Okay, let's start. Oh, there's a lot Good well, you are the brave showing up this morning Questions from your site nothing then then Yes, I should make the following remark ah interrupt sorry one question about the Store good.

Yeah, so I'm sure it does But the question is in such cases always Try and look at the semantics And that is on page 77 in this particular case.

Well that describes the effect of an instruction And it also explicitly says which flags are being set.

Yeah Well, what you have to do is to essentially Well, it will float a fire.

I guess the a bus To Sigma W probably the b bus I'll have to see the picture in front of me but you also have to activate The other such that the flags are being set Yeah, yeah.

Yeah, yeah. Yeah, so well they are already on the bus. So basically you have to tell the other pass the content of the I'm still in doubt whether the a bus or the b bus, but I have to check that explicitly Set it on pass through so the result will be on the seabus, but we don't use that And then you explicitly set the flags and Strictly spoken so I'm not completely happening happy with what is he saying?

I'm not completely happening happy with what is he saying? So basically all the flags are being set So also the fee and the set flag And the see well the see and The carry flag and overflow flag because we do not have a way to set a Particular subset of flags and I think that if there would be a next edition of this book We should add the see and overflow flag there also being always set to zero because there is no carry and overflow but the most important point for you is It also passes halfway well through the alley and we set the flags Okay, I think oh I had to make a remark. I Also have to wake up again The remark is that next week Tuesday you have some festivities.

I have no clue which festivities that are But I got the message that there will not be a lecture. So next week Tuesday There is no lecture the next lecture will be next week Thursday morning Okay, and Let's start with the ARM processor, so you basically are about to finish the part of the practicals where you make your registered transfer instructions for the processor and exercise with the processor and know how the processor is made and then the next part of the practicals are to play around with assembler just To have done that once in your life and as already said on Tuesday We had this very nicely designed PP2 processor, which was relatively easy to program but but You complained a lot at least your predecessors complaint a lot that the PP2 processor was our local processor not available widely, so we shifted to the arm microprocessor and Well at the end of the explanation, I cannot explain everything you may Form your own opinion about whether that is progress or not if it comes to programming anyhow the ARM processor is given So let's try and understand it So this is basically the Raspberry Pi model for that you will get It's a very nice Full-fledged machine actually it circle it could immediately replace this machine except that it is a little bit less capable But this is anyhow overkill So for the for most of the most tasks and it essentially Has inputs and outputs it has has an HDMI port.

I think this is the HDMI port It has some USB ports such that we can connect it to work a keyboard And there is an ARM processor somewhere and I do not even know which of these is the ARM processor There are a few components. I would guess this is the ARM processor, but I'm not completely sure Anyhow by and large that does not matter. We are just programming it and a big advantage is is that Such an ARM processor comes essentially with all it needs so all the input and output parts And components are just on the same chip So this such that you do not need complete chipsets and you can make such boards at a rather low cost what was the history of the ARM processor because that's Well It's quite a remarkable history actually so it started with the Ekorin atom in 1980, so I went through the history of computers very quickly and we saw that we had microprocessors At the end of during the 70s at the end of the 70s microprocessors are widely available at relatively low cost And you had whole ranges of them and then there were quite a large number of manufacturers that took a microprocessor Made a very primitive operating system. Well, you couldn't hardly call it operating systems But a bit a very basic system to start of the computer to show you a prompt and Often these things where had a basic interpreter in them such that you could write programs And you could essentially load often from tape So you had your tape recorder and you would get One had to wind the tape recorder the tape to the right place and then you loaded a program and in that way a program can be run on such computers one of these computers and as I said, there were dozens of them One of these computers was the Ekorin atom and I think there was nothing particular about it. It was a reasonably For its time well-designed Microcomputer, of course in no way comparable to the mainframes like the IBM 360 or the Fox machines that That big companies were using which were had proper operating systems where multi multiple people would basically Access the machine and render program simultaneously.

That was already common at these days, but for 8,000 guilders you could buy Such a machine 1,500 guilders now the BBC The soda British Broadcasting Company so that Set well we see all these computers Available so it would be nice to have a special course for all people in Britain such that you they could understand what a What a microprocessor what what what a computer would be and how you would use it to program so what they did do is they made a contest and asked all the suppliers can you provide us with a design for a Dedicated computer that can belong to this course this machine actually at the price of 2,000 guilders.

It was quite affordable and expensive And then you still had to load your programs with a tape recorder. So you had also to buy a tape recorder The be this computer the BBC computer which is basically an extension of the Echorin one and this made the BBC computer Relatively popular so the company acorn making this Became successful because success really depends on being able to sell I've seen a number of people starting company and having a very nice idea how to how they could program something that they Liked and they never thought about selling and all these companies never became a success Being able to sell is the crucial thing sometimes you have you can make something and then then then You have customers and the customers do not understand what you can make but you can also make something else So I actually saw a successful company coming into existence is of course completely irrelevant story Where people could completely? Redesign cobalt programs Make them from old cobalt to fresh cobalt automatically, which was really a big advantage for For banks or they said well, we parse the old old cobalt programs and he have any bankers all sitting Most of them old people and they did not understand anything of parsing. They never heard of what had ever heard of the word it was it was even a younger people person and they said we can then transform it with rewrite and transformation rules and all do it automatically and they all said Why interesting interesting interesting and they did not plan to buy this because they had no clue what it could do and they did Therefore they did not had a deep twist and then they do not buy it And then all of a sudden one of these persons younger person says oh interesting But can you also count the number of lines of the codes and then people here the technical people? You should not behave like that said Well, that is so simple so straightforward to count we can parse it and transform it and do things that is really of value to you But then there was a manager who he saved the company and said, oh You would like to count the number of lines? That is really difficult. That will cost you a little bit more and then they said well Well, but counting the number of lines is very then we pay a little bit more and seed Company started up and started to flourish those technical people actually left the company because they were so disappointed but this was Selling well for which they wanted to pay leads to success.

Anyhow, why I'm telling this I do not know The BBC computer got sold in large number so the company became a really established company and People had some money and with money you get ambitions And so they said we would like to really get a position in the market And therefore we need to conquer the business.

