  And then we have the concept of a half word, which is half the size of a word. So in 32 bits, a half  word is 16 bits, because it's half of 32 and 64

00:06:59 bits, that half word is 32 bits, because it's half  of 64. And then of course, we have a byte, which is always eight bits. And then we have a single  bit of data. So those are sort of the different sizes that we have available to us. So in general,  when we're working in assembly, we want to try to use these registers as much as possible. So all of  these registers are going to be available to us, with the exception that some of them are  going to have special purposes associated with them that we'll discuss. As we start  to see those special purposes later on.

00:07:30 In general, I can say that registers are zero to  our six our general purpose, we can use them for whatever storage we'd like to use them for our  seven is going to have a special functionality to us related to system calls. Essentially,  when we when we are working with assembly, sometimes we need to talk to the operating  system. And we need to ask it for maybe a resource or we need to ask it to terminate or  program when we you know, call to the operating system to ask it to do something for us, it  needs to know what we're asking it to do.

00:07:59 And the way that we communicate that is by storing  a value in R seven, and that value will be some numerical value, that it will map in a table to  some some specifically action. So for instance, if I store the value one and R seven, and then I  interrupt the program, the operating system will read that and interpret it as enter the program.  So that's an example of how we might use that. So that's one example of a special  purpose register that we have. Now there are other special purpose registers  that are actually labeled in this emulator.

00:08:29 SP, LR and PC. SP is a really interesting  one, which is related to the stack pointer. And that actually introduces us to our next  important concept, which is the concept of stuck memory. If you head over to the memory section  here, you'll see that you have a whole bunch of memory available to us. And this is referred to as  stuck memory. Stuck memory is typically stored in the RAM of the computer. And essentially, it's  slower to access and slower to write to. However, we can store a lot more data in RAM comparative  to data in registers, you'll typically see stack

00:09:07 memory used when we want to represent more complex  sets of data. Think of something like a list of numbers. For instance, if I representing a list  of numbers, I could give each location and memory a specific number. And then I can iterate through  those memory locations to get those numbers that would be an example of when we might use stuck  memory. So the SP register is always going to be telling us the address of the next available  piece of memory on the stack. So in this case, you see it's pointing to all zeros, which means  that we are sitting at this location here.

00:09:41 Now, in order to determine like the actual  locations, we can think a little bit about like where each address is, so this one is zero. We can  ask what is the address next to this? It's always going to be for larger than the previous. So this  is at address zero, this would be at address four. This would be At address eight, this would be at  address 12. And then we land here at address 16. Now, you might look at this and say, well, it says  one zeros in that 10. Remember that we're working in hex one, zero, and hex is 16 in decimal. So  just keep that in mind that you're always working

00:10:17 in hexadecimal. So that's just something to keep  in mind there. And that's the way that our stock memory grows. And you'll get really familiar with  stock memory. We're going to work with it a lot throughout this series of videos. So you'll get a  really good understanding of this as you continue to work with it. The LR register is another  interesting one, it's known as our link register. And the best way to describe this is if you've  ever worked in a higher level language before, you'll know that when you have a function, a  function has a return. And that return allows us

00:10:47 to move back to the location of what called the  function. That's what the link register stores, it stores the location that a function should  return back to. So we'll see that when we talk about functions in future videos. And then  finally, the PC is our program counter. What this does is it keeps track of the  location of the next instruction to execute. So all of our instructions are stored in  memory, like all the different things that we're asking the program to do. And the program  counter allows us to move through piece by piece

00:11:18 and determine where the next instruction is going  to be located. So those are three really important registers to keep in mind. And like I said, some  of these other registers have special purposes as well. And we'll discuss those as they become  irrelevant to us as we continue on through this. The one final thing that we'll discuss here is  our CPSR register, and our SPS, our register. Our CPSR register is a special type of register that  is used to store information about our program. So an example of some of the information that it  might start, if you subtract two numbers from each

00:11:52 other, you might end up with a negative number.  Now, if you remember, in binary, when we want to represent negative numbers, we need to use what's  known as two's complement, two's complement, essentially, you know, it goes through some  conversion mechanism. In order to convert us into that format, we have to do the ones  complement, and then we do the two's complement, and we end up with some number that represents  a negative number. The issue is that that number could also be a positive number. So it's both  positive and negative. But we need to know

00:12:22 whether or not it's representing like a negative  number or sister with the large positive number, that's where CPSR register comes in, what it  will do is it will set a flag in memory to say the result of the previous operation was negative,  we can then take a look at that register and say, if the result was negative, interpret this as  a negative number, otherwise interpreted as a positive number. That's sort of the purpose of  the CPSR register, it has a whole bunch of flags sort of similar to that, that tell us special  information about the operations, you know, if

00:12:54 a result was negative, for a result was zero, if  a result, how to carry or an overflow, those are the sorts of ideas that we would typically see. So  we'll see those again as we continue on learning about the different operations. So this gives  you a general overview of some of the important components that we need to start programming in  arm assembly. And in the next video, we'll take a look at how to create a really basic assembly  program. So that's what we'll take a look at next. So in this video, we're just going to take  a look at a really basic arm application

00:13:27 just to get a really good fundamental feel  of the structure of our programs, as well as the way that they actually run. So we're gonna go  over a lot of the important concepts of you know, where the starting point of our application is,  where the ending point of our application is, and how can we do all of the things in between. So  starting off with this idea, let's discuss the starting point of our program, you'll see that  our emulator automatically puts in two lines for us. Now, if you're programming outside  of the emulator, you won't have these two

00:13:57 lines added by default, you have to add them  yourself. But they are essential because they tell us the starting point of our application.  So we break down the lines one by one here. I'm going to start backwards with this underscore  start portion here. This is known as a label, a label is sort of synonymous with a  function in higher level languages. It's a way of being able to divide out segments of  code, such that if you go to the label, you start to execute the code that is underneath that label.  The idea of this is that we set up a start label,

00:14:33 which then we are able to say okay, go to the  start and start running this stuff that's there. That's the idea of what we're looking to do.  So that's why we define our start label. And then our global underscore start here  is a way of being able to tell people about this dirt label. So someone is going  to be running our program. So at some point something is going to be running our program,  it needs to know where the starting point is, as well. We define labels by D faults, they aren't  available to anything external to our program.

00:15:04 The dot global is telling everyone else about  our start label. So if anyone ever approaches our program and says I want to run you, where's  the start, the start is exported via this global, which means that everything is able to see  where the starting point is, therefore, things are able to go to the starting point to start  executing our program. So this is the general idea of sort of like our main starting point of  our application. So everything that we write is going to be underneath this start label. So that's  where we're going to start writing our code.

00:15:35 And the code that I'm going to write is going to  be very, very simple. All it's going to do is it's going to move some data into registers zero, that  way, you can see how we move data into registers. And then I'm going to move some data into  register seven, it might seem a little bit weird that I'm jumping straight to register  seven. The reason being is because register seven is a special purpose register. What it  does is it stores information about system calls. When we go to the operating system, we often want  to ask it to do specific things for us, right,

00:16:08 the operating system manages input and manages  output, it manages the execution of programs. So if we need the operating system to do something  for us, we need some method of communicating with it. This method of communication comes in two  pieces. The first is in system interrupts. And the second is in system called numbers. So we placed a  special number into register seven. And that tells the operating system what we would like it to do.  So basically, we placed the number to ourself, and then we call an interrupt the interrupts  go to the operating system, and it says, Hey,

00:16:44 we need something done, the operating system  comes in, and it reads register seven, which has some number inside of it, it takes that number  and it compares it to like a lookup table, and it says, Okay, well that number corresponds with  this task, and then it completes that task for us. In our case, our task is very simple. We want  our program to end. So if I want to terminate the execution of my program, what I do is I move  the number one into our seven, and then I call an interrupt the operating system then goes to  register seven, it says, Oh, I see the number

00:17:17 one here, it goes to it's a little table and it's  table says one corresponds to exit the program, and then it terminates the program for us. So  that shows us how we end our program. So let's take a look at how that's actually written. So  the first thing I want to do is move some data into our zero. That way we can see just generally  how we can move data into registers and what that looks like during execution. So the way that we  do this is as follows, we're going to start by writing in what's called an opcode, we'll use it  set in a number of different ways, either op code

00:17:49 operation, pneumonic is also used, I'm going to  refer to them as operations or op codes. So we're gonna start by running in the operation that  corresponds to moving data. And that is m o v. Now I use capital letters for mine, you don't have  to do that you can write it in lowercase as well, I just prefer this convention. And this is the one  that I'm most most comfortable with. So this is the way that I usually end up writing it. So the  way that our move operation actually works is it moves data into locations, right. Specifically,  we're going to move data into registers.

00:18:26 So what we need to do is we need to provide it  with two things, we need to give it a destination for our data, as well as a source for our data.  So the source is where we're getting the data from the destination is where the data is ending  up. The destination is the first argument given to this operation. And then the source is the  second argument. So my destination in this case, where I want to store the data is R zero. Again,  I put a capital R, you don't have to you can use a lowercase r. It's not actually case sensitive,  I just prefer the convention for capital letters.

00:18:58 So that's what you'll see me you. So I'm going to  move into our zero, the value, so I put a comma, and I'm going to specify the source. So my source  could be a number of different things. And we're going to go through these different things. As we  continue learning about assembly because we can, we can move data from registers, we can move  data from memory, so we can do all sorts of things like that. In this case, I'm going to  be moving a constant value into this register. So the way that we do this is we put in a hashtag,  and then we put in the value that we want to move

00:19:29 and I want to move in the number 30 in decimal. So  just put hashtag 30. All this does is it takes the number 30 it places it into register zero. And  that's it. That's how we move the data into the register. Now, one thing I want to point out to  you is that you can put hex into these registers as well. The way that you do that as you put 0x  and then the hex value that you want to put in so like for instance 0x Zero A, we write the value  zero a into register zero. So I just want to point out that that's the way that You do hex. In this  case, I'm working in decimal. And I'm pretty much

00:20:04 always going to be working in decimal unless  I have a very specific reason to be using hex, just because I'm most comfortable with decimal. So  that's what you'll see me using throughout this. So I moved 30 into register zero. And now we're  going to do our portion that ends the program. So remember, I said that I wanted to move into  registers seven, the value one, because one indicates that we're going to exit our program.  And then I want to do a an interrupt. The way I do this is as follows. I say SW, ei zero. SW i is a  software interrupts what it does is it interrupts

00:20:38 the program, and it lets the operating system take  over. Like I said before, the operating system then reads the value in our seven, that value  is one, it takes the one that checks the lookup table, it says, Okay, well, one says that we  should terminate the program. And so we're done. So that's really all there is to  that. So straightforward, simple. Let's go ahead and compile and run this.  So I'm going to click on compile and load. And as you can see, it compiles our program  together. And we are now in our disassembly tab,

00:21:11 and we can see each of the instructions as it  interprets them. Now, I will note to you that this emulator doesn't really do well with  us wi it doesn't really understand software interrupts all too well, I'm writing this as if  you were reading it for an actual ARM processor, if you're running this on, say, like a Raspberry  Pi that runs on arm as Wi Fi is going to be what you're going to use, but our emulator won't end  up, like executing it properly, such as note, just a little quirk of the emulator, unfortunately, but  um, if you're running on an actual ARM processor,

00:21:40 this is what you would end up using. So that's  why I'm teaching that instead of something more specific to this emulator. So let's walk  through these operations. So remember, the first thing is moving 30 into R zero. If I  press step into, it will execute that instruction. Now I want to point out a few important things.  The first important thing is you can see that r zero now has the value one e inside of it, which  I will tell you is the same as 30 in decimal, you want to verify that you can  actually come down to the settings here,

00:22:12 under format, you can go to decimal unsigned  or decimal signed, decimal signed assumes that there's negative numbers. Decimal unsigned  assumes that everything is positive. So we can go decimal unsigned, since everything is positive,  you can see that that gives us a value of 30. So you can see that's how we convert between like  the hex and the and the actual decimal value. So that's, that's something that we're able to do.  Now one additional thing that I want to mention is you can see that we started one e inside of here,  if you're familiar with converting hex to binary,

00:22:44 you'll know how to read the binary for this. And  you can see that in general, the most significant bit is on the left most side, we actually refer  to this as a specific type of storage known as Little Endian. So Little Endian refers to the  most significant bit being on the left hand side. That's the way that arm functions as well as many  other processors. But an interesting thing about arm is that actually can be used the opposite way  as well. So the most significant bit on the right hand side, which is known as Big Endian, we would  say that arm is actually a by Indian processor,

00:23:15 because we can change where the most significant  bit is depending on what we're doing. So that's just something that I've wanted to note about,  like how things are actually being stored in those registers. Then the last thing that I want to  point out here is the idea of the program counter. So the program counter is telling us where we're  currently located in terms of instructions, you can see, right now it's equal to four. And  at address four, we have our move for one and two are seven. So I just wanted to, you know, pull  some attention towards how the program counter was

00:23:47 using. That way, you can get a bit more context  compared to the last video where I just told you what it was doing. Now you can actually see it in  action. So you can see in our next instruction, we stepped into our seven gets a value of one, and  then we would end up doing a system interrupt, and that would end our program. And that's really all  there is to it. This is a really basic assembly program that we've now written. So you've now  gotten the basic fundamentals of writing a program and what the actual architecture is generally  looking like. In the next few videos that I

00:24:16 that I put up, I'm going to discuss a little bit  about some of the different addressing modes that we have available to us the different ways that  we can move data to and from registers as well as storing in the stack memory. So we'll we'll  discuss a lot more about memory storage. And then from there, we can start to take a look at  some some more advanced operations and then some actual like algorithms and building  some interesting stuff with our processor. So in this video, we're going to take a look  at the different addressing types that exist

