What This Video Assumes

- The same assumptions from Parts 1-3 of the Video Series are met.
- That you've watched Parts 1-3 of the Video Series and understand them thoroughly.
- That you have IDA Pro 7.0 or possibly above. I can only guarantee this works for IDA Pro 7.0, though. A lot of people also use Ghidra to disassemble Switch games. This is fine, but I don't want to work with Ghidra due to it being created by the National Security Administration of the United States of America. If you're using Ghidra, the functions and registers I go over will carry over to Ghidra, but the section where I actually use IDA Pro will not carry over. At that point, if you're using Ghidra, you'd want to stop watching the video.
- That you've installed the Switch Loader plugin for IDA Pro 7.0. If you haven't, you may find the plugin here: https://github.com/pgarba/SwitchIDAProLoader
- When you install the plugin, install the package from Switch64_70_122.zip It's the most updated version of the plugin.
- That you've installed NXDumpTool on your Nintendo Switch. If you haven't, you may find the tool here: https://github.com/DarkMatterCore/nxdumptool/releases

Video Contents

- Double+ Pointers (One Slide Long)
- ARM64-v8 (AArch64) ASM
- Bytes and Different Word Types
- ARM Registers (GPRs, SP, ZR)
- Instructions versus Functions
- Non-Conditional Use of ARM Instructions (SUB, ADD, MOV, LDR, LDP, STR, STP, RET, BLR, and NOP)
- Condition Codes
- Conditional Use of ARM Instructions (CMP, CMN, B, BL, Previous Instructions with Conditions Added)
- Atmosphere Cheat Codes Translate to ASM Instructions
- Instructions We'll Be Using/Writing To
- Getting the Game Executable using NXDumpTool
- Searching for Relevant Functions in IDA Pro 7.0 to make Cheats
- ARMConverter and Code Type 0 for ASM
- Decrypting ASM Cheats to Learn from them
- Sections That Transfer Over to Other Consoles

Double+ Pointers

- In Part 3 of the Video Series, I went over how to create Pointer Codes. Turns out that there's an unusual type of pointer you may encounter from using PointerSearcherSE. This is a Pointer known as a Double+ Pointer. Here's an example:

- [MAIN+759E0000]] – This is a double pointer
  or
- [MAIN+759E0000]+40]]+98C – This is a double pointer
  or even
- [MAIN+759E0000]+40]]]+98C – This is a triple pointer. You can theoretically get as high as you want with number of brackets, but I doubt that'll happen.

- Double+ Pointers are handled easily. For each bracket extra, use 580F1000 00000000 as a line. Here's an example with all three of these:

  580F0000 759E0000
  580F1000 00000000
  640F0000 00000000 80000000

  580F0000 759E0000
  580F1000 00000040
  580F1000 00000000
  780F0000 0000098C
  640F0000 00000000 80000000

  580F0000 759E0000
  580F1000 00000040
  580F1000 00000000
  580F1000 00000000
  780F0000 0000098C
  640F0000 00000000 80000000

ARM64-v8 (AArch64) ASM

- Whenever someone says something is an ASM Code for the Nintendo Switch, they mean Assembler Code. Assembler code is the type of code the processor of a computer system uses to commit changes to RAM. Assembler code is one level "lower" than the actual programming language of the game itself, where binary is the "lowest" level of code. That is, programming language gets converted into ASM Code, and then ASM Code gets converted into binary to be executed by the computer's processor. ASM is a programming language within itself.
- The Nintendo Switch uses the ARM Cortex-A57 processor, and as a result, the programming language of the Nintendo Switch's processor is ARM64-v8, also known as AArch64. This is the assembler code language the Nintendo Switch processor uses. For the rest of this video, ARM64-v8 will be referred to as ARM for the sake of simplicity. Other versions of ARM, including previous versions, do exist, but I'm just generally calling the language ARM for the sake of this video.
- The purpose of this video is not to have you knowing the entirety of ARM by the end of the video: the purpose of the video is to teach you enough ARM to make basic Nintendo Switch ASM Codes.
- The explanation of all of the instructions in the ARM section of the video can be credited to both https://developer.arm.com/ and https://www.keil.com/support/man/docs/armasm/ , as that's where got the instructions from. I also learned a good amount of ARM from both websites, so credit to them twice over.
- We'll be using ARM for the purpose of changing how the games modifies our target addresses, like money or HP.
- A great feature of creating ARM codes is that you don't need to make a pointer code for the address you're looking for. Instead of writing to the pointer address, by changing the game code, the game will write to your target pointer address for you.

Bytes and Different Word Types

- Remember hearing data types like 8-bit, 16-bit, 32-bit, 64-bit, and 128-bit in Part 1 of the video series (well, at least everything but 128-bit)? Well, they end up returning in this section. I will reintroduce the term byte and introduce you to the terms halfword, word, doubleword, and quadword.
- A byte is just a data-type with 8-bits, that is, 8 zeroes and ones, to represent a particular value. Basically, when you hear byte, think 8-bit data-type. EX: 0x7F (2 hex characters)
- A halfword is just a 16-bit data-type EX: 0x7FFF (4 hex characters)
- A word is just a 32-bit data-type. EX: 0x7FFFFFFF (8 hex characters)
- A doubleword is just a 64-bit data-type. EX: 0x7FFFFFFFFFFFFFFF (16 hex characters)
- A quadword is just a 128-bit data-type. EX: 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF (32 hex characters)

ARM Registers

- Registers are a space of memory inside a computer's processor. They're basically addresses reserved only for the processor and the code the processor executes. Data from the code executed by the processor moves in and out of these registers, allowing for several different operations to occur. Registers are significantly faster than the addresses typically stored by RAM, so these registers allow for code and data to move in and out as fast as the computer needs. In ARM, there are three basic register types that I'll be teaching you. There are more than this, but these are pretty much all you'll need to know. These three register types are known as General Purpose Registers (GPRs), the Stack Pointer Register (SP), and the Zero Register (ZR).

General Purpose Registers (GPRs)

- These registers generally store the value of addresses, addresses themselves, or data values that are having mathematical operations applied to them. There can be about anything in these registers, hence why they're called "General Purpose."
- There are 31 GPRs, named ?0-?30. ? is another character, generally dictated by the data-type of the value in the register.
- If the data value in the register is a Byte, Halfword, or Word, the 31 GPRs are generally named W0-W30. If the data value in the register is a Doubleword, the 31 GPRs are generally named X0-X30. If the number of the register can vary, I will have named it as ?n. For example, Xn would mean a register from 0-30 that contains a Doubleword. 'n' in Xn stands for the number of the Register.
- Addresses in the Nintendo Switch are always 64-bit (Quadword) and values on the Nintendo Switch are generally 32-bit (Word) or below (Byte, Halfword) , so generally, addresses are stored in a Xn register and values of addresses are stored in a Wn register. Of course, realize some address data-types are quadwords, so there may be an occasion where the value of an address is stored in a Xn register as well.
- Float and Double values are stored in a Sn register and in a Dn register, respectively. S is for single-precision, also known as Float, and D is for double-precision, also known as Double.

- There are also names for registers for values generally used in Physics. These are scalar and vector values.
- A scalar, which is a value with a measure without direction, is stored in a Bn register, Hn register, or Qn register, respectively. Bn represents a scalar register that contains a Byte, Hn represents a scalar register that contains a Halfword, and Qn represents a scalar register that contains a Quadword. Generally, you won't need to write to instructions with the Bn, Hn, or Qn register naming conventions, as we really won't be touching on editing values related to Physics. Sn and Dn registers may be used not only for float or double values, but also for scalar values too, so be careful here.
- There are also vector values, which is a value with a measure and a direction. Basically, a vector would contain two different values at once. Vector registers are labeled as Vnn.nnT, where nn are two numbers between 00-30 and T is the first letter of the data-type of the value, so B, H, S, D, or Q. Two numbers were used at once because a vector contains two components. You'll generally never need to write to instructions with this register naming convention, either.
- Realize that despite these registers being named differently that these are actually the same registers being mentioned in the processor. These registers are just being told how much data to store inside of themselves. These registers are 128-bits long, allowing for writing and reading from 8-bit data all the way up to 128-bit data. For instance, W0 and S0 are really just the first 32-bits of Register 0, but one is interpreted as a Word, the other as a Float. X0 and D0 are really just the first 64-bits of Register 0, but one is interpreted as a Doubleword and the other as a Double. Finally, Q0 is really just all 128-bits of Register 0.