So we want to make an acorn business computer And The 6502 microprocessor that was part there was an 8-bit processor that was in the BBC computer It was a little bit too slow so they did not want to use it and There was a new processor from multirola and 80 80 the 68 thousands Which was actually used at that time in the max But they found the interrupts of that system too slow and because the interrupts were too slow they said well we now have the money and the capacity so we hire a number of extra people and we design our machine and Well, as you know at that time there was research about complex instruction set computers versus reduced instruction set computers and There was a common common understanding that you wouldn't have a nicely designed reduced instruction set computer because that would ultimately be faster They designed a risk machine and they even renamed their company into advanced risk machines And then they tried to design The ABC computer, but it was very very ambitious.

So what happened was this The machine never came to existence I Do not know of I've never seen a a Picture of the machine.

There was lots of talk about it, but it's simply never bore fruit, so They did do and designed a Other computer because they wanted to have the successor of the BBC, which was far more capable But this never became a success why not let's just ask the person at the left top Why did this computer never became a success? Yes, you are looking down so intensely you need to Have an interrupt also.

Sorry. Oh Yeah, this is always a good time to do that Well go back back.

Yeah, that's so so that was part of the Answer so there were not sufficient programs for it, but go back to the history of of the computers 1987 what had happened there and what was really now in full swing and important development with computers Well, not something huge IBM released something small namely the IBM compatible machine Which other people copied by cop taking the beos in these machines? Redesigning the beos because they could not copy the one of IBM, but they could design equivalent beuses by the same set of Microprocessors and have a completely compatible machine that could run the software of this IBM Standard machine and all of a sudden you had text editors spreadsheets Games and more games and more games available for that machine.

So there was a family of machines relatively cheap with lots of software and against that the IB this is the Archimedes had to compete and that was quite impossible people rather would buy a IBM compatible because it had more value for money and basically the whole arm company More or less disappeared from from few at least I did not realize that they existed until We got a phone with an arm processor in it, and I looked up the history again So on the IBM compatibles and max s and and Intel and Linux machines basically survived So they had to survive in one way or another they had their arm processor being designed and They did not have computers to sell so arm did Do something rather remarkable? They basically said we license out our processor such that other companies that want to make a processor Can buy our design and they make the processor with our design So you have all kinds of arm like processors some extended with with extra inputs and outputs for those companies that would like to Have a processor with extra input and output others with K-passes to handle more memory all these things are being made by different companies than arm arm only licenses this This processor and that is quite remarkable in a sense that that they now have become quite big and quite impressive But they're still not making processors. They are only designing them

Okay, so This is essentially what is still happening so It was the BBC again that said but now somewhere in early 2000 that it would also be nice to have computers available that could control some hardware and and and have some kind of offset Say cheap board and at that time people designed in Manchester the Raspberry Pi as a cheap essentially one chip board And this single chip contained all that was relevant and that was also used for for education This grew a little bit at some point. You had a 32 bits arm processor now You also have the 64 bits on processor.

We will concentrate on the 32 bits on processor the 64 bit on processor is Not a little bit different than the 32 processor.

So it's not the case that it is very compatible they are quite well different machines with different bits a Similar structure, but there are quite a lot of instructions on the 32 bits machine that are not available on 64 And there are a few structures on the 32 that you will not find on the 64 etc, but we concentrate on the arm 64 bit which is in this Raspberry Pi model for that I showed you earlier Okay. Now, how does this look like? I think I show you this picture you immediately Understand what the essence and the core of this 32 bit arm processor is Namely we have registers Now we have 16 registers.

All registers are 32 bits All instructions are also 32 bits.

So it is very nicely designed in that sense You have an early and you have a status words. This is a 32 bit status words But if I show you this you will immediately recognize most most of it if you look at registers Register 15 and 14 have a special status This one is the program counter we use this day we used to call this the instruction pointer in the simple data path But now the program kind of counter is a member of the normal set of registers and this has some implications namely That you can Move the content of the program counter to another register. If you would like to do that, it's easy on this arm processor and This is actually being done at times if you would like to address some data relative to the program counter with these indexed instructions that is also possible and There are also good use of that There is also a link register So we have the branch to subroutine instruction where on the simple data path and on most Processors but not on the arm if you branch to a subroutine the return address is put on the stack That is not What the arm is doing if you branch to a subroutine Your only option is that you can optionally put the return address in the link register And if you want to have it on the stack, you have to do that yourself It costs me quite some time to get used to the idea because in my mind Branches of protein would always put something on the stack. The arm does it slightly differently. It leaves it up To the user to move things to a stack this speeds up matters Why? Because if the if you just call a procedure and the only thing is that you go back and this procedure does not call Something else you do not have to save the link register in memory You save it in a register and saving something in a register is much faster than moving it to memory even with the existence of caches Okay There is a stack pointer according to Handbooks the stack pointer is register 13, but that is not part of The arm processor per se it does not know that register 13 is Is the stack pointer you could also use any of the other registers as the stack pointer at will And then there is R12 Well, what we will see is that if you have four gigabyte of memory You can only make small jumps still jumps of many megabytes But not too many jumps the jumps cannot extend the whole memory because all our instructions had to fit is 32 bits boundary and and therefore Programs are use R12 Compilers use R12 if you would have larger jumps So the rule is that by and large we do not touch R12 because fishy things can be done with R12 with the rest By the rest of the computer, but this still leaves us with quite Quite a lot of other registers R9 is often used as the base register moving to pointing to memory Anyhow large number of registers, which is quite quite pleasant Okay, let's go to the status words.

Okay, let's go to the status words. The status word is still 32 bits long and it looks like this At the can I do this?

At the can I do this? Oh, yeah Here You have the four say I like like to look here Here you have the four status So they are well known by now And and fortunately they are exactly the same as what we know in a simple processor we also have a few bits for the interrupts So you have the interrupts Disable flag and the fast interrupt disable flag So if you don't want to get interrupts at a particular moment You have to take care that these bits are set to zero You even have a flag for the thump mode and the thump mode Allows you to write shorter instructions. We will not look at this thump mode, but if you have only Limited amount of memory you may want to use these thump instructions because then your programs are a little bit shorter I think first that's of no relevance at all and Something and I will speak about that later is the mode And the mode is very important to get security in the system.

So all modern computers have different modes One is the user mode in which you program But as a user you can only access a little bit of memory that operating system assigned to you You cannot access the input and output ports or write something to the screen This was very nicely possible on the pp2 processor, but not here Because the operating system simply does not allow you to do that the only party that can access discs inputs outputs other pieces of memory is basically the system in supervisor mode You if you are in interrupt mode, you can do a little bit more but let's see how these modes are being used to Actually take care of safety of the system.