00:24:48 inside of arm as well as other assembly languages.  And the general idea of addressing types is that there are ways that we're able to store  and retrieve data from the various memory locations that we have. So For example, in the  previous video, we used a type of addressing that is known as immediate addressing. And  that's when we want to move into a register a specific value that is constant. So whenever  we have like a constant value like this five on this side being moved into a register, or zero,  we call this immediate addressing, because we're

00:25:21 taking an immediate value, and we're placing  it into a register. A similar type of register type movement that we have is moving between two  different registers. So if I want to move now, what's an R zero into R one, this would be called  registered direct addressing. So we're directly moving a value from one register into another.  So those are two types of addressing that we have that work with the register memory, and they're  the most common ones that we'll typically see when we're working with registers. Now, the more  interesting type of addressing that we have has

00:25:54 to deal with data that's stored in the stock.  So to demonstrate this, I'm going to go ahead and show you how to get data onto this stock  first, and then we'll take a look at how we can work with the data that's on the step. So  first off, how do we get data onto the stock? To do this, we have to use a data section in  our application. And the way that we do that is we just type in data, this is going to come  below all of our like global start portion here. So you say put it just right down here. And what  we're going to do is we're going to declare any

00:26:25 data that we want to put in our stack memory.  And we do this in the way of giving it a label, which basically functions like a variable name,  that we then declare the type of the variable, and then the data that's actually stored inside  of it. So for example, I'm going to declare some data, I'm going to name the data list, I'm going  to put list colon, and then I'm going to go to the next line. And I usually like to indent to put  the next portion here just organize it nicely. And in this case, I want to store a list of  numbers. So in this case, when we're dealing with

00:26:58 numbers, numbers are going to be a specific size.  In this case, I want to work with 32 bit numbers. And as we've discussed 32 bits is considered  to be a word. So I'm going to type in dot word, this tells it that every single one of the values  that follows should be treated as a word type, which means that they are 32 bits in size. Notice  that we don't say like it's an integer, or it's a float or anything like that. It's simply it's very  basic, it's basically just got those those basic sorts of data types, usually you're gonna see  things like ASCII, or you're gonna see things like

00:27:27 the actual size, like Word or halfword, or bytes  or something like that. So in this case, now, we can just start listing off the numbers we  want to put inside of our lists. So I'll just put some some random numbers in, it doesn't really  matter what numbers you use, I'm putting in some some positives and some negatives, so you can  see the different types that exist. Now, the very first thing that I need to do is I need to be able  to retrieve where this list is located in memory. And what we do is we typically look for the first  entry in the list, and then everything is going to

00:28:00 follow sequentially from there. And basically,  what's going to happen is that they're going to appear in every single like slot in memory  sequential from the first one. So you'll see that when we actually load up this program.  So in order to get the first memory location into a register, we want to enter registers that  we can actually work with it and manipulate it, it's easier to work in the registers than it is  to work in the stock. So we want to get it into the register first. And to do this, we're going to  use an instruction known as El de R. And what ltr

00:28:30 lets us do is it lets us load data from stuck  into registers. That's the main purpose of it. So I'm going to load into register R zero,  the location of the first value in our list variable. So this equals list indicates that  I'm dealing with this list in my data section. And what it's going to do is it's going to find  out where this first value is located. And it's going to place it into registers zero. This is  known as direct addressing. And this is how we essentially initialize the location of our list.  So let's Compile and load and see what happens.

00:29:08 I'm going to go ahead and step into this  LDR instruction, and we're going to see that it's going to store the value eat into  the register R zero. So what does that mean? It means that if I come into memory, and I go  over to memory location eight, so this is zero, this is four, this is eight right here, you can  see that this looks like the start of my list. So if we look we have four five, negative 910 to  negative three. So I've got four, five, this might look a little bit weird. It helps to switch over  our view here in our settings to decimal signed

00:29:40 to see it a bit more clear. You can see that  that's negative nine. Just as a Moodle reminder, the reason why we see it like this is because  it's stored in two's complement, right? So we have all the F's, which is all the ones and then the  actual number there, right? So you can see that that's generally how that stored but this helps  us see it a little bit more clear. So you can see that these are all the entries in C I'd have our  list. So that's, that's nice and easy to look at. And now we actually have the location of the first  value in our list. So that's nice that we have

00:30:12 that location. The next question is, well, how do  we actually retrieve the value from that location? And the answer is that we have to use another  type of addressing mode. And this one is known as register in direct addressing. And the  way that that works is that we still use LDR, because we're moving from the stack into the  register. And what I'm going to do in this case is I'm going to load into register  r1, the value that exists at the address of Rs zero. So these square brackets tell our  assembler that we're really looking to find

00:30:46 the value associated with the address in  R zero. So when we compile and load this, what's going to happen is, in this case, we  get 10 as our value in R zero, or one, zero. So you can see that it changes every time that  I compile and run it, that's totally normal, every single time that you run it, the  memory structure will be slightly different, which is why we sort of have to do more dynamic  loading, like we're doing rather than hard coding numbers. And so it's just something to note as  well. So you can see that our zero is equal to 10,

00:31:15 we come into memory, we can see that indeed,  10 does look to be the location of the first element. When we do our next instruction, this  LDR R one and then the square brackets are zero, what it does is it takes a look and it  says okay, well when r is zero, we have this value here. And then what it will say  is they'll say, Okay, let's go to that memory address what's stored in that memory address,  okay, four is stored in that memory address. So we'll return that back as our results. And you  can see, that's exactly what we get is four.

00:31:44 So just to give you a little bit of an  analogy, if you're thinking about like other programming languages, it always helps  to think about this sort of like a high level world as well. In a lot of programming languages,  you might have like a list, you know, I could do like a Python, like list, for instance.  And we can have like our different values, like this. And what we're doing with this, with  this register in direct addressing is basically just trying to find the lists value at i, where i  is Rs zero. So it's really like the list value at

00:32:15 zero. That's basically what we're looking to do  with this. So that's just something to know. Now, there are other ways that we can access values  off of this list. One such way would be to use register indirect with an offset. And what  an offset does is it starts with the value in our zero. So for example, in this case,  it's currently 10, it adds some offset to it, and then retrieves the value. So for instance,  what I could do is, I could add four to it, because if I add four to it, that would take  me to the next location of memory. Remember,

00:32:48 this is 10, this is some 14 here. So I can add  four to it to get to the next location in memory and retrieve that value. So that would be an  example of something that we could do. So let's see how we actually do that, we would do LDR. In  this case, I'll put it into r two in this case, and we're gonna do R zero, has to take four. So  what this does is it takes the value in R zero, it adds the four to it, and then it retrieves that  value. If you sort of like doing R zero plus one, in the high level context, I do plus one here,  because we're just moving one index over in the

00:33:25 list. Remember, we're doing four here, because we  need to add four to this address here in order to end up at this location, which is the location  of the next value in memory. Such as generally how that ends up working, right, it's four in  hex for each stock location. And of course, we could do other things, right, like we could add  eat to get to this, we can add 12 to get to this, and we can keep going like that as well. So that's  just something to keep in mind with this too. And just to demonstrate to you that this really  does work, let's Compile and load it and let's

00:33:56 step into it, we can see we get 10. In this case,  four is the location of the first memory. And then five is what started the second location.  And we can confirm that that actually is true, we can see five follows from four. So we can  see that that does work the way that I have explained it, which is great. So the last  two types of addressing methods that we'll talk about here are known as a pre increment  and post increment. So I said previously, the the first one that we have here is the same  as doing like lists that are zero plus one.

00:34:28 With a pre increment, what we would be doing is  we would be incrementing, our zero, and then we would be accessing the value of our zero, so we  essentially increment it, and then we check it. So that's the way that a pre increment works. So  it increments before it actually gets the value. And the way that this works is we just put an  exclamation mark after this. So it's the same sort of syntax, we just put an exclamation mark  that indicates this is a pre increment now. So let's see what ends up happening. When I compile  and run this you can see that I get 10 Still

00:35:02 forest still in this, and you can see what's going  to happen as the result is going to be the same, we're still going to get five, because it's going  to increment this by four, which would take us to location 14. In location 14, we have the value  five. However, something very interesting happens that's different from the previous example and  nuts that are zero changes to 14 afterwards. In the first example, where we're doing  just an offset, our zero didn't change. When we do a pre increment, which has  the exclamation mark at the end, it

00:35:31 increments our zero, and axis is at that specific  location. So with this differs from our offset. And now finally, we'll talk about our  post increment, our post increment is very similar to our pre increment just different  than when we actually increment the value, right? It'd be the same as doing an access  to our list our zero, and then incrementing, our zero value one index further. And the way that  this is typically written is just like this would be our zero, and then hashtag for like this. So  that's generally the way that this will end up

00:36:09 looking. So let's Compile and look that and see  what happens. So you can see again, we're at 10, as our starting location, four ends up being our  value for our one. And then for our two, we end up with the value four, and then this increments  to 14. The reason why we got the value four is because remember, our zero started at the  beginning, which is 10. And then we ended up accessing at that location giving us four and  then we increment it afterwards. So remember, the increment comes afterwards, rather than before. So  that's the main thing to keep in mind with that.

00:36:45 And that covers all of the different address  loading types that we have available to us. So now you should be able to comfortably  manipulate data and be able to retrieve from registers and retrieve from Stack memory. And  we'll use this throughout various other videos as we continue on learning arm. So you'll get  very familiar with these different addressing types as we work with them, and you'll see  like good applications of them as well. In this video, we're going to discuss some  of the basic arithmetic operations that exist

00:37:19 in arm assembly. In addition to discussing  these, we're also going to discuss a few different modifications to these instructions that  tend to be present in a lot of the instructions in arm assembly. And they lost to do very  interesting things like setting the CPSR register after the instruction is run. And the reason  why we're going to discuss these first with arithmetic instructions is because arithmetic  is easy. everyone sort of knows about adding and subtracting and multiplying. So  it's a very easy sort of place for us

00:37:47 to work with these instructions. Once we get  a basic understanding of these instructions, we'll take on some of the more interesting  and complex constructions. Allowing us to be able to discuss these ideas with a solid  understanding and foundation already in place. So to start off with the very basics, we can  add, subtract and multiply numbers in arm using a set of instructions. So the first is add,  the second is subtract, and the third is multiply. Now you'll notice that division isn't  included in this division turns out to be

00:38:20 a fair bit more complicated than the other three  instructions, so requires a little bit more finesse to get it working. So we aren't going  to discuss any division we'll discuss addition, subtraction, and multiplication. So when we  want to add two numbers together, for instance, it's fairly straightforward to do, what I'm  going to do is I'm going to move some data into some registers. So I'll add some, we'll put  some numbers into these registers. And then we can use our ADD operation or add operation is going  to take in a destination and two sources. So the

00:38:53 destination for this it's going to be R two. And  then I'm going to add together two numbers. So I'm gonna add together RS zero, and R one. And what  this is going to do is it's basically going to do this, it's going to put into R two, the result of  R zero plus r1. So that's all this is really going to do for us. So just going to add the two numbers  together and store the result in our two. So that's nice and easy. Let's just take a look at  how this works just to get comfortable with it. So a compile and load. You see I can load five  into register 07 into register one, I add those

00:39:28 two together, and I get c which is equal to 12.  So five plus seven equals 12. So we can see that the addition instruction works successfully. So  that's the way that addition works for us and the other instructions work very similarly I  can change this to sub and what that will do is it will do the results are two equals R zero  minus r one. And then we can also do multiply. And what that will do is it will do R zero times  r one. The main thing to keep in mind here is that R zero is always the first operand  and then r1 is always the second operand,

00:40:02 which only really matters in subtraction, right?  And addition and multiplication, the order doesn't really matter. But in subtraction, it does,  right. And you can see that when we take five and subtract it from seven in this example here, we  would clearly get a negative number, right, we'd get negative two as a result. And that's where the  interesting ideas start to come in with arithmetic operations. So if I were to do this,  right, now, we can see that I get five and seven. And when I subtract them, I get this  big number here, this big looking number here.

00:40:34 And when I convert this to decimal sign, you  can see I get negative two, right? But if we're looking at it in hexadecimal, it doesn't really  necessarily clear. Because the reason why it's not necessarily clear is because we could have had  a very large number and RSC row and maybe a small number and our one that when I subtract them, I  would get potentially this very large number here. So the question is, is this a very large  number? Or is it a negative number? There's some overlap there, right? Because of the  way that we store negative numbers in assembly,

00:41:04 we don't necessarily know right off the bat, if  this is a negative number. I mean, I clearly know now because I put in five and seven. And I know  that subtracting those two gives me a negative number. But think for a minute, if we had, say,  a user input, and we subtracted two numbers that the user gave us, how do we know if they give us  a very large number or a negative number? Right? How do we know the difference between those  and just sort of maybe drive this point home, I'll just show you what I mean by it could have  been a very large number, if I were to put this as

00:41:33 as a hex, right? So I can say, let's say  1234567872, all F's like that. And then I would have to subtract one from that, right?  And we'll just change around this orderings, that it's the first one. Well, actually, no,  we can keep it like that, because we want to do our zero minus or one. So what I mean by  it could have been a large number is when I have this result here and I subtract it from this  number here, you see the result is the exact same, right? We get this all F's and the E at the  end. So you can see if I take this big number

00:42:07 and subtract one from it, I get this. But also,  if I take five and I subtract it from seven, like we did before we get the same results.  So this is why there's the question of well, is it a negative number, or is it a very large  number, subtract it from a very small number. And the way that we solve this problem is using the  CPSR register, which is this register right here. You'll notice some of these letters  along here, the N, Zed, the c v i, all these sorts of letters. Each of these indicate  the specific flag that is set in this register.

