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

- 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)
- Conditional Flags
- Conditional Use of ARM Instructions (CMP, CMZ, B, BL, Previous Instructions with Conditions Added)
- 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

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.

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.
- 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.

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.

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

- 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 B4, B4, B3 ; Stores the result of B4 – B3 back into Register B4
  6. SUB W0, W0, #4 ; Stores the result of W0 – 4 back into Register W0. Basically, subtract W0 by 4
  7. SUB W0, W0, #-4 ; Stores the result of W0 – (-4) back into Register W0. Basically, subtract W0 by -4. Really adds 4.
  8. SUB W0, W0, #-0x4 ; 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 B4 = 25 and B3 = 19. B4 – B3 returns a result of 6. B4 now equals 6. Something similar happens in Line 6. Assume W0 = 42. W0 – 4 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, consider the destination register is also used in the calculation here, the destination register partially determines the result of the outcome.
- 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 Instruction (EXs)

- Examples:
  1. ADD W0, W1, W2 ; Stores the result of W1 + W2 to Register W0
  2. ADD X0, X10, X11 ; Stores the result of X10 + X11 to Register X0
  3. ADD S2, S23, #0xFF ; Stores the result of S23 + 0xFF (255) to Register S2
  4. ADD S2, S23, #255 ; Stores the result of S23 + 255 to Register S2
  5. ADD B4, B4, B3 ; Stores the result of B4 + B3 back into Register B4
  6. ADD W0, W0, #4 ; Stores the result of W0 + 4 back into Register W0. Basically, add 4 to W0
  7. ADD W0, W0, #-4 ; Stores the result of W0 + (-4) back into Register W0. Basically, add -4 to W0. Really subtracts 4
  8. ADD W0, W0, #-0x4 ; Stores the result of W0 + -(0x4) back into Register W0. Basically, add -0x4 to W0. Really subtracts 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 45. W0 now equals 45. 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 add 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 511. S2 is now equal to 511. Line 4 does the same thing, but instead of adding a hexadecimal value, a decimal value is added. 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 B4 = 25 and B3 = 19. B4 + B3 returns a result of 44. B4 now equals 44. Something similar happens in Line 6. Assume W0 = 42. W0 + 4 returns a result of 46, so W0 is now equal to 46. Unlike in Lines 1-4, what the destination register originally equaled didn't matter. However, consider the destination register is also used in the calculation here, the destination register partially determines the result of the outcome.
- Lines 7-8 are basically clones of Line 6 to show you that you can place negative values in as well.

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 W0, W0 ; Sets Register W0 equal to Register W0, effectively doing nothing
  5. MOV S0, WZR ; Sets S0 equal to WZR, which is 0
  6. MOV D4, XZR ; Sets D4 equal to XZR, which is 0
  7. MOV D4, WZR ; Sets the first 32-bits of D4 equal to 0, while preserving the last 32-bits
  8. MOV X27, #-255 ; Sets Register X27 equal to -255
  9. MOV W0, #-0x7F ; Sets Register W0 equal to -0x7F (-255)

- We have nine 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 = 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.
- Line 4 literally does nothing. The function NOP, which stands for No Operation, encodes in the program to MOV W0,W0. This way, a person could insert this code here to take up space without doing anything. This is a bit of a spoiler alert, but later on, we'll be replacing in-game instructions with NOP. This basically prevents whatever function that originally existed from having any effect.
- Lines 5-6 are pretty self-explanatory. However, Line 7 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 8-9 are really clones of Lines 2-3 to show that you can move a negative value into a register as well.

LDR 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

- 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.
- 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. Particularly good for storing pointer addresses.
- 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 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 X5 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 X1, [X9, #-0x40] ; Loads Register X1 with the address contained by Register X9 - 0x40

- We have nine lines here, with fifteen instruction total. 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 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 9 follows the same logic, but by subtracting 0x40 instead of adding 0x40.
- Lines 3-5 are pretty much the same thing, but have two instructions instead of one. In Lines 3-5, 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.
- 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.
- With the exception of loading values, LDR instructions pretty much always use Xn on the Nintendo Switch, as all Nintendo Switch memory addresses are Doublewords.

STR 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, noffset] ; immediate offset
  or
- STRT rA, [rD, noffset]! ; pre-indexed offset
  or
- STRT rA, [rD], noffset ; post-indexed offset

- Basically:
- rD = Value contained by rA
  or
- rD + noffset = 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.
- Where rD is the destination register, rA is a register that contains a value, and noffset is the amount to to add or subtract the address by to get the desired address.
- 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 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

- We have eleven 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 = 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.
- 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.

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.

- 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.

RET Instruction

- Remember in the Stack Pointer Register section whenever I talked about how whenever a function is called, the current function on the stack pointer is paused, and the called function is placed on the stack pointer until it ends? Well, RET is the instruction that ends the called function, which then places the previous function back on the stack pointer, allowing it to resume.
- Basically, RET returns to the previous function.
- 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. This would keep the function called by BLR from being called, keeping any sort of calculation from happening to begin with.

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.