Okay, let's look at the instruction set So I will come back to these modes This is the instruction set The overview of the instruction set and we can look at the instructions Well, we can see that they are all 32 bit long So for instance, we have a branch instruction here Well all the instructions have these condition flags.

These are four bits Now it's an indication that this is the branch instruction You can use the link register or not use the link register and then you have I believe that these are 32 minus 8 24 24 bits That you can use to actually make your jumps and that is not enough to reach all addresses in the whole system So we have a limited offset if you want to have values here then actually here we have only 12 bits to store Actual values in the instruction, which is not too much So this will have some some consequences.

So let's first look at these condition codes, so all instructions have this field have a condition and this condition essentially says that the instruction is being executed if the condition encoded in these Four bits are valid So what are these bits? These bits basically are given here So for instance 110 is always this is the default so if these bits are set to 1110 the instruction whatever instruction you have here is always being executed but if you have 000 here the instruction is only executed if Z is Zero flag is being set and if that is 0 0 0 1 it's only executed if that is clear And here you have all the other conditions Under which that is being executed now, how do you use that in an instruction? Well, basically you have the instruction add R0 R1.

You can also say add always So these always is by default being added here and then the instruction is always executed independent of how the flags are set But if you want to only execute it when the carry is set then you take this add and this holds for any Instruction and you add this CS to it and then this is only executed if the carry is being set So you can execute all the instructions depending on how the flags are being set Now something that you already know, but I'll stress it again.

Is that If you have unsigned values then the outcomes of comparisons are different Then when you have signed value So if these are the bit patterns as four bits Then if you then then you will you apply smaller them on four and thirteen then This will actually this is actually true But if you would apply smaller than on plus four and minus three if these are two's complement numbers Then all of a sudden it should yield false. So There is this whole Table of Extensions that you apply in case of unsigned comparisons so for equality it was equal and For signed comparisons you have especially for for these Compare the comparison of two values you have to apply different conditions So typically if you go back to this slide here, you will find in this table all the possible settings Conditions such that you can apply both unsigned and signed Evaluations and interpretations of a sub of of a result Okay, most important thing here is We can Execute any instruction based on these flags and that makes it actually much easier for the processor to execute these instructions.

Why? Well Let's let's let's look at this piece of code here So if our zero are zero represent some variable and which that is register zero and if that is equal to zero Then I want to increment register one by one Now in our simple data path and on an 80 86 processor on many other processors you typically have to do this Well, you set your flex then if it is not equal we jump over the increment instruction and After that and if it's equal we actually do the add instruction now Imagine what a processor has to do it eats up all these instructions Especially it wants to eat multiple instructions at the same time then all of a sudden it encounters a branch so it has to look at the instruction and and do a little bit of branch prediction and and predict whether it will take This route or that route and that becomes very nasty and and and computationally intensive and with the year with the Arm we simply execute this in such as how the flags are being set According to our zero and we do just an ad Depending on whether the flags indicate that the result was of The result in our zero was zero and then we actually Carry out whatever we have to carry out namely increment are one by one and this is the way to write it And and this is not only one instruction as this text is saying below It's not only shorter, but it's also easier to execute and faster to execute Okay, next question that we could ask is when do we set the flags? And also here they do it in a somewhat special way, which is also handy But it puts a little bit more burden on the programmer.

What it does is In all of not all instruction, but in most instruction you have this s field And actually what the test field is saying is that when that is set? The conditions are being set and if that is not set and that is the default You do not influence the condition codes so Typically if you have an instruction you can add the letter s to it And if you add this letter s to it the conditions codes are being set so this has also an advantage suppose that here you do a calculation and the jump here depends on this calculation then on your simple processor you essentially do then calculations in between but your flags are already lost and then you have to do this calculation again to determine whether you jump or not and That means you have to redo the calculation on the arm processor you do a calculation here then all these instructions are being carried out, but they do not set the flags and You still have your flags set here in your program, and that is also something that is an advantage Also, if you are not setting the flags the processor trying to eat as quickly as possible through the instructions can see that there are lots of instructions not influencing the flags and therefore they can be executed even in arbitrary sequence and That is what the processor actually does the arm processor Well gives you the ID that instructions are being executed in sequence, but internally that's not true at all It executes instructions as quickly as it can Anyhow You can write the I add instruction add to to register our one and store the result in our zero But if you want to do it and set the flags you have to explicitly put this s there You can search for this for a long time if you silently assume that the flags are always being set Because yeah, if you just write the add and you forget the s and then do an Branch conditional branch based on that then nothing will happen and you get confused and This can take days to find You can even combine this so add EQ s is execute this instruction only if the flag the

Z flag is set and If you execute it set the flags again Okay, it's nicely orthogonal so there's not too much more to explain about this it is works for her Most Well the conditions work for all instructions and this this set option works for quite a lot of the instructions Okay, let's look at how we load and store Basically You have what's called a move instruction and this move instruction Goes from right to left not in all languages assembly languages.

They go from right left. Sometimes they go from left to right Literally confusing but at times but basically you can have some value here some operands and move that value into a register and You have so so so let's look at this example move our one Our zero our one means just move the content of our one into our zero very straightforward And then you have move negative I have no clue why they edit it I think it is a mistake and I also think it's not it's not available in the 64-bit processor anymore But it does the following you move the content of our one to our zero, but you X or it with FFF so you basically take the complement of the value That could at time be some use of that Okay, but in order just to understand how these operands look like we have to understand a little bit more and That little bit more is the following namely the other in this system has a so-called Right, there are actually five switch shifting schemes and shifting instructions and You can do that simultaneously so you can shift and then do for instance a move so You typically write it like this move our one to our zero, but shift our one Two points to the left as it is a logical shift left and that means multiply our one with two to the power two and That is just one instruction that can be executed So what does the left shift? logical shift left do It basically takes the content here it gets Well, either register value or a explicit five-bit value.

So this is the value 32 Up to 32 and it says how much? You have to shift to the left and it sets the carry flag So it basically shifts zeros inside Given this particular number. So if you put the number 10 here the shifts to 10 to the left and it's multiplication 1024 Well at sometimes that is useful Okay.

Now we have five of these. So this is one We also have the logical shift rights Basically the logical shift right means we divided by two to the power Whatever number the number of shifts we do.