General Purpose Registers (EXs)

B0 <-Contains a Byte Scalar Value in Register 0
H3 <-Contains a Halfword Scalar Value in Register 3
S5 <-Contains either a Float Value or a Word Scalar Value in Register 5
D7 <-Contains either a Double Value or a Doubleword Scalar Value in Register 7
Q1 <-Contains a Quadword Scalar Value in Register 1
W10 <-Contains a Byte, Halfword, or Word Value of an Address in Register 10
X11 <- Either contains an address or Doubleword Value of an Address in Register 11
V03.4D <- Contains two values to make a vector that is a Doubleword, with one value in Register 3, the other in Register 4
V03.4H <- Contains two values to make a vector that is a Halfword, with one value in Register 3, the other in Register 4

- Remember, the number value beside the register can be anywhere from 0-30, so you might see B0, H0, S0, D0, Q0, W0, X0, V00.1D, or any combination of numbers between 0-30 right next to the letter of the register.
- More likely than not, the registers you'll be messing with are Wn and Sn, as most values in-game are Words/Floats, smaller than a Word, and not used in Physics calculations.

A Visual Representation of GPRs

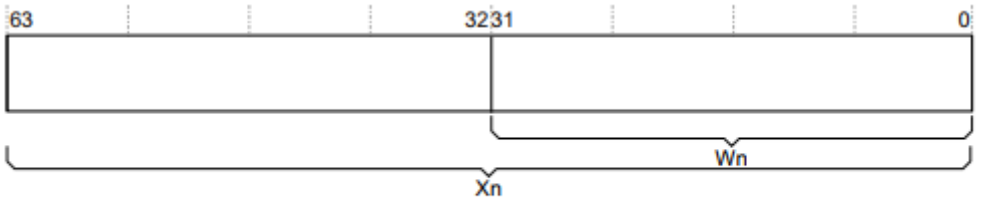

Figure 4-2 64-bit register with W and X access.

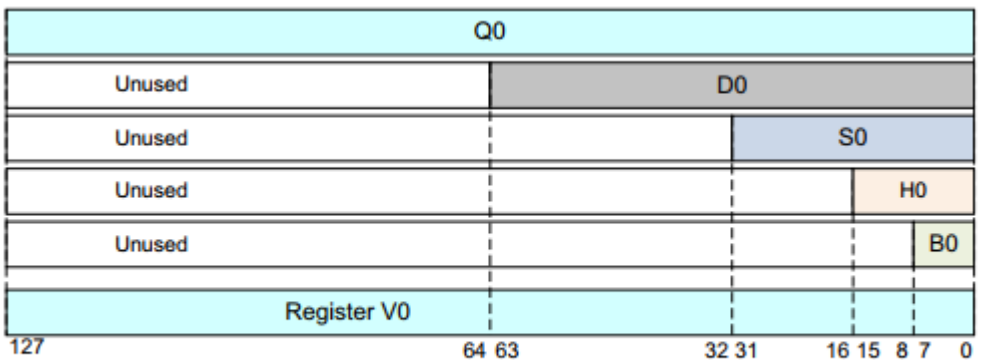

Figure 4-11 Arrangement of ARMv8 registers when holding scalar values

Image Taken From:
https://www.programmersought.com/article/91233774448/

The Stack Pointer (SP) and the Zero Register (ZR)

Stack Pointer (SP):

- The Stack Pointer 'points' to the function of the program currently being executed by the processor. In this case, our game is the program. Think of the Stack Pointer like a bookmark, but instead of it tracking a page in a book, it's keeping track of the current function being executed by the game.
- The Stack Pointer register is abbreviated as WSP if the address on the stack is a word and as SP if the address on the stack is a doubleword. This register stores the current address of the function in code.
- Never write to a line of code with SP in it. You have no need to and you'll probably crash the game.

Zero Register (ZR):

- This is a special register inside the processor that just contains the number 0.
- The purpose of this register is to be able to store 0 to other parts of the program, essentially erasing their values.
- The Zero Register is notated as WZR whenever it is zeroing a word, halfword, or byte and is notated as XZR whenever it is zeroing a doubleword. Very similar to the naming convention used by the GPRs.
- Generally, you don't write to instructions with WZR or XZR in it, but it could be useful to place WZR or XZR in an instruction. For instance, let's say a function contains the value of someone's health address. You could insta-kill that person by moving the Zero Register into the health address, zeroing the person's health out. I'll explain what I mean by this later.

Instructions versus Functions

- An instruction is one line of code with some operation executed. A function is an area of the program that contains several instructions at once. If the function is executed, every instruction of the function is executed in order. Some part of the program will call for a function to be used, the code of that function is executed, then the previous function that contained an instruction that gave an instruction call will continue.

Non-Conditional Use of ARM Instructions

- When I say something is conditional, I mean that something will only execute when a certain condition is meant. For instance, one instruction may not execute unless a certain register is greater than, less than, or equal to a particular value or register. Non-Conditional instructions, then, are functions that will execute without a condition or have nothing to do with conditions whatsoever.

SUB Instruction

- SUB rD, rA, rB
  or
- SUB rD, rA, #imm16

- Where rB is the register to subtract by, rA is the register to subtract from, and rD is result is stored.
- #imm16 is a numerical value to subtract by, from 0-4095. Keep in mind the value can be positive or negative.

- Basically:
- rD = rA - rB
  or
- rD = rA - #imm16

- Allows you to subtract one thing from another, whether it be registers or given values.

SUB Instruction (EXs)

- Examples:
  1. SUB W0, W1, W2 ; Stores the result of W1 - W2 to Register W0
  2. SUB X0, X10, X11 ; Stores the result of X10 - X11 to Register X0
  3. SUB S2, S23, #0xFF ; Stores the result of S23 - 0xFF (255) to Register S2
  4. SUB S2, S23, #255 ; Stores the result of S23 - 255 to Register S2
  5. SUB W0, W0, W1 ; Stores the result of W0 - W1 back into Register W0. Basically, subtract W0 by W1
  6. SUB W0, W0, #6 ; Stores the result of W0 - 6 back into Register W0. Basically, subtract W0 by 6
  7. SUB W0, W0, #-6 ; Stores the result of W0 - (-4) back into Register W0. Basically, subtract W0 by -4. Really adds 4
  8. SUB W0, W0, #-0x6 ; Stores the result of W0 - -(0x4) back into Register W0. Basically, subtract W0 by -0x4. Really adds 0x4

- We have eight instructions here. Realize that the number and period right next to the instructions aren't part of the instruction: they're just how I'm showing each line of code. The semicolon represents a code comment. Code comments aren't included in the actual instruction: they're just sometimes added by the developer to keep track of what their instructions do. In this case, I used comments to indicate what the above instructions were doing.

- In Line 1, assume W0 = 100, W1 = 30, and W2 = 15. W1 - W2 returns a result of 15. W0 now equals 15. In this case, the value of W0 didn't matter: it is just being used as a register to store our result in. It's likely 100 was the previous result of some other function. Line 2 is executed basically the same way, with different things to subtract by but the same destination register, but the register is being told to store doublewords instead of words.
- In Line 3, assume S2 = 0 and S23 = 256. S23 - 0xFF (255) gives a result of 1. S2 is now equal to 1. Line 4 does the same thing, but instead of subtracting a hexadecimal value, a decimal value is being subtracted. Both functions lead to the same result: just showing you that you can write in a value as either decimal or hexadecimal by using the above notation.
- In Line 5, assume W0 = 42 and W1 = 4. W0 - W1 returns a result of 38, so W0 is now equal to 38. Unlike in Lines 1-4, what the destination register originally equaled didn't matter. However, considering the destination register is also used in the calculation here, the destination register partially determines the result of the outcome. Line 6 is basically the same thing, except instead of using a register for the calculation, the number 6 is used. W0 - 6 returns a result of 36, so W0 is now equal to 36.
- Lines 7-8 are basically clones of Line 6 to show you that you can place negative values in as well.

ADD Instruction

- ADD rD, rA, rB
  or
- ADD rD, rA, #imm16

- Where rB is the register to add, rA is the register to add to, and rD is result is stored.
- #imm16 is a numerical value to add to, from 0-4095. Keep in mind the value can be positive or negative.