00:42:43 The end, for instance, stands for negative, so it  will tell you if the results of the last operation was a negative number. And that's exactly what we  want, right? We want to set that flag so that we can understand that this result is negative. And I  could do other things to it can take care of them if there was a carry in the operation if there was  an overflow, if the result was zero, and that sort of idea there. So how do we get that to actually  set because if you notice, when I compile and load

00:43:12 this and I step through these instructions,  nothing happens to the CPSR register? The answer is that we have to  use a special type of instruction called a in this case, I'll call it an arithmetic  with flags. And the way that it's represented is we just add an S to the end of our instruction,  so sub s in this case, so sub s is going to set the flags in the CPSR register. And you  might be thinking, Well, why doesn't it just always do this? Why doesn't the subtraction  operation just always set the CPSR register?

00:43:43 And the answer is because to set the CPSR register  requires at least one additional operation, right? It requires us to actually load data into  that register. And that, you know, adds a little bit of overhead to the operation. If we don't  need to do that, then we shouldn't. So that's why there's two separate instructions for this  because one of them is slightly more efficient than the other and slightly efficiencies do matter  when you're programming at at such a low level. So then the second question that you might ask  is, well, when should I use sub s over sub? And

00:44:14 the answer is you should use sub s, if one you  know that there's going to be negative numbers, or two, you don't know what the values that are  going to be subtracted are. That means you're loading them from some location that you don't  have control over. If you're just taking two constant numbers and subtracting them from each  other and you know what the result is going to be, then you can use you know, whatever instruction  is appropriate. If it's going to be positive, just use sub that's okay. If you don't know that you  should use sub s. So you can check to see if it's

00:44:45 positive or negative. So just to demonstrate what  this ends up doing, when I compile a note this, like I said before, when we do this,  now, you see that this CPSR register has changed. And you can see that n is now bolded that  means and it has identified that the result was a negative number rather than a positive one.  And that's exactly what we were looking for. Now, just to note, the reason why this is such  an eight is because so each of the four bits at the beginning of the CPSR register stands for one  of the flags. And that's the same with every bit,

00:45:19 every single bit is one of the flags. So since  it's set to 1000, that means that negative is set to one, which means that it was negative. And then  I think that the other four are the other three zeros, or the zero, the carry and the overflow,  which are all set to zero. And that's why they're grayed out here. So that's the way that those  flags are actually set. So if you're not working in an emulator, you'll just see that set as  eight rather than seeing the end highlighted, the end being highlighted is nice for when you're  learning because obviously that tells you hey,

00:45:50 it's a negative number. But if you're doing this  just like on like a Linux system or something like that, you won't see that you'll just see the  eight instead. So it's it's good to understand exactly what that's doing as well. So that would  be the idea of our arithmetic operations. Now, there's one other additional arithmetic  operation that is quite helpful for us. And it's the fact that when we add two numbers  together, in some instances, you might end up with a carry. And basically, what that means is that  the number was way too big to actually be stored

00:46:22 inside of a single register. So to give you an  example, if you have like, you know, something like this, and you want to add it to maybe  something like this, right, so we'll just say, we'll add this. And this is another example  where you would want to use something that actually sets the flags. And the reason  being is because when you do this addition, you'll notice that when you add these numbers  together, it's far too big to actually be stored in one register, because it overflows beyond 32  bits, right. So see, when you do this, it actually

00:46:55 sets the carry flag in here to indicate that there  was a carry that happened in this operation. So these are the sorts of operations where you end  up with potentially a carry on, right. So when these sorts of things happen, we want to be able  to catch them as we did here. But we may also want to use that carry at some point later on. So if we  want to add a carry operation, then what we need to do is we need to use the ADC a operation, what  this will do is it will add a query to a result. And what I mean by that is basically so if  we were doing this, this operation here,

00:47:36 the result would be r two equals R zero plus R  one plus the carry. So the carry will basically be a one if this flag is set equal to one, or it  will be a zero otherwise. So that's the way this works. So it's either set to zero if the carry  is not set, or it's set to one if the carry flag is set. So that's the way this ADC operation  works. And there are other ones as well. So we can subtract as well. Using the same sort of idea,  we can also multiply with the carry. But the most common would it would be the odd which would be  a a DC like this. So this gives you an idea of

00:48:17 some of the fundamental arithmetic operations.  And we'll see these as we discuss other assembly programs. But this just gets you familiarized  with how we add, subtract and multiply numbers and some of the little quirks of setting flags  when we do these different types of operations. In this video, we're going to take a look at basic  logical operators in arm. So these are operators such as AND or exclusive OR, AND negation that are  used to complete bitwise operations on registers. We're gonna take a look at how these operators  work. And in addition to that, I'll show you a

00:48:59 few basic things that we can do with these  operators, aside from the typical ideas of, you know, comparing if two inputs are the same  or not, or those sorts of ideas. So starting off with the instructions, the end or exclusive, or  they all work very similarly to the arithmetic operators that we saw in the previous video. And  by that I mean, so for instance, if we were going to do an end, we would have a destination for  the results and the two things that are being ANDed together. So that's the way that our and  operator will typically work. So for example,

00:49:34 if I were to do like a simple, let's move on let's  move into our zero with value. FF. Let's move into our one the value 22, for instance, and let's add  those two together and store the result inside of our two. When we compile and load this you'll  see that what we get is we get the two F's here, and this one we get 16 and then the result  of adding those together. is 16, of course, because all of the inputs in the first one are all  ones. So wherever the ones match up, we get one, wherever this year was matched up, we get zero.  And that gives us the exact same result back. So

00:50:12 you can see that that works the same way that we  would expect any typical logical end to work. So this would be the idea of our n. Now, we can also  do n and that sets flags using an S. So I just always want to note that with these instructions,  typically, there's always a complimentary version that will set flags as well. Most commonly,  you're going to see just and being used, since we don't usually have flags getting set with  the end operation, it's mostly just a matter of completing the logical operation and storing  the results. So that gives us or an instruction

00:50:46 or or instruction is similar. It's O R, R, for  or so just keep in mind, it's not or it's with two R's. So again, just to demonstrate how this  works, you can see there are two inputs are FF and 16. And when we are those together, we get FF  because of course, anywhere that there's a one, we end up with a one. And since f and f is all ones,  we always end up with ones which gives us the FF as a result. So that gives us our or operation.  And then finally, our exclusive OR is e o r. So this is exclusive, or so just to show how  this works, we have the FF the six t, and when

00:51:24 we do those together, we get e nine, remember that  an exclusive OR is going to only give us a one, if one of the inputs is one, not both of  them, right? If both of them are the same, we get a zero, otherwise, we get a one. And that's  our exclusive OR so you see that when we do this, we get exactly that result as expected. So those  are the very basic logical operations, they're very simple and straightforward. There's nothing  particularly special or interesting about them. The one that's a little bit different actually  is negation. And that's one that I want to talk

00:51:57 about now. So with negation, we do something  slightly different than just doing like a knot on the actual register itself, you might expect  that you might have like a knot operation that does that negation. That's not the case in  an arm, what we have instead is we have a special instruction called Move negative m v n  like this, what move negative does is it moves from the source the negation of the source  into the destination, so it negates the source, and then puts that result into the destination.  So it doesn't move and then negation at the same

00:52:32 time. So to demonstrate that, let's just do this  with our zero and itself, what this will do is it will move our zero, it will negate it and put  it back in the register that we started with. When we do this, you see that we start  off with all F's, and when we negate it, we get zeros in the first two bits and F's and  all of the later bits. Now, this might seem maybe a little bit strange. But remember, when  we do a negation, it negates the whole register, not just like the single bits or anything  like that it does it to the whole register.

00:53:05 Now, when we have something like this happen, we  might want to make it so that the other registers are set back to zero. And the question  is, how would we do something like that? And well, actually, we can use our and operator to  do something like that, right, what we could do is we could do an end storing the result in RS zero,  taking our zero and ending it with something that we know is going to give us zeros for this first  six bits, or those first six hex values, right, which would be 0x 123456. And then the others will  be FF to keep whatever is inside of them. This

00:53:41 will clear all of these bits because ending zero  with anything gives us zero. And since these are always one, if we add anything with them, we'll  get back what we started with right, either a zero or a one depending on what the value actually  is. And now the result of this is we start with the FF, we'd negate it. And then you can see  when we do the and clears all the other bits, giving us this result back. I know this might  become a little bit more clear. If I use like, let's do a for instance, when I do a for an  example, you see that we get five, five here,

00:54:16 rather than zeros, you see that when we query it,  we keep the five, five and get rid of everything else. That's something that we can use and to be  able to do is clear specific bits and register. So that's one of the applications of and so  this gives you a very basic overview of the different types of logical operators that exist  in arm. This is, of course, very high level and just like very simple examples, right? I just  want to give you an idea of the instructions and idea of what they might be useful for. And  then from there, we can continue to build on top

00:54:49 of those concepts. So this gives you the very  fundamentals of the logical operations and then in future videos, we'll see how they might be  used in more practical sorts of applications. In this video, we're going to start to discuss  logical shifts as well as logical rotations. And these two instructions are really useful in  the sense that they allow us to manipulate numbers at a bit level by way of shifting everything  to the left or to the right. And it has a lot of unique and interesting properties that are  actually really useful to take advantage of.

00:55:25 So what I'm going to do in this video is, it's  easier to look at this as a bit level operation rather than with hex or decimal. So I'm gonna  demonstrate to you what it looks like from a binary perspective. And then in the next video,  we'll actually take a look at some examples with instructions in arm. So this is just going to get  you that basic foundation of what is a shift? What is a rotation? What does it look like? What does  it represent? Why do we use it, so it's going to be the main call here. So we'll start off with  shifts. With shifts, there's two different types

00:55:55 of shifts that we see there's a logical shift to  the left and a logical shift to the right. And these two differ based on what direction the bits  in the binary number are shifting in. So to give you a basic example, let's just pick a binary  number, let's say 1010. This is 10 in decimal. So when we look at this number, when we're  doing a logical shift to the left, what that means is that I'm going to move everything  one position to the left of where it started. So this zero moves over here with this one  used to be this one moves over here with a

00:56:29 zero used to be the zero move to the one.  And remember, since there's eight bits, I didn't write them in here, but there's,  you know, eight bits of space here available, the one is going to move over here to this empty  bit that didn't used to have anything in it. Now the actual result of this is the following, we get  1010. And then there's an extra zeros at the end. Now when we take a look at this binary number,  we actually see a really interesting property come up. And that's the fact that this is actually  equal to 20, when you translate it to decimal,

00:57:00 and notice that this is exactly double what we  started, it's like we multiplied the value by two. And it turns out that isn't coincidental. That is  true for every single instance of a logical shift to left, it's the same as multiplying the value by  two. As such a logical shift to the left actually represents a fast way to do multiplication  by two. And that's the main value of doing this logical shift. Now, there are other  reasons why you might want to do it. But this is one that's very obvious and prevalent. So  this is something that we could take advantage of.

00:57:34 So this would be the idea of your logical shift  to the left. Now, when we take a look at a logical shift to the right, it does something very  similar, but in the opposite direction, right. So we start off with 1010, which again, is  equal to 10. If I shift everything to the right, it would mean that this one shifts over here with  a zero used to be the zero shifts over to the one, that one shifts over to the zero, and then  we just get rid of the zero at the end, right. So this zero just gets it disappears,  right. So we don't really care about it anymore.

00:58:05 Another thing that I'll just note here  quickly, remember, there's technically zeros to the left of this, right. So this zero  here moves into the bishop where this one used to be. The result of this is the following, we  get 0101. And what is this equal to? Well, this is equal to four and one, which makes five, notice  that this is exactly half of what we started with. So it turns out that when we shift to the  right, it's the same as dividing by two. So remember that we didn't really have a  way to divide numbers before there isn't

00:58:36 really a division operation in arm. This  is a way that we can implement division, we can do divisions by two, which allows us to  be able to actually do full division operations. So this allows us to have division. And again,  there's other reasons that we might want to do shifts to the right or shifts the left. But these  are the two most obvious ones that are really valuable to know about. And just in case, you're  really wondering, like, why is this true, you can you can intuitively figure out why this ends up  multiplying by two, right? If you think about what

00:59:08 shifting to the left really does is it increases  the power of all of the powers of two here by one, which is the same as multiplying by two, whereas  this one decreases all of them by one, which is the same as dividing by two. So there is actually  an intuitive reason behind that if you want to go through and actually sort of like derive why  this is true. So it gives you an intuition behind logical shifts. And hopefully it gives you a bit  of an understanding of why we care about them and what they are useful for. The next thing that  we'll talk about is a rotation, which is our O

00:59:40 R. And a rotation only really differs from  a shift in one single way and that is that's when we talked about the shift to the right I said  okay, this zero we just get rid of it right so the right mouse thing we just get rid of it. We don't  care about it anymore. With the rotation that doesn't happen. The right most thing actually loop  back over to the left most position. So let me show you an example of that. Suppose that we have  the following. So do all eight bits this time. And suppose we have something like this. So  first off, if I did a logical shift to the right,

01:00:17 what you would end up with is you would end  up with, you know, we have all the zeros here, and then we have zero, remember, everything shifts  over to the right one, so we would get 0010. This would be a logical shift to the rights in this  case, right? Now, if we were to do a rotation, but we end up with instead, as we end up with  this, we end up with 10000010. Notice that this one moved to the front rather than just getting  eliminated, since gives you the difference between the logical shifts in the rotation. Now, you  might ask, why would we want to use a rotation,

01:00:52 the uses of a rotation are actually  a lot more abstract than a shift, where we're dealing with rotations, we're  typically using it for things like hashing, we use it for crypto, we use it for graphics,  it's not an entirely common operation to see, it's typically actually left in for historical  reasons. And one thing that you'll notice because of this is that arm doesn't actually have  a rotation to the left like an arrow, l, this does not exist. So we can only rotate to the  right. If you want to do a rotation to the left,

01:01:22 you can technically do it with rotation to the  right. So if you want to shift n bits to the right, what you would do is you would do sorry,  if you want to shift n bits to the left, rather, what you would do is you would do a  rotation to the right 32 minus n times. And you know, you could sort of play around  with that and see that it actually is true, what that does is it rotates over to the left  hand side rather than the right hand side. But again, these sorts of operations, the  rotations are not very commonly used, we

01:01:54 don't see them all that much. So they're sort of  just something that I'm adding it to here to say, hey, these exist, there are some applications  of them and hashing and crypto and graphics, you might see them at some point. But they  aren't entirely common. But this is how they work just in case you ever do see them. So it  should give you a good groundwork foundation for logical shifts and logical rotations.  In the next video, we're going to actually go through and see how we can implement  these things in our arm assembly emulator.