So if I say move R1 to R0 logical shift right with two it's basically means Defines this number by two.

Defines this number by two. Well, there's a catch It puts zeros at the beginning so if it shifts this to the right you have zeros there so you looked on your phone When does this go wrong?

When does this go wrong? No you yeah, you know that you looked on your phone Yeah, so that is exactly the answer.

So if this would be a two's complement number and We would shift zeros at the beginning and this would be a negative number then what we have is is That you put zeros at the beginning so the negative number all of a sudden becomes a positive number while you divide it by Two or four or eight or sixteen and that is not intended. So there's also an arid medic shift, right? This is the arid medic shift, right? Which essentially takes this first bit and moves it at the beginning and and pets it with the same bit So if you have a negative number, you can now define it a negative two's complement number You can divide it by and say two or four or eight or sixteen any power of two And it simply remains a negative number and if it was a positive number, it simply remains a positive number So in that case we use the arid medic shift, right? Something that I now do not know is whether these flags are always being set if you do this that is something we should look up or Whether that is only being done if this s is is Explicitly part of the instruction.

So this there are some details that I'd have to double check. Yeah Yes, so we use that this notion of sign extension to divide it by a power of two Yeah, that's exactly exactly what we used there So that that is the explanation why this is a reasonable operation We have two more operations that we will use remarkably more often than you think namely the rotate, right? So in the rotate right you basically move every bit that you get here and you move that into Into here so so so you yeah, you basically shift it around and move these these these numbers to the front again And you can also have a rotator right extended where you actually extend it Rotate it through the carry. I do not know the use of this, but it's also available So well, we can write move our one to our zero and rotate this number right With two positions. So these two bits are moved to the front of the instruction Okay Now now we can we cannot play with this.

So This is what I already showed you now Let's first ask ourselves and Let's let's try and finish that before the break because this is something that also completely confused me for quite some time It's quite simple, but but I did not expect it. It took me off guard But we can move a particular value inside the register But on the little bit of a disadvantage we only have 12 bytes for this So if you would like to say move our zero with a big number, then it's not possible at all Still we would like to do that quite often So the question is if we want to move this particular number into this register are zero How can we do that without? Continuously writing nasty code and the solution to this is there is a meta instruction that's the Assembler recognizes, but it translates it to something which can be actually quite complex.

Let's Look look at this.

So we have these 12. We have these 12 bits available and These 12 bits are actually being split in 8 bits for a value and 4 bits 4 bits to do a shift So these 4 bits only indicate the value of 16.

So There is this rule that if you have the value 0 you shift 0 if you have the value 1 you shift 2 So you shift 0 2 4 6 or 30 so you can only do an even number of shifts Because we do not have the bits available Now what will happen is that if we take these 12 bits then then then we take the value the 8 bits and we shift that basically To another position so we can still make only 4,000 values and some of these shifts and values in combination get the same value So the actual number of values that we can generate in this way is less than 4,000. So If you would like to To write this then this is impossible because The value 4,000 cannot be stored in these 8 bits But the compiler, the assembler will look at it and will say ah But if we take 4 0 So this can be stored in In 8 bits and we rotate it 26 times so we put a value 13 there to the right Then that is actually the same as multiplying this value with 2 to the power 6 So rotating this to the right is the same as rotating it 6 position to the left and so what will happen if that you can type this in in assembler and This is the instruction that will actually be generated and the effect will be exactly the same So in this particular case, you don't have to worry but it could be Oh, so so so this is what you cannot do.

This is what you can do But if you write down this then it is automatically assembled to this instruction Now it could also be that you would like to put a bigger number there an arbitrary number there into R0 and if you write that then the assembler will Look at this instruction and it will essentially ask itself.

How can we interpret this? So it tries to map it on one of these instructions. So if that would be have this number 4096 it will translate it like this But if that is not possible, it does something quite remarkable in general It can solve it in different ways, but the standard way is the following you have your program. This is your load instruction And then it says that it stores that value somewhere before your rep program, but somewhere in the neighborhood Or something somewhere at the end and then it basically says load Relative to the program counter this value from here In there so it's still an instruction But it is a load relative to the program counter and that is how it will be translated So what you can write is simply load R with an arbitrary number use this is sign then this value is being loaded into R0 but internally you will not recognize the translation at all if you would look at the bits because it could be a Move instruction relative to the program account.

Okay We can do more things. So what is a little bit confusing is that this load R instruction here is a meter instruction at this point But it is also being used to actually load data from memory into registers and then it is an actual instruction so load R has two lives.

It's this meter instruction often translated to move and it is this Real instruction moving data from certain addresses for memory.

So after the break, let's look so let's start at the Carter to 10 Let's continue with With all the other move instructions and there are weird move instructions Okay, you can start again So Typically this move instruction is being used to load to move the contents of registers to registers and load Values inside a register from a single instruction and this load R Instruction if you write an is there is just a meter instruction Which is different from the load R on the next slide load R on the next slide is actually a Load instruction where you load something from memory inside a register And and yeah, as I said, I found this all quite confusing being used to Relatively straightforward a nicely designed instruction set Using the same names for in some sense quite different concepts is tricky So distinguish these three in your mind and I think it will be a little bit easier to answer after you have understood it You asked yourself why was it difficult at all? But if you have a misconception in your head, then it does not particularly work So we have actually We have load instructions.

We have store instructions. This is Intended to load one from a value from memory and store one register into memory And We have load multiple and store multiple that can actually be used to store even the values of 16 register simultaneously into memory are back again so that that is Quite a remarkable and unexpected instruction, but I'll come to this I first look at load R and store R and in essence what you can do is and I think you immediately Recognize this except then it was written as load and store You can do load Well the value from memory pointed to bit register R1 into R0 Or store it at that particular address So this is what the semantics is of such an instruction. You cannot store and load from a specific address To specific address.