- Basically:
- rD = rA + rB
  or
- rD = rA + #imm16

- Allows you to add one thing to another, whether it be registers or given values.
- ADD works the same way as SUB, but instead of subtracting, it adds. No example for this instruction will be given.

MOV Instruction

- MOV rD, rA
  or
- MOV rD, #imm16

- Where the value in rD is set to the value in rA.
- #imm16 is a numerical value to set the value of rD to, from 0-65535.

- Basically:
- rD = rA
  or
- rD = #imm16

- Really, it just copies, or ‘moves’ a value from one place to another.
- In x86-64, MOV allows you to move an address to a register, a value contained by address to a register, register to register, a given value to a register, a value contained by register to an address, and a given value to an address, whether it be a pointer or static address.
- In ARM, the current language, we can only use MOV to move a given value to a register or a register to a register. The LDR and STR functions, the ones directly after this, cover the rest of what x86-64’s MOV was able to do.
- For the rest of the video series, I will only write code in ARM. Assume any code I write throughout the rest of the video series is in ARM. I basically gave you x86 as examples early on to prepare you for this. Even further code type videos after this one, assuming I make them, will be in ARM.

MOV Instruction (EXs)

- Examples:
  1. MOV W0, W1 ; Sets Register W0 equal to Register W1
  2. MOV W0, #0x7F ; Sets Register W0 equal to 0x7F (255)
  3. MOV X27, #255 ; Sets Register X27 equal to 255
  4. MOV S0, WZR ; Sets S0 equal to WZR, which is 0
  5. MOV D4, XZR ; Sets D4 equal to XZR, which is 0
  6. MOV D4, WZR ; Sets the first 32-bits of D4 equal to 0, while preserving the last 32-bits
  7. MOV X27, #-255 ; Sets Register X27 equal to -255
  8. MOV W0, #-0x7F ; Sets Register W0 equal to -0x7F (-255)

- In Line 1, assume W0 = 99 and W1 = 42. Now, W0 is equal to 42. Line 2 sets W0 to 255. Line 3 sets X27 to 255. This is rather self-explanatory.
- Lines 4-5 are pretty self-explanatory. However, Line 6 is pretty strange. Basically, WZR means to carry 32-bits of zeroes into the destination register. XZR means to carry 64-bits of zeroes into the destination register. Considering Register D4 is currently holding 64-bits, WZR only stores zeroes to the first half of the register. This is why you can specify the length of the Zero Register: sometimes you have to set a 32-bit or a 64-bit register to zero. You can't zero a 64-bit register with WZR, it must be with XZR. Likewise, you can't zero a 32-bit register with XZR, as the destination register might overwrite bits of the rest of the register that are actually important. You must zero the register with WZR instead.
- Lines 7-8 are really clones of Lines 2-3 to show that you can move a negative value into a register as well.

LDR (Load) Instruction

- LDR is going to be a slight bit different from our other instructions. It can be written as:
- LDRT rA, [rD] ; No offset

  or
- LDRT rA, [rD, #offset] ; immediate offset

  or
- LDRT rA, [rD, #offset]! ; pre-indexed offset

  or
- LDRT rA, [rD], #offset ; post-indexed offset

- Where rD is the destination register, rA is a register that contains an address, and #offset is the amount to add or subtract the address by to get the desired address. #offset can be anything from -4095 to 4095 if you're loading a word or a byte or -255 to 255 if you're loading anything else.
- Basically:
- rD = Address contained by rA

  or
- rD = Address contained by rA + #offset

- Pre-indexed and post-indexed offsets are out of the scope of this lesson, so we'll only be focusing on the cases where we have no offset or an immediate (which means the same as given) offset. Just know that the instruction can be written like that as well.
- The function can be used to load a register with an address or load a register with the value contained by an address.
- T is the data-type, where B is an Unsigned Byte, H is an Unsigned Halfword, D is a Doubleword, Q is a Quadword, and omitting the T term is a Word. T can also be SB for a Signed Byte or SH for a Signed Halfword.
- Generally, D isn't the term of T, as you could just use an X register for rD and rA and accomplish the same thing using LDR instead of LDRD.
- You'll never need to write to instructions containing LDRT, and the basics of this tutorial will not have you inserting LDRT instructions. However, some ARM Codes might insert LDRT, but those are more advanced cheat codes.

LDR (Load) Instruction (EXs)

- Examples:

1. LDR X0, [X1] ; Loads Register X0 with the address contained by Register X0. Register X0 now also contains the address
2. LDR X1, [X9, #0x40] ; Loads Register X1 with the address contained by Register X9 + 0x40

3. LDR X5, [X10, #0x8] ; Loads Register X5 with the address contained by Register X10 + 0x40
   LDRB X5, [X5] ; Loads Register X5 with the first byte contained by the address in Register X5

4. LDR X5, [X10, #0x8] ; Loads Register X5 with the address contained by Register X10 + 0x8
   LDRH X5, [X5] ; Loads Register X5 with the first halfword contained by the address in Register X5

5. LDR X5, [X10, #0x8] ; Loads Register X5 with the address contained by Register X10 + 0x8
   LDRSB X5, [X5] ; Loads Register X5 with the first signed byte contained by the address in Register X5

6. LDR X5, [X10, #0x8] ; Loads Register X5 with the address contained by Register X10 + 0x8
   LDR W5, [X5] ; Loads Register W5 with the first word contained by the address in Register X5

7. LDR X5, [X10, #0x8] ; Loads Register X5 with the address contained by Register X10 + 0x8
   LDR X5, [X5] ; Loads Register X5 with the doubleword contained by the address in Register X5

8. LDR X5, [X10, #0x8] ; Loads Register X5 with the address contained by Register X10 + 0x8
   LDRD X5, [X5] ; Loads Register X5 with the doubleword contained by the address in Register X5

9. LDR X5, [X10, #0x8] ; Loads Register X5 with the address contained by Register X10 + 0x8
   LDRD D5, [X5] ; Loads Register X5 with the doubleword contained by the address in Register X5

10. LDR X5, [X10, #0x8] ; Loads Register X5 with the address contained by Register X10 + 0x8
    LDRD S5, [X5] ; Loads Register X5 with the doubleword contained by the address in Register X5

11. LDR X1, [X9, #-0x40] ; Loads Register X1 with the address contained by Register X9 - 0x40

- In Line 1, assume X0 = 0 and X1 = Memory Address 0x8000000000. Remember, Nintendo Switch addresses are 64-bit (Doubleword) and are 10 hexadecimal characters long. X0 now contains Memory Address 0x8000000000. Line 2 follows similar logic. Assume X1 = 0 and X9 = Memory Address 0x4900000000. X1 now contains the Memory Address 0x4900000000 + 0x40. X1 now contains Memory Address 0x4900000040. Line 11 follows the same logic, but by subtracting 0x40 instead of adding 0x40.
- Lines 3-10 are pretty much the same thing, but have two instructions instead of one. In Lines 3-10, assume X5 = 0 and X10 = 0x007E000000. In instruction 1 of 2, X5 is now contains the Memory Address 0x007E000000 + 0x8, or Memory Address 0x007E000008. In instruction 2 of 2, the value of the address contained by Memory Address 0x007E000008 is being loaded. The only difference between them is the data-type of the value being loaded. Line 3 loads a byte from X5, Line 4 loads a Halfword, Line 5 loads a Signed Byte, Line 6 loads a Word, and Line 7 loads a Quadword. Line 8, although set to store a Doubleword, is fundamentally the same as Line 7. This is because the Doubleword register was specified in Line 7, making LDRD kind of redundant. This is why LDRD is almost never used. Lines 9-10 are to show that you can load Float and Double values from memory as well. This can also be applied to other scalar and vector registers.
- Line 11 is to show that you can use negative offsets as well as positive ones.
- If you ever see a second LDR instruction below another LDR instruction with the register number being the same, even if the letter is different, assume that the value contained by the address of that register is what's being loaded.
- LDR instructions pretty much always use Xn on the rA register on the Nintendo Switch, as all Nintendo Switch memory addresses are Doublewords.