01:02:24 In this video, we're going to take a look at  how logical shifts and rotations are actually used in arm. So we're actually going to take  a look at some practical examples and see how these instructions actually work. And be able to  demonstrate that they do actually do as I said, multiplying by two and dividing by two, and make  sure that these results are actually truthful. So to start off with, let's look at a very  simple example, we'll do something similar to what we did in the previous video where  we were working on the binary numbers. So

01:02:52 what we did was we had 10, and we did some  chips on it. So we did some shifts to the right, some shifts to the left. And we'll take a  look at what those results end up being. So to start off with, I'm going to go ahead  and move the value 10 into register zero. Once we have done this, what we're  going to do is we're going to do, let's do a logical shift to the left to start  off with. So the way that we do this is we do LSL we type in the register that we want to shift. And then we're going to indicate  how many times we want to shift.

01:03:27 So what I was demonstrating in the video previous  was that we could shift one time to the left and we would be shipped one time to the left it was  the same as multiplying by two. With arm we can actually specify to shift multiple times if we  want to. And the way to think about that is that it's the same as just multiplying by two repeated  amount of times. So if I shift to the left twice, what I'm doing is I'm multiplying by two twice, or  multiplying by four, right? If I shift three times that I would be multiplying by two, three times,  right. So that's the idea of what we're looking

01:03:59 at when we're saying how many times we would  like to shift. So if I want to shift everything to the left one time, which would be the same as  multiplying by two, I simply indicate that here. There's something that's interesting about  this is that you can actually use this to shift by a variable amount, right? So I  could put in like a register like r1 and say shift by the number of times that's inside  of the register r1, right. So there's ways of being able to combine this together to be  able to utilize like a variable amount of

01:04:29 shifting. So that's one thing that you're  able to do that I just want to note here. So with that being said, let's take  a look at what this actually does. So we can pile on load, the first load is  going to put 10 inside of this register, and we're gonna go to change this to  decimal just that way you can see it a bit easier. So you can see that we have 10  here. The next time that we move through we do our shift which changes it to 20 and as  you can see it does as I said it would it multiplies it by two. And just to continue on  With this, we'll go ahead and do a rotation to

01:05:02 the right and just what we'll do is shift to the  right rather, and see what that ends up doing. Now, what I'm anticipating this is going to  do is it's going to shift to the right by one bringing us back to 10. And let's just see if  that actually does happen. So we'll go ahead and compile and load, we started at 10, we ended  up at 20. The next shift brings us back to 10. So you could see that generally, this is working  the way that we expect it to. So this is nice and simple, nothing too complicated here. Now, one  thing that we can do is shifting that's quite

01:05:36 interesting and unique is that we can combine it  with the move operator to do a move and a shift at the same time. This is a really useful sort of  thing to be able to do. To give you an example, maybe I don't want to shift R zero, maybe I want  to store the result of multiplying R zero by two into register r1. So just to demonstrate what  that would look like currently, with their current instruction sets, I'll just go ahead and  get rid of this line. What I'm saying I want to do is I want to move, you know, r into r one, the  value r zero. And what I want to do is I want to

01:06:10 do a shift on r one by one. So what I'm doing  is I'm preserving the value of R zero, right, so that way, the value of R zero doesn't  actually change R one stores that value instead. Now, rather than having to do the move and shift  in separate instructions, we can actually do them together. So what I can do is I can add  on a third piece here, I can see LSL. One, just like this, what this is going to do is it's  going to move the value stored in RS zero into r1. And then it's going to shift it by one. So  let's take a look at what happens. So we get 10.

01:06:48 And you can see we get 20 directly. So you can  see that it does that shift immediately, right. So we don't have to worry about it happening, you  know, multiple times, right, we don't have to do it in multiple separate instructions, we could  do it all in one instruction instead. So this is something sort of unique about this shifts that  we haven't seen with other sorts of instructions so far. So that shows you how the logical shifts  work inside of arm. I'm going to end off here just by showing you a rotation, just that way,  you're able to see what that ends up looking like.

01:07:22 So the instruction for rotation is, of course  are or like we said. And of course, we could do some examples of shifting by one for instance,  some, although we should pick a number that will actually flow over into the remaining, you know  end. So that goes over to the left hand side. Let's see, I think I've value perhaps like, well,  that you like 15 should definitely do it right, because we would have all ones on the right hand  side. So let's go ahead and give that a try and see what happens, we get 15. And you see when we  overflow over we get this massive number here,

01:07:58 this one that actually helps to look at the  hex to see what ends up happening, right. So you can see that we have eight as our first hex  value, which is 1000. So you can see that the one shifted over to the front, we still have seven  on the right hand side, which implies that we have you know all of the ones remaining except for  the one that was on the end, right we have 0111. And then on the other hand, we have 1000 on the  left hand side. So you can see how this rotation is generally working, it is doing sort of the idea  of what I was saying would happen with it. So this

01:08:30 gives you an idea of how the rotations work. And  I want to note as well that you could do the same thing with rotations that we did with with moving  right, we can put it into the move instruction, and we can rotate as loyal as the move right.  So the same sort of idea is what we did with the shifts before system instructs you how  rotations and shifts work in arm. So you now understand what these operations do what their  value is, and how the instructions can be used. In our next video, we'll take a look at some other  instructions through arm and we can get a better

01:09:00 understanding of some more advanced things  that we can do with this programming language. Now that we understand some of the basic  instructions in arm, we can start to take a look at some of the typical high level constructs  that we would see and find out how they map generally into the assembly language. And the way  that we're going to start with this is by taking a look at statements related to conditionals.  These will be statements sort of like IP statements in higher level languages. We want to  take instructions based on the result of something

01:09:36 else that has happened in the program already,  perhaps something related to a user input or something of the sort. And we want to be able to  take different paths based on the values that are provided. So in this video, I'm going to give  you some really basic examples of how this can be done. So the main way that we do these branching  instructions is using comparators and branches. So it can be error is something that allows us  to compare two values to determine whether or not they are greater than, less than or equal  to each other. Branches, on the other hand,

01:10:10 allow us to move around our program to different  locations based on the result of comparisons. So for example, we might want to move to a different  part of the program if a result is greater than another, or maybe if two results are equal, we  move to a different location in the program. This is very similar to the way that if statements  and L statements work in your typical programming languages. So let's take a look at a really simple  example. I'm gonna go ahead and move some values into some registers, we'll do some comparisons and  see generally how these branches actually work.

01:10:46 So let's move to our zero the value one. I'm going to move to into r1. And now I'm going to  do a comparison between these two values. And the way that we do this is we use a cmp. That's our  comparison instruction. And then we give it two arguments. And we'll discuss how this generally  works once we've got it written out here. So what a comparison is really doing is it's  going to take a look at R zero and R one, and it's going to do the following computation. Gonna  do R zero minus r one, so it's going to subtract

01:11:24 the two from each other. Why does it do this,  because this is a way of comparing two values, we know that if R zero is bigger than r1, then  the result of this will be some positive number. If our zero is smaller than r1, the result  will be some negative number. And if they're the same as each other, of course, the result is  going to end up being zero. So we can know those three results based on the subtraction of the  numbers. So what will happen is the computation for the compare will do this subtraction. And what  will happen is this CPSR register, this one right

01:12:00 here is going to get set based on the results of  the subtraction. Now as we briefly discussed in this CPSR register, we can set different things  like negative zero carry overflow and these sorts of flags. Well, it turns out that that's very  useful for this situation, because suppose if we subtract these two numbers, and I want to see if  our zero is smaller than our one, if our zero is smaller than r1, I expect to get some negative  number. And that would mean that the negative flag should get set in the CPSR register. So  you see how all this sort of fits together,

01:12:33 we've sort of put the puzzle together. And we  can see how we can use all of these different pieces to get our conditional statements. So  this is a very high level construct that we're able to replicate in assembly. For that matter,  when you compile your high level languages, it's using this sort of idea in the  background, maybe not exactly like this, but similar sorts of ideas to this. So understanding  that what happens after we do this comparison, well, in this case, maybe we want to check to  see if so let's say we want to check to see if

01:13:04 R zero is bigger than r one. So we want to see  a greater than r1, what we can do is we can use a branch greater that which is written as b g t,  so it's B as a branch and then GT is greater than. And what we do is we give this a location,  and the location is going to be some label that we did also discuss briefly the idea of  labels. So start is an example of a label. And we can actually add more labels if we would like. So  I can add in a label like greater, for instance, this would be the label that will run when the

01:13:42 branch greater than occurs. So what I can say,  I could say b g t, then I can see greater. What will happen is when it does this comparison,  if it finds that R zero is bigger than r one, it will move to this greater label, and it will start  to execute whatever is provided here. So in this case, we'll just do something to see that this  is working, we'll move the value into our two. So that would be an example of this. Now, let's  go ahead and run this and just see what ends up happening. So you can see generally, that these  values actually are greater, right, our zero is

01:14:21 not greater than our one, which means that we  shouldn't actually take this branch here. Now the question is what happens after if we don't take  this branch? And the answer is we just continue on as if we never actually executed this instruction.  Let's maybe make this a little bit more clear. Let me add in a move instruction inside of here.  Right, so we can add in this move instruction. And we could see that when we do this, the comparison  ends up not being a value that's greater. You can see the negative flag is set indicating that the  subtraction was a negative number. When we do

01:14:56 this, you see that it skips over this branch  and it just moves into the next instruction. So you can see how that generally works. Now,  if it were greater, right, if I said three, for instance here, and we ran this, what would  end up happening is we do our comparison, you see that we don't get a negative  number, instead, we actually get a carry. And this indicates the value was  bigger rather than being smaller, you'll see that we can move into this  branch for greater than you can see that

01:15:24 rather than moving into this instruction,  it skips it and goes to this greater label. So this gives you an idea of what's actually  happening with these branches. Now, generally, when we're doing these sorts of instructions, it's  important to note the flow of assembly languages. If the value isn't greater than when we have  this instruction, here, we saw that it moved to this instruction. However, once this instruction  finishes, assembly will continue moving through until it reaches the next instruction. And you  can see what ends up happening is we end up in

01:15:58 this greater branch anyways. And that is,  of course, a problem we don't like that. But we want to do is we want to skip the  greater branch, we want to make sure that we don't actually reach it if we're not greater than  that value. To do this, we can use branches as well. So there's a special type of branch called  a branch always be a L sub branch and then always, and this will always execute. So we'll always  branch this location, regardless of what happens. So for B A L, what I could do is I could say  rather than going to greater, I can go to

01:16:32 a different one, you know, maybe I'll call this  one default. So in this case, we can create a default label. And then we could do something  here. So I can do the same thing as before. Right, so we move to into our two. And we can  see that in this case, if we don't reach this greater branch, we move here. And we always branch  to default, which skips over grader and ends up here instead. This allows us to effectively bypass  this greater instruction. Now, another thing that we need to keep in mind here is once we finish  the greater portion, right, we will also fall

01:17:11 into the default case. So it's just very important  to note that the instructions are always going to run sequentially, I want you to understand that  very clearly that everything is going to run sequentially here, right. And I can show you  this right. So if I compile and load this, we have three we have to we subtract them,  we do the branch for greater than we end up in this move instruction, right. So we do this  move instruction. And you can see that we end up in default, next, right, and then it runs  out instruction. So it's very important to note

01:17:41 that it's always going to move sequentially,  right. So if we don't want to go into default, after greater, I should add in another branch to  move somewhere else. So this is just something to keep in mind when you're writing these  branches. Everything runs sequentially, that can be a little bit tricky. So just be very  careful of that stepping through the programs, it's going to make it very clear what is going  on, you'll understand it completely once you step through a few programs. But just keep that  in mind and keep thinking about that as you're

01:18:08 writing your code and running it and practicing  with it. So this gives you a general idea of the different branches and how they generally work  in our program. Now there's of course a lot of different branches. And I'm not going to like show  off, every single one of them are going to have the amount of detail for them, I'll just sort of  like lay out a few of them here that are common. So if greater than and greater than or equal  to, we have less than and less than or equal to. And then we have equals or does not equals those  the different branches that we have available

01:18:41 to us. And they're fairly intuitive, and they may  write each u equals any not equals lt is less than L e is less than or equal to so on like this, you  can get a whole table of these through the arm documentation, there's actually a lot  more than the ones that I'm showing here, I'm going to show you the most basic ones  that will come up most frequently. If you ever find yourself needing a very special branch,  it likely exists. And you can find that branch and the actual documentation. But this gives you the  main overall picture of what branches are commonly

01:19:12 used. How do we use branches, and what is the  effect of using the branches. So that's everything for this video. In the next video, we're going to  see some applications of branching in the form of looping. So branches allow us to implement  looping, good repetition as well. So we're going to take a look at this sort of concept and  be able to build some interesting applications and hopefully help you understand this idea of  conditionals and branching a little bit more. In this video, we're going to talk a little  bit about looping as a consequence of the

01:19:42 conditional statements that we learned in the  previous video. When we discussed the idea of conditional statements, one of the big benefits  that we can get out of them is the ability to repeat instructions. Of course if we could branch  to a specific label, we can always move back to a specific starting point and repeat instruct shins.  And we can always exit that repetition using a comparison and some form of branching as you  would with like a for loop or a while loop, right, so you're able to get that exact sort of idea.  Looping, while some sort of condition is true,

01:20:14 or while some sort of condition is not true,  as well as looping a specific number of time. So we can get both of those loops working in  assembly in the same sort of way that we would in a high level language, maybe with a little  bit more finesse required to make it actually work. So to better understand this, we're going to  take a look at a relatively simple example, that's going to draw from a lot of the things that we've  learned so far. So it's going to be a great way to sort of refresh your knowledge and apply some  of these instructions in a more practical way.