It's always relative to the address in a register But we have seen how you can if you really would like to store something at a specific address how to do that Simply load that value in a register Okay now We will decorate this a little bit so what you can actually do and it gets weirder and weirder But what you can actually do is you basically say load The value of with R1 Well, but you do this in an indexed way So you basically put an index at at a 12 to the content of register R1 and that provides you with the address from which you load something in R0 and In the same way you can store that and there is a special bit that indicates whether this this number and I believe this Is a 12 bits number that you can use So you can use these 12 bits, but now for an explicit explicit value And you can indicate explicitly whether it should be a positive or negative With one bit, but of course the assembler will handle this for you Okay, you could also write an exclamation mark after this and What that means is that actually you calculate this Value R1 plus 12 and then you store that in R1 So this is very convenient in in some sense namely you walk through an array and in one instruction you get the value from that array and you increment the Value of you first incremented by one you and then you get this value from that array and then you increment it by one And you only need one instruction to do this so With this exclamation mark you can write it and of course Optimizing compilers can this use this to a big benefits even even Do doing unrelated operations in one instruction Well, and it can be weirder So this is the pre indexed addressing that is what we have seen and Then then then we only address but we do not change this register R1 here we in the auto indexing addressing we store the changed value also in the register that we use to address To to calculate the address for memory And we can also do post index addressing namely.

And we can also do post index addressing namely. What does that mean? reload in R0 the value that we find at R1 and then After we essentially have done that we add up 12 to R1 so we change our one after we use it and I Would expect an exclamation mark there because we write something to our one, but why is there no exclamation mark?

Would expect an exclamation mark there because we write something to our one, but why is there no exclamation mark? Sorry So I still do not completely get it.

It should say it a little bit louder. That's yeah, so so If Suppose that we would not we would increment R1 by 12, but would not store it this whole instruction does not make sense That is the actual reason Namely If you would increment R1 so we we load from this address we would then increment R1 by 12 and Suppose that we would not write it it makes no sense to increment R1 by 12 so By default the only option is if you write this then it is intended that R1 Really changes and therefore the behavior is as if you wrote this exclamation mark But as the instruction without the exclamation mark does not make sense.

The exclamation mark has been omitted here. I Also, if you think about language design, I don't think this is optimal on the other hand It saves you writing this exclamation mark at all these points, but all of a sudden this Exclamation mark has a semantics right the results in this register here.

You also have that same semantics But you do not use this exclamation mark Yeah, okay Now we have the barrel shifter so in Addition to these three patterns that are already showed you can use Any of the shift instruction. So what you can essentially do is say we have R0 We take the value of R1 plus R2 here. We do not Index with a value anymore with with a register. So this is a register indexed Register addressing But this second value there can actually be shifted using the shifter We take R1 plus R2, but we multiply R2 by 4 and this gives us the address and that address the value at that address is moved into register R0 and If we store this with with the right if we use the right then that's actually a division So we can divide R2 by a power of 4 add that up to R1 that gives us an address And we store the content of R0 in that particular address And then that then we can do everything again in the same way. So we can We can address with an left logical shift left But with an exclamation mark we also store Change the value of R1 by simply adding this R2 times 2 to the power 2 to R1 Yeah, there can be use of that One of the uses is if you write to a stack then we would like the stack pointer to be adapted after the instruction And in the same way we can do post index addressing. We now again do not write down the exclamation mark But in essence we say we load the content of R1 into R0 and and now we write it outside these brackets and We change the value of R1 by adding up the value of R2 divided by 2 to the power 13 Yeah This can be done, okay And I think this is more or less what you can do with loads and stores So this this is what you have available it's quite a puzzle to make compact code fortunately we have compilers for that It's actually quite tricky to get working code because you can so easily make small mistakes Okay Let's look at the load multiple and the store multiple.

So there's one instruction that is this instruction not a branch instruction Where was it ah here the block data transfer instruction the store and loads That you can can actually do and here you have 16 So so you do it modal a register and here you have 16 bits that indicate which of the registers should all be stored or not and in that way Yeah, you can Select any arbitrary subset of registers and store them to memory Simultaneously or load them from memory simultaneously So how does that look like well basically? You Can say load from memory Indicated by our zero So now our zeros gives you the address in a room. So that is quite different as before and From that you load our to our tree and this is common notation with his bar our five our six and our seven From memory and you basically get them from the consecutive places in memory Of course, these are all 32 bits words. So this is from at from Address our zero and then our others are zero plus four and I think this 16 here should actually be a multiple No, that is there are four registers.

No, this is 16 is correct. So this indicates exactly the address of our seven and You can also store multiple Register so if you want to store our 14 and our 15 then simply store at address indicated with the first argument the first register and you store our 14 at an address and our 15 at a consecutive address actually using the bits you can indicate whether you should store at increasing addresses or Decreasing addresses depending on what you like to do.

That is part of this instruction now Typically If you have a piece of code and you do all kinds of calculation in that code So you use all kinds of registers at the beginning you would like to save all the registers such that you can restore them at the end so typically at the beginning of a piece of code you write down that you would like to store all your registers onto the stack and that can don't be done with this single instruction that if you want to store our zero up to 12 to the stack you Store them to the stack pointer. The stack pointer is register are it was our 13 if I'm not mistaken and You will increment With this acclimation mark the register afterwards or you store it to the stack and Then the pointer of the stack points to the new free place and you can have stacks that grow down and grow up That is both possible And In the same way you can load multiple of these instructions from stack From the stack and you can even load the program counter from the stack So suppose that your return address is stored on the stack Then with one instruction you load all the registers from the stack and the program counter And you will execute at that place where that program counter pointed at Yeah, oh The addresses are in bytes of the memory, but we store 32 bit words and That means that these take four bytes So if you store one 32 bit word at one address, then the next one should be stored four position laters because The first one takes at four bytes and then you have to store the second one four bytes later.

So that's We will find these things at more places where you sometimes have to increment by four And and sometimes the ARM processor does that automatically and sometimes it does not align to this for position boundary Okay, these are the loads multiple and store multiple.

Now, let's look at ordinary instructions Given that you now understand the previous instruction the rest is relatively simple.

That's all and in the same line so we can do an add add R2 we take our tool we add at four and we store it in R1 You see that you can take elements and store it at other registers Contrary to what we did do in our simple process if I always had to Take R1 as the first argument add up something and we always store it in the first register. These are simply different assembly sets You can add up a register With another register.