LDP (Load Pair of Registers) Instruction

- LDP is basically LDR, but with two registers instead of one and no data-type specification. It can be written as the following:
- LDP rB, rC, [rA] ; No offset
  or
- LDP rB, rC, [rA, #imm] ; signed offset (Can be Negative or Positive)
  or
- LDP rB, rC, [rA, #imm]! ; pre-indexed offset
  or
- LDP rB, rC, [rA], #imm ; post-indexed offset

- Where rB and rC are either Wn, Xn, Sn, or Dn registers. If rB is Wn, rC must also be a Wn register. So on with the other register naming schemes. rA is the address to load from. #imm is a value from -256 to 252 (if 32-bit) or -512 to 504 (if 64-bit) that is added to the address as an offset to get the real address. Negative values only apply in the signed offset version of the instruction.

- Basically:
- rB = Address contained by rA + #imm
  and
- rC = Address contained by rA + #imm + 0x4 (if 32-bit register)
  or
- rC = Address contained by rA + #imm + 0x8 (if 64-bit register)

- Once again, I'll only be giving examples of this instruction with no offset and with a signed offset.
- The instruction allows you to load an address and the address right next to the current address (0x4 is 32-bit or 0x8 if 64-bit) into two registers. The instruction can also be used to load the values of an address and the value of the address right next to the current address into two registers.
- LDP only works with 32-bit or 64-bit values, making it unnecessary to have a data-type specification.
- You'll never need to write to an instruction with LDP in it. There may theoretically be some cheats that write to these instructions, but I've never seen them. I've never had to write to these instructions, anyways.

LDP (Load Pair of Registers) Instruction (EXs)

- Examples:
  1. LDP X0, X1, [X2] ; Loads Register X0 with the Address contained by X2 and loads Register X1 with the Address contained by X2 + 0x8
  2. LDP X0, X1, [X2, #-510] ; Loads Register X0 with the Address contained by X2 - 510 and loads Register X1 with the Address contained by X2 - 510 + 0x8
  3. LDP X0, X1, [X2, #0x7C] ; Loads Register X0 with the Address contained by X2 + 0x7F and loads Register X1 with the Address contained by X2 + 0x7C + 0x8

  4. LDR X4, [X5, #0x80] ; Loads Register X4 with the Address Contained by X5 + 0x80
     LDP D5, D6, [X4] ; Loads the Double contained by the Memory Address in Register X4 into Register D5 and loads the Double contained by Register X4 + 0x8 into Register D6

  5. LDR X4, [X5, #0x80] ; Loads Register X4 with the Address Contained by X5 + 0x80
     LDP S5, S6, [X4] ; Loads the Float contained by the Memory Address in Register X4 into Register S5 and loads the Float contained by Register X4 + 0x8 into Register S6

  6. LDR X4, [X5, #0x80] ; Loads Register X4 with the Address Contained by X5 + 0x80
     LDP W5, W6, [X4] ; Loads the Word contained by the Memory Address in Register W4 into Register W5 and loads the Word contained by Register X4 + 0x8 into Register D6

- For Line 1, assume X2 = Memory Address 0x7F00000000. The Memory Address 0x7F00000000 is loaded into X0 and the Memory Address 0x7F00000008 is loaded into X1. Lines 2 and 3 follow the same logic, but with offsets instead. In Line 2, the Memory Address 0x7F00000000 - 510 is loaded into X0 and the Memory Address 0x7F00000000 - 510 + 0x8 is loaded into X1. In Line 3, the Memory Address 0x7F00000000 + 0x7C is loaded into X0 and the Memory Address 0x7F00000000 + 0x7C + 0x8 is loaded into X2. Lines 2-3 are to show that you can have positive and negative offsets set in either decimal or hexadecimal notation.
- For Line 4, assume X5 = Memory Address 0x8000000000. The first part of Line 4 loads the Memory Address 0x8000000000 + 0x80 into X4. Then, the second part of Line 4 loads the Double contained by Memory Address 0x8000000000 + 0x80 into D5 and then loads X5 with the Double contained by Memory Address 0x8000000000 + 0x80 + 0x8. Lines 5 and 6 follow the same logic, except Words and Floats are used and the value to add to Register 6 is 0x4 instead of 0x8 due to being 32-bit. Just how two LDRs were used to load a value from an address in the last two slides, an LDR and an LDP is used to load a value from an address and the address right next to it.

- Think of LDP as two LDRs in one. Basically, LDP X0, X1, [X2] is equivalent to the two following instructions:
  LDR X0, [X2]
  LDR X1, [X2, #0x8]

STR (Store) Instruction

- STR is an instruction very similar to LDR, mainly because they're often used very shortly after each other. It can be written as:
- STRT rA, [rD] ; No offset
  or
- STRT rA, [rD, #offset] ; immediate offset
  or
- STRT rA, [rD, #offset]! ; pre-indexed offset
  or
- STRT rA, [rD], #offset ; post-indexed offset

- Where rD is the destination register, rA is a register that contains a value, and #offset is the amount to add or subtract the address by to get the desired address. #offset can be anything from -4095 to 4095 if you're storing a word or a byte or -255 to 255 if you're storing anything else.

- Basically:
- rD = Value contained by rA
  or
- rD + #offset = Value contained by rA

- Pre-indexed and post-indexed offsets are out of the scope of this lesson, so we'll only be focusing on the cases where we have no offset or an immediate (which means the same as given) offset. Just know that the instruction can be written like that as well.
- The function is used to move the value of a register to an address in memory.
- T is the data-type, where B is an Unsigned Byte, H is an Unsigned Halfword, D is a Doubleword, Q is a Quadword, and omitting the T term is a Word. T can also be SB for a Signed Byte or SH for a Signed Halfword.
- Generally, D isn't the term of T, as you could just use an X register for rD and rA and accomplish the same thing using LDR instead of LDRD.
- Most of the ASM Codes you write will either change an STR instruction or they'll change an instruction right before STR in order to make sure STR stores the value you specified.

STR (Store) Instruction (EXs)

- Examples:
  1. STR W0, [X1] ; Stores the Word from Register W0 into the value of the Memory Address contained by Register X1
  2. STR W1, [X9, #0x40] ; Stores the Word from Register W1 into the value of the Memory Address contained by Register X9 + 0x40
  3. STR W1, [X9, #-0x40] ; Stores the Word from Register W1 into the value of the Memory Address contained by Register X9 - 0x40
  4. STR S4, [X19] ; Stores the Float value from Register S4 into the value of the Memory Address contained by Register X19
  5. STR D6, [X20] ; Stores the Double value from Register D6 into the value of the Memory Address contained by Register X20
  6. STRB W2, [X18] ; Stores the First Byte from Register W2 into the value of the Memory Address contained by Register 18
  7. STRSB W2, [X18] ; Stores the First Signed Byte from Register W2 into the value of the Memory Address contained by Register 18
  8. STRH W2, [X18] ; Stores the First Halfword from Register W2 into the value of the Memory Address contained by Register 18
  9. STRSH W2, [X18] ; Stores the First Signed Halfword from Register W2 into the value of the Memory Address contained by Register 18
  10. STR X2, [X18] ; Stores the Doubleword from Register W2 into the value of the Memory Address contained by Register 18
  11. STRD X2, [X18] ; Stores the Doubleword from Register W2 into the value of the Memory Address contained by Register 18
  12. STR WZR, [X21] ; Stores the 32-bit Zero Register into the value of the Memory Address contained by Register 21.
  13. STR XZR, [X21] ; Stores the 64-bit Zero Register into the value of the Memory Address contained by Register 21.

