Project 1: ARM64 Emulator

# Description

It’s not uncommon to be given an ARM64 executable that is not native to your laptop. The executable may not share the same host OS, architecture, or hardware peripherals. Therefore, emulation is a popular solution to dynamically analyze ARM64 executable code which would otherwise not be possible.

# Assignment

You are tasked with creating a simplified ARM64 assembly emulator. The emulator will be narrowed down to a few of the most common instructions. This emulator is not intended to be used for anything serious, but rather educational to understand how emulators work, ARM64 assembly, and overcoming hard technical challenges.

# Why? How will learning this help me?

*BaCk In My DaY* I didn’t have Ghidra. I didn’t have a decompiler L Decompiler’s are a modern marvel, but they are only as good as the reverse engineer who is using them. I **need** you to be better than the decompiler at reversing code. Why? Decompilers often fail and output incorrect or untrustworthy code. This is commonly seen reverse engineering embedded targets, obfuscated code or determining stack variables. Understanding the assembly enough to **fix** the decompiler is the sweet spot… for now J An advanced understanding of assembly will open doors to binary exploitation and anti-reverse engineering techniques.

# Tasks

This lab is broken down into several, smaller, steps in effort to make this larger project more digestible. Take it one small step at a time!

Note: The “Example output” is there to give you an idea of what is expected to be completed per task. You are welcome to change or improve upon the output of each task as you’d like. The important part is to make sure you output what is asked of each task. Feel free to be creative!

Note: You are permitted to use any third-party libraries you’d like to implement the emulator, so long as the third-party library does not interfere with the purpose of the assignment.

Good use case: third-party library for hexdump.

Bad use case: third-party library that does emulation for you.

If you are unsure, email me.

Note: Complete this project in pairs of 2. Each student must document which portion of the assignment they completed.

Note: Wait to implement 32-bit registers until Task 6.

## Task 1

Create an ARM64 assembly parser. You will be given a text file of ARM64 assembly instructions, and the output should be the instruction, and its operands respectively. One way to implement this is to read one line at a time from the input assembly file, parse the instruction mnemonic from the line, and pass the line to the instruction’s function handler. Each instruction will have its own parser and knows how to parse the instruction string. We will build on this in a later task to then execute/emulate the instruction.

Example input: ADD X1, X2, X3

Example output:

-------------------------------------------------------------------------------------------------------------------------------Instruction #<LINE NUMBER>:

-------------------------------------------------------------------------------------------------------------------------------

Instruction: ADD

Operand #1: X1

Operand #2: X2

Operand #3: X3

Example input: LDR X0, [SP, 0x08]

Example output:

-------------------------------------------------------------------------------------------------------------------------------Instruction #<LINE NUMBER>:

-------------------------------------------------------------------------------------------------------------------------------

Instruction: LDR

Operand #1: X0

Operand #2: [SP, 0x08] --> SP + 0x08

## Task 2

Create ARM64 registers. Since this is a simplified ARM64 emulator, vector/float instructions/registers will be omitted as they are much more complex to effectively implement. Instead, we will only handle 64-bit and 32-bit ARM64 registers (Xn) plus the processor state register.

Create all X0-X30, XZR, SP, PC registers. You can use whatever data structure you’d like to implement these registers.

For the processor state register, we only care about the N and Z bits for CBZ and CBNZ instructions. Feel free to implement storing these 2 bits of information however you see fit. Just know the real processor state register is much more complex J

Example Output:

-------------------------------------------------------------------------------------------------------------------------------

Registers:

-------------------------------------------------------------------------------------------------------------------------------

X0: 0x0000000000000000 X10: 0x0000000000000000 X20: 0x0000000000000000

X1: 0x0000000000000000 X11: 0x0000000000000000 X21: 0x0000000000000000

X2: 0x0000000000000000 X12: 0x0000000000000000 X22: 0x0000000000000000

X3: 0x0000000000000000 X13: 0x0000000000000000 X23: 0x0000000000000000

X4: 0x0000000000000000 X14: 0x0000000000000000 X24: 0x0000000000000000

X5: 0x0000000000000000 X15: 0x0000000000000000 X25: 0x0000000000000000

X6: 0x0000000000000000 X16: 0x0000000000000000 X26: 0x0000000000000000

X7: 0x0000000000000000 X17: 0x0000000000000000 X27: 0x0000000000000000

X8: 0x0000000000000000 X18: 0x0000000000000000 X28: 0x0000000000000000

X9: 0x0000000000000000 X19: 0x0000000000000000 X29: 0x0000000000000000

SP: 0x0000000000000000 PC: 0x0000000000000000 X30: 0x0000000000000000

Processor State N bit: 0

Processor State Z bit: 0

## Task 3

Prepare some stack memory. Stack memory is used extensively by programs as a temporary storage location for variables. You will create a contiguous chunk of memory to be used as stack space. You may use whatever data structure you see fit to implement this stack space. This chunk of memory’s base address will be set in the register SP before executing the first instruction.