01:20:44 What I'm going to do is I'm going to  start off by defining a list of numbers. And this number list is just like all  the numbers from one to 10. So something very simple for us. And what we're  going to do is we're going to load this list into memory, and we're going to try  to iterate the list. So the first few parts out, it's fairly simple, right, we're going  to load the list into memory using LDR. So that part is it's no big deal. And then what  I'm going to do is I'm going to start by loading

01:21:23 the first element into my into my memory, I'm  going to use R one to store the elements of the list. And what I'm going to do is I'm going to  add up all of the numbers inside of the list, and I'm going to store that result in our two  sets sort of the main idea of our program. So let's go ahead and do that. We'll start by loading  the value of the first value of the list into r1. Of course, we use that using addressing  based on the address that was placed inside of RT zero. And then we can do the  addition to add that first value into our

01:21:57 to do sort of like initialize everything right. So that gives us sort of like an initialization.  And then we can actually start to talk about the looping portion of this. So this starts up  the list. So what we've done so far is we've sort of set to the first value of our sum equal to the  first value in the list. And then what we're going to do is we're going to iterate past that point  and add those values on to that or to register to we've reached the end of the list. It is an  interesting concentrate, how do we find the end

01:22:27 of the list? Well, let's take a look at what the  list looks like in memory, good code and compile and just step into it to find the location on  my list. It's that location 10 in memory. And you can see it right here, you see all the  numbers from one to 10. And then afterwards, we get all of these A's repeated. Now it turns  out in the case of my memory, I have all A's as a representation of an empty slot in memory. So  it goes to serve that if we have this set of A's, we've reached the end of the list. So I'm going  to continue to move through the list iterating

01:23:02 until I reach something that is equal to this.  Now how we do that check is actually a little bit complicated, we actually have to do  something sort of interesting here. Now, when we work with assembly, you might think  okay, well, that's easy, I can just compare, you know that value, and then we'll be done. But  the way that assembly, specifically arm assembly handles literals is a little bit weird. So when  we have those immediate values, or literals, they can only be a specific size. And above that  size is typically two hex values. So if we have

01:23:36 something bigger than two hex values, we have  to be a little bit more creative with how we deal with this. And the main way that we're going  to deal with this is we're going to use constants. And constants isn't something that we've really  talked about so far, but they're very simple. At the very top of our program, we're going to define  our constant using a keyword that is.eq You. So EQ you like this. Similar to defining our  list, we're gonna give this a name and a value. And I'm just gonna go ahead and copy it because  I don't want to mess it up here. So just copy

01:24:14 this over there. And there you have it, that  is our constant defined. So this defines a constant called the end list, and it gives it  a value, which is this value here. And then we can load this into memory using an LDR. Like  we've done with the with the list loading it, it's going to load this into R three,  just to have it available to us. So there we have this value now  loaded into R three. And now we can actually do our loop. So I'm  gonna start by creating a label for our loop. And what our loop is going to do  is it's going to load the next value in the list

01:25:01 So what this will do is we'll increment by  four, and then it will load the value of that address into r1. And then it's going to take  a look at this value and see if it is equal to that location in memory that is empty. So this  all these A's here. So remember, we stored that in our three, so we're going to compare R one, two,  R three. If those are equal to each other, we want to leave the loop. So I'm going to say B Q exit,  I'm going to put a label exit here. And the label exit isn't going to do anything, it's just going  to be the exit point of this loop. Alternatively,

01:25:36 if we have not reached the end of the list, so  what I'm going to do is I'm going to add the value into our to, and then I'm going to loop again,  so I'm going to branch to the top of the loop. So to understand this, what's going to happen is  it's going to load the next value in the list. If that value is equal to the end of the list, so  this set of memory that sort of is initialized, then we're going to go to the exit, otherwise,  we're going to add that value to the total. And then we're going to loop again. Now this isn't the  only way that we could find the end of the list,

01:26:09 we could know the length of the list, that could  be another way that we might be able to do this. And there are likely other tricks that work in  assembly to be able to do this. However, mostly when we have the length of the list and makes  things a little bit easier for us to be able to do. There's an interesting idea of uninitialized  memory, it can be a little bit unpredictable, right? We don't necessarily note these values  are going to be stored this way. So the way that I'm doing it right now is a simple example  that's going to demonstrate to you the idea of

01:26:38 looping. In the real world, when we're doing this,  it's a little bit more complicated, what we would end up doing is we would end up putting a special  value at the end of this list, you know, something predictable, and then when it loads into memory,  we know what value is at the end of the list, I can actually give you a practical example of  this. If you're familiar with C programming, you'll know that characters and strings  have a special value at the end of it. And it's this backslash zero character, it's  called the null Terminator. What this is

01:27:07 doing is it's indicating the end of the lists  that when we iterate, we know where to stop. So that's just to give you a bit more of a  practical idea of how we do this. But for now, we'll assume that we're in this perfect  world, we're uninitialized memory stays put, and we don't have to worry about that. But  just in case, you're curious about that, that would be the way that we would usually handle  this sort of idea. So with that out of the way, let's just talk about what this program does,  let's go ahead and execute it. So you can see

01:27:35 that we get the location into memory, we then  load in that end location, so we can check for it, we then start by loading the first value, which  is the value one, and we add that on to the total. Now we get into the looping. So you can see the  next value is to recheck if that's the end of the list it's not. So we add that value to the  total. And then we go back to the top. And then we continue doing that. And you'll see that this  continues on for a while, right. So we're gonna do this for every single value inside of the list,  it will just keep looping through right, right 789

01:28:11 and 10. And once you reach 10, you'll see that  the next value is going to be that end of the list all of the A's there. So at this point, we compare  them and we see that they are equal to each other. And you see that it skips us out to the exit  rather than adding it to the total and to looping again. So at this point, we've exited the loop,  everything is done, we've iterated the list, and you can see that the value  that we got at the end was 55, which is the value of adding one to 10 inclusive.  And that's the way that we set up our loops.

01:28:41 So using this sort of concept, you could do all  sorts of different loops, you could do for loops, you could do while loops. And you know, you're  able to sort of expand your knowledge from here. This gives you the basic fundamentals, it shows  you how you can use the branching for the loops. And from here, you can apply this in many  different ways whenever looping is actually required. So that's everything for this video.  In the next few videos, we'll take a look at some other instructions that we're able to use related  to this idea of sort of structuring our program

01:29:09 through conditionals and looping and see different  ways that we can manage this sort of idea. So in this video, we're just going to discuss some  of the fundamentals of conditional instructions. Now conditional instructions allow us to be able  to complete a specific instruction based on a condition. So similar to the comparisons that  we saw with branches. What we're able to do is we're able to do a comparison, and then specify  to do another operation based on the result of that comparison without the need to branch around  the program or do anything complicated like that.

01:29:44 So it allows us to create a more condensed  form of our branching exercises that we were looking at in the previous few videos. So let's  discuss a little bit about this with an example. So suppose that I were to move some values into  registers. We'll just move some random values doesn't really matter what they are. So I'll move  to into R zero and maybe we'll move forward to r1. Now we'll do is we'll compare R zero and r1. And of course, we know that the result of this  comparison is that our zero is less than r1,

01:30:23 right? Because if we subtract two from four,  we get a negative number. Now in this case, maybe I want to add one to register two, if  the result of this was actually less than, if we wanted to do this in the previous videos,  with the knowledge that we had, what we would do is we would do something like this, we would  say, Okay, we would branch less than we could call it like, add our two, and then we would have like  an ad or two that would do exactly that, it would add to the register, r to the value one, right.  So that would be the idea of what we could do

01:31:00 in order to achieve that effect. And then we would  have to have a branch over here to skip over it just in case, you know, we didn't actually want to  do that, right, just in case, it wasn't less than that we could skip over it and say, like, maybe  skip over to like an exit. And that would be, you know, over here. So this would be the way that  we would have done this before with just branches, right. And this is not bad. But there's a lot  of lines of code for this specific instruction. And really, it doesn't require as much like of  this code as what we really want, right. So we

01:31:33 can condense this a little bit more using these  conditional instructions. So the way that these conditional instructions work is that with things  like add, you can add on a condition. So I can say add LT, what that means is add less than, so what  will happen is this instruction will only trigger if the comparison that came previous is less than  so if R zero is less than r one in this case, then we will do this addition instruction.  So like I was saying before, we'll try adding one to R two. And let's just see what ends up  happening. Right. So you can compile and load,

01:32:10 we see that we get two and four into those  registers. When we do our comparison, you see that the negative flag gets set in the CPSR registered,  that tells us that the value was less than right, so two is less than four. And now we come up  to this ADD instruction. And it should only add one to our two, if the value of the previous  comparison was less than. And we can see that it does add one because the value was less  than. So you could see that generally this is how this instruction tends to work right. And  there are other variations of this instruction,

01:32:41 it's not just add that we can do this with  we could do this with a lot of the different instructions in arm. So there's things like  um, like subtraction and multiplication, we could do with the move instruction as well. So  just to give you like another example, we could do something like this like move greater than  or equal to the value five and to our to write, this would only trigger if the value is greater  than or equal. So you can see when we run this, and we go through our instructions,  it doesn't actually do anything,

01:33:11 because the value is not greater than or equal.  So the move instruction doesn't execute. So that'd be an example of you know, whether it will  execute or not. And of course, we can flip this around to be able to get it to execute,  there's two different ways to do that, we can move a bigger number to our zero. Or we can flip  this comparison, right. If I flip this comparison, it now does R one minus r zero, so four minus  two that's a greater than or equal to value. And when we run this, now, you'll see that we  get to four, you know, we do the comparison,

01:33:44 you can see the results in the CPSR register will  result in it moving five in because the result was greater than or equal to. So just wanted to give  you have an idea of the sorts of instructions just to show that they exists in assembly, just to be  able to make your code a little bit more simple. Because having to run branches all the time is  very complex. We don't want to have to do that every single time. So this allows us to reduce  the complexity of our applications. Overall, it just write simpler code. So that's everything  for this video. In the next video, we're going

01:34:16 to dive into the idea of functions, which are a  bit more of a complicated way of doing branching. But as you probably know, core to most  higher level languages is the idea of a function. So we'll dive into more detail to get an  understanding of how functions actually work out a low level perspective, and get some familiarity  with writing functions with arm assembly. So in this video, we're going to start to  discuss the idea of functions in assembly. And specifically, we're just going to take a look  at how we can call and return to a location using

01:34:48 these functions. So we're not going to dive into  a huge amount of detail. I want to give you sort of the higher level functional flow of things.  And the purpose of doing this is that well, functions tend to be very complicated. A lot  of People approach this by showing you the full picture immediately. And we start getting into  all these complicated register interactions. And it's very hard to understand. So I want to  break it down into smaller pieces that it's a lot easier for you to be able to grasp every  single individual concept. And then we can put

01:35:15 it all together and really run with it and see  how things are working with a full example. So to start off with, we already know sort  of the base fundamental ideas of calling different locations and our program, right moving  around to different labels in our application. So let me give you a really simple example.  Suppose that we wanted to create a program that added two numbers together. And we wanted  that to be a function, because maybe we might do it multiple times throughout our program.  Well, in order to do this, we're going to go

01:35:46 ahead and start by moving some values into  some registers that we want to be added. And related to to make these values, whatever you  want, I'll just move into our zero or one. And then what I'm going to do is I'm going to branch  off to a function that's going to add the numbers together, and we're going to call that add two. So  we'll call it a su. And then I'll define it here. And all this is going to do is it's just going to  add the numbers together and store the result in our two. And there we have a very simple idea  of a function. So when we run this, of course,

01:36:31 it should work the way that we expect it to  right, we get one to three, we branch over to our add function, it adds the numbers together,  we end up with four and everyone is happy. Now, my question to you is, what if I  wanted to do something slightly different? What if I wanted to continue execution directly  after this function call? So what I mean by that is that maybe after this function call, I want  to do something different, right? I want to, you know, move some more numbers into registers, you  know, maybe I want to like move into R three, the

01:37:00 value four, right. So if I want to do this, I have  to have some way of coming back to this location after this function is called. Now there are a  few ways that you could achieve this sort of idea, right, like I showed in a previous video, what  we could do is we could branch after this ad and we could branch to this location here. And then  after that location, we can branch beyond the the add two functions, so we don't actually  run back into it. This creates a little bit of complexity. And we don't really like that  complexity, it's a little bit weird to follow.