- In Line 1, assume W0 = 40 and X1 = Memory Address 0x8000000000. Remember, Nintendo Switch addresses are 64-bit (Doubleword) and are 10 hexadecimal characters long. The Memory Address 0x8000000000 now has 40 as its value. Line 2 follows similar logic. Assume W1 = 40 and X9 = Memory Address 0x4900000000. The Memory Address 0x4900000040 now has 40 as its value. Line 3 follows the same logic, but by subtracting 0x40 instead of adding 0x40.
- In Line 4, assume S4 = 1 (0x3F000000) in float and X19 = Memory Address 0x8000008000. The Memory Address X19 now has 0x3F000000 as its value. Line 5 works basically the same way, but with a Double value instead of a Float Value and with different registers.
- Lines 6-9 all basically work the same way, except with the data-type they're storing. Assume X18 = Memory Address 0x4F00000000. Line 6 stores the first Byte of W2, Line 7 stores the first Signed Byte of W2 into the value of Memory Address 0x4F00000000, Line 8 stores the first Halfword of W2 into the value of Memory Address 0x4F00000000, and Line 9 stores the first Signed Halfword of W2 into the value of Memory Address 0x4F00000000.
- Line 10, although set to store a Doubleword, is fundamentally the same as Line 11. This is because the Doubleword register was specified in Line 10, making STRD kind of redundant. This is why STRD is almost never used.
- Lines 12 and 13 are storing the Zero Register to the Address contained by X21, which means the value of that address now becomes zero.
- STR functions on the Nintendo Switch, if they're not scalar values, will mostly have a Wn register or a Sn register as the first term, as most memory addresses on the Nintendo Switch contain values that are Words, Floats, Halfwords, or Bytes. The first term can also have a Xn or a Dn register if the value is a Doubleword or a Double. The second term will pretty much always be a Xn register, as all memory addresses on the Nintendo Switch are Doublewords.
- Generally, LDR will be used to calculate the address to store to, and then STR will be used to store a value to that address. There might also be instructions before the STR and after the LDR in order to calculate the value to store the address, whether that be through things like ADD, SUB, MOV, or by calling other functions to handle it.

STP (Store Pair of Registers) Instruction

STP is an instruction very similar to LDP, mainly because they're often used very shortly after each other. It can be written as:

- STP rA, rB, [rD] ; No offset
  or
- STP rA, rB, [rD, #imm] ; signed offset
  or
- STP rA, rB, [rD, #imm]! ; pre-indexed offset
  or
- STP rA, rB, [rD], #imm ; post-indexed offset

- Where rA and rB are either Wn, Xn, Sn, or Dn registers. If rA is Wn, rB must also be a Wn register. So on with the other register naming schemes. rD is the address to store to. #imm is a value from -256 to 252 (if 32-bit) or -512 to 504 (if 64-bit) that is added to the address as an offset to get the real address. Negative values only apply in the signed offset version of the instruction.

- Basically:
- rD = Value contained by rA
  and
- rD + 0x4 (if 32-bit) = Value contained by rB
  or
- rD + 0x8 (if 64-bit) = Value contained by rB

- Once again, I'll only be giving examples of this instruction with no offset and with a signed offset.
- The function is used to move the value of a register to an address in memory and the value of another register to the address in memory right next to the previous address (which is 0x4 or 0x8 away, for 32-bit or 64-bit data-types, respectively). Allows you to write to two addresses at once with two different values.
- STP only works with 32-bit or 64-bit values, making it unnecessary to have a data-type specification.
- I've never had to write to an STP function in order to make an ASM Cheat. Still, it's theoretically possible you might have to, especially if two values related to each other are stored right next to each other. So, it's good to at least know about the STP instruction.

STP (Store Pair of Registers) Instruction (EXs)

- Examples:
  1. STP W0, W1, [X2] ; Store the Word in Register W0 to the Memory Address Contained by Register X2 and the value in Register W1 to X2 + 0x4
  2. STP W0, W1, [X2, #0x80] ; Store the Word in Register W0 to the Memory Address Contained by Register X2 + 0x80 and the Word in W1 to X2 + 0x80 + 0x4
  3. STP W0, W1, [X2, #-128] ; Store the Word in Register W0 to the Memory Address Contained by Register X2 - 128 and the Word in W1 to X2 - 128 + 0x4
  4. STP WZR, WZR, [X2] ; Stores the 32-bit Zero Register to the Memory Address Contained by Register X2 and X2 + 0x4
  5. STP XZR, XZR, [X2] ; Stores the 64-bit Zero Register to the Memory Address Contained by Register X2 and X2 + 0x8
  6. STP WZR, W4, [X2] ; Stores the 32-bit Zero Register to the Memory Address Contained by Register X2 and stores the Word in Register W4 to X2 + 0x4
  7. STP X1, XZR, [X2] ; Stores the Doubleword in Register X1 to the Memory Address Contained by X6 and stores the 64-bit Zero Register to X6 + 0x8
  8. STP X0, X1, [X2] ; Store the Doubleword in Register X3 to the Memory Address Contained by Register X5 and the Doubleword in Register X4 to X5 + 0x8
  9. STP W0, W0, [X2] ; Stores the Word in Register W0 to the Memory Address Contained by Register X2 and to X2 + 0x4
  10. STP D3, D4, [X5] ; Store the Float in Register D3 to the Memory Address Contained by Register X5 and the Float in Register D4 to X5 + 0x8
  11. STP S3, S4, [X5] ; Store the Double in Register S3 to the Memory Address Contained by Register X5 and the Double in Register S4 to X5 + 0x8

- In Line 1, assume X2 = Memory Address 0x6000000000, W0 = 2147483647 and W1 = 1. The value in Memory Address 0x6000000000 is set to 2147483647 and 0x6000000000 + 0x4 is set to 1. Lines 2-3 basically show the same thing, but with offsets instead. In Line 2, the value in Memory Address 0x6000000000 + 0x80 is set to 2147483647 and the value in Memory Address 0x6000000000 + 0x80 + 0x4 is set to 1. In Line 3, the value in Memory Address 0x6000000000 - 128 is set to 2147483647 and the value in Memory Address 0x6000000000 - 128 + 0x4 is set to 1. Lines 2-3 are to show that you can have positive and negative offsets set in either decimal or hexadecimal notation. Line 8 is there to show that you can use Doublewords and Words in the STP instruction.
- Lines 4-7 are to show that you can use the Zero Register in the STP instruction. You can decide to use a GPR and a zero register, the zero register twice, or two GPRs in the STP instruction.
- In Line 9, assume X2 = Memory Address 0x6000000000 and W0 = 2147483647. The value in Memory Address 0x6000000000 is set to 2147483647 and the value in Memory Address 0x6000000000 is also set to 2147483647. Line 9 is there to show you can use the same register twice in the STP instruction. Something like Line 9 might be used to store a value to a visual address and the actual address at the same time. This would likely be the case if the visual address and actual address of a particular value were 0x4 or 0x8 away from each other.
- Lines 10 and 11 are there to show that you can also use the STP instruction to store Float and Double values.

An Image of LDR (and LDR) and STR (and STP)

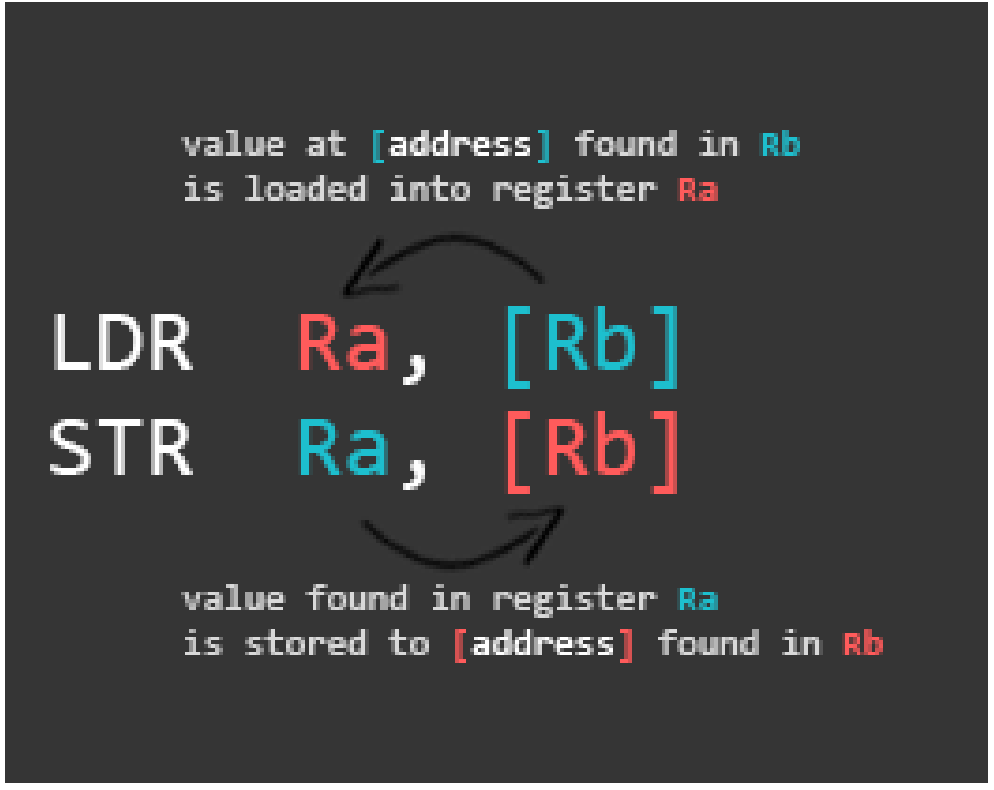

Image Taken From: https://azeria-labs.com/memory-instructions-load-and-store-part-4/

With the exception of STR and STP, pretty much all functions work like LDR, where the Register on the right goes into the Register(s) on the left.