On my laptop, my default stack size is 8MB. This is far more memory than what we need and will be cumbersome to keep track of and print to the console. Instead, create a stack size of 256 bytes. This will be plenty to execute our simplified ARM64 instruction set, while still digestible by printing to the console. We will not be implementing any other type of memory. You can set the stack’s base address to whatever you’d like, in this example its set to 0x0.

Example output:

-------------------------------------------------------------------------------------------------------------------------------Stack:

-------------------------------------------------------------------------------------------------------------------------------00000000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|

## 00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|

## 00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|

## 00000030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|

## 00000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|

## 00000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|

## 00000060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|

## 00000070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|

## 00000080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|

## 00000090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|

## 000000a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|

## 000000b0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|

## 000000c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|

## 000000d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|

## 000000e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|

## 000000f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|

## 00000100

Example output: (these are just a bunch of random bytes without meaning)

-------------------------------------------------------------------------------------------------------------------------------Stack:

-------------------------------------------------------------------------------------------------------------------------------00000000 65 98 d8 4e de 3e ef ef 76 fb a2 dd 92 64 26 d4 |e..N.>..v....d&.|

00000010 6b 50 24 e9 91 b8 62 b7 12 40 7e 0f 49 b6 7d b1 |kP$...b..@~.I.}.|

00000020 72 74 66 26 e9 a9 6f 18 02 93 20 e6 ed d6 d5 02 |rtf&..o... .....|

00000030 56 6f 15 7e 32 7e 6d 04 ea 3c 1f 5b 4a 49 7f c1 |Vo.~2~m..<.[JI..|

00000040 a8 66 80 10 68 46 bc f7 6f 23 56 9b 3f 0e 98 b5 |.f..hF..o#V.?...|

00000050 03 f0 09 92 1e 4e a0 ee ed ce 78 ab fb b7 5e fb |.....N....x...^.|

00000060 f2 ad 6a 67 3f eb b4 21 cd ba ba 84 91 85 d6 58 |..jg?..!.......X|

00000070 e2 6d 09 68 10 00 09 aa cc a0 86 13 d9 7f bb 50 |.m.h...........P|

00000080 09 ad 90 58 61 c7 3e a4 bd db b2 8f d7 c4 cb 38 |...Xa.>........8|

00000090 36 c7 f7 99 1c dc 50 15 8d 9e af 4d f8 f9 31 c8 |6.....P....M..1.|

000000a0 43 7e 41 6f 99 85 c1 25 34 76 41 7d 99 fb f5 ba |C~Ao...%4vA}....|

000000b0 5c 97 23 11 6b 49 a7 ac 2f f9 2f 6a 97 c7 b4 d0 |\.#.kI.././j....|

000000c0 7b b0 88 ad ab d8 af 09 a9 19 d8 13 87 8f 18 d5 |{...............|

000000d0 3c 72 8a 06 da 36 54 e6 d2 8e 37 70 7b 6e b9 24 |<r...6T...7p{n.$|

000000e0 c8 d9 9c 5c e9 e4 5b ae 03 e4 5f 95 d1 e9 a2 3c |...\..[...\_....<|

000000f0 f6 94 95 1f 6d de 22 21 63 35 f6 9d 03 da 5e 23 |....m."!c5....^#|

00000100

## Task 4

This task builds upon Task 1. For each line of assembly read from file, update/store the address in PC. All ARM 64-bit instructions are exactly 32-bits in length, or 4 bytes. Start with address 0x0 and increment by 4. Make sure this value updates properly during branching.

## Task 5

This task builds upon Task 1. For each line of assembly read from file, you will have already parsed the instruction into its mnemonic and operands. Now you will execute/emulate the instruction. This could be adding two registers and storing its value or reading/writing to stack memory. Each instruction has its own purpose, and you will write the code to perform that single instruction. Below are the following instructions to implement:

* SUB
* EOR
* ADD
* AND
* MUL
* MOV
* STR
* STRB
* LDR
* LDRB
* NOP
* B
* B.GT
* B.LE
* CMP
* RET
  + Use this instruction to end your emulation

## Task 6

This task builds upon all previous tasks. Now that you have all the instructions implemented for your emulator in 64-bit mode. Now handle 32-bit registers. The change for this is much smaller than you think! Remember, 32-bit Wn register *targets* are zero extended, **but only when they are the destination/target**. Also, I recommend not creating a separate set of 32-bit registers. Use the existing 64-bit registers and handle 32-bit operations accordingly.

Whenever you need to read from a register in 32-bit mode, only read the lower 32-bits (little endian). Whenever you need to write a register in 32-bit mode, only write to the lower 32-bits (little endian) and make the upper 32-bits zero. Done!

## Task 7

Bask in the glory of your emulator and run it against all the supplied input files. The beauty of this project is you will know if you get a 100% A+ grade by knowing whether your emulator output is correct! J

## Task 8

Create a 10-minute presentation discussing what the 2 students learned by creating an ARM64 emulator. Please be creative, no one wants to sit through an entire class of every student presenting the same information!

Example ideas:

* How does your emulator compare to a professional emulator such as qemu or unicorn?
* How could you use your emulator to research a device that you do not physically have?
* Besides adding more supported instructions, what other features would you add to your emulator?
* What technical challenges exist emulating a single ELF executable vs. an entire OS?