01:37:35 So in these sorts of situations, where we're just  calling this as a function, what we can do is we can add in what is called a return address. And  with a return address, what we're able to do is we're able to return to the function where we  started. So we're able to at the end of this function returned back to the location directly  after we called that function. So this is the idea of a return. So that's what we're going to take a  look at here to take a look at how we can do this. The first thing that we're going to do is we're  going to change this branch to a B L, which

01:38:07 means a branch links. And what that's going to do  is it's going to store the location that follows this branch inside of one of our registers,  which is this one here, the LR register or link register. So without doing anything further,  let me just show you what that looks like. So when we go to this point, here,  B L, you'll see that LR gets 00000 and then C, right? What that location corresponds  to is the address of the instruction that follows the branch. So you can see here at sea, there's  this move instruction, which is right after

01:38:47 our branch. So you can see that what this  link register stores is it stores the location directly after the call or branch to the function.  So in order to make this work, after we finish this here, we should go back to the location  that stored inside of the link register. So in order to do this, we can use a another  instruction known as A B x, this will branch back to the location stored in registers. So in this  case, we can branch back to LR which will branch us back to the link register. So let's pull that  all together. And let's see what that looks like.

01:39:29 When we do this, we move the values into our  registers, we branch into the add function, you can see that it's storing see as our return  location, which is this move instruction right after this branch. You'll see that we add the  values together and our two and then when we branch it takes us back to this move instruction.  So you can see that that actually worked. It brought us back to the location where we started.  This mimics the sort of flow of a typical function right if you are familiar with higher level  languages, when you have a function you have

01:39:58 a return the return takes you back to where you  started. This is the exact same idea. And this is what your higher level languages are doing in  the background when it translates into assembly, this would be the idea of what is  happening with that translation. So this gives you a bit of an intuition behind  how we actually do these branches with returns. Now I'm I'm excluding a little bit of detail  here, there's more things that need to happen to make this work completely functionally, right.  You know, if we have more complex programs,

01:40:29 there's certain safeguards that we need to  take in order to make this week as as good as possible. However, for the time being, you have  a good understanding of branching with the links, how that link register works, and how to return  back afterwards. So that's what we're going to talk about in this video. This gives you the  fundamental ideas of functions and subroutines with returns. And now in the next video, I'm going  to discuss a little bit more detail of how we can say preserve things before we move into  these functions and then be able to

01:40:58 utilize that to our advantage. So that's what  we'll sort of go into next after this video. So one notable idea about functions and  sub routines is that they often have to do calculations inside of them, in order to  get the result that we're looking to get. In order to do these various calculations, we may  need to use different registers in our processor. Now the number of registers that we have an  arm are, of course, limited, we don't have an infinite number of registers to work with. So  because of this, we may need to reuse a register

01:41:31 inside of a function that's already being used  outside of the function. In these sorts of situations, we need to be able to preserve what we  had before we called the function and then be able to restore it once we're done with the function.  You can think of this concept similar to local variables in the function, when we declare a local  variable in the function, its scope is to live inside of the function. Once we read the function,  we no longer need that variable. So the variable is just de allocated and a garbage collector  will come and you know, remove the memory and

01:42:03 all of that fun stuff. So what we're doing here is  we're emulating that sort of idea, we're going to allocate some registers in our main sort of aspect  of our program, we're going to call our function, we're going to use the same registers, then we're  going to restore the values back to the main, this is done in the background in high  low level applications, typically, the compiler is responsible for setting up the sort of  register allocations. So this is something that we would see in sort of the low level assembly  that comes out of compiling of programs. So

01:42:32 it's something that is done by your high level  languages without you really knowing about it. So let's take a look at the details of this. So  suppose that we have a function where, you know, some stuff was happening beforehand, and then  we call a function, and then we continue on as normal afterwards. So maybe some of the stuff  that was happening beforehand is using the registers R zero and R one, maybe they have  some values inside of them. So let's say, you know, R zero has the value of one.  Let's say r1 has the value of three.

01:43:03 Now what's going to happen is we're going to do  a branch for a function, I'm just gonna call a function get value. And then we'll go ahead and  create that function. So we'll say get value. And what this function is going to do is,  it's going to need to use our zero at r1, we'll just pretend all the registers are full  right now. And R zero and r1 are the only ones that we can really work with. So let's suppose  that we're going to use those two. So we're going to say I'm not going to move some values into  R zero, we're going to do some values into r1.

01:43:34 And then we're going to do some sort of  calculation. So I'll just add the two together, perhaps and store the result in R two. And then  we'll return back to the link register. And we're done. So in this situation, you can see quite  clearly that when we move into this function, R zero and R one are overridden, right. So  if I were to compile and load this right now, you know, we would have one in three,  then we branch we get five and seven, we do our calculation. But when we come back  to the beginning of our function, we no longer

01:44:06 have these results available to us, right, we  no longer have the R zero and R one values. So to be able to help resolve that what we're  going to do is something interesting here, what we're going to do is we're going to push  the values of r zero and R one onto the stack memory. So we're gonna do this or is a push R zero  R one. What this is going to do is going to place the values of r zero and R one onto our stack,  right? So it's gonna place them onto the stack. And then what's going to happen is once we're  done doing everything related to this function,

01:44:42 we're gonna go ahead and pop them off the  stack like this, or does it pop R zero, and R one. And then what I'm gonna do is I'm  just gonna I'm just gonna branch to like an end label so we can just sort of like get somewhere  that isn't going to have, you know, the function interfering with it because otherwise we just move  right into the function again. And that would be the end of this. So what's happening here is when  we move the values into R zero and R one, we then push them on to the stock of memory. So we'll  get pushed over here onto stock memory somewhere.

01:45:12 And then what's going to happen after that is  we're going to call this function, we overwrite R zero and R one, we do our calculation,  when we return back to the link register, we come back here. And then what we do is we pop  the values off of the stack back into our zero at r1. So it's going to pop those values and place  them back where they started. This is preserving the values of r zero and R one on the stack, and  then placing them back when they're needed again. So you can see that the result of this is that the  values for r zero and R one don't get overwritten,

01:45:41 they get used locally in the function,  but we don't overwrite them permanently. So let's run this and see what happens. So you  can see that we get one in three, two, R zero and R one, and then we push them onto this doc, if  I come over to the stack memory, here, you can see we have three and one. So when we're thinking  about how popping works, just to just to sort of clarify why we're doing this pup, the way that  we are, what's gonna happen is we're gonna pop the first thing into the first register. And then  the second thing into the second register, right,

01:46:11 since the stack pointer currently points  at this, you know, FF FF, eight address, what happens is, it's going to take the value  off of that address, which in this case is one, it's going to pop it on to the first register,  which is our zero, which is this one here. And then we're going to pop three, onto r1,  which is this one here, right. So you can see that it just pops one after the other like that,  starting at the top and moving down to the bottom. So hopefully helps you understand a little bit  about the ordering of these two operations.

01:46:41 Now when we do our branch, you'll see that  we come into here, we get five in our zero, we get seven and r1. We do some calculations with  them. And then we branch back to the beginning, where the branch back to the beginning, you see  that we hit this pop instruction. Now watch what happens, right, the stack pointer currently points  at the location of one and three, right here. And here. When we go ahead and run this operation,  what you'll see is that our zero is now one and our one is three, you can see that it actually did  preserve those values, right. So it actually did

01:47:11 place them back into their respective locations,  this shows that we can actually preserve these values, so they can be used in the function  and then sent back to their original values after the function, right. And then of course,  we have our branch that just takes us to the end. So hopefully, this helps you understand the  idea of how we can preserve values in our functions. And we can use these sorts of  ideas in a lot of different ways, right? One other way that we can use this is  actually with return values, right?

01:47:40 Suppose that our two was also being used, what  we could do is we could do something like this, we could say, okay, so push our two onto the  stack. And then we come back here, we pop them, we'd currently be pointing at our two. So what we  can do is we can place it in some unused register, right? So if I want to get that return address  value, I could say, Okay, let's place it into an unused register. And maybe, you know, our nine  is unused for instance, right. So that would be able to place us at our nine, instead of having to  be you know, somewhere else, right? So instead of

01:48:10 having to keep it in our two, we can place it on  the stack. And then we can retrieve it from the stack afterwards, this would be a more traditional  way of returning values, although I think often values are just returned on the into register  itself. So we don't really have to worry about this sort of idea. Because the thing is, with  values being returned, if you return a value, odds are you're going to be using it at some point  in the code, right at some point soon in the code usually. So what happens is we place it into an  unused register, and then we just use it right.

01:48:40 So that's the typical case with these, with these  returns, you know, otherwise, you'd have to pop it off and place it in into its position immediately.  So you need it in an empty register regardless. So that's why we don't usually do that, I just wanted  to show you that just in case you ever see it, it is something that you can't do, I think it's not  usually the case that you would end up doing that. So with this, you now have a full picture  of subroutines and functions in arm, you understand more of how to preserve values,  how to set values onto the stack and remove them

01:49:07 from the stack. And this allows you to be able  to write fully functional sub routines for your programs. Set sort of sums up a lot of the logical  ideas of what we would typically talk about first sort of an introduction to our assembly type  course, for the next few videos that I put out, I'm going to be showing you the ideas of working  with hardware with the armet processor, which is actually going to be pretty straightforward  because we actually have some hardware devices here on the right hand side. So I'm gonna show  you how to interact with those hardware devices

01:49:35 get you familiar with hardware, and then  we can start to take a look at some other concepts maybe at a little bit of a higher  level and see how we can interact with our many other system related processes. So that's  sort of where we'll take the next few videos. In this video, we're going to take a look at  how we can interact with a variety of different hardware devices using our ARM processor. On  the right here we have a variety of different devices is going to note right now you need  to be using the arm D one S O C emulator to

01:50:06 have these devices for using just the arm  one general emulator, you won't have these devices available to you. So you have to use  the D one SOC emulator. So with this emulator, we have a number of different hardware devices  that are available for us to be able to interact with. And I'm going to show you how to interact  with a few of these different pieces of hardware. And once you have an idea of some of the general  devices that we're able to interact with, you'll have a better understanding of how to interact  with the other devices that are available to you.

01:50:35 So primarily, I'm just going to show you how to  work with the LEDs and the switches. And these are really just a rudimentary inputs and outputs, the  switches are an input, and the LEDs are an output. So let me show you how the input  works first, which is the switches, the very first thing that we need to do is we  actually need to declare a variable that's going to store the location of our switch, and later our  LED as well. The way that we do this if we say.eq. And then we put the name of the variables.  So in this case, I'm gonna call mine switch.

01:51:12 And then we put the location of the variable. So  in this case, the location is right here, this FF 200040, right here, so we're going to put that in,  so we'll put it in hex, right, so 0x, FF 200040. Now there's a good reason why we do this, the  reasoning is because you can't directly load something this large into arm directly. So  you can't use an LDR and just load in this value as a constant value, there's a limit the  size of the constant values that are allowed to be loaded in directly. So if your value is larger  than that, you actually have to declare it as

01:51:56 a constant value like I've done here. And then  what you can do is you can load it in like this, you could say LDR. And then I can go into  our zero, for instance, the value of switch. And what you'll see is that when we  do this, we can compile and load. And once we do that, we should be able to see the  actual value Oh, sorry, I forgot a comma here. So don't forget the comma here. So it's  the EQ switch, comma, and then the address. Now everything will compile. And when we do  that, you can see we just get that exact value

01:52:26 loaded into R zero. So all we've really done  is we've declared it here as a constant, and then we've loaded it into our R zero register  to be used later on. So if I want to be able to use this register, what I have to do is I have  to be able to retrieve the value that is located at its memory address. So what's going to  happen is each of these switches is going to correspond with a different number.  And a different number is going to be represented in binary based on the switches  that are on. So this right most switch here

01:52:58 is going to represent like the two to  the power of zero portion of the binary, this one is two to the power of one, this one is  two to the power of two, and so on like that. So that's the way that this is working. If it's on  it's like a one in binary. And if it's off, it's like a zero in binary. So for example, if I check  this off, it would be the same as one, zero and binary, which is two, of course. So let's take a  look at how that works. What we do is we say LDR, I get alluded to r1, the value that's stored  in R zero. So remember, this is the address

01:53:29 of the switches R zero is and we're loading  from the location, right, R zero into r1. So let's Compile and load and see what that does,  you'll see that our zero is currently equal to this memory location. When I step into the next  line of code, you see that we got the value two. And that's because I have the one checked off.  So remember, it's one zero, which means that it's equal to two. And we get to this with all  sorts of different numbers, right? So I could check off like all three of these, right, we would  get 111. Let's just reload it and give that a try.

01:54:01 You see that? That gives us seven, right? Because  it's four plus two plus one, which is seven, of course. So using this, we can actually get inputs  from users, right? So I can actually do things based on the input from these switches, right, so  I just put in whatever value I want in the switch, when it loads that value, then we can manipulate  it as we need. So this is a very simple way to gain inputs from a hardware device. Right? So you  may have been wondering throughout this course, like how do we actually get inputs? This  is one way that we're able to do it.