So R2 plus R3 and store that in R1 You can add with carries if it's useful if you have to read do big calculations and and add Big numbers which by and large we do not do but if you do it, then then you have an add with carry instruction you have a subtract and I have no clue why this thing is there you have a reversed subtract and Subtract subtract R2 minus R3 and the reversed subtract simply changes the arguments But you could also write down subtract R3 R2 And with almost the same effect except that this last argument Can be used in a using a shifter So you can apply a shift Operations any shift operations to this last argument and then you get add up R2 and R3 So I think this is subtract. So this should be a minus And at the same time apply one of the shifters that you have available so that is possible and then of course a subtract can and the reversed subtract cannot be Changed to a subtract because if you apply the shift to this R3 Then this cannot be done to our don't to our to you cannot apply the the shifter The the battle shifter to to our tree, but that is Probably the only reason so I also believe that the reverse subtract does not exist anymore in arm 64 Well, we can do an and we can do an or all these things have been written with three letters. So the or is or Or we have an exclusive or that is the E or You can clear particular bits in a particular word Effectively, yeah, and there are a few more operators, but this is by and large what you can do Okay, then it's time for jumps and branches and now the remarkable thing is That there's only one jump instruction. That is just branch And Yeah, what is the reason that you only have this branch instruction? Yeah, so so so so we have convinced we Any instruction can be given with condition codes So automatically you can have branch and you add this condition codes to it. So that is essentially why we do not have a special branch in searching because we silently have them available anyhow, so What's happening here if you do a branch? Then well now you have to understand that the program counter is pointing two instructions ahead And it's not pointing to the current instruction But two instructions that had these are 32 bit words is that it is actually has the value Eight higher than the current instruction being executed now If you do a branch to a particular address then basically well because these last Two bits do not matter because all instructions are at this boundary of four four Bits so they are all 32 bits lungs the processor automatically multiplies this address with four and It subtracts the value Minus eight to compensate for the fact that the program counter is always to a head and The same is no we can also do that under a particular condition. So if That is equal to zero. We do this particular branch to this particular address now, you can also do a branch with link And that means that basically do you do a branch instruction? So you change your program counter and you set the link register to the value PC minus 4 by minus 4 3d Either I am deaf or you are all speaking Yeah, so so and and the next instance the instruction pointer points to a structure ahead But we want to get the next instruction So we have to so it points eight the value eight bytes ahead and we have selected by four to get at the next instruction directly after the branch instruction and that is the address that we store in the link register and Store instruction or a store multiple instruction to the stack if you would would reuse the link register later, so Branch with link is essentially the same as branch to sub protein and of course this can also be done on the conditions So we can do branch to sub protein under a condition Yeah, that is possible now at the end of the code quite a number of ways to do a return from sub protein and One of my favorites is this one But you may not always recognize that immediately if you say move the link register back to the program counter Then the return address that you put here will move be moved back to the program counter So return from sub protein can be implemented as a move In The practicals, I think you will often see that you do a branch with an exchange where you basically also branch to The content of the link register.

That's also a way of doing this Yeah Okay, these are the branches now more instructions multiply Multiply and add if you want to have big big numbers being multiplied We have the compare instruction, which is of course very useful to work to subtract two numbers and set the flags I think the compare instruction always sets the flags There's no option to do that without otherwise it would not make sense you have a compare negative where you Flip to an X or you flip all the bits of the second argument Might have some use you can test the bits And see whether things are equal Yeah, so so so this standard there There are a few more instruction namely a swap instruction, which is really necessary for multi-threading.

So Let's let's skip that for the moment and we have a software interrupt in search So so you can imagine that we have an interrupt from the outside world saying there's something happening But there's actually a software interrupt in search image. I also found somewhat confusing Why have a software interrupt instruction? Couldn't be encoded but it's being used for basically the security of the system, so Let's let's look at that So let's look a little bit at the security of the system and how that is being being realized Generally, this applies to all modern computers. It's all being done in essentially the same way and all computers, I think starting from the 80 to 86 although These mainframes in six seven in the 70s also had this have different modes So essentially if the computer starts up it It it does this in the supervisor mode the supervisor mode The processor basically has access to all possible hardware and can can set up the whole machine Assign blocks of memory do all administration Loads an operating system at some point and after a while When it is ready setting up the computer It basically says I'm now going to the user mode and the user Has only very restricted access to to To the machine so it cannot even access an input output port We have to do that fire the operating system and operating system shields everything because if we could just access input and output port we could probably just get all the data from the disk and inspect a complete disk and find out what's on the disk and Especially from other users and that is not allowed Therefore everything goes through the supervisor even if you want to send a message to fire a port it goes through the supervisor and operating system so Here we have only very very limited access.

So here we start Here we go to normal operation and and Yeah, unless you become a root then you can Access more of the machine but for the rest you can only well You can run an algorithm and that is generally enough for all people except a few If there's an interrupt Then it goes to enter modes in the intro mode.

It has more access and it can set interrupts registers jump to pieces of code if You that handle the interrupts so The rights are extended and if you do a software interrupt then you change back to The supervisor mode so basically a software interrupt is a call to the operating system To do something which you are not allowed to do yourself and the operating system has code In there that checks whether this particular operation is allowed if you want to read a file Then then you request read a file and store it at a particular address in memory Or read a next line from a file or read a character from a file and then the operating system Will actually in software check whether you have the rights to access that file and if not it will basically say what what does it say something like long a nose or I Cannot translate that how to say that you get an error message in some sense so This is the purpose of software interrupt So a lot of programming now means setting up the right software interrupts carrying them out and then the operating system does the actual work for you and that is What a lot of these programs look like so software interrupt is just a software interpret a number and actually this number is Not being used by the processor at all That's rather weird But you can I do a software interrupt with number 17 Then the processor can in software extract that value because it knows the next instruction And Because the next instruction was in the link register is a bit subtract for it finds a software Interrupt instruction I can load that and can actually get the value and then decide something based on that Often this number is not not actually being used at all But all the information is passed in explicit register So if you want to have a particular service from your operating system, you do the software interrupt instruction and you say in register 7 Print the particular string that is pointed to with these other registers And then the system does that and it will return and you will get in your restricted world where you can write your program So what it will essentially do is it goes to the supervisor mode it puts the next instruction into the link register Yeah, this I also find the boy weird.