Bringing MOV, LDR, and STR All Together

- Here is an example of a function that might be used to calculate HP after a player is damaged:

- _CalculateHP:
  LDR X18, [X1, #0x3C] ; Calculates Address of HP (Address contained by Register X1 + 0x3C) and Loads it Into Register X18
  LDR W17, X18 ; Loads Current Value of HP into Register W17
  SUB W17, W17, W16 ; Subtracts Current Value of HP from Current Value of Damage (in Register W16) from W17, stores result in W17
  STR W17, [X18] ; Stores the Calculated HP Address from the Last Instruction into the Memory Address Contained by Register X18
  RET ; Return from function call. Basically, end of function. I'll go into slightly more detail about this next slide.

- Here is an example of a function that might be used to insta-kill the player whenever the player hits a kill zone:

- _KillHP:
  LDR X18, [X1, #0x3C] ; Calculates Address of HP (Address contained by Register X1 + 0x3C) and Loads it Into Register X18
  MOV W17, WZR ; Moves the Zero Register into W17. W17 now equals 0.
  STR W17, [X18] ; Stores Register W17, which is 0, into the player's HP Address. This kills the player.
  RET; Return from function call. Basically, end of function. I'll go into slightly more detail about this next slide.

- I also said previously that x86's MOV can be used to move a given value into an address. That cannot be done in one step in ARM like it can be done in x86, but it can still be done. Here's an example of me making the program store the value 127 (0x7F) to an address:

  MOV W0, #0x7F ; Sets W0 equal to 0x7F (127).
  STR W0, [X19] ; Stores W0 (currently 127) into [X19], our target address's value. Our target address's value is now equal to 127.

RET Instruction

- RET stands for Return. Basically, RET ‘returns’ to the previous function. It might as well stand for END.
- There is no special notation or usage of the instruction, it’s literally just this:

- RET

BLR Instruction

- BLR stands for Branch with Link to Register. In a later section of the video, I'll get even more in-depth into Branching. Basically, branching is "branching out to another function." That is, a function call. This is one of the instructions that places a new instruction onto the stack pointer by calling it. As previously said, RET would be the instruction to end a function called by BLR.

- This is the notation of BLR, at least for the Nintendo Switch:

- BLR Xn

- Addresses on the Nintendo Switch are always Doublewords. The BLR function calls a function located at the address contained by the X register on the Nintendo Switch, pausing the current function.

- Personally, I almost always see a BLR before a STR function. This means you can replace a BLR with a MOV, allowing you to load any value into whatever address you're targeting. I've done this before. This would keep the function called by BLR from being called, keeping any sort of calculation from happening to begin with. If you want to just keep the calculation from being called without moving a particular value into it, just use a NOP instruction. I've never had to do this with a BLR instruction before, though.

BLR Instruction (EX)

- Remember our previous function to calculate HP? Let's spice it up with a BLR:

- _CalculateHP:
  LDR X18, [X1, #0x3C] ; Calculates Address of Function (Address contained by Register X1 + 0x3C) and Loads it Into Register X18
  BLR X18 ; Branches to Register X18, which contains the address of a function to Calculate HP. Calls that function to calculate HP. Pauses function before STR, so STR doesn't execute yet.
  STR W15, [X17] ; Stores the HP Value calculated by the function loc_8000000000 into the HP Address calculated by the same function.
  RET ; Return from function call.

- Let's say X18 is currently 0x8000000000. The function from this memory address is called.

- sub_8000000000: ; Function at Memory Location 0x8000000000
  LDR X17, [X1, #0x40] ; Loads Calculated Damage Address into Register X17 from the Memory Address X1 + 0x40. The value for X17 is saved, allowing the result to be stored into the HP address in the _CalculateHP function.
  LDR W16, X17 ; Loads Calculated Damage Value from Damage Address into Register W16
  LDR X15, [X1, #0x44] ; Loads HP Address into Register X15 from the Memory Address X1 + 0x44
  LDR W15, X15 ; Loads HP Value from HP Address into Register W15
  SUB W15, W15, W16; Subtracts Calculated Damage from HP, storing result in HP Register
  RET; Returns from function call. Stack Pointer returns to CalculateHP. After the Stack Pointer returns to CalculateHP, the STR instruction is immediately executed.

NOP Instruction

- NOP Stands for No Operation. It does exactly what it says, which is absolutely nothing.
- There is no special notation or usage of the instruction, it's literally just this:
- NOP
- It is equivalent to the instruction MOV W0, W0. Assuming W0 = 1, setting W0 to W0 would make W0 equal to, 1. Basically, nothing happened.
- NOP is useful because of two reasons. For one, it allows a coder to place code that does nothing in order to fill the space of the program without doing anything to compromise the programming. Secondly, it's useful to us as code creators because it allows us to stop an instruction from doing anything.
- Basically, we could replace an instruction in code with NOP in order to render it useless. Whenever we get to the section of actually writing our own ASM Cheats, we'll be changing instructions to NOP to see if that's the instruction that writes to the address or function we want. By using NOP, we can do a process of elimination on instructions, just like we did with addresses.. Whenever we NOP a function and our targeted address either stops updating or changes to zero, we know we've found the instruction we're looking for.

Condition Codes

- Condition Codes are simply abbreviations that come after the abbreviation of an instruction. There are 16 instructions. I will only cover nine in this video for the sake of simplicity. Here are the following condition codes for this video:

- EQ (Equal)
- NE (Not Equal)
- MI (Negative)
- PL (Positive or Zero)
- GE (Greater Than or Equal To)
- LT (Less Than)
- GT (Greater Than)
- LE (Less Than or Equal To)
- AL or Omitted (Always)

- For instance, if you were to use the instruction ADDEQ (which is a combination of the instruction ADD and the condition code EQ), addition would only occur if a particular register was equal to another register from a previous condition check. I'll explain this a little more in-depth later, just keep this condition codes in the back of your mind.

CMP (Compare) Instruction

- Usually comes before some instruction that executes based on a particular condition. It can be written as follows:

- CMP rA, rB
  or
- CMP rA, #imm

- Where rA is a register to Compare to and rB is a register to compare from. #imm a number to compare rA to if the programmer decides to compare to a given number instead of a register.
- I do not personally know the range of #imm, as the websites that document these functions haven't appeared to have documented that.

- Basically:
- Compare rA to rB
  or
- Compare rA to a given number

- The instruction behaves a lot like SUB, but doesn't store the result of the calculation.

- The first instruction basically does this: rA - rB
- The second instruction basically does this: rA - #imm

- Then, certain flags in memory are set based on the outcome of that calculation.
- If the result of rA - rB or rA - #imm was 0, that means both numbers were either zero or rA is equal to rB/#imm. Any instruction directly after the CMP instruction with a EQ (Equal), PL (Positive or Zero), GE (Greater Than or Equal To), or LE (Less Than or Equal To) condition code would execute.
- If the result of rA - rB or rA - #imm was greater than 0, which means that either rA is larger than rB/#imm or that means rB or rB/#imm was such a large negative integer that the result was positive, then any instruction directly after the CMP instruction with a NE (Not Equal), PL (Positive or Zero), GE (Greater Than or Equal To), or GT (Greater Than) condition code would execute.
- If the result of rA - rB or rA - #imm was less than 0, which means rB or #imm is larger than rA or that rA was negative and rB/#imm isn't a large enough negative integer, so the result is negative. Any instruction directly after the CMP instruction with a NE (Not Equal), MI (Negative), LE (Less Than or Equal To), or LT (Less Than) condition code would execute.
- No matter the result, an instruction with an AL or an omitted condition code will execute, regardless of what the CMP instruction returns.
- I've never had to do this, but you could change a CMP instruction to change if a function is called or not.