01:54:33 And then similarly, how do we output  values, we can do that using our LEDs, and the LEDs are going to work in a very similar  way. When we take a look at the right hand side led the rightmost led represents the first  digit of binary and then the second digit, the third digit, the fourth digit, so on and so forth  like that, right? So it's the same sort of idea. So let me show you what happens  with this. So let's say.eq you because I have to declare the location of the LED  right so it's gonna be led ie comma 0x, FF 200000,

01:55:06 like that. So once we have that located, what  we're going to do is we're going to go ahead and load that value into memory like we did before.  But instead of loading data from that location, we're actually going to store  data in that location, right, because we're using it as an output rather  than an input. So we're going to put data into that memory location instead of taking data  from that memory location. So to demonstrate this, let's go ahead and load the data that  we got from the switches into the LEDs.

01:55:37 So in order to do that, we couldn't we're going  to do is this, we're going to LDR. And I'm just gonna go to overwrite our zero because I've  already gotten the data from the switches, so I don't need it anymore. So we'll say from  our zero, we're going to load in the LED address. And then what I'm going to do is I'm going to  go ahead and store the data that's currently in R one into the location of R zero. And the way  I do that, as I say, S T R, R one into R zero. So what this is going to do is, so Well, actually, we  could just run the program and see what it's going

01:56:11 to do, right, so we can step in, you see that  the first line loads location of the switches, and then the second line gets the value from the  switches into register r1. Right. So currently, this value is B, because that's what's represented  with this. It's 1011. Right, so that's the value that's in here, that'd be eight 910 11,  right, which corresponds to be in hex. Now, once we've done this, what we do next is  we load in the location of the LEDs into memory, which is FF 200000. So you could see that those  two match up with each other, or zero and the

01:56:45 LED value here. And then what we do is we store  the value that's currently in our one into the location of our zero. So when I step into this,  you see what happens is the LEDs light up, right, they light up, and based on the values that we  placed inside of register are zero or one rather. So you can see what happened is these are  matching up with each other. Now the switch matches up with the LEDs. The reason being is  because we took the value from the switches and stored it into the LEDs. So you can see that  that actually lights those values up. So we can

01:57:19 actually represent a binary using these LEDs. So  this gives you a very simple explanation towards how we can actually work with inputs and outputs  using various different hardware devices. There's a lot of different hardware devices that exist  here, right, there's push buttons, there's seven segment displays, there's J tags, there's there's  a lot of different ones. And really the best way to learn these different hardware devices is just  a play around with them, right. There's lots of documentation online that can tell you information  about how these hardware devices typically work.

01:57:50 And I encourage you to try each of these devices  and just see what they do understand them and play around with them, they'll be the best way  for you to really learn those hardware devices. I showed you the very basic ways that you  can interact with them. And from here, you should be able to interact with all of these  hardware devices and see what each of them do. Hello, and welcome back. I hope  that you're all doing well. In this video, we're going to start to  take a look at the idea of programming arm

01:58:20 through an actual operating system, in  this case, a Linux operating system. Now the Linux operating system that I'm going to  be using for these videos is called raspian. It's a Raspberry Pi based operating system. And the  reason why I'm going to be using this operating system is because the Raspberry Pi computer runs  an ARM processor. And this actual operating system is developed completely compatible with arm  which makes it a really good choice for us to actually program various different arm things  on it. Now of course, there's lots of different

01:58:51 operating systems that use ARM however, this one  is one that is lightweight, it's easy to emulate, and it will support everything that we need. So  it's a pretty decent choice for our purposes. Now, the reason why we're choosing to emulate  these operating systems is because I'm aware that most people probably don't have an ARM device  lying around. You know, Raspberry Pi isn't too expensive. But of course, it's a little bit much  to ask someone to go out and buy a device to learn arm assembly. So rather than doing that, I'm going  to show you how to emulate it so that you can

01:59:22 just write it on your computer. And then at some  point down the line if you want a physical device, you can go out and get one of those knowing of the  arm stuff that you will know from these videos. So for the purposes of this video, I'm  using VirtualBox I have an Ubuntu machine. It's running Ubuntu 20.04 I believe and it is a  64 bit operating system, you should be able to follow along on any Linux system. The instructions  are actually fairly straightforward. So let's go ahead and jump right into it. What you're going  to want to do is you're gonna want to download

01:59:52 two files onto your computer and I'll put links to  them into the description. The first file is from this downloads that raspberry pi.org work. And the  file is this 20 1704 10 Raspbian Jessie dot zip. This is the operating system image that we are  going to use in order to work with emulating this operating system. There are other versions of  it, there's more recent versions, this one is the one that seems to work most stably with QEMU. So  it's the one that I am going to be utilizing. Now, in addition to this, we're also going to need  a kernel that is working for this operating

02:00:29 system. And we can get that from this GitHub link.  Again, I'll put this into the description as well. This GitHub link is going to give you this  QEMU four point 4.34 hyphen Jessie file, if you go to the root of this, you can actually  see all the different kernels that we have. So we have support for a variety different types of  kernels. This one is the one that corresponds with our operating system, though. So we're  gonna go ahead and download this one. Once you have these files downloaded, I've placed them  inside of this QEMU underscore VMs folder, you can

02:00:57 put it anywhere on your computer doesn't really  matter. And you're going to want to unzip this 20 1704 10 Raspbian Jessie dot zip file, to turn  it into that img file that I have in a directory as well. Once you have this available, I want you  to install QEMU. And you can do that using sudo apt get install Q and you system. So I've already  installed this, it will take a number of minutes on your computer, depending on how fast it is. And  then once it's completed, you will be good to go. Once you have that all set, we're gonna go ahead  and just run our emulator and the way that we

02:01:34 do that as we say QEMU hyphen system hyphen, arm,  and then hyphen kernel. And I'm going to direct it towards the kernel that we downloaded, which is in  QEMU underscore VM slash kernel, this file here, we then want to give it the CPU that we want to  use for the emulation, I'm going to use ARM 1176. Again, this is the one that seems to  work most reliably with this particular distribution. So it's the one that I  will show hyphen M is the amount of RAM or virtual RAM that we want to allocate.  So I'll say 256. That's a good size for this

02:02:08 pipe and capital M is the emulated machine  type we are going to use versatile PP. This one is just like one of the various types  that's included with QEMU. Again, one that works very well for this particular example, we're going  to provide a serial as stdio just told everything to work through our standard input output. And  then I'm going to use the Append command to append the following. We're gonna say root dev  SDA to root Fs type equals e x t for rewrites. This just helps us set up the general file system  that we're emulating. So this gets everything set

02:02:46 up the way that we need it. And I'm going to put  hyphen, HDA. And I'm going to point it towards the ISO that we have on our computer set is in QEMU  underscore VMs slash 2017. This one here to ing. And then finally, what I'm going to do is I'm  going to forward a port from this computer to our SSH port, which is 22 on our machine. The reason  I'm going to do this is because when you work through QEMU, it's a little bit hard to get things  to fullscreen properly. So rather than doing that, what I'm going to do is I'm going to SSH into  the emulator through my virtual machine here.

02:03:25 So to do that, we can just do the following. We do  hyphen, Nic, they're gonna say user come of host, forward equals TCP, and then two colons and 5022  and 22. So 22 is the port that we're forwarding. So that's the SSH port. 5022 is the port that  we're forwarding to. So when we want to connect to ssh on this computer, we're going to use  the port 5022. So that'd be the idea of this. And then we will launch this we're going  to say no reboots, just because I don't want the system to actually reboot I just  want it to boot as is. Once we do this,

02:04:06 it looks like maybe I just spelt this wrong. The  machine type let me just double check it should be vs. vs. Tile. Yes, it's spelt wrong.  So you versatile PB like this. Now once we run this, you should see that your  QEMU starts to boot. So this is booting into the Raspberry Pi operating system. If you've  gotten this far, Congratulations, you now have a Raspberry Pi machine running on your computer.  I will note that it's a little bit tricky to get this working reliably. I am able to get it working  reliably every time that I've tried it so far on

02:04:43 my computer, which is why I'm showing this set  of instructions. If you have any troubles with getting things set up. It should be relatively  easy to troubleshoot the errors that you're getting. But feel free to comment them as well  and I will try to help you as much as I can. I also plan to try to post this image if possible  somewhere so that it's able to be downloaded. So keep an eye out on the description as well, I'm  going to try to put a link to this virtual machine image so that you're able to access it and use it.  Rather than having to do all this setup yourself.

02:05:12 That way you get one that is for certain working  through virtual box. And then from there, you can just sort of fiddle around and get it to work on  your particular system. There's a few things like setting up emulation inside of emulation, like  allowing Virtual Box to run emulators like this, there may be some settings that you need to  tweak in virtual box to get that to work. But regardless, again, it should be relatively  easy to get set up. If not just feel free to leave some comments, I'll try to help you with  troubleshooting some of those errors if possible.

02:05:42 So once we have things set up here, what  I'm going to do is I'm just going to run the following command, I'm going to open up  the terminal on this emulated device. And we're going to do is I'm going to forward SSH on  it. So I'm going to say sudo service, SSH start. Once that runs successfully, we're going to just  try to SSH into this machine through my computer and see what happens. So it looks like everything  is good to go. I'll create a new tab here. Zooming in a little bit. I'm going to ssh pi one to 7.0  point 0.1. And then the port, like I said before

02:06:22 is going to be 5002 or 5022. Right there. When  that goes through ask you if you want to continue, you can say yes, the password for this is Raspbian  R A s or sorry, it's I think it's Raspberry Pi. Okay, well let's let's double check and see  what it is. We want to say Raspbian password. It should be raspberry. So Raspberry should  be the default password. Let's just try that. There we go. So now it has brought us into our  device. And we are now inside of our emulated device. So with this, we now have an ARM device  that is running on a Linux system. And now we're

02:07:13 going to be able to do a lot of really cool and  interesting things with arm we're going to learn about things like system and software interrupts.  And we're going to really just build up a lot of operating system understanding as well, I really  dig deeper into the sort of things that we can do with arm assembly. So thank you for watching this  video. I hope that you enjoyed it. Again, if you have any sort of trouble throughout this process,  please do leave comments, let me know, I'll try to help out as much as possible. And I'm sure some  of the other viewers will be able to help out too.

02:07:41 And then we'll be able to get things up and  running like this. And then we'll be able to learn all these different operating system  techniques. So again, thank you for watching, and I'll see you in the next video. In this video,  we're going to start to take a look at operating system interactions by seeing how we can output  data onto the screen. So we're finally going to do like an actual formal Hello World program  where we output hello world onto the screen. And while we're doing this, we're going to explore  the ideas of system calls and system interrupts

02:08:13 done through arm assembly. So let's go ahead and  get started. For the purposes of these videos, I'm going to be working through our Raspberry Pi.  In this case, I'm actually logged on to a physical machine, but you can use the emulator we set up  in the previous video. Or you can use a physical machine of any kind that has an ARM processor. If  you want to follow along on something physical, the choice is really yours, it should work on  pretty much any ARM based device. So we're gonna start off by creating a new file, which I'm going  to call Hello World. I'm going to do that through

02:08:41 the Nano text editor. And it's just my text editor  of preference you can use whatever you prefer. The extension for this file is dot s dot s is  the typical file extension for assembly based programs. And what we're going to do here is we're  going to set up our program in a very similar way to what we have done in the emulator past. Right,  so let's take a look at how we set that up. We say dot global underscore start, and then we have  our start label. And then for this, we're going to need some data. And the data is going to be  what we actually want to print onto the screen,

02:09:13 I'm going to do that declared as a message. Now  we want this message to be text. There's a lot of different ways that we can declare text in  arm and I want to discuss each of them. Now, the most basic type of text type is ASCII. ASCII  is just general characters. And there's something about the ASCII format that doesn't really work  well for a lot of text applications. And that is the fact that it is not null terminated. And  what that means is basically in some lower level programming languages, if you've programmed in C,  you'll be familiar with this idea. When we create

02:09:47 a string sits it's really just like an array of  characters. We have something that indicates the end of the array of characters because of course,  as we're going into memory in memory is not always predictable what an empty slot of memory will  contain Okay, so what we do is we pick a special character that indicates the end of a string,  and we call that the null Terminator. So it's a special character set aside, that always indicates  the end of a string, the ASCII format does not have that. So this sort of format works best for  things like individual characters, or situations

02:10:18 where we don't need the null Terminator.  In almost every single case, though, you're going to want to have a null Terminator for your  string. So instead, we're going to use this here, ASCII set, I'll call it. And what this does  is it declares a string, but it adds on a null Terminator at the end, which allows us to  know where the end of the string actually is. Now, one other note that I want to make here is,  sometimes you might see dot string, dot string is actually an alias for dot ASCII set. So dot  string is the same as this declaration here.

02:10:53 So I just want to note that just in case you ever  see dot string, it's the same as this dot ASCII set. So those are the same as each other. It's  a null terminated string. And then from here, we could just type in the name or what what text  we want to write. So I could say, Hello World, I could put backslash, and we could put this  new line character. And so all that does is it indicates a new line, backslash n.  And essentially, we can add that in because it will get interpreted through the  operating system to write a new line character.