I double-checked it, but it jumps at the instruction at address 8 in memory And there often is again a jump instruction that takes care that you jump at the appropriate routine and You stay safe all the status flags that you have So you get new status flags and you can execute and supervise remote But when you go back the old status flags are being restored such that you live in User mode for the pp2 processor you had also a similar instruction there It was called a trap not so much a software interrupt But the effect was essentially the same that caused the software interrupt and it caused a jump towards the operating system That would then do something for you Okay, so how do programs in arm look like Well, there is a Standard keywords often Start but this is different for all kinds of assemblies other assemblies use other keywords This is command. These are commands commands are very useful because If you look at these instructions, it's it's quite incomprehensible very quickly So looking a week later and you have no clue what was intended. So try to always write down clearly what? each instruction, but each each piece of code is doing And write down what the inputs and outputs are in terms of register. So In this particular case The Code is for a small game a game that you are supposed to change a little bit and and run and Basically in the code you generate a number and then you have to buy guess the number by going up and down so here in this code you do a branch with a link to generate number routine somewhere else and Then at some point you get back then in this routine stores some kind of Generated number in our zero and you move it to register our aid so it's now stored in our aid that is the generated random number and Then you store the value 10 in our nine This is an allowed move of data because this value 10 fits in a in 32 bit instruction, so you say I have 10 guesses and then You loads the address of the new.

Oh Of the text new game So this is actually a pointer in memory where you find all the bytes In ASCII code and this gives you the address of that string You move that in our one and then here you have codes to print that string and then at the end exit code At the end you basically say I Put one in our seven and that means I want to stop my program so this corresponds to stopping the program and you do a software and throat and Then basically your operating system says ah this program wants to kill itself Kill removed from memory and go back to the operating system prompt I think that Would say that that should be 10 So I have no clue But but but book Sorry, oh, maybe yeah.

Sorry, oh, maybe yeah. Yeah. Yeah Okay, let's let's look at some other other piece of codes this if we want to print some string to the monitor Then we we have the address of the string in our one and our two So this is codes that we can write ourselves and this is typically how a lot of code looks like So if you apply your disassembler on your windows machine Then then on your 8086 you will also recognize quite a lot of push instruction at the beginning of each routine and pop instructions at the end So here the code basically says we have in our one the address of the string are to the length of the string Yeah, and now basically what does this say? Save the link register because we will do a software interrupt here. So we will destroy the link register So we first want to save that we also use our seven So we want to save that too and our zero is a scratch register anyhow. So We do not save that although we could have saved that to on our on our On the stack and then basically do we do a software interrupt?

On the stack and then basically do we do a software interrupt? We load four Which which is the number to print a string? Say that it should be to standard out So this is all looking at the interface of the system routines And if it should be printed at standard out, then we apparently have to put one into our zero R1 and R2 are already set up.

These are also important for the software interrupt We call do the software interrupts then the operating system takes care that the strings are being printed to the output we go back we restore our seven and the link register and now we do a Branch exchange with a link register Which also has the effect of moving the link register in the program counter could also be done with a move instruction So what you could try is replace that with move link register to the program counter. That will also work More Well, let's do this very quickly So print the loose statements.

Well, we store a number of registers on the stack Then we loads the string lost game So you lose the number was well, it was not zero zero. It was an auto number. So We will actually calculate what this offset was this offset is given here.

So we load this offset We Essentially write the number that had to be written by our own routine So this is routine that that we also provide and write ourselves and we write the right number there then We put The address of of this string in our one We put the length of the string in our tool and we call this our we call this Branch with a link to this printer's instruction that we gave on the previous page now when we return everything has printer has been printed and here we restore our registers and Simple and you well Essentially all these instructions are not difficult, but you get killed by the quantity So at some point you have thousands and thousands of lines and no clue what this is doing but it works So there are still many banks and other Organizations that have this software there and you are not allowed to touch it Even if you see in the code one plus one is three You are not allowed to change that into one plus one should be two because at some other place in the code You could have code that actually replaces every occurrence of three by two such to compensate for this.

So these The quantity is is actually killing by by and large what's happening is very straightforward Okay Okay.

Now you are the perfect arm programmer good now five minutes of history Now Dutch history So what happened in the Netherlands? I think that if you look at France Then you have similar like similar kinds of histories if you go to the United States then The history is also similar but some of these companies did survive Let's let's look what happened in the Netherlands but but but As I said quite quite a lot of these stories are similar.

So in the Netherlands We set up a nice Institute Center for mathematics and computer science.

It was then called the mathematical Center In Dutch it's called savy e-centrum for this kind of an informatica and I happen to be very lucky to do my PhD there Not even knowing what the Institute was but I heard of the Institute.

I just just did a phone call because got the People people at the reception and they basically said well, I'd be hurt that that person got a project For which they still search people so they connected me and two days later. I had a job Which is quite remarkable. This is the way how things can work two weeks later. I started Anyhow CWI in Amsterdam Center for the so later they called it the Center for mathematics and and computer science but in the early days it was called to the mathematical center and There we had from van harden the first director and he was a mechanical engineer, but very much interested in Calculations because he had observed that lots of calculations had been done when you do mechanical engineering So you would like to have a well now something that could do mechanical calculations cheaper So One of the things that they did do was they basically hired a lot of women. This is one of them. I Once attended a short interview with her And these people that was a time where the Dutch were very nasty with women So if you were a woman, then you really had less chances than you have these days So many of the very skilled people at secondary school did not study But they're good in mathematics and were hired at the Center for mathematics and we're basically doing all the calculations So in the early days lots of people were sitting there basically doing calculations the whole day Generally Women so they were called the Reagan nastas They changed their job from doing calculations to actually writing programs Anyhow from van van harden went to Great Britain and heard about the

Colossus and said we want to have something like that too, so he said we will start with the Arithmetic they can a parade something like that and The first hour was made it was particularly unreliable so they they invited the Minister of Education and the major of Amsterdam and They really set it to work and they thought well if we let it calculate 5 plus 5 Then even the Minister of Education may observe that if it says 17 due to a book Then that is not the right that will not give the right impression. So let's do something smarter Regenerate a random number and if the number is not correct Then then they will not recognize that and they presented that as a big thing. So we generate a random number and That's the only thing that I have one ever did So it was too unreliable then then we had this guy Here at Blau. So he actually went to 20. I studied in 20 whether that was the best choice I can this course for hours, but I think ain't hoeven it a much better. Well was certainly a much better choice Anyhow, he came to see blue blue. I he had worked at IBM with the deck 20 and he said we should do these things with a proper engineering mind We will not just connect everything.

We will have systematic boards systematic connections everywhere everything is done in the same way and The other tool was a working computer in quite a short time Due to the systematic approach by Blau a Blau were died two three years ago.