CMN (Compare Negative) Instruction

- Usually comes before some instruction that executes based on a particular condition. It can be written as follows:

- CMN rA, rB
  or
- CMN rA, #imm

- Where rA is a register to Compare to and rB is a register to compare from. #imm a number to compare rA to if the programmer decides to compare to a given number instead of a register.
- I do not personally know the range of #imm, as the websites that document these functions haven't appeared to have documented that.

- Basically:
- Compare rA to rB
  or
- Compare rA to a given number

- The instruction behaves a lot like ADD, but doesn't store the result of the calculation.

- The first instruction basically does this: rA + rB
- The second instruction basically does this: rA + #imm

- Then, certain flags in memory are set based on the outcome of that calculation.
- If the result of rA + rB or rA + #imm was 0, that means both numbers were either zero or you had an integer and its opposite added together. Any instruction directly after the CMN instruction with a EQ (Equal), PL (Positive or Zero), GE (Greater Than or Equal To), or LE (Less Than or Equal To) condition code would execute.
- If the result of rA + rB or rA + #imm was greater than 0, which means that at least one register/number was a non-zero number and that a negative integer large enough to go under zero wasn't present, then any instruction directly after the CMN instruction with a NE (Not Equal), PL (Positive or Zero), GE (Greater Than or Equal To), or GT (Greater Than) condition code would execute.
- If the result of rA + rB or rA + #imm was less than 0, which means that at least one register/number was a negative integer large enough to go under zero, then any instruction directly after the CMN instruction with a NE (Not Equal), MI (Negative), LE (Less Than or Equal To), or LT (Less Than) condition code would execute.
- No matter the result, an instruction with an AL or an omitted condition code will execute, regardless of what the CMN instruction returns.
- I've never had to do this, but you could change a CMN instruction to change if a function is called or not.

A Little More on CMP and CMN

- Programming-wise, instructions like CMP or CMN are basically the “If” part “If… then statements”.
- An “If… then statement”, for those unfamiliar with programming, is an instruction that says ‘If certain condition is met, then execute certain line of code. Otherwise, do either something else (else if) or nothing.
- After this, I’ll get into what makes up the “Then” part of an instruction.

- I said previously that CMP and CMN do not store the result to memory. That is sort of true, and sort of not. Whether the number from the result is negative, zero, or positive is stored, but the overall result of the calculation is not. Whether or not the number was negative, zero, or positive is stored using two flags in the Switch’s Processor: the Z flag and the N flag. It isn’t stored into the Nintendo Switch’s memory itself, but rather, on the temporary memory of the Nintendo Switch’s processor.
- A flag is something that can either be a 0 or 1, 0 for false and 1 for true.
- The Z flag stands for Zero and the N flag stands for Negative. After a CMP or CMN result is executed, these two flags are updated to what I’ve typed below, based on what the result was:

- If Z=0 and N=1, then the result was Negative.
- If Z=1 and N=0, then the result was Zero.
- If Z=0 and N=0, then the result was Positive.

- Then, whenever an instruction with a condition code is loaded, these two flags are checked. If the flags match up, then the instruction is executed. Otherwise, the instruction is skipped. Furthermore, when an instruction is executed that does meet the condition flags, the flags are cleared afterwards, not set again until another comparison instruction of some sort is used.
- There are other ways to compare two values without using CMP or CMN. Excluding one thing near the end of the ARM lesson, I will not be covering other ways to do comparisons. If you’re curious about this, look up some of the ARMv8 Opcode Documentation.

B (Branch) Instruction

- A Branch function is basically BLR but with a specified address instead of using the address contained by a particular register.
- The format of the instruction is basically this:
- B.(cond) #0x??????????
- Where (cond) is one of the previously specified condition codes.
- The Branch instruction is basically just a function call.
- Variations of the Branch Instruction:
  B
  BEQ
  BNE
  BMI
  BPL
  BGE
  BLT
  BGT
  BLE
- Might also appear with a period before the condition code, like:
  B.LT (Still means the same thing as BLT)
- Rewriting a Branch instruction with a NOP instruction prevents a function from being called. I've had to do this before.

B (Branch) Instruction (EX)

- Let's reuse the example we had in the BLR Instruction (EX):

- _CalculateHP:
  LDR X17, [X1, #0x40] ; Loads Calculated Damage Address into Register X17 from the Memory Address X1 + 0x40. The value for X17 is saved, allowing the result to be stored into the HP address in the _CalculateHP function.
  LDR W16, X17 ; Loads Calculated Damage Value from Damage Address into Register W16
  CMP W16, 0 ; Checks to see if Calculated Damage is Zero
  BNE #0x8000000000 ; If Calculated Damage isn't Zero (that is, there's actually damage), then branch to function located at 0x8000000000. Makes it so storing to HP Address only occurs if damage occurs. It'd be pointless to repeatedly store the same HP value to the address repeatedly without a change.
  RET ; Return from function call.

- Let's say X18 is currently 0x8000000000. The function from this memory address is called.

- sub_8000000000: ; Function at Memory Location 0x8000000000
  LDR X15, [X1, #0x44] ; Loads HP Address into Register X15 from the Memory Address X1 + 0x44
  LDR W15, X15 ; Loads HP Value from HP Address into Register W15
  SUB W15, W15, W16; Subtracts Calculated Damage from HP, storing result in HP Register
  STR W15, [X17] ; Stores the HP Value calculated by the function loc_8000000000 into the HP Address calculated by the same function.
  RET; Returns from function call. Stack Pointer returns to CalculateHP. After the Stack Pointer returns to CalculateHP, the RET instruction is immediately executed.

BL (Branch with Link) Instruction

- The Branch with Link function basically is just calling a labeled function. Instead of setting an address, you set the name of the function, and the processor loads the address of the function in, branching to that address.
- The format of the instruction is basically this:
- BL.(cond) Function_Name
- Where (cond) is one of the previously specified condition codes.
- The Branch instruction is basically just a function call.
- Variations of the Branch Instruction:
  BL
  BLEQ
  BLNE
  BLMI
  BLPL
  BLGE
  BLLT
  BLGT
  BLLE
- Might also appear with a period before the condition code, like:
  BL.LT (Still means the same thing as BLT)
- Rewriting a Branch with Link instruction with a NOP instruction prevents a function from being called. I've had to do this before.

BL (Branch with Link) Instruction (EX)

- Let's reuse the example we had in the B Instruction (EX):

- _LoadHP:
  LDR X17, [X1, #0x40] ; Loads Calculated Damage Address into Register X17 from the Memory Address X1 + 0x40. The value for X17 is saved, allowing the result to be stored into the HP address in the _CalculateHP function.
  LDR W16, X17 ; Loads Calculated Damage Value from Damage Address into Register W16
  CMP W16, 0 ; Checks to see if Calculated Damage is Zero
  BLNE CalculateHP ; If Calculated Damage isn't Zero (that is, there's actually damage), then branch to function named CalculatedHP. Makes it so storing to HP Address only occurs if damage occurs. It'd be pointless to repeatedly store the same HP value to the address repeatedly without a change.
  RET ; Return from function call.

- Let's say X18 is currently 0x8000000000. The function from this memory address is called.

- _CalculateHP: ; Function named CalculatedHP
  LDR X15, [X1, #0x44] ; Loads HP Address into Register X15 from the Memory Address X1 + 0x44
  LDR W15, X15 ; Loads HP Value from HP Address into Register W15
  SUB W15, W15, W16; Subtracts Calculated Damage from HP, storing result in HP Register
  STR W15, [X17] ; Stores the HP Value calculated by the function loc_8000000000 into the HP Address calculated by the same function.
  RET; Returns from function call. Stack Pointer returns to CalculateHP. After the Stack Pointer returns to CalculateHP, the RET instruction is immediately executed.

Previous Instructions with Conditions Added

- SUB(S)(cond) rD, rA, rB
- ADD{S}(cond) rD, rA, rB
- MOV{S}(cond) rD, rA or MOV{S}(cond) rD, #imm16
- LDRT(cond)
- LDP(cond)
- STRT(cond)
- STP(cond)
- CMP(cond)
- CMN(cond)

- {S} updates the condition flags if included. S is basically like adding a CMP instruction inside of another instruction.
- An instruction with a condition will execute whenever that condition is meant.
- Here's a perfect example of what I mean:

SUBS W0, W0, W1 ; Does the calculation W0 - W1, then stores the result to Register W0
MOVMI W0, 0 ; If Register W0 resulted in a negative number, move 0 to Register W0. This prevents Register W0 from ever being negative. This could be used for things like Health values in game to prevent them from ever going negative.