02:11:21 So that's something that we're able to put in as  well. Now, I will eventually show you what this looks like in memory. But for now, we're just  going to sort of take this for granted and say, Okay, this will work. And we will continue on  with this. Now, one other thing that I want to note is that when you when you make the call to  write data to the screen, in this case, which is what we're going to be doing, we're actually using  components of the Linux operating system, which is really interesting sort of idea. So you'll see  that from this point on, when we start using

02:11:51 the system interrupts and system calls, there's  gonna be a lot of parallels between assembly and C. So if you're familiar with C, it will be very  helpful for you. If you're not familiar with C, not to worry, we will eventually discuss C  and its concepts in future videos as well. So once you've declared the text here,  what we could do is we could declared the length of the text. So I could  say le N equals dot minus message. What this is going to do is it's going to declare  a variable called Le n, as gonna be equal to the

02:12:23 length of message This is the way that we declare  that the way that this works is it starts at the beginning of the text. So it starts at this each  character here, and it continues moving through it until it reaches that null Terminator that I  mentioned previously. Once it reaches that null Terminator, it knows it's reached the end of  the string, that is the length and it stores the length in this LGN variable for us to use,  send us all the data that we're going to need to create our Hello World program. Now in order for  hello world to work, in order for us to write data

02:12:53 to this screen, we're going to need to declare and  have ready three different pieces of information. The first piece of information is where do we  want to write the message or text or data to the second thing that we need is a reference to  tell us what what we want to actually output. And then the third thing we want is the length,  how long is the thing that we want to output, and those are going to be stored in very  specific locations. So we're going to move into our zero the value one, this one indicates that  we want to place that we want to place the output

02:13:30 in the standard output for our computer. So this  would be the standard output, which is where all things output by default, in the case of me being  SSH into my actual computer, the standard output would typically be the command line interface. So  I'd be without output. So this is just generally the way that we output things into the standard  output, we can output the standard in by using zero as our argument, we can output your  standard error using two as our inputs. Now in general, we can also add in  different values other than 01. And two,

02:14:06 this will actually take in any valid file  descriptor as a an argument. So we can actually write to files as well. For those of you who are  not as familiar with the Linux operating system in the background, what Linux does is it assigns  things called file descriptors to the files on the system. What happens is when you open up a file,  we are given a unique identifier for that file, which is typically represented I believe, as an  integer. And that allows us to know where we're reading and writing from. So whenever we make  calls to read and write to files, we use this

02:14:36 file descriptor which is an integer which uniquely  identifies the file. Because this R zero is taking in some integer value, we can actually provide it  with any file descriptor of a file on our system and it will actually write to that file set  something that's interesting about this. Now, like I said before, we also have to give it the  value that we want to print as well as the length. So we could do that using LDR. So I can LDR into  r1 Send the message. So that's going to load the address of message like we've discussed previously  into r1. So I will tell it where a message is

02:15:08 currently located in memory. So it's kind of like  a buffer, right, it tells it the buffer location of the message. And then what we could do is  we can LDR into our to the value of length. This point says the length into register two.  So now we know the length of the actual string itself, we know the actual string that we want  to print, and we know where we want to print it, the last thing we need to do is tell the computer  to print the thing to the screen. Now the way that we do this is actually quite interesting.  What I'm going to do is I'm going to move into

02:15:38 our seventh the value for Now that might seem  a little bit random. What's happening here is, when we communicate with the operating system, our  seven is a special register, that actually keeps track of what we want the operating system to  do. So when we do an interrupt, which will allow the operating system to take over the processing,  what the operating system is going to do is it's first going to look at register RS seven, and  it's gonna check what number is written there, that's going to tell the operating system what  it needs to do next, when I give it the value

02:16:11 for what the operating system CS is, it sees that  I want to write something to the screen, it's then going to take a look at R zero, R one and R two to  answer the questions of where to write the data, what data needs to be written, and what the length  of that data is, it will then write out the data, and then it will return back to the program  to allow it to continue executing. That's the general flow of what happens when we do this sort  of interrupts. If we want to go into a little bit more detail, what's actually happening here is  in Linux, there's a system called called rights.

02:16:41 What this is doing is it's actually calling  that write function. And again, if you have familiarity with C programming, and specifically  like system level C programming, you are likely familiar with the right function, you use that  actually to write files on a Linux system, that's the same function that's being called  here, we're just calling it through a system call in a little bit more of an abstract  way. So I just want to bring out that parallel, just in case you're  familiar with C programming, because

02:17:08 it's a really valuable one to keep in mind that we  are actually able to communicate with the system calls in this way. So we set up the value that is  needed in order to know which system call to make. Now let's make the system call, which we do  using SW I zero SW I mean software interrupts so what it does is it interrupts and it tells the  operating system Hey, we need you to do something. The operating system helps us handle  hardware things like input an output, so it is able to do the output for us. Again, it  takes a look at our seven it sees the value four,

02:17:39 it says that's an output it then outputs  based on r one r zero R one and R two values. Once that is done, it returns back to our  program and lets our program continue to execute. At this point, what's going to happen is we're  going to flow into the data and weird things are going to happen. In order to prevent that we  have to properly tell our program to exit which hasn't been something that we've really had to do  throughout the emulation type videos. To do that, what we do is we say move into our seven  the value one and then we do an SW AI. So

02:18:12 what's happening here is again, we're telling  the operating system, what we need it to do, the argument of one is telling the operating  system to terminate our program. So this time when we do our software interrupts, the  operating system will interrupt it will take a look at our seven it sees one and it says oh,  that means I should terminate the program, and it will terminate it and we will be done. So this is  everything that's involved in creating our simple Hello World program. You can see it's it's not  necessarily all that simple. But now that you have

02:18:41 a familiarity with a lot of the general concepts  of assembly, hopefully this does actually seem a little bit easy to actually do right. So a lot of  these instructions are already familiar. There's a few new ones a few new concepts, but this  should be relatively straightforward, hopefully. Now, last thing that I need to show you here is  how to actually run your program. We're gonna do two steps, we're gonna first assemble the program.  So we're gonna do as HelloWorld dot s, hyphen O, hello world dot O. This is creating  an intermediate form file known as an

02:19:15 object file. The object file can then be combined  with other object files if required and compiled into what is called a binary. The binary is what  is actually run on the system to execute the code. So to create the binary we do LD hello world  dot o hyphen o hello world. So you can see right now we have three files you have Hello  World, which is the binary I just created hello world dot O which is the object file. And  then hello world dot s which is our assembly code. Now we want to execute Hello  World v2 dot slash hello world.

02:19:51 And as you can see, hello world is printed to the  screen so it looks like everything is working as expected. So from this video, you should now have  a better understanding Think of how system calls work and how we're actually able to output data  onto the screen. So thank you for watching this video. In the next video, I'm actually going to  walk through gdb to show you how you can get a better understanding of the actual layout of  memory on this Linux system. Because you can't really see a lot of the details just running it,  I want to show you how to debug and understand the

02:20:20 code that you're running with a little bit better  than what we have now. So thanks for watching, and I'll see you in the next video. In this  video, we're going to take a look at how we can actually debug our assembly programs through  the Linux command line. And what I want to do is I want to show you that a lot of the things that we  saw in our online emulator can also be replicated through the actual Linux command line using  programs like gdb. And that's primarily what we're going to focus on, we're going to take  a look at the same program that we had before

02:20:52 that was this, this Hello World program that was  able to print out Hello World to the screen, I gonna show you how we can step through it and see  all of the different pieces that are required in order to make this work. So first off, we're going  to launch this program into debug mode using GDB, which is a debugger that comes typically default  on most Linux distributions. And it's a really valuable debugger for assembly as well to other  programs like C you'd often see it used with. So to do this, we just do gdb. And then we type in  the name of our program, which is hello world. And

02:21:26 you'll see that we'll get up this screen that will  say all this information about gdb. And then it's been launched successfully. So in order to get  this running, what we have to do is we have to add in a breakpoint, what a breakpoint is going to do  is it's going to stop the execution of our program at a point specified by us, allowing us to be  able to see what the current state of the memory is at that point and step through to the next  instruction and continue stepping one by one, the typical thing that we break on is we break  on the Start label. So we break underscore start.

02:21:59 So what this will do is when we run our program,  it will stop at that start label and allow us to start executing step by step from there. So once  we have our breakpoint set, we could just type in run, and you'll see that it started the program.  And it's hit the breakpoint at start. But you can't really see any of this information right  now. So what we have to do is we have to change the layout of GDB. So we could say layout, ASM.  And you can see that this gives us a layout that shows all of the instructions in our application.  So you can see the first instruction which is

02:22:33 moving one into our zero. And then we have the  loading of our strings and our length. And then we have our system call. And then we have our  end of our program with the system call for that. So this is giving us an idea of how we can  actually look at the instructions that are associated with our program. Now the next  question is, how do we see the registers, there's two different ways that you could  typically see the registers. The first is, you could say, info, register, and then type in  the register that you want info on. So I could

02:23:03 type in, for instance, r 00. Sorry, I had to do  it as a lowercase r. So it'd be info register, R zero. And you could see that it has a value of  zero. So there's there's nothing currently in it, it's set to zero currently. So that's  one way that we could see the register. Another way that we could see the registers  we can actually type in layouts are EGS. And you can see that that will show all of  the registers above our instructions. Now what you could do is you could type in control x,  and then Oh. And that will allow you to switch

02:23:39 between the instructions and the registers. So you  could see now I can scroll with the arrow keys. And I could see each of the different registers  that exists, you can see like the stack pointer, the link register the program counter the CPSR  registers, you can see all the information that we would typically have when we were running through  the emulator. Again, that's CTRL X and then type O after doing that, and that will let you switch  your interface between the layout ASM and the layout of registers. So you can have both of them  going and switch between them with Ctrl X Oh.

02:24:12 Now from here, we can step to the next  instruction using step i. So I type in step I and you could see that it highlights the next  line. So it executed the first line and moves on to the next one. And we can confirm that  because if I look up at my register view, I see that our zero has been set to one, which  is what happens when we do this move instruction, right we move our one into our zero  and we get to the actually did happen. What step by again. And you could see  what's happened now is it's loaded

02:24:42 our string into r1. And you could see that it  has the address of the string now stored in r1. Now the next thing I'm going to talk about here is  well how do we actually see the string in memory? So how do we actually see the stuck memory? The  way that we do this is we do the following we do X This slosh. So basically what x is doing is  it's saying examine, so we're gonna examine a part of memory. And now what we have to do is we have  to tell gdb. How many pieces of memory do we want to analyze? And what format do we want to see it  in. So for example, you could say 10x, what that

02:25:19 would do is it would show us 10 hexadecimal memory  slots. So what's going to happen is we specify a starting point. So I would say address. So say I  want to see the address of r1. Oh, put dollar sign r1, what's going to happen is, it's going to  start at the address of r1, which is 20098. And it's going to show me 10 hexadecimals, starting  at that point, so it starts at 20098. And then it goes to the next memory slot, and then the next  one until it showed me 10 hexadecimal values. You can see when I press Enter, that's  exactly what happens. Right? So we see

02:25:55 the 20098, right, this is the first slot, and  then the second slot, and then the third, fourth, fifth, all the way up to 10. Right. So you can  see that that generally shows us the hexadecimal values. Now looking at these values here, it's not  necessarily clear what each of these represents, right, we can see that each of these is a  single piece of our stack memory. You know, this really parallels the same ideas as our  emulator, right? Nothing's really changed, they're still the same number of like  hexadecimal values per sort of memory,

02:26:27 we're still working with 32 bits for every  single piece of memory here, right? So you can see that we have 1032 bit slots, that's  what's really happened here. Now, of course, looking at the hexadecimal numbers is a little  bit confusing. So what you can actually do is you could specify different formats, I give you these  in decimal by changing the x to a D, like that's, you can see that it gives us decimal  values. Again, still not entirely clear. So let's do something different. We can actually  specify character using 10 C. So C is character,

02:27:00 D is decimal, just to give you a few more, here  you is unsigned decimal. So that sort of gives you all the same ones that you would have had in the  emulator. write x for hexadecimal D for decimal, you for unsigned decimal, but then we also  have to see for character, which is nice. When I do this one, what you'll see is our  string, right? Hello, world, I don't quite have enough numbers here. Let me put in 15, should be  enough. Yeah, you see, there we go. We have hello, world, backslash n. And then you see that  backslash 000. Here, let me highlight it here.

02:27:35 This right here is the null Terminator that I was  talking about. It's the special character that indicates the end of the string, set something  interesting to sort of point out here, that's the null Terminator for our string. So you could  see that this is each of the different values that exists inside of our memory, right, you can  actually see that each of these does represent the characters that we were expecting, right,  we have the Hello World string with the null Terminator, and the backslash n for the new line,  all stored in stock memory. So this allows you to

02:28:05 actually see what's being stored in that memory.  So you could see that you have a lot of the same troubleshooting techniques as we had when we are  working in the actual emulator itself. And again, we can continue through this, we can continue  stepping through as much as we'd like, you could see that we have generally the length of  the string being stored and that we do our system calls and we move on as it's right. If you want  to just like complete the execution, you can just type in run again. And what will happen is, we'll  actually run will take us back to the beginning,

02:28:35 we can remove the breakpoint, for instance, and  then run it again. But generally, the way they interact with the application through gdb would  be stepping through step by step to sort of debug problems. And then the way that you would run it  is what we saw in the previous video where we did that dot slash, you know, and then running  the program as is. So this should hopefully give you a bit of a better understanding of  how we can actually debugger applications. And you're going to see this a fair bit. Whenever  we're working in assembly, I'm going to be using

02:29:03 this debugger to sort of show you the general idea  of what is happening when we're going through each of these different memory locations. So not to  worry, you'll get a lot of practice with this and I recommend just try and get out with a few  of the other programs that we've done so far. Try writing them in this Linux format, and  then try debugging them and just see what happens. You know, it will help you a lot  with understanding the various components of GDB so thank you so much for watching this  video and I will see you in the next one.