Unfortunately We have this other guy Willem from the pool who is still alive and Willem from the pool worked in in Delft and in Delft He was hired as a PhD student by Nicolas de Brane who actually moved to ain't hoeven because he said ain't hoeven. He was often invited to go back so de Brane is still seen as one of the big and most impressive mathematicians that we have in the Netherlands and he actually Made well, I don't think you have already seen that but that made provers proof checkers such that you could give a precise logical proof and The computer would check that that was one of his major Projects for which he is now but he did do a lot more but in the early days the brain was in Delft and Well, they asked him back I Should not tell this because we don't have time But they asked him back to go back to work to Delft and then he said well Let's go to the main building It's now called Atlas and he walked up with the visitors And then he would open the door at the top and stand on the roof and then say look around here So look what you see. Well person said we see a lot of trees Yes, we see a lot of trees. We see a lot of nature. It's much more pleasant to work here So please nice that you have visited me, but I will not go back to where Dallas full of houses and buildings, etc I like to live in Noonan Well totally irrelevant in the Willem from the pool Actually started to set up Also machines and he wanted to make them very simple. So he made the tortudo That was a very simple machine to calculate with light waves and it was much slower than than a human But what they would do was they would actually say Well put an exercise and then a calculation in there then it would calculate throughout the night and in the morning The calculation was actually ready and that was still faster than human because humans could do it in eight hours during a whole working Day, they could also program it set it to work and read of the results later on here. He went on He made the different other machines among which the zebra anybody and that is a Dutch question What the zebra stands for neighbor lance zir in father Binaire bacon upper-hat So it's a very simple binary calculation device It was difficult for them to get actually to get these machines being produced by actual companies so the zebra has been produced in Great Britain by a Stantec and With the other they said well, we should now start with production and the party that really made a difference here was a Insurance company an insurance company said well, we want to have calculations A lot we do a lot of very boring calculations and we rather have these machines So we will invest in this and they set up a company Electrological basically making the x1 the name stems from the fact that they said well We do not know the name.

We do not know the name. So let's call it X. So they wrote X on a blackboard and They never replaced it by anything else So they got the x1 and that was a very reasonable Machine, it was not yet a machine that you would would call Well, it could be used by specialist they Replaced it by well, there have been versions of x2's and x4's and but they also had the x8 And The x8 was eight times faster and that was really one of the nicest computers that we developed in the Netherlands Dijkstra you may have heard that name actually designed partly even the instruction set and the operating system and This machine was completely built around the operating system that Dijkstra Constructed and in 1965 for a very short time It was really the nicest machine that that was around but then other companies went on and they essentially lost the competition so Let's look at a few pictures. So this was actually the console with which you could program this Machine, so we managed to save one of these computers There's also a very long story, but for a short time the whole x8 has been in in the How is this mission museum called the museum in Leiden?

How is this mission museum called the museum in Leiden? I Know the name of the museum, but that will come up later Unfortunately People took it away, but there is still the console the console from Philips is still there.

It was a 27 bit computer a long story why it had to be 27 bits which I could I did not study and I do not understand and you could actually well set the address to run it and set the address to write down a word into memory so you could write program it by setting the Switches and then actually say write this word and then set the switches again.

Switches and then actually say write this word and then set the switches again. You can see that this machine Programming is very delicate. I cannot imagine that I would be able to do that correctly But they used this console for the machine to actually run it now this picture you have already seen But I think it's very illustrative for how a simple computer looks like if you would have explicit wires Actual machines are far more Are far more complex and have far more wires Somewhere inside and and any idea what this is It was a modern machine 1965 it was Amazing what it could do What is this it's not the stove to heat up the machine I think you should all know this value of of course are of SSD memory and not of hard disks anymore But this is the high disk So this is this whole thing is the high disk It's a this height it would slowly start to spin when it was on speed it would move up a little bit and Then well, they did not have the notion of heads Being able to track and go up and down So all these connections here basically are the right and read heads of the high disk Yeah, this is how the hardest look like this was a machine with a hard disk It's much better than the tape recorder we had later on. So this is the hardest now What happened further?

Well, they made a number of these computers Phillips at that time. Well first said we do not want to have to do anything with computers because they had an agreement at IBM And they said to IBM I've IBM said if you make the components then we make the computers and That is a profitable deal, but you are then not allowed to think about making computers They made a computer at the not lob but not Really for production. So Phillips refrained from touching computers until IBM said ah But we can also make our own components that can be done more cheaper than you deliver them. So Go away and then Phillips decided to wait very nasty until electrokocha then broke and Then they took the whole thing over and then they looked at these very nicely designed x8 And they said well we from Phillips know better we make the Phillips P1000 which is completely different and I really do not know whether there's was logic behind it But it took them quite a lot of time They had a family of machines that they managed to sell So at some point especially Dutch banks had these Phillips machines with their own very specific processor But at a particular moment also these Philip machines could not compete against IBM They say they lost more than a billion guilders. That's half a billion Euros on trying to design and sell the computer. So it was in no way profitable for a phillips So this was the end of the story. Well, they tried to revive themselves a little bit by Designing a Phillips P2000, but that was a machine that was more or less this this consumer machine that you would say would have with a processor taken from elsewhere and At a particular moment there were a lot of IBM clones. So Phillips made IBM clones, but they had his nasty trick So we had one student inventor Who of which the parents worked at phillips?

Who of which the parents worked at phillips? He got a scholarship from phillips So that was all were all and he also had a IBM computer IBM compatible computer from phillips And then we wanted to connect the printer to it, but the printer did not work. Why not because they inverted to the The ready signal such that only printer from phillips could be connected to To this particular machine standard printers could not be connected. That was a nasty trick of such a company This is something that that gave that still give me a bad feeling about About buying a computer from phillips this you should not do this as you should not trick consumers in that way Anyhow, what happened in the Netherlands further at some point? We had the company tulip tulip for a long time made IBM compatible computers But they could not compete against say these big companies that now the Dells etc So they went broke and by and large this is the set end of this story. We do not have anything anymore Regarding constructing computers.

Regarding constructing computers. Well, we have ASML, which is completely essential to build the chips with building computers Well, we all buy them Yeah, so what did we do? Well, you now know the arm 32 processors and you know a little bit about how security is Constructed inside a computer. I think this is an important thing to know and There was an impressive time During the history of Dutch computers If you really played a role of significance and that by and large at least if it comes to the construction of computers Has gone and we simply all buy them anyhow next week Tuesday there will not be a lecture on Thursday I will show up again. Okay. See you then