- Realize that S can be combined with condition codes too, like this:
  SUBSEQ W0, W0, W1

- That is a combination of the instruction SUB, the S condition flag, and the EQ (Equal) condition. Basically, it checks if something was equal in the last comparison, executes the subtraction of W0 - W1, stores the result to Register W0, and then updates the condition flags based on what the result of the SUB instruction was. The condition flags would then be referenced by an instruction with a condition code, like another SUBEQ or a Branch.

- Compare functions can have a condition code in them as well. For instance, take this for example:

  CMP W0, #1 ; Compares Register W0 to the Number 1
  CMPEQ W0, W1 ; Sees if Register W0 is equal to the Number 1, and if it is, Compare Register W0 to Register W1.

- The rest are self-explanatory: like a STREQ only storing if the previous comparison gave an equal result.

- You'll probably never need to write most of these yourself: this is just so you understand these if you see them in the Assembly Code.

Atmosphere Cheat Codes Translate to ASM Instructions

- In the previous videos, I presented Atmosphere Cheat Codes using the x86 Assembler Language. This time, I'll present Atmosphere Cheat Codes using ARMv8 as the Assembler Language. You'll see some similarities between the two languages, but also a lot of differences.

- Code Type 0 (EXs):
  01000000 0D000000 0000007F ; STRB WriteRegister, [r0] or as STRB WriteRegister, [MAIN, #0x00D0000000]
  02000000 0D000000 00007FFF ; STRH WriteRegister, [r0] or as STRH WriteRegister, [MAIN, #0x00D0000000]
  04000000 0D000000 7FFFFFFF ; STR WriteRegister, [r0] or as STR WriteRegister, [MAIN, #0x00D0000000]
  08000000 0D000000 7FFFFFFF FFFFFFFF ; STRD WriteRegister, [r0] or as STRD WriteRegister, [MAIN, #0x00D0000000]
  04100000 0D000000 80000000 ; STR WriteRegister, [r0] or as STR WriteRegister, [HEAP, #0x00D0000000]

- WriteRegister is set to the value provided by the YYYYYYYY term of Code Type 0, so in the first line, WriteRegister is set to 0x7F and in line two it's set to 0x7FFF. You can see the pattern I'm going with. r0 is Register 0 of the Atmosphere Code Handler. Register 0 is set to the term provided by AAAAAAAAAA in Code Type 0. r0 is not a register designated by the Processor of the Switch, but rather, the Atmosphere Code Handler. As a result, we're not using other registers in our calculation, so we must specify the length of the data we're writing. Furthermore, this also means that Register 15 of the game you're playing will not be affected in the slightest, as that's Atmosphere's register, the not register stored in the Nintendo Switch's processor. Registers 0-15 are basically GPRs like those seen in ARM that can contain up to 64-bits.
- I'm calling this Register 15 instead of Register F. Considering those are the same number, but in decimal and hexadecimal respectively, consider them to be equivalents.
- Basically, we're storing the value we specified in Code Type 0 to the address contained in Register 0 of the Atmosphere Code Handler, which was also specified in Code Type 0.
- Keep in mind that WriteRegister isn't a real register, even in the Atmosphere Firmware. I'm using that here so it'll be accurate to ARM's code documentation, as STR cannot be used to store given values without a preceding MOV instruction. Likewise, MOV cannot be used to store given values into an address, so that's why I made the decision to use the term WriteRegister.

Atmosphere Cheat Codes Translate to ASM Instructions (Cont)

- Code Types 5, 7, and 6 (Pointer EXs):

```
580F0000 759E0000 ; LDRD r15, [MAIN, #0x00759E0000]
780F0000 0000098C ; ADD r15, r15, #0x98C
640F0000 00000000 7FFFFFFF ; STR WriteRegister, [r15] ; Writes 0x7FFFFFFF to Pointer Address [MAIN+759E0000]+98C

581F0000 759E0000 ; LDRD r15, [HEAP, #0x00759E0000]
780F1000 0000098C ; SUB r15, r15, #0x98C
640F0000 00000000 7FFFFFFF ; STR WriteRegister, [r15] ; Writes 0x7FFFFFFF to Pointer Address [HEAP+759E0000]+98C

580F0000 759E0000 ; LDRD r15, [MAIN, #0x00759E0000]
580F1000 00000040 ; LDRD r15, [r15, #0x40]
580F1000 00000000 ; LDRD r15, [r15]
780F0000 0000098C ; ADD r15, r15, #0x98C
610F0000 00000000 0000007F ; STRB WriteRegister, [r15] ; Writes 0x7F to Pointer Address [MAIN+759E0000]+40]]+98C
```

- Basically, the same logic is being used for WriteRegister as it was in Code Type 0. Basically, the code handler is storing the value stored by HEAP+0x?????????? or MAIN+0x?????????? to our specified register. ADD or SUB is then subsequently used for the part of the pointer without a bracket. For multi-level pointers, the previous value of Register 15 has itself added to, and then the value of that memory address is loaded back into Register 15. ADD or SUB is only used at the end of a pointer code if there's a part of the pointer address without a bracket. For the double pointer, the value of the address is used for a double bracket, which is why the address in Register 15 is loaded into Register 15. This is exactly what the code handler does. Finally, the STR instruction is what's storing our deisgnated value to our pointer address.
- Register 15 is essentially being used to calculate the current address of our pointer address.

Atmosphere Cheat Codes Translate to ASM Instructions (Cont)

- Code Types 8 and 2 (Conditional EX):
  80000040 ; CMP ButtonRegister, #0x80
  02000000 0D000000 00008000 ; STRHEQ WriteRegister, [MAIN, #0x00D0000000]
  20000000 ; RET

- The first line compares the current button pressed to 0x80 (the R button).
- The second line stores the Halfword 0x8000, currently contained by WriteRegister, to MAIN+0x00D0000000 if the Button pressed is currently equal to 0x80 (the R button).
- The third line terminates the code. End of function.

- If you'd like to see something disassemble an Atmosphere Cheat Code for you, you can visit this website by WerWolv, the original creator of Edizon: https://edizon.werwolv.net/cheat_emu/index.html

- This website above might help you to understand ARM better by pulling apart Atmosphere Cheat Codes.

- Basically, paste a code into the left-hand panel. There should be an ID number above the code you entered. In the right-hand console, type disas <ID> and it'll convert an Atmosphere Code to ARM Pseudocode. It may be slightly different from what I've wrote, but it's incredibly similar. For instance, if your code has an ID of 1, typing disas 1 will displace the Atmosphere code as ARM Code.

Instructions We'll Be Using/Writing To

- MOV rD, #imm16 ; We'll be using this to move a value of our choice into a register, which is then promptly stored to an address.

- BLR Xn ; This is an instruction we'll change into MOV rD, #imm16 if it's above a particular STRT instruction. This prevents the value of an address from being calculated, allowing us to insert our custom value instead.

- STRT Wn, [Xn] ; Because most values are Bytes, Halfwords, or Words. If we're looking for these data-types, we'll be looking for this instruction
  or
- STRT Sn, [Xn] ; Because Floats are often common. If we're working with a float address, we'll be looking for this instruction.

- STP Wn, Wn, [Xn] ; Same reason as STRT Wn, [Xn]
  or
- STP Sn, Sn, [Xn] ; Same reason as STRT Sn, [Xn]

- Theoretically, you may also be looking for STRT or STP with Xn or Dn as well if you're working with a Doubleword or Double address. I haven't had to do this yet, though.

- NOP ; We'll be using these to either test to see if a function contains our address or to stop a function call from being executed. We'll be NOP'ing STRT or STP to see if an address we're targeting changes to 0 or stops updating. If it does either of these, we found our target instruction. We'll also be NOP'ing BL.cond functions in order to stop functions from being executed. BL.cond have the function labeled in almost plaintext, so if you see a function you'd like to stop working in the BL.cond instruction, NOP it.

- No other instructions will be necessary to use or write to, at least for the purpsoses of this video. I've never had to write to anything but MOV, BLR, STRT, or NOP. I've never even had to write to STP, but it could be possible you might have to, so I included it. Also, NEVER write over a RET, or you'll freeze the game.